In Praise of Numerical Computation

Dedicated to Kurt Mehlhorn on his 60th Birthday

Chee K. Yap*

June 11, 2009

Abstract

Theoretical Computer Science has developed an almost exclusively discrete/algebraic persona. We have effectively shut ourselves off from half of the world of computing: a host of problems in Computational Science & Engineering (CS&E) are defined on the continuum, and, for them, the discrete viewpoint is inadequate. The computational techniques in such problems are well-known to numerical analysis and applied mathematics, but are rarely discussed in theoretical algorithms: iteration, subdivision and approximation. By various case studies, I will indicate how our discrete/algebraic view of computing has many shortcomings in CS&E. We want embrace the continuous/analytic view, but in a new synthesis with the discrete/algebraic view. I will suggest a pathway, by way of an exact numerical model of computation, that allows us to incorporate iteration and approximation into our algorithms' design. Some recent results give a peek into how this view of algorithmic development might look like, and its distinctive form suggests the name "numerical computational geometry" for such activities.

You might object that it would be reasonable enough for me to try to expound the differential calculus, or the theory of numbers, to you, because the view that I might find something of interest to say to you about such subjects is not prima facie *absurd; but that geometry is, after all, the business of geometers, and that I know, and you know, and I know that you know, that I am not one; and that it is useless for me to try to tell you what geometry is, because I simply do not know.*

— G.H.Hardy, in "What is Geometry?"

1925 Presidential Address to the Mathematical Association

1 Introduction

This article celebrates the scientific work of Professor Kurt Mehlhorn, a special friend and colleague. Few computer scientists can match the impact that Kurt has had in computer science. Even to summarize the scope of his work would be a daunting task. Since this essay is about numerics, I may let the numbers speak for themselves: his current webpage lists 207 papers, 9 books and 6 software systems. I propose to only highlight one aspect of Kurt's experimental work, as it is a special tribute to say that any theoretician had significant experimental contributions. Over twenty years ago, Kurt began a quest to put the corpus of data structures and algorithms produced by the theoretical Computer Science community into code. That was the birth of the software library known as `LEDA` [31, 32]. Indeed, around this time, computational geometry witnessed a spurt of experimental geometric software development. But major software development requires sustained effort over a long period of time which, as theoreticians, we may not have the constitution for.

*This work is supported in part by NSF Grants CCF-043086 and CCF-0728977.

Yet today LEDA is the basis of a successful commercial company. Like most large software, LEDA is the work of many hands: Kurt's collaborators include Stefan Näher with whom he wrote the LEDA book [30], Stefan Schirra, Christian Uhrig, Christoph Burnikel and others.

¶1. What LEDA has Achieved. LEDA has implemented the best practical data structures and discrete algorithms that have been developed in the last 40 years. But the unique part of LEDA lies in its collection of geometric algorithms. Since the late 1980's, computational geometers have become acutely aware of numerical nonrobustness issues in geometric computation. Of all the areas of algorithms, we are especially afflicted. Some have declared the problem intractable, even for problems as simple as the robust intersection of two line segments. In retrospect, what is remarkable about Kurt's foresight was his insistence, from the very first, that LEDA must be fully reliable and practical, *even for geometric algorithms*. Twenty years ago, that was a big wish for a geometric library. A few "robust geometric algorithms" were beginning to appear in the literature, but nothing with which to stock an entire library. Each problem required special treatment, and many approaches were contending to solve nonrobustness issues (see my survey in [54]).

