Recent Progress in Exact Geometric Computation [⋆]

C. Li

Courant Institute of Mathematical Sciences
New York University, New York, NY 10012, USA.

S. Pion

INRIA Sophia Antipolis, France.

C. K. Yap

Courant Institute of Mathematical Sciences
New York University, New York, NY 10012, USA.

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) has emerged as one of the most successful. This survey describes recent progress in EGC research in three key areas: constructive zero bounds, approximate expression evaluation and numerical filters.

Key words: Exact Geometric Computation, Constructive Zero Bounds, Nonrobustness Problems, Robust Algorithms, Approximate Expression Evaluation, Precision-Driven Computation, Numerical Filters, Interval Arithmetic, C++ Libraries.

[⋆] 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 and by the IST Programme of the EU as a Shared-cost RTD (FET Open) Project under Contract No IST-2000-26473 (ECG - Effective Computational Geometry for Curves and Surfaces).

Email addresses: `chenli@cs.nyu.edu` (C. Li), `Sylvain.Pion@sophia.inria.fr` (S. Pion), `yap@cs.nyu.edu` (C. K. Yap).

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 [93,36]. 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.

There are many possible approaches to solving the non-robustness problem, but in recent years, an approach called Exact Geometric Computation (EGC) has emerged as one of the most successful. Major software libraries such as `LEDA`[48] and `CGAL`[46] offer robust algorithms built on EGC principles. The purpose of this paper is to describe recent developments in three key aspects of EGC: constructive zero bounds, approximate expression evaluation and filter techniques.

Paper Outline. In Section 2, we briefly review three general approaches to numerical non-robustness: arithmetic, geometric and the EGC approaches. Section 3 addresses the problem of *constructive zero bounds*, the theoretical tool that makes EGC possible. Sections 4 treats the problem of *approximate expression evaluation*, the algorithmic centerpiece in any general EGC number library. Section 5 treats *numerical filtering*, the key to practical efficiency in EGC. We conclude in Section 6.

2 Literature Review

The problems of numerical non-robustness have received much attention in the computational geometry community in the last 15 years ([29,45,92,58,20,30]). In [100], this literature is classified along two lines: papers that aim to make fixed-precision algorithms more robust, and those that aim to make the exact algorithms more efficient. Below, we refer to these as the **inexact** and **exact** approaches, respectively. However, our review uses a somewhat orthogonal classification. We focus on two general approaches: the **arithmetic approach** tries to make algorithms robust by making the arithmetic more accurate, while the **geometric approach** achieves robustness by ensuring certain geometric and topological properties of the problem at hand. For more detail surveys, see [100,79,72].

2.1 Arithmetic Approaches

This is a natural first place to look for a solution since 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 surveys of multi-precision numbers, see [101,39]. 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. But for irrational numbers, arbitrary precision no longer ensures error-freeness. The usual understanding of "number representation" is that it is some form of positional number system (basically, as strings of digits in some base). In this sense, the algebraic number $\sqrt{2}$ cannot be represented exactly. However, it is well-known from computer algebra that we can represent and perform all the usual arithmetic operations on algebraic quantities, including exact comparison. Furthermore, by appeal to a general result that goes back to Tarski, any problem that is algebraic can be computed without errors [97]. This general result is the basis for the development of the exact approaches.

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 lengths 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. Thus, Yu [102] concluded that exact rational arithmetic for 3-D polyhedral modeling is impracticable. Karasick et al. [52] 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. But by introducing interval filters, they show that this computation can be sped-up three orders of magnitude. Such results give hope for the exact approach.

Approximate arithmetic is used by the inexact approaches, mostly in the form of machine arithmetic. An early example is Ottmann et al [69] who showed that the use of machine arithmetic with a scalar product primitive can improve the robustness of geometric algorithms. But approximate arithmetic is increasingly used in exact approaches as well. This is a surprising development because in the early days of exact geometric computation, researchers assumed that all arithmetic operations must ultimately be reduced to exact integer computations. In this context, perhaps the first example of the use of approximate arithmetic is [26] where the exact implementation of Fortune's sweepline algorithm is studied. The critical predicates here involve the comparison of square-root expressions. Previously, exact approaches reduce such comparisons to exact integer arithmetic, using repeated squaring. But the use of approximate arithmetic, combined with zero bounds, turns out to be superior. It is also one of the first uses of zero bounds.

It is important to see how approximate arithmetic solves the efficiency bottleneck of naive exact arithmetic. The latter ultimately reduces to big rational arithmetic. For various reasons, rational computations tend to be very slow compared to big integer computations. Big integer arithmetic serves as the baseline of comparison in multi-precision arithmetic. For all practical purposes, multi-precision approximate arithmetic may be identified with big float arithmetic. Big float arithmetic is fast because its complexity is basically that of big integer arithmetic, plus some modest overhead. Equally important, approximate arithmetic algorithms tend to have adaptive complexity which makes them practical.

Other ideas to improve robustness at the arithmetic level (without necessarily completely eliminating non-robustness) are to improve the accuracy or range of fixed precision arithmetic [21,60]. Other techniques here include providing accurate scalar products [69] 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 [50], Pascal-SC [10]).

2.2 Geometric Approaches

The geometric approach leads to considerably more diverse forms than the arithmetic approach. We briefly touch on some major representatives. The general idea is to ensure that certain geometric properties are preserved by the 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 [88]. Topological approaches are subsumed under geometric approaches in our current classification. The first decision facing the robust algorithm designer is the choice of which properties to preserve, and this is dictated by efficiency considerations and needs of the application.

Finite Resolution Geometry. If we compute in fixed precision arithmetic, then one approach is to invent novel "finite resolution geometries" [40] 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 model for finite precision geometry is the integer grid. Greene and Yao [40] investigated line arrangement computation in this geometry. See also [43,41]. Here, line segment may become polygonal lines so that their intersections preserve properties such as non-braiding and connected intersection.

Approximate Predicates and Fat Geometry. Another approach focuses on the imprecise nature of predicate evaluations. The **Epsilon Geometry** of Guibas et al. [42] uses "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. 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, the widespread programming trick called **epsilon tweaking** amounts to a similar fattening of geometric objects [97,81]. Milenkovic [64] 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 Approaches. In the introduction, we noted that the evaluation of predicates in a geometric algorithm determine the combinatorial relations among geometric objects in the algorithm. The **consistency approach** ensures that the computed combinatorial relations are consistent with some perturbation of the numerical input. This amounts to requiring that *the outcome of evaluating any predicate is never inconsistent with previous outcomes.* Fortune [31] noted that in principle it is possible to make most algorithms "parsimonious" (such algorithms never perform conditional tests whose results are implied by the results of previous tests). The reason is that, assuming all predicates are polynomial sign evaluations, the statement that a predicate evaluation is a logical consequence of previous evaluations can be phrased in the language of the existential theory of the reals. The decision problem for this language is at least NP-hard but in polynomial space [19].

In practice, we want more than just consistency. Otherwise, we can give the non-redundant predicates any answer we like! 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 by Sugihara and Iri [90,89,86,87,91]. In making decisions, the combinatorial part is given primacy over the numerical part. As a result, the combinatorial structures are guaranteed to be valid in the special sense of satisfying certain selected properties such as planarity. Similar ideas are advocated by Schorn [80], but phrased in terms of ensuring properties of primitive operations.

For any particular problem, the topology-oriented approach leaves open as to which topological properties the algorithm designer should pick. A general theory is possible using the idea of "realizable" combinatorial structures. This was advocated by Hoffmann et al. [44]. They noted that achieving consistent algorithms is tantamount to theorem proving (akin to Fortune's parsimonious algorithm). Thus the consistency approach is generally infeasible.

