ELSEVIER

Computational Geometry 7 (1997) 3–23

Computational Geometry
Theory and Applications

Towards exact geometric computation ☆

Chee-Keng Yap [1]

Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012, USA

Communicated by Anna Lubiw and Jorge Urrutia; submitted 27 November 1993; accepted 10 October 1994

Abstract

Exact computation is assumed in most algorithms in computational geometry. In practice, implementors perform computation in some fixed-precision model, usually the machine 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 should 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.

This paper formulates the theoretical framework for exact computation based on algebraic numbers. We then examine the practical support needed to make the exact approach a viable alternative. It turns out that the exact computation paradigm encompasses a rich set of computational tactics. Our fundamental premise is that the traditional "BigNumber" package that forms the work-horse for exact computation must be reinvented to take advantage of many features found in geometric algorithms. Beyond this, we postulate several other packages to 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 nonrobust if it can precipitate unpredictable failures during execution. It is clear that such failures occur with a sufficiently high probability to cause widespread concern. This phenomenon is reflected in diverse communities, and various approaches and special solutions have been proposed. The unexamined premise in many of these solutions is the commitment to *fixed-precision computation*. Our general theme is that the alternative approach based on *exact computation* has a much larger role to play than currently practiced or suspected. In any case, the goal of reliable

☆ Work on this paper is supported by NSF grant #CCR-90-02819. A preliminary version appeared as an invited paper in the proceedings of the 5th Canadian Conference on Computational Geometry, University of Waterloo, August 5–9, 1993 (pp. 405–419).

[1] E-mail: yap@cs.nyu.edu.

SSDI 0925-7721(95)00040-2

computation is better served when both approaches are well-represented. Of course, we are partisan in this quest, and this paper only hopes to contribute to the development of exact computation.

Exact computation is *the* computing standard in the field of computer algebra (a.k.a. symbolic computation). Most problems of computer algebra have little use for fixed-precision arithmetic—a floating-point calculation usually cannot shed light on whether an integer is prime. In some sense, we are just advocating a wider role for this computing standard. What makes the new role for exact computation interesting is that (as we hope to demonstrate) it raises *uniquely* geometric issues. Although we address ourselves to geometric algorithms, it will be clear that many of our ideas apply to related fields. For instance, it is somewhat surprising that entire areas of "scientific computing" that are concerned about robust algorithms simply overlook the use exact computation techniques. But that goes beyond our present scope. This paper outlines some thoughts on a research agenda that forms the basis of ongoing research with Tom Dubé [11]:

- In Section 2, we describe the two approaches to geometric computing: fixed-precision versus exact computing, emphasizing special features of "geometric" computing.
- Section 3 expands on our concept of exact computation, basically identifying current problems in computational geometry with the class of algebraic problems.
- Section 4 identifies a subclass of the algebraic problems called "rational bounded-depth problems" (RBD) for which exact computing seems to be promising.
- In Section 5, we discuss the "BigNumber package", the traditional piece of software for achieving exact computation. Using two published work on exact computation as anecdotal evidence, we suggest that practical goals for revising this piece of software.
- Section 6 argues the need to go beyond a number package. It postulates some software infrastructure ("beyond BigNumber") for exact computation: BigFloats, expressions, geometric objects.
- We conclude in Section 7.

2. Two approaches to numerical computing

2.1. Fixed-precision computation

The root cause of nonrobustness[2] seems clear: whereas algorithms are described in exact terms, their implementations replaces exact arithmetic with fixed-precision arithmetic. Floating-point arithmetic is the usual example of fixed-precision arithmetic. The nontrivial probability of catastrophic loss of significance in such computations in practice is confirmed in theoretical models (e.g., [14]). More powerful fixed-precision models (e.g., level-index arithmetic [7]) may be useful but only in *delaying* the onset of nonrobustness problems. Similarly, the growing acceptance of the IEEE standard 754-1985 in computer architecture[3] should not obscure the fact that its main purpose is to make floating point errors predictable and *architecture-independent*. But it does not make the nonrobustness of floating-point computation disappear. So the logical step is to work towards standards for exception handling (cf. [21]).

[2] The term "robust" in this paper will be an informal catch-all term for all the difficulties of fixed-precision computation. Technical definitions of robustness come in several flavors and are model-dependent. But a precise definition is irrelevant for us as we are ultimately interested in exact computation where all these concepts disappear. For this reason, the term "robust algorithm" is only applied to algorithms that are based on fixed-precision arithmetic.

[3] See [26, Appendix A] for a description of the IEEE standard.

In practice, nonrobustness in algorithms is frequently corrected using some ad-hoc method that, at best, decreases the 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 (this is really a trite observation nowadays). For relatively simple numerical problems, the difficulties can be analyzed and kept under control; embedded in larger problems, it becomes a matter of educated guesses. But our main interest is in geometric computing which has an additional dimension: its essence may be captured in the aphorism,

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

Numerical computing is exemplified by the problem of solving linear systems of equations; graph searching is a typical problem of combinatorial computing. The convex hull problem for a set of points is a canonical example of geometric computing: the points are specified by numbers but the convex hull is essentially a combinatorial structure (a labeled graph). Linear programming counts as geometric computing but solving linear systems does not qualify. We justify this exclusion on grounds that the combinatorial structure in geometric computing ought to be implicitly determined by the numerical data. We similarly exclude the usual problems of shortest paths or minimum spanning tree on weighted graphs. But there are "geometric versions" of shortest paths and minimum spanning trees (see Section 3.1).

With these clarifications then, the extra of difficulty of geometric problems arises because it is not just numerical outputs we seek, but the associated combinatorial structure must somehow be consistent with the computed numbers. Robustness issues arising this interplay between numerical and combinatorial elements of geometric algorithms is treated in the survey of Fortune [17] (see also [19]). 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 unknown for many basic and even conceptually simple problems. For instance, Yu [41, p. 110] concluded that robust algorithms for performing Boolean operations on solids lie in the distant future. Yet such algorithms are fundamental in the field of solid modeling. Even in the plane, the complexity of a solution proposed by Milenkovic [23] suggests that there is more work to be done.
- When robust algorithms are achievable, they seem to require inordinate effort relative to the known exact algorithms. Moreover, the techniques do not easily generalize. As a consequence, only a handful of certifiably robust algorithms are known.
- 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 an "approximate line" has variously been modeled by (i) using a suitable set of pixels [computer graphics], (ii) fattening the line into a tubular region [Milenkovic], (iii) a suitably "monotone" polygonal path [Greene–Yao], or (iv) an actual line whose equation has bounded coefficient sizes [Sugihara]. Beside losing many desirable properties of lines, these models give rise to complicated algorithms. We refer to Fortune's survey [17] for a more detail description.
- A more basic approach is to 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". Interval arithmetic is a well-known version of this approach. 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 geometric computation (although such intervals seem useful in some purely numerical computation).

