Towards Exact Geometric Computation *

Chee-Keng Yap
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, NY 10012

July 12, 1993

Abstract

Exact computation is a fundamental premise of most algorithms in computational geometry. In practice, implementors perform computation in some fixed-precision model, usually the native floating-point arithmetic. Such implementations have many well-known problems, here informally called "robustness issues". To reconcile theory and practice, authors have suggested that theoretical algorithms ought to be redesigned to become robust under fixed-precision arithmetic. We suggest that in many cases, implementors could make robustness a non-issue by computing *exactly*. The advantages of exact computation are too many to ignore. Many of the presumed difficulties of exact computation are partly surmountable and partly inherent with the robustness goal.

We examine the practical support needed to make the exact approach a viable alternative. It turns out that exact computation encompasses an extremely rich set of computational tactics. Our fundamental premise is that the traditional "BigNumber" package that forms the work-horse for exact computation must be re-invented to take advantage of many features found in geometric algorithms. Beyond this, we postulate several other packages must be built on top of the BigNumber package.

1 Introduction

In recent years, there has been considerable interest in "robust" geometric algorithms. In practical terms, an algorithm is termed non-robust if it can precipitate unpredictable failures during execution. It is clear that such failures occur

*Work on this paper is supported by NSF grant #CCR-87-03458

with a sufficiently high probability to cause widespread concern. This concern is expressed in many diverse communities, and many ingenious and practical solutions have been proposed. The unexamined premise in most of these solutions is the commitment to *fixed-precision computation*. We suggest that the alternative approach based on *exact computation* has a much larger role to play than currently practiced. In any case, the goal of robust computation is better served when both approaches are well-represented. Of course, we are partisan in this quest, and only hope to contribute to the development of exact computation. This paper outlines our thoughts on a research agenda that forms the basis of some on-going research with Tom Dubé [5]. Although we address ourselves to geometric algorithms, it is clear that many of these ideas apply to related fields.

Fixed-precision Computation. The root cause of non-robustness seems clear: whereas algorithms are described in exact mathematical terms, their implementations replace the exact arithmetic with fixed-precision arithmetic. Floating-point arithmetic is the usual example of fixed-precision arithmetic. The high probability of catastrophic loss of significance in such computations in practice is confirmed in theoretical models ([8]). More powerful fixed-precision models (e.g., level-index arithmetic [2]) may be useful in delaying the robustness problem. In practice, the problem is fixed using some ad-hoc method that, at best, decreases this failure probability. It often amounts to what is known in the trade as "epsilon tweaking" (choosing the right constant for some epsilon parameter in the code). Observe that robustness issues already appear in purely numerical computation (e.g., [21]); one can only imagine such problems to be compounded in geometric computing, whose essence is captured in the aphoristic equation,

$$\text{Geometric Computing} = \text{Numerical+Combinatorial Computing}.$$

Robustness issues arising from the interplay between numerical and combinatorial elements of geometric algorithms is illustrated in, for example, Hoffmann [11]. To address this problem, some have insisted that algorithmic design should take account the use of fixed-precision arithmetic. This has led to the following difficulties:

- Robust algorithms are not known for many conceptually simple problems. For instance, the problem of intersecting two polygons [25].
- When robust algorithms are achievable, they seem to require inordinate effort relative to the known exact algorithms. Moreover, the techniques do not easily generalize. But even the true significance of these robust algorithms may be open to interpretation (see §3.2).
- Fixed-precision geometric models to approximate the original continuous models are invariably hard to work with and retain very few of desired

properties. For instance, the concept of a fixed-precision "line" has variously been modeled by (i) using a suitable set of pixels, (ii) fattening the line into a tubular region, (iii) by some "monotone" polygonal path, or (iv) an actual line whose equation has bounded coefficient sizes. Beside losing many desirable properties of lines, these models give rise to complicated algorithms.

