White Paper on Exact Computation and Reliable Geometric Software

CHEE K. YAP
Courant Institute of Mathematical Sciences
New York University
251, Mercer Street
New York, NY 10012

May 17, 1996

ABSTRACT

Fixed-precision computation, typically in the form of floating-point arithmetic, is inherently incapable of delivering reliability in geometric computation. The alternative is an emerging computing paradigm called *exact computation*. We describe the main features of this computational mode, its applications and challenges.

1 Introduction

High integrity in software involves many factors, depending on the application domain. Perhaps the main source of non-integrity in scientific computing is *numerical non-robustness*. This is a well-known problem and widely documented. These problems stem the *fixed-precision computation paradigm* for algorithm design: *assuming that all numerical values are to be computed to a fixed-precision, devise effective methods to reduce the loss of accuracy in a computation.* This entails techniques such as error analysis and restructuring of computation, all of which are highly developed in numerical analysis. The fixed-precision is usually the native machine precision. In most modern workstations, this is 32 bits (with hardware support for doubling this precision). Here the de facto standard for number representation is floating-point. The computing community is so committed to the *fixed-precision floating-point representation* that it is now an axiom that high performance computers must support this representation with specialized hardware. It leads to the paradoxical situation where integer multiplication is slower than floating-point multiplication in many RISC machines, even though floating-point arithmetic is much harder than fixed-point arithmetic in hardware. Indeed, von Neumann in his original computer design rejected floating-point arithmetic as being too complicated, in favor of fixed-point arithmetic. The advantage of floating-point notation is that, given a fixed amount of precision, it represents a greater dynamic range of values than any fixed-point notation. The backwards error analysis of Wilkinson greatly helped to remove the mysteries of floating-point errors, leading to the present ascendent position of floating-point computation. For further references on floating point hardware, see [11, 14].

A major development in floating-point computation occurred in the 1980s, when high performance computer architectures proliferated. There are many (non-obvious) possibilities when implementing floating-point arithmetic, with the result that practically no two architectures have compatible floating-point conventions. The result was a confusing non-portability of numerical code. This led to the now widely accepted IEEE standard [11], which greatly ameliorated the confusion. Kahan won the 1989 Turing award for his key role in this standard. Nevertheless, the achievements of this IEEE standard should be put in perspective: it simply ensures that numerical code performs *consistently* across machine architectures. It does little to remove the inevitable failures of floating-point code. Other proposals such as level-index arithmetic [6, 7] are still fixed-precision and only serves to delay the onset of such errors.

Geometric Computation. In this white paper we want to distinguish between *(purely) numerical computing* and *geometric computing*. The latter involves some *geometric structure* D which is some combinatorial object (say a graph G) that is associated with numerical data (say, numerical labels on the nodes and edges of G), together with some *consistency predicate* constraining the possible variations on the numerical data. See [17] for one formalization. Geometric computation normally constructs such a D or searches within D. Examples of geometric computing include (i) computing an arrangement of hyperplanes, (ii) intersecting two polyhedra, and (iii) point location in a planar map. Thus, the combinatorial object in (i) is the incidence graph and the numerical data amounts to labeling the vertices with faces of the arrangement, etc. In contrast, (iv) computing determinants and (v) shortest paths in edge-weighted graphs are not examples of geometric computation since the associated combinatorial structures (matrices and graphs, respectively) in these problems are independent of the numerical data. Areas such as computer graphics, CAD/CAM or robotics clearly involve significant amounts of geometric computations.

In practice, software tries to minimize numerical errors by allowing users to control parameters such as setting "epsilon parameters" and iteration limits. For purely numerical computation, this ad hoc approach may be serviceable (even this becomes an issue when the errors become too large). In geometric computing, there is the possibility of combinatorial errors which is of a qualitative nature. The latter can easily lead to unexpected states or inconsistencies (e.g., the boundary of a convex-hull may self-intersect) that are catastrophic. Michelucci[1] pointed out that even infinite looping may arise. See also Hoffmann [12].