2.3 Exact Geometric Computation

We now describe an approach that, strictly speaking, ought to be classified under arithmetic approaches. In [97], we call this the **Exact Geometric Computation** (EGC for short) to emphasize that the "exactness" is in the geometry, not in the arithmetic. EGC is the most successful approach in this informal sense: any problem that has been successfully treated by other approaches can also be treated using EGC, but EGC has done more. Moreover, the EGC solution is often more general and has better properties (since the underlying Euclidean geometry is preserved). But how does an arithmetic approach ensure anything about geometry? The answer is simple. The consistency approach ensures the outcome of the predicate evaluations is correct for some perturbed input (and hence consistent). But EGC insists that *the outcome of every predicate evaluation is correct for the given input* (not some perturbed input). EGC thereby guarantees that the combinatorial output of the algorithm is the exact one. Two important computational consequences follow:

(1) *EGC is computationally feasible.* EGC avoids two infeasibility traps. First, it avoids the infeasibility of the consistency approach: by ensuring the combinatorial output is exact, we achieve consistency without having to do theorem proving. Second, it avoids the infeasibility of naive numerical exactness: we only need to compute to sufficient precision to make the correct predicate evaluation. This adaptive complexity is vital in practice.

(2) *It is possible to create a software EGC library whereby programmers can write robust programs just by calling the library to perform their arithmetic.* For a large class of problems, such an EGC library is a "general solution", as opposed to being a problem-specific or algorithm-specific solution. This contrasts with many geometric approaches, or the earlier work in EGC [26,13],

which offer problem-specific solutions.

The class of problems which is known to be amenable to such a treatment are the algebraic problems [97]. The algebraic problems constitute the majority [1] of problems treated in contemporary computational geometry.

We use the term **EGC numbers** to refer to any number type that supports exact comparisons. The main service of a EGC library is to provide such number types. Early studies in EGC focused on integer or rational number types (e.g., [32]). The `Real/Expr` Package [101] extends this to the class of constructible real numbers (numbers definable over $\{\pm, \times, \div, \sqrt{\cdot}\} \cup \mathbb{Z}$). The `Core Library` [51,47], as the successor of `Real/Expr`, is aimed at achieving a user-friendly interface, especially for user's access of numerical accuracy [96]. The EGC numbers in `Core Library` are called `Expr`; it is the first [2] EGC numbers to incorporate arbitrary real algebraic numbers. In contrast to the `Core Library`, the major libraries `CGAL` and `LEDA` offer, in addition to their EGC number types, also a large repertoire of algorithms, data structures and related services. The EGC number type in `LEDA` is called `LEDA_real`[14,48]. The `CGAL` library [46] offers its own rational EGC numbers, but for more general EGC numbers, `CGAL` uses either `LEDA_real` or `Core Library`.

The Challenges Ahead. EGC is in the midst of an exciting development that seemed quite remote 10 years ago. It has emerged as the dominant approach to non-robustness. Using EGC libraries, programmers can now routinely write completely robust and reasonably efficient geometric code for a large class of problems. Both `LEDA` and `CGAL` have been commercialized and used in industrial applications. For the rational bounded-depth class of problems [97], the consensus is that their nonrobustness problems have essentially been solved (in theory if not in practice). Such problems include convex hulls of points, mesh generation and polyhedral Boolean operations. But what lies ahead?

1. The perennial challenge of efficiency for EGC is currently focused on high degree algebraic computation. Such computations as found in CAD applications remain a severe challenge, as all current CAD software are nonrobust. These issues are just now beginning to be addressed in earnest (e.g., [3,53,28]).

2. When EGC algorithms are embedded in larger application systems (such as a mesh generation system), we need to cascade the output of one algorithm as input to another algorithm. As the output of an EGC algorithm may be in high precision, it is desirable to reduce this precision in the cascade. The

[1] Most problems found in standard references [76,27,68,11,66,22,37] are all algebraic. Non-algebraic examples do arise, e.g., some kinds of Voronoi diagrams, shortest paths with circular obstacles and in non-holonomic motion planning.
[2] Since Version 1.6 (June 2003).

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). We *do not* require that T' is topologically equivalent to T. Otherwise, it is easy to run into NP-hardness, as in [63]. In particular, we allow topology to change in T' (e.g., if two close points in T may collapse to a single point in T'). In the literature, "rounding problems" are often a composition of two problems: a construction problem combined with a bona-fide rounding problem. The snap rounding problem [43] for intersecting line segments is such a composite problem. We prefer to solve such composite problems in two steps: first compute the exact geometric object T, and then rounding T to a lower precision version T'. The first step is considered solved using EGC. So we focus on the second step only. See also [38].

3. The "fundamental problem of EGC" is the **zero problem**. Relative to any set Ω of real algebraic operators, the problem is to decide whether $E = 0$ for any given expression E over Ω. The main open question here concerns the decidability of the zero problem for non-algebraic expressions. This problem will be treated in detail in the next section. See [25] on the introduction of hypergeometric functions into EGC. Without resolving the zero problem for such functions, a weak form of EGC must be adopted.

4. At the practical level, users of EGC libraries often find surprising efficiency penalties for innocuous decisions. For instance, in traversing the vertices of a polygon represented by a circular list $(P_0, P_1, \ldots, P_n)$, one might use a while-loop with the exit test "$P_i = P_0$" (for $i = 1, \ldots n$). In an actual[3] example, where the P_i's are computed points on the unit 2-sphere, this slowed the computation to a halt. 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. For that matter, using machine floating-point arithmetic is not without its pitfalls for the unwary user. EGC arithmetic is no different. Such phenomenon could be avoided by using smart compiler technology in EGC libraries. This area must be addressed if EGC libraries are to be widely used.

5. A model of the EGC mode of computation and its complexity awaits development. A proper foundation requires a theory of real computation that treats the zero problem properly: current theories like computable analysis or TTE [95] make the zero problem undecidable, while the algebraic theory of Blum-Shub-Smale [7] makes the zero problem trivial. We propose in [99], a theory of real approximation that avoids both extremes. Another fundamental issue is the complexity model. Standard complexity analysis based on input size is inadequate for evaluating the complexity of real computation; we need

[3] Private communication, Professor Siu-Weng Cheng.

to express the complexity as a function of the output precision. This is the bit-analogue of output-sensitivity in computational geometry. The study of **precision-sensitive algorithms** is initiated in [83,2].

3 Constructive Zero Bounds

The possibility or impossibility of EGC computation ultimately hinges on the computability of the sign of an expression. For algebraic expressions, it is possible to determine the sign by purely algebraic or symbolic means. However, the current EGC libraries use a numerical approach based on zero bounds. This approach was first used in `Real/Expr`[101].

Throughout 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 is 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 $\Omega = \{0, 1, +, -, \times, \sqrt{\cdot}\}$, then $\mathcal{E}(\Omega)$ is the set of division-free radical expressions. Note that Ω may include 0-ary operators or constants. Thus $E \in \mathcal{E}(\Omega)$ denotes a real value $\mathrm{val}(E)$, defined inductively in the natural way. Since the algebraic operators may be partial, $\mathrm{val}(E)$ may be undefined, denoted $\mathrm{val}(E) =\uparrow$. We say E is **invalid** if $\mathrm{val}(E) =\uparrow$, and E is **valid** otherwise. Thus "expressions" in this paper are essentially straightline programs (cf. [94]), viewed as a rooted, labeled directed acyclic graph (DAG). The sharing of subexpressions is allowed. We generally follow the usual abuse of notation by writing "E" instead of $\mathrm{val}(E)$ when the context is clear.