- A more basic approach is go back to the arithmetic model and to introduce uncertainty there. Logically, this means we have at least a third truth value corresponding to "not-sure". There are many forms of this approach. For instance, the well-known interval arithmetic. Symptomatic of this general approach, we find that the intervals in interval arithmetic can quickly grow into fairly worthless bounds in the course of a computation.

These criticisms cannot be attributed to a lack of effort (although, it is true that computational geometers are relative new-comers in this enterprise). The optimist might still say that we need more time. But perhaps the difficulty is more basic: there will be no satisfactory solution until we confront the specter of exact computation and understand what is inherently involved there. We hope to show that there is much more to the idea of exact computation than what we might initially suspect.

Exact Computation. We now switch to a discussion of exact computation. Roughly speaking, "exact computation" means that we do not use any kind of approximation, and so no errors are ever committed during the computation. The first and foremost advantage of exact computation is that "robustness" is a non-issue! All classical geometric concepts are preserved. In contrast to the obscure theories of approximate geometry, classical geometries (Euclidean or otherwise) have many theorems and many important cases (planar geometry!) are relatively well-understood. So we can reason with classical objects with relative confidence. More pertinent, practically all geometric algorithms in the literature pertains to classical geometries. Thus, exact computation is a "generic" solution (cf. [22]) to the robustness issue, not a special fix for each particular problem. Another advantage is that we can sometimes use symbolic perturbation methods to automate the handling of degeneracies, thus simplifying our coding of algorithms (cf. [7,23]).

Given these advantages, why is exact computation is almost never used in practice? We suggest that misconception and culture each has a role. For many, it is simply assumed that except for very special domains such as number theory and algebra, all continuous domain computations must be approximate. This common misconception is easy to dispose of. The claim that exact computation is too inefficient is much harder to counter. The floating point culture is so entrenched and enjoys so much infrastructural support (hardware or otherwise) that this claim is partly self-fulfilling. It is true in some sense that exact computation is inherently slower than floating point. But by the same token, one can

claim that floating-point is inherently non-robust. Then it is up to the user to decide which horn of this dilemma to choose. (Of course the truth is somewhere between these two positions.) While we cannot make that decision for any user, we believe that the user should be presented with viable alternatives. It is the premise of this research that the true viability of exact computation has not been well-represented. So this is our first goal:

(G1) To improve the practical cost of exact computation.

The emphasis here is on "practical", although we indicate interesting theoretical issues as well. For now, we just note that what makes (G1) interesting is the fact that exact computation turns out to be extremely rich – it is not just a matter of carrying out each arithmetic operation without error (which would be boring indeed).

With respect to the user dilemma above, it is clear that certain users are unwilling to pay the inherent cost of exact computation. For instance, [25] concluded that "exact computation is not feasible for the problem of point classification". There may be hidden assumptions that make this conclusion true, but in general, such unqualified conclusions are unjustified. Surely if robustness is important enough (say, it relates to the success of a mission into space), then exact computation may be the only choice. The literature contains many such claims based on ad hoc or unjustified criteria. We need some theoretical framework to mediate the true differences between exact and finite-precision computation. This is the basis for a second goal:

(G2) To study the inherent tradeoffs between speed and precision, between fixed-precision and exact computation.

This is a more abstract goal, involving the construction of theoretical models and posing paradigmatic problems. In this regard, we may recall the conceptual framework that complexity theory provides for the entire field of algorithms.

Varieties of precision. We should acknowledge that any simple characterization of exact versus fixed-precision approaches will runs into gray areas. For instance, we may distinguish between degrees of fixed-precision: the most restrictive form of fixed-precision prevails in most practice, where there is a universal precision depending on the machine word-size for all computations. A local form of fixed-precision is where each variable carries its own precision which is fixed throughout the computation. This idea is very useful if the caller of the geometric algorithm is only capable of using a certain of precision. For instance, the caller may be a graphics display unit that has a certain screen resolution. Indeed, as we will show, many exact algorithms can be carried out using this local version of fixed precision. This shows exact computation may well allow some internal approximation, which is indicative of the richness of what comes under the rubric of exact computation. Conversely, not all algorithms that use

some local fixed-precision qualifies as "exact". Other variations are possible: for example there is a language developed for numerical computation there is the concept of a *precision block*. One iterates the computation of that block with increasing precision until a desired goal is attained.