I will classify these approaches into two camps: those wishing to make fast machine arithmetic reliable and those wishing to compute exactly in order to achieve reliable software. Kurt's approach falls under the latter "exact" camp. Many researchers in our community did not think the exact camp could be practical or could compete with machine floating point computation. To place yourself in context, by the late 1980's, machine floating point had become the dominant mode of numerical computation (and has remained so today). Floating point arithmetic has become standardized, enjoys full industry support, and has moved from software into standard hardware in the form of co-processors. This view is summed up by Steve Fortune's foreword in an Algorithmica special issue on implementation issues [22]: "*Floating point arithmetic has numerous engineering advantages: it is well-supported ... the Challenge is to demonstrate that a reliable implementation can result from the use of floating point arithmetic.*"

What about exact computation? It was (and still is) regarded as the domain of specialists and specialized applications. Yu [59] wrote a thesis under Chris Hoffmann that concluded that exact computation will not be practical for Boolean operations on polyhedral objects in the foreseeable future. But LEDA did find a general and systematic solution to nonrobust geometry — not by implementing specialized "robust algorithms" for each problem — but by introducing a general number type called `Leda Real` that has the remarkable property that comparisons are error-free. According to the principles of **exact geometric computation**, this implied that the geometry would be exact and hence free from nonrobustness issues. Now, if a computation involves only rational operations, then this property might not appear impressive (just use a BigRational number package, although you would still run into the efficiency bottleneck described by Yu). But `Leda Real` included square-roots and later, arbitrary real algebraic numbers. Despite this, it remains practical for all the common geometric problems. Today, such algorithms are reasonably competitive with nonrobust machine-precision algorithms. Any programmer can implement a fully robust geometric algorithm (provided the primitives are algebraic) using software such as LEDA. Superficially, it appears that the exact camp has won in a healthy contest of ideas. But lurking behind this triumph, we see some ideas of the other camp are also firmly embedded.

¶2. Exact Numerical Computation. How does `Leda Real` do this? There are five key ingredients, the first two well-known and next three novel:

(1) You must use arbitrary precision — but use **BigFloats** (for efficiency, do not use BigRationals).

(2) Track errors automatically — use **interval arithmetic**. Interval arithmetic tells us when a comparison between approximate values is valid.

(3) All numbers must have an exact representation — use **expressions**. This representation supports the the ability to approximate each number to any desired absolute precision. Such approximations must be available on demand.

(4) You must solve the **zero problem**, described later. In practice, we use some **constructive zero bounds** which tell us when a numerical approximation is small enough that we may declare the exact value to be zero. The BFMSS bound [11] from the LEDA group is one of the best zero bounds in this area.

(5) You should exploit adaptivity of numerical computations. A highly effective technique here is **numerical filters** which can decide most comparisons quickly. Thus, through filters, the "engineering advantages of floating point arithmetic" of Fortune is restored. Work from `LEDA` is in the vanguard of trying to extend such techniques, from cascading filters to filtering of general algorithms [12, 23].

These ideas also appear in my earlier work on the `Real/Expr` [57], the precursor to `Core Library`. Another major library founded on similar principles of exact numerical computation is the `CGAL` library [21]. The computing principle that urges us to such a distinctive mode of computation is exact geometric computation. But in this paper, I want to look at the broader implications; for this purpose, I call this mode of computation **exact numerical computation** (ENC). Note that "numerical" often has the connotation of inexactness, but no such inference is[1] intended here. Of course, we will use numerical approximations, but they are used to derive exact conclusions with the help of zero bounds. Actually, exactness in ENC cannot be taken for granted: very little is known about the zero problem in transcendental cases [42]. In such cases, as applied mathematicians know very well, we need carefully circumscribed conditions (smoothness, Morseness, non-singularity, Lipschitz, etc) that allow exact solution. Another way to restore exactness is to modify correctness in the sense of backwards error analysis. All these are within the parameters of ENC.

Exact computation is traditionally the domain of symbolic computation and computer algebra. Nevertheless, ENC has no parallel in the computer algebra literature (e.g., [10] or [17, Chap. 4]). Algebraic computation in computer algebra is greatly influenced by the great subject of algebraic number theory, focusing on algebraic and arithmetical properties of number fields $\mathbb{Q}(\alpha)$ (e.g., [37]). But such approaches do not have the flexibility and adaptivity necessary to be deployed in practical geometric computations. My favorite illustration is the following: to compute a number of the form $\alpha = \sum_{i=1}^{100} \sqrt{n_i}$ (for positive integers n_i), standard computer algebra methods require the computation of a defining polynomial of α which generally has degree 2^{100}, a daunting task. Yet, in a geometric application like computing Euclidean shortest paths, we may have to handle thousands of such α's. Using our ENC approach, most of these computations can be dispatched quite routinely since we only need to construct an expression for each α. The comparison of such α's could be time consuming, but in practice we are saved by ENC's adaptive complexity.