Definition 1 *We call $b > 0$ a* **zero bound** *(or root bound) for an expression E if the following holds: if E is valid and $E \neq 0$ then $|E| \geq b$. We also say $(-\log_2 b)$ is a* **zero bit-bound** *for E.*

Note that b is a conditional bound: it is *not* a bound when E is invalid or zero. Although there are many kinds of "zero bounds", our definition makes it plain that we are interested in bounding zeros away from 0. To determine the sign of E from a zero bound b, we compute a numerical approximation $\widetilde{E}$ such that if $E =\uparrow$ then $\widetilde{E} =\uparrow$; otherwise, $|E - \widetilde{E}| < \frac{b}{2}$. Then

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

The zero bound determines the worst-case complexity in sign determination. Thus, it is important to find (efficiently computable) zero bounds that are as large as possible. Such bounds may vary greatly depending on the operators in Ω. Call the set Ω a (computational) **basis** if Ω contains $\{\pm, \times\} \cup \mathbb{Z}$ (where $\mathbb{Z}$ are the integers). The following hierarchy of bases is useful in practice:

- Polynomial basis: $\Omega_0 = \{\pm, \times\} \cup \mathbb{Z}$.
- Rational basis: $\Omega_1 = \Omega_0 \cup \{\div\}$.
- Radical (or constructible) basis: $\Omega_2 = \Omega_1 \cup \{\sqrt[n]{\cdot} : n \geq 2\}$.
- Algebraic basis: $\Omega_3 = \Omega_2 \cup \{\mathrm{Root}(P, i) : P \in \mathbb{Z}[x], i \in \mathbb{Z}\}$. Here $\mathrm{Root}(P, i)$ means the i-th largest real root of the polynomial P (useful interpretations can be given when $i \leq 0$). If the coefficients of P are allowed to be other expressions, we get the "diamond operator" of [16] and denote the corresponding basis by Ω_3^+.
- Elementary basis: $\Omega_4 = \Omega_3 \cup \{\exp(\cdot), \ln(\cdot)\}$.
- Hypergeometric basis: $\Omega_5 = \Omega_4 \cup \mathcal{H}$ where $\mathcal{H}$ denotes the set of real hypergeometric functions [25].

Note that $\mathrm{Root}(P, i)$ is a 0-ary operator, used for introducing arbitrary real algebraic values into expressions, while the diamond operator is an anadic operator to create real algebraic values out of other expressions. No efficient implementation of the diamond operator is currently available. In many areas of computational sciences, non-algebraic operators are needed: Ω_4 is the simplest basis beyond the algebraic case. Basic functions such as the trigonometric functions are captured in Ω_5. No known root bounds are known for Ω_4, but Richardson [77] has an important conditional result; partial results are also known from transcendental number theory.

3.1 Review of Constructive Zero Bounds

Zero bounds have been extensively studied in the classical literature (e.g., [59] or [62, chap. 2]). From an algorithmic point of view, many classical results are non-constructive. A "constructive zero bound" for a class $\mathcal{E}$ of expressions is an effectively computable function $B : \mathcal{E} \to \mathbb{R}$ such that $B(E)$ is a zero bound for each $E \in \mathcal{E}$. All the current bounds have the form $B(E) = \beta(u_1(E), \ldots, u_m(E))$ where β is some easy-to-compute **bounding function**, and $u_1(E), \ldots, u_m(E)$ is a set of numerical parameters that are maintained via a set of recursive rules. One of these parameters is invariably an upper bound $D(E)$ on the degree of $\mathrm{val}(E)$. For $E \in \mathcal{E}(\Omega_3)$, we define $D(E)$ to be the product of the degrees of each node in E; the degree of a $\sqrt[k]{(\cdot)}$-node is k, and degree of a $\mathrm{Root}(P, i)$-node is the degree of P, and otherwise the degree of a node is 1. Most of constructive zero bounds cited below are applicable for the class Ω_2, but the exact operator set Ω may be deduced from the given tables.

Canny's bound. Canny [18] shows that given a zero-dimensional system Σ

E	$d(E)$	$\ell(E)$	$h(E)$
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

Table 1
Rules for Degree-Length and Degree-Height bounds

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 Σ. Canny's bound has a proviso, that the homogenized system $\widehat{\Sigma}$ has a non-vanishing U-resultant. Equivalently, $\widehat{\Sigma}$ has finitely many zeros at infinity. Yap [98, p. 350] gave a bound without such a proviso. Such multivariate zero bounds are easily translated into bounds on expressions over Ω_3^+, in the style of [15].

Degree-Length and Degree-Height bounds. The *Degree-Length bound* [98] maintains two numerical parameters: $d(E)$ and $\ell(E)$ which are upper bounds on the degree and length (i.e., 2-norm) of the minimal polynomial of E. The bound here is $B(E) = 1/\ell(E)$ (Landau's bound). A similar *Degree-Height bound* [101] maintains the parameters $d(E)$ and $h(E)$. The bound here is $B(E) = 1/(1 + h(E))$ (Cauchy's bound). These parameters, as with all the parameters in the zero bounds to be discussed, are maintained by induction on the structure of the expression DAG, using a set of mutually recursive rules. Table 1 shows the recursive rules. Note that d_i, ℓ_i, h_i $(i = 1, 2)$ in the table is short hand for $d(E_i), \ell(E_i), h(E_i)$. These rules are justified by the resultant calculus. In practice, we can replace $d(E)$ by the more accurate $D(E)$ defined above.

Degree-Measure bound. Mahler's measure for a complex polynomial $P(x) = a\prod_{i=1}^{n}(x - \alpha_i) \in \mathbb{C}[x]$ is defined as $m(P) = |a| \cdot \prod_{i=1}^{n} \max\{1, |\alpha_i|\}$. For an algebraic number α, $m(\alpha)$ is defined as the measure of its minimal polynomial. It is not hard to show that if $\alpha \neq 0$ then

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

For an expression $E \in \mathcal{E}(\Omega_3)$, let the parameter $M(E)$ be maintained using the rules in column 5 of Table 4. Mignotte [62] shows that $M(E)$ is an upper bound on the Mahler measure $m(E)$. Thus $B(E) = 1/M(E)$ is a zero bound, which Burnikel et al. [15] calls the *Degree-Measure bound*. The Degree-Measure bound turns out to be always better than the Degree-Length bound [15]. Sharpened forms of the original Degree-Measure bound are reported in [57,82].

	E	$u(E)$	$l(E)$
1.	integer a	$\|a\|$	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.	$\sqrt[k]{E_1}$	$\sqrt[k]{u(E_1)}$	$\sqrt[k]{l(E_1)}$

Table 2
BFMS Rules

BFMS bound. One of the best constructive zero bounds for the class of radical expressions is from Burnikel et al. [15] (hereafter called the **BFMS bound**). For division-free expressions, it is an improvement over previously known bounds. 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 recursive rules in Table 2. Clearly, if E is division-free, then $L(E) = 1$ and $\mathrm{val}(E)$ is an algebraic integer. The BFMS bound is

$$B(E) = (u(E)^{D(E)^2-1} l(E))^{-1}. \tag{3}$$

If E is division-free, the bound improves to

$$B(E) = (u(E)^{D(E)-1})^{-1}. \tag{4}$$

BFMSS bound. The zero bit-bound in (3) is quadratic in $D(E)$; this factor can become a serious efficiency issue. Consider a simple example: $E_0(x, y) = (\sqrt{x} + \sqrt{y}) - \sqrt{x + y + 2\sqrt{xy}}$ where x, y are L-bit integers. This expression is identically 0 for any x, y. The BFMS bound with $D(E_0) = 8$ yields a zero bit-bound of $17.5L+\mathcal{O}(1)$ bits. But in case, x and y are viewed as rational numbers (with denominator 1), the bit-bound becomes $157.5L+\mathcal{O}(1)$. Thus introducing rational numbers at the leaves of expression DAGs has a major impact on the BFMS bound. Burnikel et al. [16] extended the BFMS bound to the so-called **BFMSS bound** which is applicable to expressions in $\mathcal{E}(\Omega_3^+)$. But their key improvement was a simple device to avoid the above-mentioned quadratic behavior in many cases. Specifically, in the last row of Table 2, when $E = \sqrt[k]{E_1}$, they propose the new rule $u(E) = \sqrt[k]{u(E_1)l(E_1)^{k-1}}$ and $l(E) = l(E_1)$. But one could equally use $u(E) = u(E_1)$ and $l(E) = \sqrt[k]{u(E_1)^{k-1}l(E_1)}$. Yap noted if we use the symmetrized rule

$$u(E) = \min\{\sqrt[k]{u(E_1)l(E_1)^{k-1}}, u(E_1)\}, \quad l(E) = \min\{l(E_1), \sqrt[k]{u(E_1)^{k-1}l(E_1)}\},$$

then the BFMSS bound is never worse than the BFMS bound. With this modification, the BFMSS bound can now be based on (4). For our expression

$E_0(x, y)$ above, the BFMSS gives a bit-bound of $42L + O(1)$ when the L-bit integers x, y are viewed as rational numbers with denominator 1.