Remark. There are *genuine* problems of rounding or approximation. That is to say, there are rounding questions that are inherent in the problem formulation, not just artifacts of using fixed-precision arithmetic to approximate exact arithmetic. A simple example is the problem of transforming a simple polygon so that it remains a simple polygon but such that each vertex is "snapped" to one of the four corners in the lattice square containing the vertex. It is not surprising that such problems can be NP-hard in view of their combinatorial nature. But they are outside our scope.

2 What is Exact Computation?

We first clarify our use of the term. By an "exact computation", we mean that a computational process that
(i) represents the underlying mathematical objects in an *exact* manner, and
(ii) in the course of computation, never makes an error in its decision.

We understand in (i) that mathematical objects are characterized by suitable numerical parameters. To say that that the parameters "exactly" represent an object means that we can decide when two such objects are equal or not equal from these parameters. The representation (i.e., parameters) need not be unique.

Example. These concepts are illustrated by the representation of algebraic numbers. By definition, an algebraic number is the root of an univariate polynomial with integer coefficients. For instance, the number $\sqrt{5}$ is an algebraic number as it is a root of $X^2 - 5$. We know that there is no finite representation of $\sqrt{5} = 2.236068...$ in positional notation. But $\sqrt{5}$ can be represented *exactly* as the pair $(X^2 - 5, [1, 4])$, interpreted as the unique root of the polynomial $X^2 - 5$ lying in interval $[1, 4]$. This is called the *isolating interval representation* of real algebraic numbers. Of course, $(X^3 - 5X, [2, 3])$ would represent the same number exactly, while $(X^2 - 5, [-3, 3])$ represents no number because the range $[-3, 3]$ does not contain a unique root of the equation.

Clearly the precision of numbers used in such representations must be arbitrarily large. The fact that we can represent $\sqrt{5}$ exactly suggests that in some sense, we have infinite precision. However, the terms "arbitrary precision computation" or "infinite precision computation" are inadequate substitutes for "exact computation". We said that exact computation means that nothing is done approximately. In some sense, $[1, 4]$ is an approximation to $\sqrt{5}$, and

[2, 3] is an even better approximation. But our representation of $\sqrt{5}$ itself is no approximation.

We understand in part (ii) in our definition of exact computation that, with respect to the representation of objects, there are effective procedures to compute and make decisions about these objects. In the context of algebraic numbers, this usually means that we can perform the usual arithmetic operations $(+, -, \times, \div)$ and determining the sign of real algebraic numbers. Henceforth, we focus only on real algebraic numbers since the complex ones can be represented as a pair of real algebraic numbers. Note that the set of real algebraic numbers is closed under the arithmetic operations. For instance, if α is a root of $\sum_{i=0}^{n} c_i X^i$ then $-\alpha$ is a root of $\sum_{i=0}^{n}(-1)^{n-i}c_i X^i$. If α, β are roots of $P(X), Q(X)$ (respectively) then $\alpha + \beta$ and $\alpha\beta$ are roots of

$$\mathrm{res}_Y(P(Y), Q(X - Y)), \quad \mathrm{res}_Y(P(Y), Y^n Q(X/Y))$$

where $\mathrm{res}_Y(P(Y), Q(Y))$ denotes the classical resultant of two polynomials in Y. Since a resultant is a determinant, and using some classical bounds on the separation of roots, we conclude that the the basic arithmetic operations and the sign of algebraic numbers can be effectively computed. For instance, we should be able to give the isolating interval representation of $\sqrt{5} - \sqrt{3}$ and determine the sign of $2\sqrt{5} - 2\sqrt{3} - 1$. Actually, algebraic numbers have another important closure property: the root of a polynomial with algebraic coefficients is algebraic. We roughly call this the *root extraction* operation (begging the question as to how one specifies the particular root of interest). Note that division and subtraction can be viewed as root extractions of linear polynomials. For more details on computing with algebraic numbers, see for instance [3,24].