Some examples of recent work are: epsilon geometry (Salesin, Stolfi and Guibas), backward error analysis (Fortune), approximate combinatorial consistency (Milenkovic; Hopcroft, Hoffmann and Karasick), randomization and sampling (Dobkin, Silver). The optimist might say that we need more time to resolve these difficulties. But perhaps the difficulty is intrinsic: there will be no satisfactory solution until we confront the "specter" of exact computation and understand what is inherently involved there.

2.2. *Exact computation*

We switch to a discussion of exact computation. For now, we simply say that "exact computation" means that numerical values are computed exactly in a suitable sense and only error-free decisions are made.

- The foremost advantage of exact computation is that "robustness" is a non-issue! (A "robust exact algorithm" is an oxymoron.)
- All classical geometric concepts are preserved.
- In contrast to the obscure[4] theories of approximate geometry, classical geometries (Euclidean or otherwise) have a wealth of theorems and many important cases (planar geometry!) are relatively well-understood. So we can reason with classical objects with relative confidence.
- Practically all geometric algorithms in the literature pertain to classical geometries. This means we avoid the daunting prospect of trying of "robustify" all known geometric algorithms.
- Sometimes we can use symbolic perturbation methods to automate the handling of degeneracies, thus simplifying our coding of algorithms (cf. [13,40]). These methods are meaningful only with exact computation.
- There are applications that simply require exact computation. Examples include geometric theorem proving, checking geometric conjectures or checking topological properties of specific geometric configurations. Often such applications are one-shot deals and one is willing to devote considerable time to their (off-line) computation.

Thus, exact computation is a "generic" solution (cf. [39]) to the robustness issue. Given these advantages, why is exact computation almost never used in practice? We suggest that misconception and culture each plays a role. Many authors simply assume that, except for very special domains such as number theory and algebra, all continuous domain computations are necessarily approximate. This surprisingly common misconception is easy to dispose of. The claim that exact computation is too inefficient seems harder to counter. The floating point culture enjoys so much[5] infrastructural support (hardware or otherwise) that such claims are partly self-perpetuating. Goldberg [26, p. A-12] concludes that "given the predominance of the floating-point representation, it appears unlikely that any other representation will come into widespread use". 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 nonrobust. 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 starting point

[4] As in "unfamiliar", not in the sense of being imprecise.

[5] For instance, on CRAY systems and some RISC machines, floating-point multiply is faster than integer multiply! The latter is emulated in software.

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, [41] concluded that "exact computation is not feasible for the problem of point classification". But surely if robustness is important enough (say, it relates to the success of a mission into space), then exact computation may well be the only choice. The literature contains many such claims about the infeasibility of exact computation which need not have universal validity. We need some theoretical framework to mediate the true differences between exact and finite-precision computation. This is motivation for our 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. we may recall the conceptual framework that complexity theory provides for the entire field of algorithms. We shall not have more to say for this goal in this paper.

Varieties of precision. We should acknowledge that any simple characterization of exact versus fixed-precision approaches will run into gray areas. For instance, we may distinguish between degrees of fixed-precision: the most restrictive form of fixed-precision prevails in 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. As we will show, many exact algorithms can be carried out using this local version of fixed precision. Exact computations may use internal approximations. Other variations are possible: for example the language *Numerical Turing* [20] developed at the University of Toronto for numerical computation has the concept of a *precision block*, which is syntactically like a do-loop. The computation of such a block is iterated with increasing precision until some 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 for approximating exact arithmetic. An 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 of the unit square of the integer lattice that contains the vertex. Milenkovic and Nackman [24] has shown that such problems can be NP-hard. Another class of examples is where the input is an approximation to some ideal data. For instance, the input may be visual data collected by a camera. We may want to do various feature extractions on this imperfect data. Such problems are outside our scope.

3. What is exact computation?

We clarify our use of the term: by an "exact computation", we mean 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 decisions.
Of course, exactitude and error are relative to the underlying mathematical model. In computational geometry, the mathematical model is usually (but not necessarily) Euclidean geometry.

We understand in (i) that mathematical objects are characterized by suitable numerical parameters. To say that the parameters "exactly" represent an object means that we can decide whether or not two such objects are equal from these parameters. The representation (i.e., parameters) need not be unique. In applications where the input values are approximations to unknown values, we must nevertheless treat these approximations as exact. If this is not possible, we face a bona-fide problem of approximation which, as noted before, is outside the present scope.

3.1. Algebraic numbers

These concepts are illustrated by the representation and manipulation 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 X^2-5.

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", since neither entails exact computation. In some sense, the interval $[1,4]$ 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) of our characterization 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 means that we can perform the usual arithmetic operations $(+,-,\times,\div)$ and can determine the sign of real algebraic numbers. For our purposes, we focus only on real algebraic numbers since the complex ones can be represented as a pair of real algebraic numbers. It is well-known 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 $-\alpha$ and $1/\alpha$ (respectively) are roots of

$$\sum_{i=0}^{n}(-1)^i c_i X^i \quad \text{and} \quad X^n \sum_{i=0}^{n} c_i X^{-i}.$$

If α, β are roots of $P(X)$, $Q(X)$ (respectively) then $\alpha+\beta$ and $\alpha\beta$ are roots of

$$\mathrm{res}_Y\big(P(Y),Q(X-Y)\big), \qquad \mathrm{res}_Y\big(P(Y),Y^nQ(X/Y)\big),$$

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 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 $2\sqrt{5}-2\sqrt{3}-1$.

Algebraic numbers have another important closure property: the root of a polynomial with algebraic coefficients is algebraic. In addition to the above operations, we shall also include in the set of algebraic operations the *root extraction* operation, which, given the coefficients $a_0, a_1, \ldots, a_n$ of a polynomial

$$P(X) = \sum_{i=0}^{n} a_i X^i,$$

extracts a root of $P(X)$. Depending on the context and application, this can be variously interpreted to mean one of the following: *any* root, any *real* root, or *all* roots. 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 [9,38].

3.2. Algebraic problems

Our example of algebraic numbers is felicitous[6] because *almost all contemporary 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 are algebraic numbers. We informally call such problems *algebraic*. We can be slightly more precise, using *Tarski's language* which is the first order language of the real closed fields. Briefly, a predicate $P(x_1, \ldots, x_n)$ in this language may be assumed to have the form

$$Q_1 y_1 Q_2 y_2 \cdots Q_k y_k \big[M(y_1, \ldots, y_k, x_1, \ldots, x_n)\big],$$

where the Q_i's are quantifiers ($\forall$ or $\exists$) and M is a Boolean combination of polynomial inequalities of the type $P(y_1, \ldots, y_k, x_1, \ldots, x_n) = 0$ or $P(y_1, \ldots, y_k, x_1, \ldots, x_n) > 0$. Here x_i, y_j are real variables. A *semi-algebraic* is the set definable by such a Tarski predicate. An *algebraic decision problem* is given by a sequence of Tarski predicates,

$$\varphi_1, \varphi_2, \cdots, \tag{1}$$

where φ_n is a predicate on the appropriate number of variables for describing an input instance of "size n". For instance, if the input instance are points in d-space, then there are dn real variables for inputs of size n. To formulate *algebraic construction problems*, first assume that the output size is a function of the input size. Then such a problem is again given by the sequence (1) where each φ_n involves both the input and output variables. In case the output size is not a function of the input size, we replace φ_n by a set of predicates, one for each output size. To state complexity results for algebraic problems, we need to place suitable complexity restrictions on the sequence (1); for instance, that they be constructed in logarithmic space. This definition of algebraic problems is simple but serves our purposes[7].