Eigenvalue bound. This bound is based on matrix eigenvalues [78]. 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 easily seen that $\Lambda(n, b)$ is a finite set of algebraic integers. Moreover, if $\alpha \in \Lambda(n, b)$ is non-zero then $|\alpha| \geq (nb)^{1-n}$. Hence, if E is a division-free radical expression, and $n(E)$ and $b(E)$ are parameters such that $\mathrm{val}(E) \in \Lambda(n(E), b(E))$ then we can use the bounding function $\beta(n, b) = (nb)^{1-n}$. The parameters $n(E)$ and $b(E)$ are maintained by Table 3.

	E	$n(E)$	$b(E)$
1.	integer a	1	$\lvert a\rvert$
2.	$\sqrt{cd}$	2	$\max\{\lvert c\rvert, \lvert d\rvert\}$
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	$\overline{P}(n_1 b_1)$

Table 3
Scheinerman's Rules

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 [78], but can be deduced from an example he gave. E.g., to test if $\alpha = \sqrt{2} + \sqrt{5 - 2\sqrt{6}} - \sqrt{3}$ is zero, the Eigenvalue bound requires calculating α to 39 digits [78] while the BFMS bound says 12 digits are enough.

Conjugate bound. This bound from Li and Yap [57] was originally proposed to handle expressions in $\mathcal{E}(\Omega_3)$. The basic idea is exploit the relation that, for any algebraic number $\alpha \neq 0$,

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

where $\deg(\alpha)$ is the degree of α, $\mu(\alpha) = \max\{|\xi| : \xi \text{ is a conjugate of } \alpha\}$, and $\mathrm{lead}(\alpha)$ the leading coefficient of the minimal polynomial $\mathrm{Irr}(\alpha)$. Hence, we may use the bound

$$B(E) = (\overline{\mu}(E)^{(D(E)-1)} \mathrm{lc}(E))^{-1}$$

where the parameters $D(E)$, $\mathrm{lc}(E)$ and $\overline{\mu}(E)$ are upper bounds on (respectively) $\deg(E), \mathrm{lead}(E)$ and $\mu(E)$. In the presence of division, we maintain three more parameters: $\underline{\nu}(E)$ to lower bound $\mu(E)$; $\mathrm{tc}(E)$ to upper bound $\mathrm{tail}(E)$; and $M(E)$ to upper bound $m(E)$. Here $\mathrm{tail}(E)$ is the tail coefficient, i.e., the constant term of $\mathrm{Irr}(E)$. The rules are given in Table 4. The last entry in Line 3 of this Table is missing; this special entry is taken to be $\max\{M(E)^{-1}, (\overline{\mu}(E)^{D(E)-1} \mathrm{lc}(E))^{-1}\}$, and it is justified by (2) and (5).

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

Table 4
Recursive rules for Conjugate Bound

The Factoring Method. Pion and Yap [75] introduced a zero bound technique based on the following idea: maintain a zero bound b for an expression $E = E_1E_2$ in the factored form, $b = b_1b_2$ where b_i $(i = 1, 2)$ is a zero bound for E_i. The b_i's are maintained using one of the previous methods. The trick is to choose an easily factorable form for E where this has an advantage. Indeed there is such an form, namely, $E = E_1 2^n 5^m$ where $n, m \in \mathbb{Z}$. When the input numbers in E are binary or decimal numbers, we can easily extract such powers of 2 and 5 at the leaves. These powers of 2 and 5 are propagated throughout the various nodes. This technique seems generally applicable to the known zero bounds. For instance, applied to the BFMSS bound, we obtain the BFMSS$[2, 5]$ bound. Both the BFMSS$[2, 5]$ and Measure$[2, 5]$ bounds have been implemented in `Core Library`. It is proved that these bounds are never worse than the original version, but can be much better for some expressions (e.g., determinants).

3.2 Comparison of Constructive Zero Bounds

Comparisons between various constructive zero bounds are found in [15,57]. In general, a direct comparison of these zero bounds is difficult because they use different set of parameters and bounding functions. Our strategy to gain some insights is to compare their performance on various special subclasses of expressions. Of the known zero bounds, there are three that are not dominated by any other methods: the BFMSS, Conjugate and Measure bounds. Furthermore, these three are mutually incomparable.
1. For division-free radical expressions, the BFMS bound and the Conjugate bound agree, and both are not dominated by the other known bounds.
2. For general algebraic expressions, in terms of zero bit-bound, Conjugate bound is at most $D \cdot M$ where D is the degree bound, and M is the zero bit-bound from the Degree-Measure bound.
3. For expressions of the form $\sum_{i=1}^{n} \sqrt{a_i}$, the Conjugate bound and the Degree-Measure bound can be better than each other, depending on the size parameters used for the expressions. But both bounds are always better than the

BFMS bound.
4. If E is a radical expression with rational values at the leaves, and E has no divisions or shared radical nodes, the Conjugate bound 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, the Degree-Measure bound requires $(64L + 12)$ bits, and the Conjugate bound requires $(19L + 9)$ bits. To illustrate the effect of these bounds, consider the running time (in seconds) for testing if $E = 0$ in Table 5. Random input numbers with different L values are generated. The platform is a Sun UltraSPARC with a 440 MHz CPU and 512MB main memory.

L	10	20	50	100	200
Conjugate	0.01	0.03	0.12	0.69	3.90
BFMS	0.03	0.24	1.63	11.69	79.43
Degree-Measure	0.03	0.22	1.62	10.99	84.54

Table 5
Timings for Fortune's expression

3.3 Treatment of Special Cases

Zero separation bounds. In both the `Core Library` and `LEDA`, the comparison of two expressions α and β is obtained by computing the zero bound of $\alpha - \beta$. However, more efficient techniques can be used. If $P(X) \in \mathbb{C}[X]$ is a non-zero polynomial, $\mathrm{sep}(P)$ denotes the minimum $|\alpha_i - \alpha_j|$ where $\alpha_i \neq \alpha_j$ range over all pairs of complex zeros of P. When P has less than two distinct zeros, define $\mathrm{sep}(P) = \infty$. Suppose $A(X)$ and $B(X)$ are the minimal polynomials for α and β, then $|\alpha - \beta| \geq \mathrm{sep}(AB)$. If the degrees of A, B are at most d, d', and the heights of A, B are at most h, h', then a zero separation bound for $A(X)B(X)$ (which need not be square-free) is given by

$$|\alpha - \beta| \geq \left[2^{(n+1)/2}(n+1)hh'\right]^{-2n} \tag{6}$$

where $n = d + d'$ (see Corollary 6.33 in [98, p. 176,173] and use the fact that $\|AB\|_2 \leq (n+1)hh'$). The advantage of using (6) is that the zero bit bound here is linear in $d + d'$, and not dd', as would be the case if we use resultant calculus. We compute α and β to an absolute error $< \mathrm{sep}(AB)/4$ each, then declare them to be equal iff their approximations differ by $\leq \mathrm{sep}(AB)/2$ from each other. Otherwise, the approximations tell us which number is larger.