2.1 Algebraic problems

Our example of algebraic numbers is felicitous because *most problems in computational geometry can be computed exactly, via a reduction to exact algebraic number computations.* Of course, it must be assumed that the inputs to a problem is exact. We call such problems *algebraic.* Let us illustrate some algebraic problems.

Consider the problem of finding the shortest path between two points avoiding a set of polyhedral obstacles in Euclidean n-space. Euclidean distances between two points involve the taking of square roots. Thus the length of a polygonal path in n-space is a sum of square roots, and thus an algebraic number. All known shortest path algorithms can be reduced to making decisions by comparing the relative lengths of two polygonal paths. Thus such problems can be solved exactly. Similar problems include the Euclidean versions of spanning tree and traveling salesman.

Certain problems that apparently involve transcendental functions are actually algebraic problems in disguise. For instance, consider the so-called motion planning problem [14] where the robot and the obstacles have piecewise algebraic

boundaries, and we seek the feasibility of an obstacle-avoiding motion between two positions. Since the robot can rotate, we might generally expect that the calculations would involve trigonometric functions. However, it turns out that we can exploit the algebraic relations among such functions and avoid transcendental decisions. Thus we view $\sin x$ and $\cos x$ as two algebraic quantities connected by the relation $\sin^2 x + \cos^2 x = 1$.

The class of algebraic problems is very large. We caution that in general, these problems have exponential complexity. Essentially, these problems can all be reduced to the decision problem in Tarski's language, which is the first order theory of real closed fields. The general upper bound on these problems has seen tremendous progress in recent years. These results yield the following important meta-theorem: *algebraic problems in computational geometry have a single-exponential space complexity.* Furthermore, some of these problems are provably intractable – by reduction to the decision problem of the theory of real addition, which Rabin and Fisher have shown to be nondeterministic exponential-time complete.

3 Rational Bounded-degree Problems

Since algebraic problems in general are intractable, we seek tractable subclasses. Many problems do not require the full power of algebraic computation, requiring only the four arithmetic operations but not root extraction. *Assuming the problem inputs involve only rational numbers*, we call these *rational problems.* For instance, linear programming or constructing hyperplane arrangements [6]. Note that essentially the convex hull problem is reducible to hyperplane arrangements.

Another important restriction is the concept of *degree of derivation* (cf. [23]): relative to a set U of numbers, a number x is of degree 0 if $x \in U$; x is of degree at most $d+1$ if x is obtained by one of the rational operations applied to numbers of degree at most d, or by root extraction from a degree k polynomial where each coefficient of the polynomial has degree at most $d - k + 1$. An algorithm has *degree* at most d if there is a finite set K of numbers (i.e., the constants in the algorithm) such that on any input set X, all intermediate values computed by the algorithm are of degree at most d relative to $U = K \cup X$. A problem is *bounded-degree* if it can be solved by an exact algorithm of at most some fixed degree.

A problem is *rational bounded-degree* (RBD, for short) if it can be solved by an algorithm of bounded-degree that performs only rational operations. Of course, the (actual) *degree* of an algorithm or problem is the least d such that it is of degree at most d. We note that the class of RBD problems encompasses most of the computational problems in contemporary computational geometry (for instance, see the problems treated in the standard texts [6,13]). The following is an obvious but key property of RBD algorithms:

There is a constant D such that if the input instance to the algorithm involves rational numbers of size (at most) n, then the intermediate computation involves only rational numbers of size Dn.