The computational geometry community started investigating nonrobustness issues in geometric algorithms about a decade ago. The initial approaches taken by these researchers stayed within the fixed-precision computing paradigm: that is, *assuming that we have a fixed-precision for our arithmetic, how can we device geometric algorithms that are robust?* See surveys of these approaches in [10, 17]). Briefly, the approaches can be classified into basic groups:

Interval geometry: Introduce the geometric analogue of *interval arithmetic*. That is, replace classical geometric objects by the "thick objects" (points by suitable "balls", lines by "linear zones", etc). Inevitably, the unusual geometry of these thick objects leads to undesirable properties (or the thickness may become catastrophically big).

Topologically consistent distortion: Allow the geometric object to be distorted while preserving certain basic properties (e.g., combinatorial consistency). Thus a line might be allowed to become polygonal lines or pseudolines. A metric form of this approach leads to *stable algorithms* in which the output is guaranteed to be close to some bounded perturbation of the actual input. This is a form of backwards analysis.

Geometric Rounding: Assuming that only certain geometric objects are representable because of the finite precision numbers used, we therefore try to "round" an arbitrary geometric object into one of these representable ones. This may leads to some kind of simultaneous approximation of numbers, which is a difficult problem.

These approaches are difficult to apply, for even conceptually simple problems. Some are difficult to generalize. Others may achieve technically correct notions of robustness, but the result leaves much else to be desired. Perhaps the biggest practical barrier to such approaches is that the "robustified algorithms" remains a handful; it is daunting to think of all the geometric algorithms that remains to be treated. We suggest in [16] that the approach based on *exact geometric computation* is generally preferable (even though fixed-precision approaches may be useful for specific applications). To give just one reason for this remark, we observe that in the cases where researchers succeeded in getting robust fixed precision solutions, the exact solution is conceptually simpler (superior) and no harder to apply. Specifically, all the treated problems are of *bounded depth* and the exact approach to such problem do not run into high precision issues, the usual bane of exact computation. For a survey of recent papers using exact geometric computation, see [18, 17].

[1] Private communication.

2 Exact Computation

Briefly, *exact geometric computation* is the approach to algorithm design and implementation that demands all branching tests to be performed without error. A corollary is that numerical values must be computed to sufficient precision to ensure this.

Note that a branching test amounts to determining the sign of a numerical value and doing a three-way branch, depending on whether the value is positive, zero or negative. The combinatorial structures constructed during geometric computations depend only on these branchings. Hence errorless branching implies that the combinatorial structures are correctly computed. Thus non-robustness, the bane of fixed-precision computation, is a non-issue in exact geometric computation.

The term "exact" is chosen advisedly: we do not call this "arbitrary" or "infinite" precision computation even though exactness does entail arbitrary and (in some sense) infinite precision. After all, one could use arbitrary or infinite precision numbers without being exact. We also emphasize that it is "geometric". Otherwise, "exact computation" is no different from computer algebra in which every numerical quantity is expected to be computed exactly. This leads to high precision problems, which might be called the "bane of exact computation". But in exact *geometric* computation, we can exploit the use approximations when appropriate. Nevertheless, for short, we often this the "exact approach".

Let us answer three common questions about this approach.

(I) *How can we represent all real values exactly?* This is a natural question since the basic premise of exact geometric computation flatly contradicts assertions commonly found in the literature. For instance, a discussion of non-robustness issues might begin with the remark "numerical errors are inevitable in real computers". The answer to (I) is that, assuming that the input numbers are *exact*, then we can often represent and manipulate all computed values exactly. The simplest situation where this holds is when all input and computed values are rational. Clearly we must give up fixed-precision. A more sophisticated example is the exact representation of algebraic numbers such as $\sqrt{2}$. There are several known ways to do this, and we can perform the usual arithmetic operations on such representations including testing the sign of a real algebraic number. Informally, let us say that a problem is *algebraic* if it does not require anything beyond algebraic numbers. All the problems treated in contemporary computational geometry (e.g., [15, 8]) are algebraic. Many problems that appear to require transcendental functions are actually algebraic. For algebraic problems, it is known that the problems can in principle be solved within single-exponential space (this is related to Tarski's theory of real closed fields). There is a corresponding single-exponential time lower bound. So in principle all these problems can be computed exactly. This framework is described in [16].