To sum up, I believe that `LEDA` represents an important achievement in computing history: through the work of `LEDA` and related work in the computational geometry community, we now understand the fundamental barriers to robust geometric computation and have identified key elements for solving this problem in a systematic way. The existence of commercial libraries such as `LEDA` and `CGAL` prove that robust geometric computation is a practical reality today. Kurt's broad insights and leadership in this area have played a major role in this achievement. To read some of Kurt's thoughts on this area, I recommend[2] his article [29]. The societal benefits of robust geometric computation are potentially immense: nonrobust numerical computation has negative impact on programmer/researcher productivity (many of us experience this), represents a huge economic cost [41], stands in the way of full automation in industry, and often plays a role in dramatic disasters.

¶3. An Apology. The above quote from Hardy [25] expresses my own ambivalence about writing on numerical computation, for I know that you know that I do not do much numerical computing. What little I know is the combination of numerical computation with algebraic computation. It is this synthesis that I will talk about. My praise of numerical computation represents a slow personal conversion that has grown over time. When I told a colleague what I will write about, the reaction was — *but surely computation* is *discrete*? I hope to show that there is a deeper issue at stake.

2 Return to the Continuum

What I have discovered over the years, as an unintended consequence of the pursuit of robust geometric computation, is that numerical computation has many virtues that theoretical algorithms fail to recognize. We have been enamored with discrete computation (which is good in itself) but to the exclusion of the continuous. We feel that if a problem or algorithm is numerical, it is the domain of numerical analysts and

[1] There is an important and related issue of inexact data, which I do not address in this essay.

[2] It was written for another similar occasion, the festschrift of Thomas Ottmann.

applied mathematicians. It is true that we should not be amateurs in what others can do better. But the study of ENC has convinced me that some of these numerical concerns should be our concern.

¶4. Problems in Computational Science & Engineering Let me briefly clarify what I mean by the "continuum". In some literature, it refers to the real numbers $\mathbb{R}$. But we may expand its reference to any locally compact topological space such as $\mathbb{R}^n$ or $\mathbb{C}$. By **continuum problems**[3] we mean the problems of computing functions whose domains and/or ranges are continua. Unfortunately, the theory of continuum computing, often pronounced as "real computation", is in its relative infancy because its foundations are still very much in dispute [56]. We have no consensus similar to Church's thesis in discrete computation. This is an exciting opportunity for the computability and complexity theorist, but this is not my focus in this essay.

First, I point out what we are missing out on by our totally discrete view of computing. I am especially interested in problems arising in a constellation of subdisciplines, collectively known as **Computational Science & Engineering** (CS&E). For any discipline X of science, mathematics, or engineering, it is possible to identify a subdiscipline called "Computational X". Thus we have computational biology, computational physics, computer algebra, etc. In the earth, atmospheric and ocean sciences, the computational aspect is so central that it is redundant to attach the "computational" prefix. There has been an explosive growth in computational activities in CS&E. Keen observers of the scientific enterprise have identified the CS&E phenomenon as representing a third pathway to scientific discovery. Alongside the two traditional pathways based on theory (deduction) and experimentation (induction), we now have computation (simulation). In many disciplines X, computer simulation is increasingly seen as an alternative to physical experimentation. Computational labs vie with traditional wet labs to provide insights for X.

Where is the Computer Science in computational X? Taking a highly Computer Science-centric view, imagine computational X as a collection of computational problems, and so CS&E is the union of these collections. Let us also regard Computer Science as a collection of computational techniques. Then the relationship between Computer Science and CS&E can be pictured as a matrix where each problem is represented by a column, and each technique represented by a row. Each matrix entry has a numerical score between 0 and 1, indicating the relevance of a technique to a problem. Of course, this is only a cartoon view to make a point. In Table 1, I have further simplified the column space by identifying each computational X with only one column.