Here, the size of a rational number p/q is just the maximum of the bit sizes of p and q. Thus, assuming that n is the machine word size, then we can implement all arithmetic exactly by representing an integer with D words. We conclude: *RBD algorithms can be implemented using exact arithmetic with only a constant factor C slow-down.* Clearly, C depends on D. Using classical algorithms for arithmetic, we have $C = O(D^2)$.

The constants C and D are crucially important to our goal (G1). Note that we are outside the realm of asymptoptics when we discuss these constants. We can take $D = 2^d$ if the RBD algorithm has degree d. This 2^d bound could be improved in some cases.

Example. Suppose our algorithm only has to compute (repeatedly) determinants of $k \times k$ matrices, for some fixed k. Moreover, assume that the determinants are evaluated on the input values. It is well known that convex hulls of point sets in dimensions $k-1$ can be solved by such algorithms.[1] The degree of the algorithm depends on how one implements the determinant computation. For instance, using standard Gaussian elimination, the degree is $d = O(k^3)$ but $D = 2^k$ is overly pessimistic. In the full paper, we discuss this in more detail, showing how homogeneous coordinates can help to reduce D.

3.1 Unbounded-degree problems

There are not many natural problems in traditional computational geometry that are rational but not bounded-degree. But imagine a geometric editor involving just points and lines, and where we are allowed to construct a new point as intersection of two prior lines and we can construct a new line through two prior points. Note that this is not a "computational problem" in the usual understanding of the word. Alternatively, imagine a solid polyhedral modeler in which we can do rational transformations of solids and perform Boolean operations on solids. Each transformation and operation on these solids increases the degree of derivation. Clearly each intersection or transformation step increases the degree of derivation of the object, and we will generally need more and more precision to represent the objects exactly. It is not surprising that fixed-precision fails notoriously. It would be interesting to formalize and prove that this must be so (goal (G2)). An artificial form of this phenomena which is useful for numerical experimentation was invented by Dobkin and Silver [4]: it is based on repeated application of two operations (going-in, going-out) on an

[1] Such algorithms can also solve convex hulls in dimension k but the determinants are evaluated on values of degree 1 rather than degree 0 (input values).

initial pentagon and the fact that in the exact world, going-in and going-out are inverses.

3.2 Some robust algorithms

There are some apparent successes in achieving robust algorithms using fixed-precision. For instance, a systematic approach to robustness has been outlined by Sugihara and Iri in several papers (e.g., see [18,17,19,20]). They propose to view geometric algorithms as constructing combinatorial structures guided by numerical computations. If we can structure such algorithms so that no redundant combinatorial decisions are made, then the algorithm can be made robust. More generally, we may say that the philosophy is to give priority to combinatorial data over numerical data. The Sugihara-Iri approach has been applied to several examples such as Delaunay triangulations and the gift-wrapping 3-D convex hull algorithms. Still, stability does not seem easy to achieve [20]. As another example, Fortune [9] has described two robust $O(n^2)$ algorithm for planar Delaunay triangulation.

It turns out that these success stories all fall under the RBD class. So we could solve these problems exactly, at the cost of some constant C multiplicative factor. Why would one exchange an $Cn \log n$ exact algorithm for an $C'n^2$ approximate algorithm as in the Delaunay triangulation problem? This depends on C and C'. It is believable that with fine-tuning, one can make this C competitive, even assuming $C' = 1$. One of our goals is to achieve this using techniques that are general, rather than just special to say, Delaunay triangulations. We have here a concrete research question: carry out empirical studies comparing the robust fixed-precision algorithms to exact algorithms for these problems.

4 Re-inventing BigNumbers

We now address research goal (G1), which is simply to reduce the cost of using exact computation. Just as a floating-point package is the basis of most fixed-precision computation, a "BigNumber package" is the basis of all exact computation. Naturally this must be the first place to begin our investigation.