(II) *What is the point of computing exactly when the input data may be inherently inexact*? In other words, it seems silly to perform calculations with more precision than the input data. To answer this, it is important to make a distinction between "robustness" and "accuracy": even if we have no interest in high accuracy, we may desire robustness. So if high accuracy can lead to robustness, we may be willing to pay the cost of the high accuracy. So one answer to (II) is that *exact computation is the only general and systematic way that we know for achieving robust algorithms.*

The second answer to (II) is that if the input has some known error-bounds, we can often modify the problem to solve a related problem in which the output object has a corresponding error-bound, albeit exactly! E.g., [13, 3].

(III) *How practical is exact computation?* There are two aspects. First, we are implicitly comparing exact approaches to floating-point computation. Of course, exact computation is slower. We are trading off speed for accuracy and robustness. The user must decide which is more important for the application. If robustness is important, the user may have no choice. In practice, exact approaches are currently under-developed and under-supported as compared to fixed-precision computation. One research goal is therefore to level the playing field by developing tools and support for exact computation.

Second, there are problems for which exact computation computation is infeasible, in the worst case. We think this is inherent – in other words, there are no better "robust" solutions in the fixed-precision setting. This poses the challenge of developing new (exact) approaches to such problems: the central theme here is to understand and control the precision of a computation. Although we cannot avoid high precision in some cases, we hope to only pay this cost when it is necessary (so the cost is "fair"). This could lead to practical solutions if most input instances do not need high precision. A formalization of this idea is the paper [4] that introduced the concept of *precision-sensitive algorithms*. They applied this to the Euclidean shortest path problem (which is known to be NP-hard). But whether we modify the criteria of judging performance of algorithms (as in introducing precision-sensitivity) or by modifying the problem (as in Papadimitriou [13]), we remain committed to the principles of exact geometric computation.

3 Applications

Exact computation is a potentially useful in any application that places a high premium on reliability. Here we use the term *reliability* to encompass two aspects of a computation: accuracy and robustness. Such applications include any mission-critical code (used in space shots, say), solid modelers, solver engines in constraint logic programming languages and geometric theorem proving. But there are problems where it does not make sense to do exact computation. This includes any computation that is performed, not so much for the precise results as for the insight that it affords (recalling a famous dictum of Hamming). For instance, in computer visualization and computer graphics, the object of computation is often a "pretty picture". It is not important that such pictures (usually) reveal local inconsistencies when examined in detail. Similarly, it is unimportant that when rendering a computer graphics image, any particular ray is cast with numerical errors.

Next we mention four applications.

Dimensional tolerancing and metrology (DT&M). Here, physical objects are specified to within certain tolerances, and the metrological question is whether a manufactured object conforms to its specification. A typical problem here is to compute the minimum width of two parallel planes that contain an input set of points. Errors in the software for such problem has legal (contractual) consequences. Hence there is vital interest in the industry to validate the performance of commercial software in this area, especially in the context of the Coordinating Measuring Machine (CMM) technology. The proliferation of CMM's in the 1980s and the unreliability of CMM software led to a minor crisis (Walker 1988). In response, the European community and the USA have started projects to construct "reference software" for the purpose of validating CMM software. In particular, NIST recently announced an *algorithms testing and evaluation program* (ATEP) to administer this service for the CMM community. We believe that exact computation is only technology that can deliver on the basic goals of a reference software, and have begun work in this area.

Physical Simulation. This is a difficult but growing area of interest (see [1]). A typical system is the interaction of solid bodies in a dynamically evolving situation. E.g., bodies tumbling down a staircase. Numerical inaccuracies in such simulations quickly accumulate, leading to qualitative errors in the final result. No present simulation system can give a qualitatively guaranteed performance. A major problem is the current numerical software packages which are used in such simulations [2].