[6] We do not know of any counterexample in the standard computational geometry literature. No doubt, as the field expands, we will see some exceptions.

[7] A less artificial way to formulate problems in natural domains of computation is to introduce "object logic" to represent the natural objects (hyperplanes, points, segments, etc.). This is basically a typed logic with functions to extract the real

Algebraic problems are essentially solved by reduction to two related general results: algebraic decision problems can be solved using a decision procedure for Tarski's language, and algebraic construction problems can be reduced to the construction of suitable algebraic cell complexes. The *cylindrical cell decomposition* of Collins [8] is the most notable method for constructing cell complexes, and it also solves the decision problem. The general upper bound for both these problems has seen great progress in recent years, and lead us to the following important metaresult: *most problems in computational geometry can be computed exactly in a single-exponential space complexity*. The superpolynomial complexity is inevitable because some of these problems are provably intractable—by a reduction from the decision theory of real addition (which Fischer and Rabin [15] have shown to be nondeterministic exponential-time hard).

Let us illustrate the above by considering three problems in $\mathbb{R}^d$:

(i) Euclidean minimum spanning tree (EMST): given a set of points in $\mathbb{R}^d$, find the minimum spanning tree connecting these points.
(ii) Euclidean traveling salesman problem (ETSP): given a set of points in $\mathbb{R}^d$, find a tour of these points of minimum length.
(iii) Shortest path amidst polyhedral obstacles (ESP): given a set E of polyhedral obstacles and points s, t, find the shortest path from s to t which avoids E.

It is not hard to see that the first two problems can be reduced to determining the signs of a finite set of algebraic numbers. For instance, for ETSP, there are finitely many tours and for any two tours π_1, π_2, it suffices to determine the sign of $L(\pi_1) - L(\pi_2)$, where $L(\pi)$ is the Euclidean length of the tour π. (A tour is a polygonal path $\pi = (s_0, \ldots, s_m)$ where $s_0 = s_m$ and $L(\pi)$ is the sum of the Euclidean distances between consecutive points s_{i-1} and s_i.) So $L(\pi)$ is a sum of square-roots and it is not hard to determine the sign of $L(\pi_1) - L(\pi_2)$ (but the obvious method takes double exponential time). With a bit of care, we can do this in single exponential space, but not much better is known. In the case of EMST, the argument[8] is even simpler: because of the matroid properties of forests, the EMST problem can be reduced to comparing distances between pairs of points. But comparing two such distances is trivial and can be done efficiently. So that EMST is exactly solvable in polynomial time (as is well-known).

But let us argue the exact solvability of ETSP in another way: note that the length function $L(s_0, \ldots, s_m)$ is an *algebraic function*, meaning that there is a polynomial $R(z, s_0, \ldots, s_m)$ with integer coefficients such that $R(L(s_0, \ldots, s_m), s_0, \ldots, s_m) = 0$ is valid. Hence, if the points s_i are algebraic (meaning that its coordinates are algebraic numbers), then the length $L(s_0, \ldots, s_m)$ is also an algebraic number. In the case of ETSP, we only need consider $L(s_0, \ldots, s_m)$, where s_i are input points. So $L(s_0, \ldots, s_m)$ are algebraic. Since we know how to compute and compare algebraic numbers we again conclude that ETSP can be solved exactly. For shortest paths in the plane, a similar remark is true. But for $d > 2$, it is not so clear that the problem can be exactly solvable. A first step would be to show that the problem is algebraic. By definition, this means the set of shortest paths is a semi-algebraic set; it need not imply that all shortest paths must be algebraic. This is because algebraic sets of positive dimension contain non-algebraic points. This is not really a problem because Collin's

parameters characterizing the objects. For instance, we can have "point variables" P and functions to extract the coordinates of P (in dot-notation, $P.x$, $P.y$, etc.).

[8] We are thankful to Edelsbrunner for this remark.

decomposition procedure can return an algebraic point (called a "sample point") in any semi-algebraic set. But in the case of ESP, every shortest path is algebraic. We show this in Appendix A.

Disguised algebraic problems. Some problems that apparently involve transcendental functions are actually algebraic problems in disguise. For instance, in the so-called motion planning problem [30] where the robot and the obstacles have piecewise algebraic boundaries, we seek the feasibility of an obstacle-avoiding motion between two positions. Since the robot may rotate, we might generally expect that the calculations would involve trigonometric functions. It turns out that we can exploit the algebraic relations among trigonometric functions and avoid making transcendental decisions. For instance, we view may $\sin x$ and $\cos x$ as two algebraic quantities connected by the relation $\sin^2 x + \cos^2 x = 1$.

3.3. Approximation problems

One approach to the intrinsically difficult algebraic problems is to modify the problem into an *approximation problem*. That is, we modify the problem to accept a prescribed amount of approximation in its solution. It is not a self-contradiction to speak of "the exact solution of approximation problems". It is best to clarify this via an example. Canny and Reif [5] have shown that shortest paths (ESP) in 3-dimensions is NP-hard. The corresponding approximation problem can be defined to take inputs as in the original problem, plus an additional $\varepsilon > 0$ parameter. The output is to be a path that is at most $(1 + \varepsilon)$ longer than the shortest path. On the other hand, Papadimitrou [25] has shown that the approximation version of the problem is polynomial time in the size of the original input and in $1/\varepsilon$. Algorithms that drive computer graphic displays can use such approximation algorithms since there is only a bounded resolution in displays. The possibility of approximation problems is an indication of the richness of what can come under the exact computation paradigm.

4. Rational bounded-depth problems

Since algebraic problems are intractable in general, we seek tractable subclasses. Many problems do not require the full power of algebraic computation, but can be solved using only [9] the four arithmetic operations but not root extraction. We call such problems *rational*, provided the inputs only involve rational numbers. Linear programming and constructing hyperplane arrangements [12] are examples of rational problems.

Another important subclass of algebraic problems comes from the concept of *depth of derivation* (cf. [40]): relative to a set U of numbers, a number x is of depth 0 if $x \in U$; x is of depth at most $d + 1$ if x is obtained by one of the rational operations applied to numbers of depth at most d, or by root extraction from a degree k polynomial where each coefficient of the polynomial has depth at most $d - k + 1$. An algorithm has *depth* at most d if there is a finite set K of numbers (i.e., the constants in the algorithm) such that on any input whose numerical parameters form the set X, all intermediate values computed by the algorithm are of depth at most d relative to $U = K \cup X$.

[9] Other simple but non-algebraic functions such as computing the sign of a number or taking the floor of a number may be needed. In the context of the rational functions, these are reasonable operations.