BigNumber packages, although widely available, have no hardware support (and barely a priority for software support). A notable attempt to put large integer multiplication in hardware is reported in [1,16] using the concept of programmable active memories. Their hardware multiplies 512-bit integers; when coupled to low-end workstations, it apparently outperforms the fastest computers of its day (circa 1990). One should note that the motivation there is cryptography, which has different concerns than us. Still, such a piece of hardware would go a long way towards making exact solution of RBD problems competitive and practical. While this is surely an avenue for more work, we henceforth focus on software solutions.

First, we can simply try to improve on traditional BigNumber packages. One attempt is reported by Vuillemin, Hervé and Serpette [15]. They also suggested that any BigNumber package written in a high level language stands to gain a factor of $4 - 10$ when hand-crafted code is employed. In view of the anecdotes (see below), this improvement alone is insufficient.

Some anecdotes. The first reality we face when using BigNums is not encouraging: *off-the-shelf use of this package incurs a tremendous overhead.* For instance, in the case of exact *integer* computations, Fortune and Van Wyk [10] said that their program becomes slower by a factor of 100-140. Note that the comparison is made against a floating-point implementation. This methodology seems standard and we will keep it for this discussion. If exact *rational number* computations are used off-the-shelf, Karasick, Lieber and Nackman [12] reports a slowdown factor of 15,000. The good news is that in both cited papers, careful fine-tuning eventually reduce these factors to a small constant factor (say, less than 10).

What is the significance of this "anecdotal number 10"? Note that a factor of 10 in the numerical part of an algorithm would only slow the overall algorithm down by a factor of 3 if the algorithm uses only 25% of its time in number crunching. For many applications, we believe such a small penalty tilts the balance in favor of exact computation if robustness is important. In any case, comparing an exact algorithm against an approximate algorithm (assuming that robustness has been achieved) which is thrice faster must come down to user priorities. But more can be said. For the sake of argument, let us assume that the anecdotal number 10 is technology-independent, that is, it will not change with improving hardware. In a world where machine speed doubles every other year, a small technology-independent constant seems negligible.[2] Again, with the increasing commercial availability of medium-scale parallel computers (of a dozen nodes, say), small technology-independent constants will be even less significant.[3]

It is not hard to identify one source of inefficiency with standard BigNum packages. One pays a large overhead for its generality, and in particular, one pays for its space management facilities. The heritage of "BigNumber packages" seems to come from computer algebra applications. The basic assumption in computer algebra systems that we cannot predict the precision needed during a computation is reasonable. But in computational geometry, as we have argued, the opposite assumption usually holds. Hence one approach is to build a *poor-man's BigNum package* to exploit this property. In other words, we wish the constant factor C to as close to $1 \cdot D^2$ as possible. In fact, more sophisticated $o(D^2)$ techniques seems possible with handcoding for small D. Alternatively, we want C to approach the anecdotal number 10. ($D^2 = 10$?)

[2] Just wait a few years instead of doing any research. : −)

[3] Just throw some $$$ at the problem instead of waiting. : −()

5 Beyond BigNumbers

To support a rich geometric computing environment, a number of other packages must be built on top of the BigNum package. We suspect that this, rather than achieving the ultimate value of the constant C, will encourage more use of exact computation. We outline some of these.

BigFloats. We had already mentioned that the idea of computing exactly is capable of a variety of interesting interpretations. One is the idea of computing each number x_i up to some prescribed precision p_i. In the usual fixed-precision, there is a fixed p for the entire computation, but here, p_i is localized to each number x_i. Although in general, we may want p_i to change dynamically, this is not important for us. The basic principle of exact computation is preserved in the sense that we ensure that p_i is sufficient precision to make the necessary decisions exactly. For instance, in many problems we only need the sign of a determinant, not its value. Hence some low precision may suffice for this determination. This suggests that we introduce a number representation with arbitrary but specified precision. Moreover, we want this precision to be independent of the magnitude of the number. The latter idea is of course embodied in floating numbers: a floating number is a pair (e, f) where e, f are integers representing $f \cdot 2^e$. Since e, f will be represented by BigNums, we call such representations *BigFloats*. We are currently developing a BigFloat package [5] which incorporates the idea of having a prescribed absolute and relative precision. For integers a, r, we say that a real number x *is approximated by* another real number $\widehat{x}$ with *composite error* (a, r) if *either* the absolute error $|x - \widehat{x}|$ is at most 2^{-a} *or* the relative error $|(x - \widehat{x})/x|$ is at most 2^{-r}. We write