Mixed-mode of computation. We feel that with the advent of faster and parallel computers, many applications can begin to migrate to exact computation. Nevertheless, fixed-precision computation will always be around. An important line of research is to investigate *mixed-mode computations* where we inter-mix fixed-precision with exact computation. The simplest example is where we can run a program in either one or the other mode. The exact mode is presumably used for testing purposes, since it may be too slow in production runs. But there are more sophisticated interaction of the two modes. One idea that has been used effectively in geometric algorithms is to use fixed-precision computation as a "filter" to catch those cases for which there is no need for exact computation.

Robot surgery. This is an example of mission-critical situations. There is an increasing use of robots in surgery (e.g., [5, 19, 9]). Grenoble University reported over 200 robot interventions in neurosurgery. The computer reconstructs a 3-D image of the brain which is displayed to the surgeon, who then specifies the trajectory using visual feedback. This trajectory is linearized and transferred to the robot kinematics. It is clear that we would like geometric software that have some guarantees in the computed trajectories. Again, the important issue here is not so much accuracy as robustness: mechanical systems such as robots are inherently inaccurate, but their inaccuracies are localized and orders of magnitude smaller than the potential errors in floating-point computation. Exact computation techniques can match the computed errors to the mechanical errors.

4 What do we do next?

We believe that the importance of exact geometric computation will grow in scientific computation as it becomes better known, as misconceptions are corrected, and as exact techniques are more developed. The bane of exact computation is high precision. Understanding and controlling this precision is necessary for exact computation to be viable: there is much room for theoretical analysis. At the practical level, the software infrastructure for exact approaches need to be developed. There are several current efforts along this line. Experience with exact approaches is growing. The dream is that someday, by paying only a modest or fair cost, we can compute "exactly" as routinely as we now compute nonrobustly.

References

[1] David Baraff. Issues in computing contact forces for non-penetrating rigid bodies. *Algorithmica*, 10:292–352, 1993.

[2] David Baraff. Fast contact force computation for nonpenetrating rigid bodies. *Computer Graphics (Proc. SIGGRAPH)*, 28:23–34, 1994.

[3] J. Choi, J. Sellen, and C. Yap. Approximate Euclidean shortest path in 3-space. *10th ACM Symposium on Computational Geometry*, pages 41–48, 1994. Also: NYU Computer Science Technical Report No. 670, September 1994.

[4] J. Choi, J. Sellen, and C. Yap. Precision-sensitive Euclidean shortest path in 3-Space. *11th ACM Symposium on Computational Geometry*, June, 1995.

[5] P. Cinquin, S. Lavallee, and J. Troccaz. IGOR: image guided operating robot. Methodology, applications. *IEEE Eng. in Medicine and Biology, 14th Annual Conference*, 3:?, 1992.

[6] C.W. Clenshaw and F.W.J. Olver. Beyond floating point. *Journal of the ACM*, 31:319–328, 1984.

[7] C.W. Clenshaw and P.R. Turner. The symmetric level-index system. *IMA J. Num. Anal.*, 8:517–526, 1988.

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

[9] Young et al. Application of robotics to stereotactic neurosurgery. *Neurological Research*, 9(2):123–128, 1987.

[10] Steven J. Fortune. *Progress in Computational Geometry*, chapter 3, pages 81–127. Information Geometers, 1993. Editor: R. Martin.

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

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

[13] C. H. Papadimitriou. An algorithm for shortest-path motion in three dimensions. *Inform. Process. Lett.*, 20:259–263, 1985.

[14] David A. Patterson and John L. Hennessy. *Computer Architecture: a quantitative approach*. Morgan Kaufmann Publishers, Inc, San Mateo, California, 1990. (with an appendix on Computer Arithmetic by David Goldberg).

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

[16] Chee K. Yap. Towards exact geometric computation. In *Fifth Canadian Conference on Computational Geometry*, pages 405–419, Waterloo, Canada, August 5–9 1993. Invited Lecture. To appear CGTA.

[17] Chee K. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *CRC Handbook in Computational Geometry*. CRC Press, 1996. (to appear).

[18] Chee K. Yap and Thomas 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.

[19] Y.S.Kwoh, J. Hou, E.A. Jonckheere, and S. Hayaty. A robot with improved absolutositioning accuracy for CT guided stereotactic brain surgery. *IEEE Trans. on Biomedical Eng.*, 35(2):153–160, 1988.