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 the application of EGC to theorem proving [94], the truth of a geometric theorem is equivalent to the zero outcome in many cases. In our numerical approach based on zero bounds, the complexity of sign determination is determined by the zero bound when the outcome is zero. Since zero bounds can be overly pessimistic, 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 iterative numerical method. Yap and Blömer [6] observed that for expressions of the form $E = \sum_{i=1}^{n} a_i\sqrt{b_i}$ ($a_i \in \mathbb{Z}, b_i \in \mathbb{N}$), zero testing is deterministic polynomial time, while the sign determination problem is not known to be polynomial time. Blömer [6,4] extended this to the case of general radicals using a theorem of Siegel; he also gave a probabilistic algorithm for zero test [5]. When the radicals are nested, we can apply denesting algorithms [49,55], but these methods are non-numerical.

4 Approximate Expression Evaluation

At the core of any EGC number library is an algorithm for **approximate expression evaluation**. This is the problem where, given an expression E and a relative or absolute precision p, we want to compute an approximation of $\mathrm{val}(E)$ to within precision p.

The precision measure p we consider here will be either relative and absolute measures: if x, p are real numbers, let $\mathtt{Rel}(x, p)$ denote the set of numbers $\widetilde{x}$ such that $|x - \widetilde{x}| \le 2^{-p}|x|$, and $\mathtt{Abs}(x, p)$ denote the set of numbers $\widetilde{x}$ such that $|x - \widetilde{x}| \le 2^{-p}$. We say that y is a **relative p-bit** approximation of x if $y \in \mathtt{Rel}(x, p)$; **absolute p-bit** approximations are similarly defined.

Let $\mathtt{Approx}(E, p)$ denote such an algorithm to compute an absolute p-bit approximation to $\mathrm{val}(E)$. The choice of absolute (rather than relative) precision here is not arbitrary: the absolute precision version is a more basic problem [99]. The main computational paradigm for such an algorithm is the **precision-driven approach** [101]. Intuitively, this can be viewed as an iterative "downward-upward process" operating on the input expression DAG. In the downward direction, we propagate precision values (starting with p at the root) down to the leaves. Each node u in E represents a subexpression E_u. Let p_u be the precision propagated to u. If u is a leaf node, we assume the ability to compute an absolute p_u-bit approximation for $\mathrm{val}(E_u)$. In the upward direction, we propagate approximations V_u for $\mathrm{val}(E_u)$ up to the root. This upward propagation amounts to a bottom-up evaluation of the expression E.

Our propagation rules provide the following guarantee:

$$\text{If the expression } E_u \text{ is valid, then } V_u \in \mathtt{Abs}(\mathrm{val}(E_u), p_u). \tag{7}$$

Conditional and Unconditional Approximation. An approximation algorithm satisfying (7) is said to have **conditional approximation**. When E_u is invalid, the value V_u can be anything. The algorithm has **unconditional approximation** if, in addition to (7), we have $V_u = \uparrow$ iff E_u is invalid. Here we view $\uparrow$ as a distinguished value, which the algorithm must output (this is unlike standard treatment of partial functions in computability theory). We can convert conditional approximation of E into an unconditional approximation if we can compute zero bounds for all subexpressions E_u. In terms of our hierarchy $\{\Omega_i : i = 0, \ldots, 5\}$, we only know how to compute zero bounds for $E \in \mathcal{E}(\Omega_3^+)$.

REMARKS:
1. Our precision-driven mechanism generalizes the standard "lazy evaluation" technique: the lazy technique has no downward propagation, but has an iterated upward propagation of approximate values, together with some stopping criteria at the root.
2. The optimal method of propagating precision is an open problem. Detailed algorithms for the propagation of "composite precision" (a generalization of relative and absolute precision) are given by Koji [70]. Our simpler approach of propagating only absolute precision follows [56,99].

4.1 Propagation Rules for Elementary Expressions

Let E is an **elementary expression**, i.e., $E \in \mathcal{E}(\Omega_4)$. Table 6 gives the recursive rules to down-propagate precision, and to up-propagate approximations at any subexpression E_u of E. If E_1, E_2 are the subexpressions at the children of the root of E, let p_1, p_2 be the precision to be propagated to E_1, E_2. The downward rules specify the values of p_1 and p_2. This depends on the operator at the root of E. For example, if the operator is $\pm$, then the downward rules in Line 1 says that $p_1 = p_2 = p + 2$. Note that some rules refer to the quantities μ_i^+ or μ_i^-; we will return to this later.

The upward rules tell us how to obtain an approximation $\widetilde{E}$ for E from the approximate values $\widetilde{E_i}$ of E_i $(i = 1, 2)$. These rules are quite uniform: let $E = E_1 \circ E_2$ where $\circ$ is a binary operator in Ω_4. Then $\widetilde{E}$ is defined to be $(\widetilde{E_1} \circ \widetilde{E_2})_{p+1}$. The latter expression describes an absolute $p+1$ bit approximation of $\widetilde{E_1} \circ \widetilde{E_2}$. In general, if X and p are reals then $(X)_p$ denotes any absolute p-bit approximation of X. If the operator at E is unary, an analogous rule applies (see Lines 4,5,6). The proof that Table 6 is correct (i.e., property (7) holds) is found in [99].

		Downward Rules		Upward Rules
	E	p_1	p_2	$\widetilde{E}$
1.	$E_1 \pm E_2$	$p+2$	$p+2$	$(\widetilde{E_1} \pm \widetilde{E_2})_{p+1}$
2.	$E_1 \times E_2$	$\max\{a_1, p+1+\mu_2^+\}$	$\max\{a_2, p+1+\mu_1^+\}$	$(\widetilde{E_1} \times \widetilde{E_2})_{p+1}$
		where $a_1 + a_2 = p+2$		
3.	$E_1 \div E_2$	$p+2-\mu_2^-$	$\max\{1-\mu_2^-, p+2-2\mu_2^- + \mu_1^+\}$	$(\widetilde{E_1}/\widetilde{E_2})_{p+1}$
4.	$\sqrt{E_1}$	$\max\{p+1, 1-(\mu_2^-/2)\}$		$(\sqrt{\widetilde{E_1}})_{p+1}$
5.	$\exp(E_1)$	$\max\{1, p+2+2^{\mu_1^+ +1}\}$		$(\exp(\widetilde{E_1})_{p+1}$
6.	$\ln(E_1)$	$\max\{1-\mu_1^-, p+2-\mu_1^-\}$		$(\ln(\widetilde{E_1})_{p+1}$

Table 6
Absolute Precision Approximation of Elementary Expressions

REMARKS:
1. The approximate value $\widetilde{E}$ is represented by big floats in practice. Thus $\widetilde{E_1} \circ \widetilde{E_2}$ could be computed exactly for $\circ \in \{+, -, \times\}$. But our rules (following [56]) no longer require exact computation, making them more sensitive to the actual precision needed.
2. In [99], rules for propagating relative precision in elementary expressions are given with one exception: no rule for $E = \ln(E_1)$ is possible.

4.2 Propagating Bounds on Magnitude of Expressions

For any expression E, define $\mu(E) = \lg|\mathrm{val}(E)|$. By definition, the $\mu(0) = -\infty$. We can maintain an upper bound μ_E^+, and/or lower bound μ_E^- on $\mu(E)$ using the rules in Table 7. These bounds are the same parameters we encountered earlier in Table 6.