$$x \sim \widehat{x} \ \mathrm{err}(a, r)$$

The important thing to note about this concept of composite error is that, for any given the *approximate number* $\widehat{x}$ $\mathrm{err}(a, r)$, we can essentially decide whether the approximation is in the absolute error regime or the relative error regime.

Expression package. We want to build a *expression evaluator* on top of the number packages (BigNums, BigFloats, etc). In our work on data degeneracies [23], we have already postulated the use of such an evaluator which has some additional properties to allow symbolic perturbation. A form of this idea was implemented by Fortune-VanWyk [10].

The most basic class of expressions is the class of multivariate polynomials. This class includes determinants which arise frequently in computational geometry. It is important to understand why BigEval (the name of the evaluator function) can be a major advancement over the traditional uses of number packages: traditionally, the algorithm calls the package for each arithmetic operation. The package has no idea how these calls are interrelated, thereby it

must forgo any possible optimizations across calls. In contrast, BigEval has opportunity for

- preprocessing of the expression (E.g., global analysis of the expression and restructuring of the expression)
- run-time optimization (E.g., look ahead in an evaluation and lazy evaluation methods).

The form of these expressions can be optimized as follows. In the vanilla version, we have the usual arithmetic operators with constant or variable operands. But there is opportunity to improve the evaluation process if we allow generalizations of these operators: product operator $\prod_{i=1}^{n}$, summation operator $\sum_{i=1}^{n}$, multiplication by a constant, addition by a constant, and raising to a constant power. For instance, in the summation operator, it may make sense to classify the arguments according to their signs (if they can be determined).

Geometric Objects. There ought to be packages whose fundamental objects are larger units than numbers. The simplest of these are *points* and *hyperplanes* (both can be viewed as types of vectors). The natural operations involving these objects can be specially implemented rather than left to the number packages (or even BigEval). Again, the fact that the nature of these objects are known to the packages means more opportunity for optimization. We mention that the use of homogeneous coordinates of vectors has certain benefits. Indeed, rational number (BigRat) can be viewed as a special case of homogenous vectors.

Heterogeneous representations. The traditional BigNums assume a homogenous internal representation, usually as the positional notation. It is sometimes useful to allow other internal representations: for instance, number expression such as $2^{1000} - 1$ may be superior to explicit binary notation using 999 bits. Of course, this complicates the internals and conversion routines must be provided. This idea applies equally to other domains such as BigFloats as well. This is not so much another package as the idea that there may be many flavors of packages, and these should be tied together in a seamless way. Object oriented languages can be effectively used here.

6 Summary

1. Exact computation is not much used despite the promise of many benefits. In fact, reported attempts to try exact computation often result in its rejection as too inefficient. We have argued that this inefficiency does not appear inherent, although theoretical confirmations of this remains to be done with appropriate models. Perhaps an even more compelling reason for exact computation is that the alternative of fixed-precision computation is even less hopeful.

2. We began by clarifying the concepts of exact computation, including its scope for algebraic problems. We suggest that the most promising problems for exact computation are in the pervasive class of rational bounded-degree (RBD) problems.

3. We gave an outline of how the development of exact computation can proceed. This already reveals that exact computation embraces many important computational techniques (tactics) which are clearly unexplored.

4. The first step involves a re-thinking about the traditional BigNumber package. Indeed, there are several different forms of BigNum packages that should be implemented. Beyond this, several additional layers must be added to address a variety of computational domains.