	Atmospheric Sciences	Comput. Biology	Material Science	Comput. Physics	$\cdots$	Electrical Engineering
Huge Datasets	1.0	0.8	0.2	0.6	$\cdots$	0.1
Optimization	0.2	0.2	0.5	0.2	$\cdots$	0.3
Symbolic Computation	0.1	0.0	0.2	0.7	$\cdots$	0.5
String Algorithms	0.0	0.8	0.0	0.2	$\cdots$	0.0
Parallel Algorithms	0.5	0.2	0.2	0.4	$\cdots$	0.1
$\vdots$	$\vdots$				$\ddots$	

Table 1. The CS&E Matrix.

Practitioners of Computational X tend to identify themselves with a particular column (as "column scientists"), while computer scientists might view themselves doing row science. The glue that makes CS&E coherent is Computer Science. When we develop algorithms in a particular row, we are often oblivious to the applications. But by aligning our row activities to particular columns we may gain new insights for the science of computing, and we would share in the advancement of X. This would be the "best practice".

But the record of involvement of Computer Science in the Computational X's shows an uneven record. In areas such as computational biology, there is a great synergy, while in many others, the Computer Science component is[4] basically non-existent. This dichotomy is highly correlated with the division between discrete and continuous computation. Computational biology can be reduced to discrete algorithms (strings and trees), and it is easy for computer scientists to make contributions at this level of abstraction. But

[3] Sometimes called "continuous problems", but this terminology is confusing, especially for geometric computation which is inherently discontinuous.

[4] Just because Computational X uses computers does not mean that there is Computer Science development in it, any more than there is carpentry in Computer Science although computer scientists use tables and wooden cabinets. David Bindel reminds me that numerical analysis is present in these fields, and so it is clear that in this essay, I use "Computer Science" in a narrow sense without including numerical analysis.

Computer Science involvement falls off rapidly as the need for numerical computation increases. My general thesis is that there is a large role for theoretical algorithms in such computation, and this ought to be most clearly understood by those of us who work in the field of computational geometry.

¶5. Two Worlds of Computing. When I suggest that Computer Science should engage in the continuum problems of CS&E, it invites a clash of two world views on computing. One view is motivated by computing ideal mathematical objects and the other, by physical modeling. Most of us live exclusively in one of these worlds and are oblivious to the other. General claims about computing from one perspective can be quite wrong in the other. We cannot afford to fall into this trap, as we intend to work at their interface.

In the **mathematical world view**, the continuous makes perfect sense and its use in mathematical modeling has been highly successful. Exact computation is also meaningful here, whether it is computing arbitrarily accurate values of π or in automated proof of geometric theorems. The ideas of exact geometric computation sit comfortably in this world. The algorithmic problems studied in theoretical computer science are also exact ones.

The **physical world view** disdains exact computing, however. It is argued that the physical world is discrete and nondeterministic. In other words, the continuous and deterministic world is a myth. When such arguments are used to dismiss the reality of the other world of computing, they miss the point of myths, whether in folklore or in science. We know that a "point mass" in Physics is a fiction, but try abolishing it from Physics text books. The point (no pun intended) is that continuous models satisfy Occam's Razor in their description of many physical phenomena. Thus, the term "continuum mechanics" is no oxymoron even when applied to the study of particle systems like fluids and gases.

Besides discreteness, the physical world view advances a related argument about finiteness. It is noted that physical constants have limited accuracy and that current 64-bit machine precision seems more than adequate for physical modeling, from the subatomic to astrophysical scales. There is a simple counter to this thought: a well-known phenomenon in ENC is that in order to compute a value up to (say) 1-bit accuracy, the intermediate computed values might require arbitrarily many bits of accuracy. In fact, it is remarkably easy to run out into very high precision.