A problem is *bounded-depth* if it can be solved by an exact algorithm of at most some fixed depth. Suppose α is a number that arises in a bounded-depth computation. Then α satisfies a polynomial $P(X)$ whose degree m is bounded. If the input numbers have bit sizes at most s, then the coefficients of the $P(X)$ have $O(s)$ bits. It is shown in [38] that if $\widehat{\alpha}$ satisfies

$$|\widehat{\alpha} - \alpha| < \frac{1}{K^{ms}}$$

(for a suitable constant K), then applying Newton's approximation to $\widehat{\alpha}$ is guaranteed to converge to α. Using known techniques, we can obtain an initial approximation $\widehat{\alpha}$ satisfying this bound. But subsequent refinements can use Newton's method which is known to be very efficient. This can be the basis for using lazy evaluation techniques in exact computation.

A problem is *rational bounded-depth* (RBD, for short) if it can be solved by an algorithm of bounded-depth that performs only rational operations. Of course, the (actual) *depth* of an algorithm or problem is the least d such that it is of depth at most d. The class of RBD problems includes the majority of problems in contemporary computational geometry (say, as treated in the standard texts [12,28]). The following is an obvious but key property of RBD algorithms:

(P) *There is a constant D such that if the input instance to the algorithm involves rational numbers of size (at most) s, then the intermediate computation involves only rational numbers of size $Ds+O(1)$.*

Here, the size of a rational number p/q is just the maximum of the bit sizes of the integers p and q.

To state a consequence of (P), let us define the *algebraic complexity* of an algorithm to be the function $T(n)$ where $T(n)$ is the worst case number of arithmetic or root-extraction operations used by the algorithm on inputs of size n. *Now assume that the numbers in the inputs fit into the machine word size.* Then we can implement exact arithmetic by representing each integer using D words, and hence: *for any RBD algorithm with algebraic complexity $T(n)$, there is a constant $C > 0$ such that the algorithm can be implemented in time $C \cdot T(n)$ using exact computation.* Note that $T(n)$ is the usual accounting function used for measuring complexity of geometric algorithms. So the theoretical time bounds for RBD algorithms do reflect the running times of *exact* implementations of these algorithms.

Clearly, the above constant 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 asymptotics when we discuss these constants. We can take $D = 2^d$ if the RBD algorithm has depth d. This 2^d bound can often be improved.

Example. Say an algorithm is of type $A_{k,b}$ if its numerical computations consist only of repeated evaluations of $k \times k$ determinants. Moreover, the determinants are evaluated on values of depth $\leqslant b$. It is well known that convex hulls of point sets in dimensions $k-1$ can be solved by such an algorithm with $b = 0$. Such algorithms can also solve convex hulls in dimension k with $b = 1$. Again, it can solve Voronoi diagrams in $k-1$ dimensions, but with $b = O(\log k)$ which comes from the use of the lifting map. The depth of the algorithm depends on how one implements the determinant computation. For instance, using standard Gaussian elimination or the division-free version due to Bareiss (e.g., see [38]), the depth is $d = O(k^3)$ but we can take $D = k$ (not 2^d) and $C = O(k^2)$.

Remark. As Schönhage [29] pointed out, root finding for bounded degree algebraic numbers has the same asymptotic complexity as integer multiplication. So bounded-depth problems are, in principle,

not much harder than RBD problems. But the implicit constants may make such problems somewhat less attractive than RBD.

4.1. Unbounded-depth problems

It is rare to find a problem in traditional computational geometry that is rational but not bounded-depth. But imagine a solid polyhedral modeler in which we can do rational transformations of solids and perform Boolean operations on solids. This is not a "computational problem" as usually understood in algorithmics, with well-defined inputs and outputs. Each transformation and operation on these solids increases the depth of derivation. It is not surprising that fixed-precision fails notoriously. It would be interesting to show that this must be so (goal (G2)).

An artificial form of this iteration phenomenon which is useful for numerical experimentation was invented by Dobkin and Silver [10]: it is based on repeated application of two operations (going-in, going-out) on an initial pentagon, and on the fact that in the exact world, going-in and going-out are inverses. Another variant is as follows: suppose that we have just points and lines, and we are allowed to construct a new point P as intersection of two prior lines, $P \leftarrow \text{intersect}(L, L')$; similarly, we can construct a new line L through two prior points, $L \leftarrow \text{span}(P, P')$. To see the kind of precision needed, suppose we start out with a set of points in the plane, and in each iteration, we form all possible lines from the points at the start of the iteration, followed by forming all possible lines from these constructed lines. After i iterations, the coordinates of the points have size $4^i(n+1)$ assuming the initial data uses n-bit integers. Thus exponential behavior apparently occurs with iterated rational operations, but we have no proof that $\Omega(c^i)$ is necessary.

4.2. Some robust algorithms

There are some 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 [35,34,36,37]). 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-dimensional convex hull algorithms. "Robustness" in Sugihara–Iri approach means that certain (problem dependent) combinatorial properties hold; this is distinct from the "stability" concept which roughly says that the output is correct for some small perturbation of the input. It seems that stability is not easy to achieve with their method [37]. On the other hand, Fortune [16] described two robust $O(n^2)$ algorithms for planar Delaunay triangulation which are stable.

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 multiplicative constant C. Why would one exchange a $Cn\log n$ exact algorithm for a $C'n^2$ stable approximate algorithm for 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.

So the RBD class is ideal for comparing these two competing approaches: for fixed-precision, because that is where robust algorithms have been successfully constructed, and for exact computation, because our observations suggest that it will be especially effective here. Unfortunately, the next two sections suggest that the necessary software infrastructure for exact computation is not quite ready for this exercise.

5. Reinventing BigNumbers

We now address research goal (G1), which seeks to reduce the cost of using exact computation. Just as a floating-point package is the engine of most fixed-precision computation, a "BigNumber package" is the basis of 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 [2,32] 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 will focus on software solutions below.

First, we can simply try to improve on traditional BigNumber packages. One attempt is reported by Vuillemin, Hervé and Serpette [31]. They 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. This improvement alone is insufficient in view of the anecdotes next.

Some anecdotes. The first reality we face when using BigNums is not encouraging: *off-the-shelf use of such packages incurs a tremendous overhead.* For instance, in the case of exact *integer* computations, Fortune and Van Wyk [18] said that the geometric primitives in their program becomes slower by a factor of 40–140. Their programs spend only between 20% and 50% of the running time on such primitives. 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 [22] reported an initial slowdown factor of 10,000. The good news is that in both cited papers, careful fine-tuning eventually reduces these factors to a small constant factor (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 (say) 25% of its time in number crunching. For certain applications, such a small penalty tilts the balance in favor of exact computation. To be sure, we do not claim that this penalty is tolerable for all applications. 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. [10] Again, with the

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

increasing commercial availability of medium-scale parallel computers (of a dozen nodes, say), small technology-independent constants will be even less significant.[11] These remarks about technology-independent gaps should be understood as follows: in many application areas, there is a lower threshold on the acceptable computation speed. Once machine speeds exceed this threshold, other factors such as robustness and user-friendliness become increasingly important. As we believe that such thresholds are being crossed for a growing number of problems, the advantage of fixed-precision over exact computation will correspondingly diminish.