5. Although we believe there ought to be hardware support for exact computation, this is unlikely in the near future. Impetus for its development may have to come from successful software exploitation first.

Acknowledgements

I would like to thank my colleague and collaborator Tom Dubé for many discussions on these issues. His insights into implementations and number packages has been invaluable.

References

[1] Patrice Bertin, Didier Roncin, and Jean Vuillemin. Introduction to programmable active memories. Research Report 3, Digital Paris Research Laboratory, June, 1989.

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

[3] J. H. Davenport, Y. Siret, and E. Tournier. *Computer Algebra: systems and algorithms for algebraic computation.* Academic Press, 1988.

[4] David Dobkin and Deborah Silver. Recipes for Geometry & Numerical Analysis – Part I: An empirical study. *ACM Symp. on Computational Geometry*, 4:93–105, 1988.

[5] Thomas Dubé and Chee Yap. A basis for implementing exact computational geometry, August, 1993. (to appear).

[6] Herbert Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer-Verlag, 1987.

[7] Herbert Edelsbrunner and Ernst Peter Mücke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Symp. on Computational Geometry*, 4:118–133, 1988.

[8] A. Feldstein and P.R. Turner. Overflow, underflow, and severe loss of significance in floating-point addition and subtraction. *IMA J. Numer. Analysis*, 6:241–251, 1986.

[9] Steven Fortune. Numerical stability of algorithms for 2-d Delaunay triangulations and Voronoi diagrams. In *Proc. 8th Annu. ACM Sympos. Comput. Geom.*, pages 83–92, 1992.

[10] Steven Fortune and Christopher van Wyk. Efficient exact arithmetic for computational geometry. *ACM Symp. on Computational Geometry*, 9:163–172, 1993.

[11] Christoph M. Hoffmann. *Geometric & Solid Modeling: An introduction.* Morgan Kaufmann Publishers, Inc, San Mateo, California 94403, 1989.

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

[13] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry.* Springer-Verlag, 1985.

[14] Jacob T. Schwartz and Micha Sharir. On the piano movers' problem: II. General techniques for computing topological properties of real algebraic manifolds. *Advances in Appl. Math.*, 4:298–351, 1983.

[15] B. Serpette, J. Vuillemin, and J.C. Hervé. BigNum: a portable and efficient package for arbitrary-precision arithmetic. Research Report 2, Digital Paris Research Laboratory, May, 1989.

[16] Mark Shand, Patrice Bertin, and Jean Vuillemin. Hardware speedups in long integer multiplication. *2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, 1990. Crete.

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

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

[19] K. Sugihara and M. Iri. A numerically stable method for Voronoi diagram construction. *Proceedings of the 1988 Fall Conference of the Operations Research Society of Japan, Tokyo*, pages 20–21, September 28–29, 1988.

[20] Kokichi Sugihara. Robust gift-wrapping for the three-dimensional convex hull. *Journal of Computer and System Sciences*, 1993. To appear.

[21] von S.M. Rump. How reliable are results of computers? *Jahrbuch Überblicke Mathematik*, pages 163–168, 1983. Trans. from German *Wie zuverlässig sind die Ergebnisse unserer Rechenanlagen?*

[22] Chee K. Yap. A geometric consistency theorem for a symbolic perturbation scheme. *Journal of Computer and System Sciences*, 40(1):2–18, 1990.

[23] Chee K. Yap. Symbolic treatment of geometric degeneracies. *Journal of Symbolic Computation*, 10:349–370, 1990. Proceedings, *International IFIPS Conference on System Modelling and Optimization,* Tokyo, 1987, Lecture Notes in Control and Information Science, Vol.113, pp.348-358.

[24] Chee-Keng Yap. *Fundamental Problems in Algorithmic Algebra.* Princeton University Press, (to appear).

[25] Jiaxun Yu. Exact arithmetic solid modeling. Technical Report CSD-TR-92-037, Computer Science Department, Purdue University, June, 1992. PhD dissertation.