Many applied fields are ostensibly uninterested in exactness. It may be hard to see why such fields might have any interest in exact computation, so let me provide some examples. In computer graphics, it is arguably unnecessary to compute beyond the accuracy of screen resolution (this is analogous to the limited accuracy of physical constants). Over the years, on quizzing experts in this field, I have been repeatedly surprised by acknowledged[5] nonrobustness issues. Or consider protein folding, an inherently approximate process. Nevertheless, the folded protein may have several distinct minimal energy states: how could we ensure that our numerical simulation has sufficient accuracy to be[6] *qualitatively* correct? Or consider the fact that approximate computation may be best modeled by an idealized mathematical model. Then, the best policy might be to compute exactly, or to try to emulate exact computation. As another interesting example, floating point computer arithmetic must ultimately rely on exact computation to solve the exact rounding problem [58]. Therefore I believe that, like the myths of point masses and continuum mechanics, exact computation has a role to play even in the approximate computations of CS&E.

¶6. Is Geometry Continuous or Discrete? Many continuum problems are geometric in nature. So it is useful to understand the general character of geometric computation. Computational geometers have much insight to offer in this regard, as they have had to grapple with this question as they confronted nonrobustness in geometric computation.

Geometry comes in two main forms: analytic geometry and synthetic geometry. The former uses equations and coordinates to define geometry while the latter (e.g., Euclidean geometry) proceeds from axioms. Interestingly, Hardy [25] regards analytic geometry as mundane and considers synthetic geometry as the "higher geometry". But computationally, we see that analytic geometry is by far the more important (cf. [7]). In automatic geometric theorem proving, for instance, the synthetic approach has had limited success, while the analytic approach, especially influenced by Wu Wen-Tsun's insights, has flowered today. Henceforth, I focus exclusively on analytic geometry.

[5] You are unlikely to see these issues discussed in print.

[6] i.e., closer to the correct minimal energy state than to any others.

Analytic geometry is the interplay of the continuous and discrete. The continuum enters in two ways. To make this concrete, allow me to introduce a little framework. In the first place, geometric objects are **parametric objects**. Geometric prototypes are points and lines in the plane. A point p is given by $Point(x, y)$ and a line ℓ is given as $Line(a, b, c) : aX + bY + c = 0$ where $x, y, a, b, c \in \mathbb{R}$ are numerical parameters. These parameters might be constrained (e.g., $a^2+b^2 > 0$). The **parameter space** of geometric objects of each type is therefore a continuum. The space of points may be identified with the Euclidean plane $\mathbb{R}^2$, and the space of lines is a subset of the projective space $\mathbb{P}^2(\mathbb{R})$.

In general, we can treat more complex geometric objects, such as a convex polytope in $\mathbb{R}^n$, as a cell complex in the sense of algebraic topology. The **(combinatorial) type** of a geometric object can be represented by a directed graph G with parametric variables $X_1, \ldots, X_m$ associated with its vertices and edges, together a constraint predicate $C(X_1, \ldots, X_m)$. We will write $G(X_1, \ldots, X_m)$ for this type, with $C(X_1, \ldots, X_m)$ implicit. An assignment of values $a_i \in \mathbb{R}$ to each X_i is valid if the predicate $C(a_1, \ldots, a_m)$ holds. E.g., $C(a, b, c)$ might say that $a^2+b^2 > 0$. A valid assignment $(a_1, \ldots, a_m)$ to $G(X_1, \ldots, X_m)$ is called an instance, and we write "$G(a_1, \ldots, a_m)$" for the instance. All geometric objects with which we compute can be put in this form (see [54]). For each type $G(X_1, \ldots, X_m)$, we obtain a parametric space comprising all of its instances.

Suppose we have a surface S, viewed as a parametric object $S = \text{Surface}(\mathbf{x})$ with parameters $\mathbf{x} \in \mathbb{R}^m$. These parameters can be approximated by some $\widetilde{\mathbf{x}}$. If $\|\mathbf{x} - \widetilde{\mathbf{x}}\| \leq \varepsilon$, we call $\text{Surface}(\widetilde{\mathbf{x}})$ a **parametric ε-approximation** of S. But we will see a (for us) more important kind of approximation.