It is not hard to identify some sources of inefficiency with standard BigNum packages. One pays a large overhead for its generality; in particular, its space management facilities and possibly its pointer structures. The heritage of "BigNumber packages" seems to come from computer algebra applications. It is a reasonable assumption in computer algebra that we cannot predict the precision needed during a computation. But we have indicated that in computational geometry, the opposite assumption usually holds. One approach is to build a poor-man's BigNum package to exploit just this property, avoiding the overhead of generality. In other words, we wish the constant factor C to approach $1 \cdot D^2$. In fact, more sophisticated $o(D^2)$ techniques seems possible with hand-coding for small D. Such a poor-man's package may be useful for demonstration but the real goal is to achieve such performances in a general package that can adapt to the application. As a concrete target, we want C to approach the anecdotal number "10" in some reasonably general setting.

6. Beyond BigNumbers

To fully exploit geometry, we must go beyond numbers. In this section, we outline some general features of geometric computing that can be exploited. Each of these features could be incorporated into a software package, built on top of BigNum (or any substitute number package). We envision a rich environment where exact geometric computing is achieved with relative efficiency and convenience. Perhaps this convenience, rather than achieving the ultimate efficiency, is key to encouraging more exact computation.

Before leaving the subject of number packages, we mention that there are packages for multiprecision arithmetic that seems to be very good in pure number crunching. A well-known system is from Brent [3] (see also Bailey [1]). It is possible that some of these packages, properly retargeted for geometric computing, may be useful as a basis for building other packages.

Although we have focused on general packages below, it is also very interesting to design packages for subdomains of geometric computation. For instance, exact computation of determinants or their signs would constitute an important software package—it can be viewed as a subpackage of the "BigExpression package" below. Clarkson [6] has shown an interesting approach for building such a determinant sign evaluator.

6.1. BigFloats

It is often assumed that exact computation entails computing with integers or rationals. But we have indicated that approximate number have a role. The theoretical basis for this is the concept of

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

root-separation bounds for algebraic numbers (see [11]). We need a package for computing of each variable 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 x_i. In general, p_i may change dynamically. The basic principle of exact computation is preserved if we ensure that p_i is sufficient to make the necessary error-free decisions. For instance, in problems where we only need the sign of a determinant, not its value, some low precision computation may suffice.

So we want a number representation with arbitrarily specified precision. Moreover, it is desirable to decouple this precision from the magnitude of the number. This decoupling is, of course, embodied in the idea of floating numbers. To describe this, let us fix any integer $d > 1$ (the *base*), and any integer in the range $[0, d-1]$ is called a *digit*. Usually $d = 10$ or a power of 2. For any integer f, let $\langle f \rangle_d$ (or, simply $\langle f \rangle$) denote the number which in d-ary notation is $\text{sign}(f)0.f_1 f_2 \cdots f_k$ where $f_1 f_2 \cdots f_k$ is the standard d-ary notation for f. Thus with $d = 10$ and $f = -123$, we have $\langle -123 \rangle_{10} = -0.123$ and $\langle -123 \rangle_2 = -0.1111011$. We call $\langle f \rangle$ the *normalization* of f (to base d). A *floating number* is a pair $(e, f)_d$ (or, simply (e, f)) of integers representing the number $\langle f \rangle_d \cdot d^e$. Here f carries the precision while e indicates the magnitude. Since e, f will be represented by BigNums, we call such representations *BigFloats*. We are currently developing a BigFloat package [11].

Each BigFloat carries an error bound. Since BigFloats are approximations even for rational numbers (e.g., it cannot represent $1/(d+1)$ exactly), errors are inevitable. Of course, a straightforward implementation need not include any explicit error information, just that the last digit is uncertain. This is reasonable when $d = 2$ but for large d, this loses too much information. We keep track of an error that ranges between 1 and $d-1$. Below, we describe another notion of error bound that is under user-control.

6.2. *Expression package*

The basis of this package is the observation that individual arithmetic operations in geometric algorithms are not completely random. Rather, *they usually exhibit well-defined local structures*. For instance, a convex hull algorithm may perform all its arithmetic operations *only* in the context of computing determinants, which has a well-defined structure. We want to exploit such local structures.

These structures are captured in suitable classes of "expressions", the most important of which is the class of multivariate polynomials. We envision the following use of the package: before the actual computation begins, the algorithm constructs the expressions that it will need. Often, only a single expression, e.g., a determinant, is needed. Then it calls the package to preprocess each expression: the result is, for lack of a better word, a "compiled expression". During the actual computation, we make repeatedly calls to a functor *BigEval* to apply a compiled expression to specific arguments. In our work on data degeneracies [40], we have already postulated the existence of such an evaluator (albeit with additional properties to allow symbolic perturbation). A form of such an evaluator was implemented by Fortune–Van Wyk [18]. With BigEval, the user never need to directly call the BigNumber package.

It is important to understand why the use of BigEval can be a major advance over BigNumber package: traditionally, the algorithm calls the BigNumber package for each arithmetic operation. The number package, having no idea how these calls are interrelated, must forgo any possible optimizations across calls. In contrast, BigEval has opportunity for

- *Preprocessing of the expression:* compilation of expression, global analysis of the expression including static error bounds, restructuring of the expression.

- *Run-time tactics:* floating point filter, dynamic error bounds, incremental and/or lazy evaluation methods.

Many of these ideas are discussed in [18].

In the vanilla version of polynomial expressions, 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. In the summation operator, it may make sense to classify the arguments according to their signs (if they can be determined). It seems that the design of this package can use many ideas from classic compiler technology.

Expression Variables. In general, we expect to evaluate expressions only up to some prescribed precision, using BigFloats. Since our users (we may suppose) intend to compute exactly, we expect to maintain some error bounds on these approximate values. This gives rise to the concept of an *expression variable* (or E-*variable* for short). An E-variable is essentially an object with three associated components: *defining expression* E (as above), an *error bound* ε (below), and an *approximate value* $\widehat{v}$. The leaves of E can be explicit values or, recursively, E-variables. The (*exact*) *value* of the E-variable is defined to be the value $v(E)$ of E provided the expression, upon recursive expansion of E-variables at the leaves of E, is ultimately well-founded in explicit values. The approximate value $\widehat{v}$ is a BigFloat, guaranteed to be approximate $v(E)$ to within the error bound ε.