E	μ_E^+	μ_E^-
rational $\frac{a}{b}$	$\lceil \lg(\frac{a}{b}) \rceil$	$\lfloor \lg(\frac{a}{b}) \rfloor$
$E_1 \pm E_2$	$\max\{\mu_{E_1}^+, \mu_{E_2}^+\} + 1$	$\lfloor \lg(\lvert E \rvert) \rfloor$
$E_1 \times E_2$	$\mu_{E_1}^+ + \mu_{E_2}^+$	$\mu_{E_1}^- + \mu_{E_2}^-$
$E_1 \div E_2$	$\mu_{E_1}^+ - \mu_{E_2}^-$	$\mu_{E_1}^- - \mu_{E_2}^+$
$\sqrt{E_1}$	$\lceil \mu_{E_1}^+/2 \rceil$	$\lfloor \mu_{E_1}^-/2 \rfloor$
$\exp(E_1)$	$\left\lceil 1.45 \cdot 2^{\mu_{E_1}^+} \right\rceil$	$\left\lfloor 1.43 \cdot 2^{\mu_{E_1}^-} \right\rfloor$
$\ln(E_1)$	$\lceil \lg(\mu_{E_1}^+) \rceil$	$\lfloor \lg(\mu_{E_1}^-) \rfloor - 1$

Table 7
Rules for upper and lower bounds on $\mu(E)$

Most entries in Table 7 are straightforward: these rules ensure that μ_E^+, μ_E^- are integers. In practice, these values can be big integers or machine doubles. When using machine doubles, we replace the ceiling and floors of Table 7 by appropriate rounding modes; the resulting bounds would be more accurate. Also, the arithmetic would be very fast, but the danger of overflow is greater. The main subtlety in this table is the entry for μ_E^- when $E = E_1 \pm E_2$. Call this the **special entry** because, due to potential cancellation, we cannot derive a lower bound on $\mu(E)$ in terms of $\mu_{E_1}^-$ and $\mu_{E_2}^-$ only. There are two ways to determine this entry. (1) We could approximate E with increasing precision until we see its most significant bit, or reach the zero bound in which case $E = 0$. This method determines $\lfloor \mu(E) \rfloor$ exactly. (2) Method two is applicable only under certain conditions: when E_1 and E_2 have the same sign in case of addition, or have opposite signs in case of subtraction, or their magnitudes differ by more than 1 bit (determined by their μ bounds). In these cases, provided these signs or magnitudes are available, we can deduce μ_E^- directly. Method two acts as a fast filter for method one.

The ComputeMu Algorithm. Based on Table 7, we could develop a simple algorithm called `ComputeMu`(E) to compute μ_E^- and μ_E^+. The `Approx` Algorithm calls `ComputeMu`, but to compute the special entry of Table 7, `ComputeMu` may have to call `Approx`. Hence these 2 algorithms are mutually recursive. This mutual recursion does not lead to infinite loops.

Computing the special entry by method one requires zero bounds. Since zero bounds for elementary expressions are not known, even the *conditional* approximation of arbitrary elementary expressions is an open problem. But in order for `Approx` to succeed on as large a subclass of elementary expressions as possible, `ComputeMu` must avoid computing any μ_E^- or μ_E^+ unless strictly required. We sometimes also need the sign, and this too can be folded into `ComputeMu` in a natural way. The upshot is a multi-argument function,

$$\text{ComputeMu}(E, \texttt{uflag}, \texttt{lflag}, \texttt{sflag}) \tag{8}$$

where the three extra variables are Boolean flags specifying whether the values $\mu_E^+, \mu_E^-, \text{sign}(E)$ (resp.,) are needed. For instance, if $E = E_1/E_2$ then `Approx`(E) will call `ComputeMu`$(E_1, true, false, false)$ and `ComputeMu`$(E_2, false, true, true)$. The reason that the `sflag` is true in the second call is because we need to know whether $E_2 = 0$ or not. These flags originate during the evaluation of E according to Table 6:

- $E = E_1 \pm E_2$. No μ bounds on E_1, E_2 are needed.
- $E = E_1 \times E_2$ or $E = \exp(E_1)$. Only the $\mu_{E_i}^+$'s are needed.
- $E = \sqrt[k]{E_1}$ or $E = \ln(E_1)$. Only $\mu_{E_1}^-$ is needed.
- $E = E_1 \div E_2$. Only $\mu_{E_1}^+$ and $\mu_{E_2}^-$ are needed.

`ComputeMu` recursively calls itself (or `Approx` in case of the special entry), and propagates these flags according Table 7. At any node u in E, when any of the values $\mu_{E_u}^+$, $\mu_{E_u}^-$ and $\text{sign}(E_u)$ have been computed, it is stored at u. `ComputeMu` checks for such values before trying to compute any of them – this is important because nodes are shared.

This completes our description of a precision-driven evaluation. In practice, other features built on top of this overall design (e.g., `Core Library` has a floating-point filter). While our new design is an improvement over an older one [70], it is still suboptimal. For instance, to determine the sign of E_1E_2, we always determine the sign of E_1. But if $E_2 = 0$, this computation is wasted. One solution is to simultaneously determine the signs of E_1 and E_2, stopping immediately when either one returns a 0. The idea is to expend equal effort for the 2 children, but this may be complicated in the presence of shared nodes.

5 Numerical Filters

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, and only go for the slower exact methods when this fails.

In EGC, certifiers are usually **(numerical) filters**. These filters certify property of computed numerical values, 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**. We can also consider a cascade of certifiers of increasing effectivity for the problem. Such cascades can be quite effective [34,17]. There is an obvious connection between the notion of certifiers and the area of program checking [8,9]. It is also worth mentioning that similar techniques have later been used on large determinant sign evaluations based on distance to the nearest singularity [71].

There are two main classifications of numerical filters: static or dynamic. Static filters are those that can be computed at compile time for the most part, and they incur a low overhead at runtime. However, static error 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.

Certifiers can be used at different levels of granularity: from individual machine operations (e.g., arithmetic operations for dynamic filters), to subroutines (e.g., geometric predicates [54]), and to algorithms (e.g., [61]). See Funke et al. [35] for a general framework for filtering each "step" of an algorithm.

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. How can you compute an upper bound on $|z|$ where $z = xy$? We first 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 we can change the rounding modes [93]. Thus changing the rounding mode to round towards $+\infty$, we will have $\widetilde{z} \geq |z|$. 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. We will describe the way to use the rounding modes later, in the interval arithmetic section. But here, instead to avoid 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 the approximation to E evaluated using IEEE double precision arithmetic. Assume the input numbers in E are IEEE doubles and E has $k \geq 1$ operations.*
(i) We can compute an IEEE double value `MaxAbs`(E) *satisfying the inequality* $|E| \leq$ `MaxAbs`(E)*, in $3k$ machine operations.*
(ii) If all the input values are positive, $2k$ machine operations suffice.
(iii) The value $\widetilde{E}$ is available as a side effect of computing `MaxAbs`(E)*, at the cost of storing the result.*

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.

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

Table 8
Parameters for the FvW filter

5.1 Static Filters

Fortune and Van Wyk [33] were the first to implement and quantify the efficacy of filters for exact geometric computation. Their filter was implemented via the `LN` preprocessor system. Let us 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 $\mathtt{MaxErr}(E)$ on the absolute error,

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

It is easy to use this bound as a filter to certify the sign of $\widetilde{E}$: if $|\widetilde{E}| > \mathtt{MaxErr}(E)$ then $\mathrm{sign}(\widetilde{E}) = \mathrm{sign}(E)$. Otherwise, we must resort to some fall back action. For simplicity, assume this action is to immediately use an infallible method, namely computing exactly using a Big Number package.

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

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

We assume that each input variable x is assigned an upper bound $\mathtt{MaxLen}(x)$ on its bit length. Inductively, if F and G are polynomial expressions, then $\mathtt{MaxLen}(E)$ and $\mathtt{MaxErr}(E)$ are defined using the rules in Table 8.

Observe that the formulas in Table 8 assume exact arithmetic. In implementations, we 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 $\mathtt{MaxLen}(E)$ is an integer, we can evaluate the corresponding formulas using IEEE arithmetic exactly. But the formulas for $\mathtt{MaxErr}(E)$ 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 reminds us that this is the "slowdown" factor. Clearly, $\sigma = \sigma(L)$ is a function of the bit length L as well. For instance, with random $L = 53$ bit signed integers, the factor σ becomes 100. Going back to $L = 31$, but using static filters implemented in `LN`, the factor σ ranges from 13.7 to 21.8, for various platforms and CPU speeds [33, Figure 14]. For simplicity, we say $\sigma = 20$, for some mythical combination of platforms and CPUs. Thus the static filters improve the performance of exact arithmetic[4] by the factor

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

In general, using unfiltered exact integer arithmetic as base line, the symbol ϕ (or $\phi(L)$) denotes the "filtered improvement". We use it as a measure of the efficacy of filtering.