The error bound ε is given by a pair of integers $\varepsilon = [a, r]$. 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}. This is denoted

$$x \sim \widehat{x} \ \mathrm{err}[a, r].$$

The expression "$\widehat{x}$ err$[a, r]$" is be viewed as an approximation to some unknown x. Given any $\widehat{x}$ err$[a, r]$, we can essentially decide whether the approximation is in the absolute error regime or the relative error regime. By decreasing the error bound (by increasing a and/or r) of an E-variable we are in effect asking for a possible reevaluation of the expression. One motivation for using composite error is that as the alternatives (relative or absolute errors) are less flexible: relative errors are gracefully maintained when we multiply approximate numbers but not when we add. Similarly, absolute errors are gracefully maintained during addition but not during multiplication.

6.3. Geometric objects

It is an obvious remark that geometric computation involves *geometric objects which can be viewed as aggregates of numbers with associated object operations.* There ought to be packages to encapsulate such object classes. The simplest of these objects are *points* and *hyperplanes* (both can be viewed as types of vectors). The natural operations (intersection of hyperplanes, point-hyperplane incidence predicates, etc.) on these objects can be implemented as part of the package. Beyond convenience, the fact that the nature of these objects are known to the packages means more opportunity for optimization.

Common representation of vectors. We give one example where dealing with vectors rather than individual numbers can be exploited. This is through the use of homogeneous coordinates. Indeed,

rational numbers ("BigRat") can be viewed as homogeneous 2-vectors of integers. In general, each n-vector $(a_1, \ldots, a_n)$ of rational numbers can be represented as the $(n+1)$-vector of *integers*

$$(m_0 : m_1 : \cdots : m_n),$$

where $a_i = m_i/m_0$ and the colon separators suggest equivalence up to proportionality. So m_0 is the common denominator and we call this the *common representation* of rational n-vectors. We take the *bit size* of a vector (in common representation or not) to be the maximum of the bit sizes of its entries. Suppose H_i $(i = 1, 2, 3)$ is the plane with equation $a_i x + b_i y + c_i z = d_i$. We want to compute their common intersection point $(x, y, z)^{\mathrm{T}}$, where

$$\begin{pmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \end{pmatrix}.$$

Hence

$$x = \frac{1}{\Delta} \sum_{i,j,k} d_i b_j c_k,$$

where Δ is the determinant of the system. We consider several scenarios:

- Assume that the coefficients of H_i are rational numbers of bit size at most s. Then we see that $\sum_{i,j,k} d_i b_j c_k$ has bit size at most $18s$, and so x has bit size at most $36s$.
- If the coefficients of H_i are integers of bit size at most s, then x has bit size at most $3s + \log_2 6$.
- If the coefficients are rational numbers but we first convert H_i, viewed as the vector (a_i, b_i, c_i, d_i), into the common representation $(E_i : A_i : B_i : C_i : D_i)$ where E_i is the common denominator, then the integer A_i has bit size at most $4s$. Ignoring the component E_i, we can compute (x, y, z) whose bit size is at most $12s + \log_2 6$.

In all three cases, the output point (x, y, z) can be in the common representation $(U : X : Y : Z)$, for no additional cost.

This analysis can be carried out in any dimension as well. It suggests that we should [12] try to convert rational vectors into the common representation. To estimate the cost of this conversion, consider the rational k-vector $(a_1, \ldots, a_k)$ of bit size s. Let $a_i = n_i/d_i$, $D_0 = \prod_{i=1}^{k} d_i$ and $D_i = D_0/d_i$ $(i = 1, \ldots, k)$. So one common representation is $(D_0 : n_1 D_1 : \cdots : n_k D_k)$. We can compute D_0 using $k - 1$ multiplications. Using a balanced binary tree T to compute D_0, we can extend this to the computation of $D_1, \ldots, D_k$: let each node u of the tree store a value $V(u)$ equal to the product of the the d_i's stored in leaves below u. These values are computed in a bottom-up fashion to obtain D_0. Now, in the top-down fashion, let each nonroot u compute the product $W(u) := W(p)V(s)$ where p, s (respectively) are the parent and sibling of node u. By definition, $W(u) = 1$ if u is the root of T. Our desired values D_i are obtained as $W(u)$ where the u are leaves. The total number of multiplications to obtain the common representation from T is thus $2k - 2$. Note that we have avoided GCD computations in this conversion process, as this could be expensive. By the same token, we do not assume that rational numbers are automatically in reduced form.

[12] In computer graphics it has been suggested that homogeneous coordinates is mainly a mathematical device which, in practice, is a needless extra dimension.

6.4. Heterogeneous representations

The traditional BigNums assume a homogeneous internal representation, usually the positional notation. It is sometimes useful to allow other internal representations: for instance, the *number expression* $2^{1000} - 1$ may be superior to an explicit binary notation using 999 bits. Of course, allowing number expressions destroys the unique representation property and makes the equality testing or sign-determination highly nontrivial. We must also provide conversion routines. This idea applies equally to other domains such as BigFloats. 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.

7. Summary

1. Exact computation is not much used despite the promise of many benefits. Perhaps the most compelling reason for exact computation is that the fixed-precision alternative is even less hopeful if we are serious about robustness.

2. We described a framework for exact geometric computation. General results related to Tarski's decision problems imply that most problems in computational geometry can be solved exactly. We identified the pervasive class of rational bounded-depth (RBD) problems for which exact computation seems particularly promising. Known robust algorithms also fall under this class, so it is a good place to compare the two approaches.

3. Exact computation embraces a broad range of computational tactics and includes approximation problems. We must rethink the traditional BigNumber package to fit the needs of geometric computation. Beyond this, we need additional layers to be added on top of BigNumber. With such software infrastructure, we believe more users will migrate to exact computation.

4. Although 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.

5. It may be appropriate to end with a perspective of the floating-point culture. The debate over the viability or nonviability of exact computation is an issue of comparative advantages. There is no inherent reason against exact computation, only that the fixed-precision approach is more efficient. But efficiency is not the only issue. How do we account for the historical ascendency of the floating-point culture from 1950–1990 (see [27, p. 249ff])? After all, it started ignominously when Von Neumann rejected floating-point numbers for being too complicated. If floating-point computation became less exotic and understood, it is mainly due to the influence of Wilkinson and his influential error analysis of floating-point calculations. But the importance of floating-point arithmetic derives from an era where computing cycles is the main bottleneck of most applications. In the world of rapidly increasing computing power, with no end in sight, a growing number of applications is no longer *cycle-critical*. For such problems, other factors (robustness, adaptability, etc.) begin to weigh in more. These factors seem to favor exact computation. We imagine many applications to move from being cycle-critical to being noncycle-critical. To be sure, cycle-critical problems will always be around. It is possible that just as the floating-point culture reached its current (apparently unassailable) ascendant position, new technological advances may work to subvert it.

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 have been invaluable. The comments of Steven Fortune, Christopher Van Wyk, Herbert Edelsbrunner are greatfully acknowledged, having helped in weeding out some wishful thinking.

Appendix A

We show that the set of solutions to the Euclidean shortest path (ESP) problem is a finite semialgebraic set. In particular, this shows that every shortest path is algebraic.