The above experimental framework is clearly quite general, and estimates the efficacy of a filter by a number ϕ. 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 used random 31-bit integers). Another measure of efficacy is the fraction ρ of approximate values $\widetilde{E}$ which fail to pass the filter. In [24], a general technique for assessing the efficacy of an arithmetic filter is proposed based on an analysis which consists

[4] The formula is $\phi = \sigma_1/\sigma_2$ where σ_1 is the slowdown factor in an exact number package, and σ_2 is the slowdown factor when the *same* exact number is filtered. But we do not have σ_1 for LN's exact number type, so the LEDA number is used. Remark that since σ_1, σ_2 are both ratios, we need not insist that they be determined on exactly the same set of tests. This flexibility is important if we want to use our framework to compare published filtering results from different papers.

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 written as $\sigma(n, L)$, $\phi(n, L)$, etc.

Realistic versus synthetic problems. Static filters have an efficacy factor $\phi = 3$ (see (14)) in evaluating the sign of randomly generated 4-dimensional matrices ($L = 31$). Such problems are called "synthetic benchmarks" in [17]. It would be interesting to see the performance of filters on **realistic benchmarks**, i.e., actual algorithms for natural problems that we want to solve. But even here, there are degrees of realism. Let us equate realistic benchmarks with algorithms for problems such as convex hulls, triangulations, etc. 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 ought to be different from (in fact, less than) the synthetic setting. To quantify this, suppose that a fraction

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

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 $1 + (\sigma - 1)\beta\phi^{-1}$. So "realistic" efficacy factor ϕ' for the algorithm is

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

It is easy to verify that $\phi' < \phi$ (since $\phi > 1$). 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 [33] noted that $\beta \in [0.2, 0.5]$. Burnikel, et al. [17] suggest a simple method 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.

Some detailed experiments on 3D Delaunay triangulations have been made by Devillers and Pion [23], comparing different filtering strategies; they conclude that cascading predicates is the best scheme in practice. Other experiments on interval arithmetic have been done by Seshia, Blelloch and Harper [84].

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 $\texttt{MaxErr}(E)$ and $\texttt{MaxLen}(E)$ as found in Table 8 for static error. The only difference lies in the base case: for each variable x, the $\texttt{MaxErr}(x)$ and $\texttt{MaxLen}(x)$ can be directly computed from the value of x. It is possible to make a dynamic version of the FvW filter, but we will not detail it here due to lack of space.

The BFS filter. This 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 an upper bound on $|E|$. The second, called the **index** of E, is a natural number whose rough interpretation is that its base 2 logarithm is the number of bits of precision which are lost (i.e. which the filter cannot guarantee) in the evaluation of the expression. Together, they satisfy the following invariant:

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

The value 2^{-53} may be replaced by the unit roundoff error $\mathbf{u}$ in general. Table 9 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

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

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

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

Table 9
Parameters of the BFS filter

Assuming (16), we have 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 10 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. The last column adds compilation to the filtered case. It is based on an expression compiler, EXPCOMP, somewhat in the spirit of LN (see Section 5.3). 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 BFS filter is more efficacious than the FvW Filter. But the use of different Big Integer packages, platforms and compilers, etc, does not justify this conclusion.]

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

Table 10
Random 3×3 determinants

While the above results look good, it is possible to create situations where filters are ineffective. Instead of using matrices with randomly generated integer entries, we can use degenerate determinants as input. The results recorded in Table 11 indicate that filters have very little effect. Indeed, we might have expected it to slow down the computation, since the filtering efforts are strictly extra overhead. In contrast, for random data, the filter is almost always effective in avoiding exact computation.

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

Table 11
Degenerate 3×3 determinants

The original paper [17] 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.

Dynamic filter using interval arithmetic. As mentioned, a simpler and more traditional way to control the error made by floating-point computations is to use interval arithmetic [65,1]. Some work has been done by Pion et al. [73,74,12] in this direction.

Interval arithmetic represents the error bound on an expression E at runtime by an interval $[E_m; E_p]$ where E_m, E_p are floating-point values and $E_m \leq E \leq E_p$. Interval operations such as $+, -, \times, \div, \sqrt{}$ can be implemented using IEEE arithmetic, exploiting its rounding modes. Changing the rounding mode has a certain cost (mostly due to flushing the pipeline of the FPU), but the remark has been made that it can usually be done only twice per predicate: at the beginning by setting the rounding mode towards $+\infty$, and at the end to reset it back to the default mode. This can be achieved by observing that computing $a+b$ rounded towards $-\infty$ can be emulated by computing $-((-a)-b)$ rounded towards $+\infty$. A similar remark can be done for $-, \times, \div$. Therefore it is possible to eliminate most rounding mode changes, which makes the approach much more efficient.

Most experimental studies (e.g. [23,84]) show that using interval arithmetic implemented this way usually induces a slowdown factor of 3 to 4 on algorithms, compared to floating-point. It is also noted that interval arithmetic is the most efficacious dynamic filter, failing rarely. This technique is available in the CGAL library, covering all predicates of the geometry kernel.

5.3 Tools for automatic generation of code for the filters

Given the algebraic formula for a predicate, it is tedious and error-prone to derive the filtered version of this predicate manually. Therefore tools have been developed to generating such codes.

We have already mentioned the first one, LN, which targets the FvW filter [33]. This tool does not address the needs of more complex predicates, which may contain divisions, square roots, branches or loops. Another attempt has been made by Funke et al. [17] with a tool called EXPCOMP (standing for expression compiler), which parses slightly modified `C++` code of the original predicate, and produces static and semi-static BFS filters for them.

The `CGAL` library implements filtering using interval arithmetic for all the predicates in its geometry kernel. The filtered versions of these predicates used to be generated by a Perl script [73,74], but the current approach uses template mechanisms to achieve this goal entirely within C++. The advantage of dynamic filters is that the code generator need not analyze the internal structure of the predicate.

Most recently, Nanevski, Blelloch and Harper [67] have proposed a tool that produces filters using Shewchuk's method [85], for the SML language, from an SML code of the predicate.Seeing these past and ongoing works, it seems important to have general software tools to generate such numerical code. Such work is connected to compiler technology and static code analysis.

6 Conclusions

Exact Geometric Computation is both general and simple: its analysis of the non-robustness phenomenon is completely general, and its prescription for what to do is conceptually simple. It does not require a problem-by-problem analysis of how to apply some meta principle. The availability of EGC libraries such as `LEDA`, `CGAL` and `Core Library` shows that EGC can be made widely accessible to the general programmer. We discussed some key issues in this paper. The sign determination problem and the application of constructive zero bounds are critical. Progress in zero bounds has made them fairly effective for many problems, but it remains a major open problem to construct a zero bound whose bit-length is linear in the expression degree. Adaptive numerical computation is another essential area for improving efficiency of EGC. We discussed two aspects: precision-driven computation and filtering. A challenge in precision-driven computation is to prove, among its many possible variants, one variant that is optimal in a reasonable model of complexity. Filter technology itself can be developed into a profoundly interesting subject on its own right, including making connections to program checking.

Through EGC, numerical non-robustness for a large class of geometric computation has been brought out of the realm of "unsolved" to the realm of "solved but practically challenging". These are problems in low dimensions and low degree problems. For high degree problems, such as arise in computation over curves and surfaces, efficiency remains a serious challenge. This is an area where researchers are beginning to increasingly focus upon. For non-algebraic problems, the fundamental question whether they even admit EGC solutions is completely open. This is the main theoretical open problem of EGC, and relates to some deep questions in mathematics.

In conclusion, we have reasons to be optimistic that EGC concepts will become more and more a part of the computing landscape.

References

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

[2] T. Asano, D. Kirkpatrick, and C. Yap. Pseudo approximation algorithms, with applications to optimal motion planning. In *18th ACM Symp. on Comp. Geometry*, pages 170–178, Barcelona, Spain, 2002. ACM Press. To appear, Special Conference Issue of J.Discrete & Comp. Geom.