Initially, let us fix the lines $L_1, \ldots, L_k$ in $\mathbb{R}^d$ and points $s, t \in \mathbb{R}^d$. By an $(s, L_1, \ldots, L_k, t)$-*path* we mean a polygonal path $\pi = (s_0, s_1, \ldots, s_{k+1})$ from $s = s_0$ to $t = s_{k+1}$ such that $s_i \in L_i$ $(i = 1, \ldots, k)$. The only restriction we have on $L_1, \ldots, L_k$ is that $L_i \neq L_{i+1}$ for $i = 1, \ldots, k-1$. In our application, L_i and L_{i+1} may not be skew and possibly $L_i = L_j$ provided $|i - j| > 1$.

We make each line L_i directed by picking an arbitrary vector u_i parallel to L_i and let[13] the *angles of incidence and reflection at* s_i to be $\alpha_i = \angle(u_i, s_i - s_{i-1})$ and $\beta_i = \angle(u_i, s_{i+1} - s_i)$ (respectively). The angle α_i is undefined if $s_i = s_{i-1}$ and similarly for β_i. The following is well-known (cf. [33, Lemma 3.1]):

(Snell's law) *If* π *is a shortest path and* α_i, β_i *are defined then* $\alpha_i = \beta_i$.

Let $I \subseteq \{1, \ldots, k\}$ and consider the predicate $\Phi_I(s_1, \ldots, s_k)$ given by

$$\bigwedge_{i=1}^{k}(s_i \in L_i) \wedge \bigwedge_{i \in I}(s_i = s_{i+1} \vee s_i = s_{i-1}) \wedge \bigwedge_{i \notin I}(\alpha_i = \beta_i).$$

[The symbol "$\bigwedge$" denotes the "anadic logical-and", used analogously to the summation $\sum_{i \in I} x_i$ notation.] It is easy to rewrite Φ_I as a Tarski predicate involving kd real variables ($s, t, L_1, \ldots, L_k$ are held constant).

Lemma 1. *If* Φ_I *is satisfiable then it has a unique solution.*

Proof. Suppose $s_1, \ldots, s_k$ is a solution. Then s_ℓ is uniquely determined whenever $\ell \in I$. It remains to consider $\ell \notin I$. It is enough to consider a maximal subsequence $s_i, s_{i+1}, \ldots, s_{j-1}, s_j$ such that $\{i+1, i+2, \ldots, j-1\} \cap I = \emptyset$. Then s_i, s_j is uniquely determined. To see that each s_ℓ $(i < \ell < j)$ is also uniquely determined, we may proceed as in [33, Lemma 3.3]. Basically their argument is valid in any dimension provided α_ℓ, β_ℓ are well-defined). □

Let $\Phi_{L_1,\ldots,L_k}(s_1, \ldots, s_k)$ be the disjunction of $\Phi_I(s_1, \ldots, s_k)$, ranging over all $I \subseteq \{1, \ldots, k\}$. It is not hard to see that every shortest path $(s, L_1, \ldots, L_k, t)$-path satisfies $\Phi_{L_1,\ldots,L_k}$. So far, we have not taken into account the obstacles.

[13] For nonzero vectors $u, v \in \mathbb{R}^d$, we define $\angle(u, v) := \cos^{-1}(\langle u, v\rangle / \|u\| \cdot \|v\|)$, where $\langle u, v\rangle$ denotes the scalar product and $\|u\|$ the Euclidean length.

Now we fix the input to ESP, namely, $s, t \in \mathbb{R}^d$ and a set E of polyhedral obstacles. We can think of E as a union of simplices where the simplices are given explicitly. If $S_1, \ldots, S_k$ are edges in E, an $(s, S_1, \ldots, S_k, t)$-path is any path $(s, s_1, \ldots, s_k, t)$ where $s_i \in S_i$. In analogy to $\Phi_{L_1,\ldots,L_k}$ above, we can construct a Tarski predicate $\Phi_{S_1,\ldots,S_k}(s_1, \ldots, s_k)$ defining a finite set of $(s, S_1, \ldots, S_k, t)$-paths. It is easy to conclude from Lemma 1 that $\Phi_{S_1,\ldots,S_k}(s_1, \ldots, s_k)$ has only finitely many solutions, provided S_i, S_{i+1} are non-collinear $(i = 1, \ldots, k-1)$. If we ignore obstacles, then all shortest $(s, S_1, \ldots, S_k, t)$-paths will be included in this finite set.

To account for the obstacles, we write a Tarski predicate $\mathrm{FREE}(s_0, s_1)$ that asserts that the path segment (s_0, s_1) avoids the relative interior of the polyhedral obstacles E. Also let $\mathrm{FREE}(s_1, \ldots, s_k)$ be the conjunction of $\mathrm{FREE}(s_{i-1}, s_i)$ for $i = 1, \ldots, k+1$ where $s = s_0, t = s_{k+1}$.

Define $\mathrm{PATH}_k(s_1, \ldots, s_k)$ to be disjunction of $\Phi_{S_1,\ldots,S_k}(s_1, \ldots, s_k)$'s, varying over all $S_1, \ldots, S_k$. Although we do not know the number k of intermediate edges that a shortest path goes through, we can easily bound k by the total number n of edges in the obstacles. Let

$$\mathrm{PATH}(s_1, \ldots, s_n) := \bigvee_{k=0}^{n} \left(\mathrm{PATH}_k(s_1, \ldots, s_k) \wedge s_k = s_{k+1} = \cdots = s_n\right).$$

Thus, there is again only a finite set of solutions to $\mathrm{PATH}(s_1, \ldots, s_n)$, and all shortest paths that avoid obstacles is among this set.

Let $D(s_1, \ldots, s_k)$ denotes the Euclidean length of the path $(s, s_1, \ldots, s_k, t)$. The assertion "$z = D(s_1, \ldots, s_k)$" can be written as the Tarski formula

$$(\forall v_0, \ldots, v_k)\left[\bigwedge_{i=0}^{k} \left(v_i^2 = \|s_{i+1} - s_i\|^2 \wedge v_i \geqslant 0\right). \Rightarrow .z = \sum_{i=0}^{k} v_i\right]. \tag{2}$$

Notice that we can as well replace the universal quantifiers in (2) by existential quantifiers. Finally the Tarski predicate $\mathrm{SHORTEST}(s_1, \ldots, s_n)$ that asserts that $(s, s_1, \ldots, s_n, t)$ is a shortest path that avoids E is given by

$$\mathrm{PATH}(s_1, \ldots, s_n) \wedge \mathrm{FREE}(s_1, \ldots, s_n) \wedge$$
$$(\forall s'_1, \ldots, s'_n)\left[\mathrm{PATH}(s'_1, \ldots, s'_n) \wedge \mathrm{FREE}(s'_1, \ldots, s'_n) \Rightarrow D(s_1, \ldots, s_n) \leqslant D(s'_1, \ldots, s'_n)\right].$$

The result that the set of shortest paths is a finite semi-algebraic set follows from the following lemma.

Lemma 2. $\mathrm{SHORTEST}(s_1, \ldots, s_n)$ *has finitely many solutions and represents the set of all shortest paths from* s *to* t *avoiding obstacles* E.