[3] E. Berberich, A. Eigenwillig, M. Hemmer, S. Hert, and K. M. und E. Schmer. A computational basis for conic arcs and boolean operations on conic polygons. In *10th European Symposium on Algorithms (ESA'02)*, pages 174–186, 2002. Lecture Notes in CS, No. 2461.

[4] J. Blömer. Computing sums of radicals in polynomial time. *IEEE Foundations of Computer Sci.*, 32:670–677, 1991.

[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] L. Blum, F. Cucker, M. Shub, and S. Smale. *Complexity and Real Computation.* Springer-Verlag, New York, 1997.

[8] M. Blum and S. Kannan. Designing programs that check their work. *J. of the ACM*, 42(1):269–291, Jan. 1995.

[9] M. Blum, M. Luby, and R. Rubinfeld. Self-testing and self-correcting programs, with applications to numerical programs. *J. of Computer and System Sciences*, 47:549–595, 1993.

[10] 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.

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

[12] H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. *Discrete Applied Mathematics*, 109:25–47, 2001.

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

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

[15] 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.

[16] C. Burnikel, S. Funke, K. Mehlhorn, S. Schirra, and S. Schmitt. A separation bound for real algebraic expressions. In *Lecture Notes in Computer Science*, pages 254–265, 2001.

[17] C. Burnikel, S. Funke, and M. Seel. Exact geometric computation using cascading. *IJCGA (Special issue)*, 11(3):245–266, 2001.

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

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

[20] 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).

[21] 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.

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

[23] O. Devillers and S. Pion. Efficient exact geometric predicates for Delaunay triangulations. In *Proc. 5th Workshop Algorithm Eng. Exper.*, pages 37–44, Jan. 2003.

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

[25] Z. Du, M. Eleftheriou, J. Moreira, and C. Yap. Hypergeometric functions in exact geometric computation. In V.Brattka, M.Schoeder, and K.Weihrauch, editors, *Proc. 5th Workshop on Computability and Complexity in Analysis*, pages 55–66, 2002. Malaga, Spain, July 12-13, 2002. In Electronic Notes in Theoretical Computer Science, 66:1 (2002), `http://www.elsevier.nl/locate/entcs/volume66.html`.

[26] 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`.

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

[28] I. Z. Emiris, A. V. Kakargias, S. Pion, M. Teillaud, and E. P. Tsigaridas. Towards an open curved kernel. In *Proc. 20th Annu. ACM Sympos. Comput. Geom.*, 2004. to appear.

[29] 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.

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

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

[32] 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.

[33] 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.

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

[35] S. Funke, K. Mehlhorn, and S. Näher. Structural filtering: A paradigm for efficient and exact geometric programs. In *Proc. 11th Canadian Conference on Computational Geometry*, 1999.

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

[37] J. E. Goodman and J. O'Rourke, editors. *Handbook of Discrete and Computational Geometry.* CRC Press LLC, 1997. Second edition expected in 2003.

[38] 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.

[39] P. Gowland and D. Lester. A survey of exact arithmetic implementations. In J. Blank, V. Brattka, and P. Hertling, editors, *Computability and Complexity in Analysis*, pages 30–47. Springer, 2000. 4th International Workshop, CCA 2000, Swansea, UK, September 17-19, 2000, Selected Papers, Lecture Notes in Computer Science, No. 2064.

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

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

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

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

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

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

[46] The CGAL Homepage. Computational Geometry Algorithms Library (CGAL) Project. URL `http://www.cgal.org/`.

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

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

[49] 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.

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

[51] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A Core Library for robust numeric and geometric computation. In *Proc. 15th ACM Symp. on Comput. Geometry*, pages 351–359, June 1999.

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

[53] M. I. Karavelas and I. Z. Emiris. Root comparison techniques applied to computing the additively weighted Voronoi diagram. In *Proc. 14th ACM-SIAM Sympos. Discrete Algorithms (SODA)*, pages 320–329, 2003.

[54] L. Kettner and E. Welzl. One sided error predicates in geometric computing. In K. Mehlhorn, editor, *Proc. 15th IFIP World Computer Congress, Fundamentals - Foundations of Computer Science*, pages 13–26, 1998.

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

[56] C. Li. *Exact Geometric Computation: Theory and Applications*. Ph.d. thesis, Department of Computer Science, New York University, Jan. 2001. Download `http://cs.nyu.edu/exact/doc/`.

[57] 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.

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

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

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

[61] K. Mehlhorn, S. Näher, T. Schilz, R. Seidel, M. Seel, and C. Uhrig. Checking geometric programs or verification of geometric structures. In *Proc. 12th ACM Symp. on Computational Geom.*, pages 159–165. Association for Computing Machinery, May 1996.

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

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

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

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

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

[67] A. Nanevski, G. Blelloch, and R. Harper. Automatic generation of staged geometric predicates. In *International Conference on Functional Programming*, Florence, Italy, 2001. Also Carnegie Mellon CS Tech Report CMU-CS-01-141.

[68] J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, second edition edition, 1998.

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

[70] K. Ouchi. Real/Expr: Implementation of an exact computation package. Master's thesis, New York University, Department of Computer Science, Courant Institute, January 1997. URL `http://cs.nyu.edu/exact/doc/`.

[71] V. Y. Pan and Y. Yu. Certification of numerical computation of the sign of the determinant of a matrix. *Algorithmica*, pages 708–724, 2001.

[72] 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 Conf.*, pages 139–140, 1995.

[73] S. Pion. *De la géométrie algorithmique au calcul géométrique.* Thèse de doctorat en sciences, Université de Nice-Sophia Antipolis, France, 1999. TU-0619.

[74] S. Pion. Interval arithmetic: An efficient implementation and an application to computational geometry. In *Workshop on Applications of Interval Analysis to systems and Control*, pages 99–110, 1999.

[75] S. Pion and C. Yap. Constructive root bound method for k-ary rational input numbers, September, 2002. Extended Abstract. Submitted, 2003 ACM Symp. on Comp. Geom.

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

[77] D. Richardson. How to recognize zero. *J. of Symbolic Computation*, 24:627–645, 1997.

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

[79] 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.

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

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

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

[83] J. Sellen, J. Choi, and C. Yap. Precision-sensitive Euclidean shortest path in 3-Space. *SIAM J. Computing*, 29(5):1577–1595, 2000. Also: 11th ACM Symp. on Comp. Geom., (1995)350–359.

[84] S. A. Seshia, G. E. Blelloch, and R. W. Harper. A performance comparison of interval arithmetic and error analysis in geometric predicates. Technical Report CMU-CS-00-172, School of Computer Science, Carnegie-Mellon University, 2000.

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

[86] 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.

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

[88] 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.

[89] 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.

[90] 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.

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

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

[93] 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.

[94] D. Tulone, C. Yap, and C. Li. Randomized zero testing of radical expressions and elementary geometry theorem proving. In J. Richter-Gebert and D. Wang, editors, *Proc. 3rd Int'l. Workshop on Automated Deduction in Geometry (ADG'00)*, number 2061 in Lecture Notes in Artificial Intelligence, pages 58–82. Springer, 2001. Zurich, Switzerland.

[95] K. Weihrauch. *Computable Analysis*. Springer, Berlin, 2000.

[96] 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. For abstracts, see `http://www.cs.brown.edu/cgc/cgc98/home.html`.

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

[98] C. K. Yap. *Fundamental Problems in Algorithmic Algebra.* Oxford Univ. Press, 2000.

[99] C. K. Yap. On guaranteed accuracy computation. In F. Chen and D. Wang, editors, *Geometric Computation.* World Scientific Publishing Co., Singapore, 2004. To appear.

[100] C. K. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 41, pages 927–952. CRC Press LLC, Boca Raton, FL, 2nd edition edition, 2004.

[101] 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.

[102] 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.