This construction yields other information: since $\mathrm{PATH}(s_1, \ldots, s_n)$ is quantifier-free, and (2) uses universal quantifiers only, we see that $\mathrm{SHORTEST}(s_1, \ldots, s_n)$ is a universal formula.

References

[1] D.H. Bailey, MPFUN: a portable high performance multiprecision package, Technical Report RNR-90-022, NASA Ames Research Center, 1990.

[2] P. Bertin, D. Roncin and J. Vuillemin, Introduction to programmable active memories, Research Report 3, Digital Paris Research Laboratory, June, 1989.

[3] R.P. Brent, A Fortran multiple-precision arithmetic package, ACM Trans. Math. Software 4 (1978) 57–70.
[4] J. Canny, Some algebraic and geometric configurations in PSPACE, in: Proc. 20th ACM Symp. Theory Comput. (1988) 460–467.
[5] J. Canny and J.H. Reif, New lower bound techniques for robot motion planning problems, IEEE Found. Comput. Sci. 28 (1987) 49–60.
[6] K.L. Clarkson, Safe and effective determinant evaluation, IEEE Found. Comput. Sci. 33 (1992) 387–395.
[7] C.W. Clenshaw, F.W.J. Olver and P.R. Turner, Level-index arithmetic: an introductory survey, in: P.R. Turner, ed., Numerical Analysis and Parallel Processing, Lecture Notes in Mathematics 1397 (Springer, Berlin, 1987) 95–168.
[8] G.E. Collins, Quantifier elimination for real closed fields by cylindrical algebraic decomposition, in: 2nd GI Conf. on Automata Theory and Formal Languages, Lecture Notes in Computer Science 33 (Springer, Berlin, 1975) 134–183.
[9] J.H. Davenport, Y. Siret and E. Tournier, Computer Algebra: Systems and Algorithms for Algebraic Computation (Academic Press, New York, 1988).
[10] D. Dobkin and D. Silver, Recipes for geometry and numerical analysis, I: an empirical study, ACM Symp. Comput. Geom. 4 (1988) 93–105.
[11] T. Dubé and C. Yap, A basis for implementing exact geometric algorithms, September, 1993, Extended Abstract.
[12] H. Edelsbrunner, Algorithms in Combinatorial Geometry (Springer, Berlin, 1987).
[13] H. Edelsbrunner and E.P. Mücke, Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms, ACM Symp. Comput. Geom. 4 (1988) 118–133.
[14] A. Feldstein and P.R. Turner, Overflow, underflow, and severe loss of significance in floating-point addition and subtraction, IMA J. Numer. Anal. 6 (1986) 241–251.
[15] M.J. Fischer and M.O. Rabin, Superexponential complexity of presburger arithmetic, in: R.M. Karp, ed., Complexity of Computations, Vol. 7 (1974), Proceedings SIAM-AMS Symp. on Applied Mathematics.
[16] S. Fortune, Numerical stability of algorithms for 2-d Delaunay triangulations and Voronoi diagrams, in: Proc. 8th ACM Symp. Comput. Geom. (1992) 83–92.
[17] S. Fortune, Progress in Computational Geometry, in: R. Martin, ed., Information Geometers, 1993, Chapter 3, 81–127.
[18] S. Fortune and C. van Wyk, Efficient exact arithmetic for computational geometry, ACM Symp. Comput. Geom. 9 (1993) 163–172.
[19] C.M. Hoffman, Geometric and Solid Modeling: An Introduction (Morgan Kaufmann, San Mateo, 1989).
[20] T.E. Hull, A. Abraham, M.S. Cohen, A.F.X. Curley, C.B. Hall, D.A. Penny and J.T.M. Sawchuk, Numerical Turing, ACM SIGNUM Newsletter 20(3) (July 1985) 26–34.
[21] T.E. Hull, M.S. Cohen, J.T.M. Sawchuk and D.B. Wortman, Exception handling in scientific computing, ACM Trans. Math. Software 14(3) (1988) 201–217.
[22] M. Karasick, D. Lieber and L.R. Nackman, Efficient Delaunay triangulation using rational arithmetic, ACM Trans. Graphics 10 (1991) 71–91.
[23] V. Milenkovic, Robust polygon modeling, Computer-Aided Design, to appear, fall 1993. (Special issue on Uncertainties in Geometric Computations.)
[24] V. Milenkovic and L. Nackman, Finding compact coordinate representations for polygons and polyhedra, ACM Symp. Comput. Geom. 6 (1990) 244–252.
[25] C.H. Papadimitriou, An algorithm for shortest-path motion in three dimensions, Inform. Process. Lett. 20 (1985) 259–263.
[26] D.A. Patterson and J.L. Hennessy, Computer Architecture: A Quantitative Approach (Morgan Kaufmann, San Mateo, CA, 1990) (with an appendix on Computer Arithmetic by D. Goldberg).

[27] D.A. Patterson and J.L. Hennessy, Computer Organization and Design: The Hardware/Software Interface (Morgan Kaufmann, San Mateo, CA, 1994).
[28] F.P. Preparata and M.I. Shamos, Computational Geometry (Springer, New York, 1985).
[29] A. Schönhage, The fundamental theorem of algebra in terms of computational complexity (1985).
[30] J.T. Schwartz and M. Sharir, On the piano movers' problem: II. General techniques for computing topological properties of real algebraic manifolds, Adv. Appl. Math. 4 (1983) 298–351.
[31] 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.
[32] M. Shand, P. Bertin and J. Vuillemin, Hardware speedups in long integer multiplication, 2nd Annual ACM Symposium on Parallel Algorithms and Architectures, Crete, 1990.
[33] M. Sharir and A. Schorr, On shortest paths in polyhedral spaces, SIAM J. Comput. 15 (1986) 193–215.
[34] K. Sugihara and M. Iri, Two design principles of geometric algorithms in finite precision arithmetic, Appl. Math. Lett. 2 (1989) 203–206.
[35] 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 (29 August–2 September 1988).
[36] K. Sugihara and M. Iri, A numerically stable method for Voronoi diagram construction, Proc. 1988 Fall Conference of the Operations Research Society of Japan, Tokyo (28–29 September 1988) 20–21.
[37] K. Sugihara, Robust gift-wrapping for the three-dimensional convex hull, J. Comput. System Sci., 1993, to appear.
[38] C.K. Yap, Fundamental Problems in Algorithmic Algebra (Princeton University Press, Princeton, NJ), to appear. Available on request from author and from the URL ftp://cs.nyu.edu/pub/local/yap/Algebra.
[39] C.K. Yap, A geometric consistency theorem for a symbolic perturbation scheme, J. Comput. System Sci. 40(1) (1990) 2–18.
[40] C.K. Yap, Symbolic treatment of geometric degeneracies, J. Symbolic Comput. 10 (1990) 349–370; Proc. International IFIPS Conference on System Modelling and Optimization, Tokyo (1987), Lecture Notes in Control and Information Science 113 348–358.
[41] 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.