The continuum enters geometry in a second way. Geometric objects such as points, hypersurfaces, cell complexes, etc, must live in a common **ambient space** in order to interact. Each geometric object $G = G(a_1, \ldots, a_m)$ is associated with a subset $\lambda(G(a_1, \ldots, a_m))$ of its ambient space, say $\mathbb{R}^n$ (for some n). Call $\lambda(G)$ the **locus** of G; in practice, we often identify G with its locus. For instance, if G is a curve, its locus is a 1-dimensional subset of $\mathbb{R}^n$.

Two geometric objects S and T, not necessarily of the same type, living in a common ambient space, are said to be **ε-close** if the Hausdorff distance between their loci is $\leq \varepsilon$. Thus, we can approximate continua, such as surfaces S, by discrete finite objects such as triangulations T, and we call T an **explicit ε-approximation** of S. The **explicitization problem** is to compute an explicit ε-approximation T from (the parameters of) S. For instance, a real function $f : \mathbb{R}^n \to \mathbb{R}$ can be explicitly approximated by a triangulation T that approximates its graph $gr(f) := \{(\mathbf{x}, f(\mathbf{x})) : \mathbf{x} \in \mathbb{R}^n\}$. Such approximations are central to our concerns.

Interaction among geometric objects is captured by **geometric predicates** which define relationships via loci. Thus, we have the $OnLine(p, \ell)$ predicate which holds if p lies on ℓ, or the $LeftTurn(p, p', p'')$ predicate which holds if we make a left turn at p' as we move from p to p' and then to p''. Let R be a set of geometric predicates, and let $O_1, \ldots, O_m$ be m sets of geometric objects, each set O_i having a fixed type. For instance, let $m = 2$, O_1 be a set of points, and O_2 a set of lines. The sum total of the geometric relationships defined by R on $(O_1, \ldots, O_m)$ constitute the "geometry" of $(O_1, \ldots, O_m)$ induced by R. *The field of computational geometry is concerned with computing, representing, and querying such geometries.*

Geometric objects can also be constructed from other data: besides constructing the objects directly from their numerical parameters (e.g., $p \leftarrow Point(x, y)$ or $\ell \leftarrow Line(a, b, c)$), we may construct them from other geometric objects (e.g., $p \leftarrow Intersect(\ell, \ell')$ or $\ell \leftarrow Line(p, p')$). Such predicates and constructors become the primitives for geometric computation as we shall see in the next section.

3 Abstract Computational Models

Computational Geometry is primarily concerned with discrete and combinatorial algorithms. These algorithms are largely non-numerical. This last characterization must strike the casual observer as an anomaly. Since geometric data arises from the continuum, surely numerical computations must be central to geometric computation? By nature, computation is discrete: each computational step (sequential or parallel) is a discrete event that transforms the computational data in a well-defined (not necessarily deterministic) way. Despite this discrete nature, we can develop computational models for continuum problems and geometric applications. This paradox also appears in mathematical logic: we build theories of the continuum using a logical language that is countable.

Computation and mathematical logic have much more in common. In discussing computational models, especially for continuum computation, we can take another page from logic. The language of any first order theory is comprised of two parts: (A) a "logical part" that has the standard logical symbols such as Boolean operators, equality, quantifiers and a countable set of variables; (B) an "extra-logical part" that[7] has predicate and operator symbols which are unique to the particular theory (e.g., [46]). Standard rules of logical deduction are supplemented by special axioms for the extra-logical parts of the theory.

We apply a similar approach to computation: each computational model can be divided into a logical part, called the **base model**, and an extra-logical part. Because of the extra-logical part, such models are called **abstract (computational) models**. The base model may be any standard computational model; Turing machines and random access machines (RAM) [1] are commonly used. Typically, the base models are at least equivalent to Turing machines. The choice of a base model determines the kind of data structures and type of control structures of our algorithms. With Turing machines as the base model, we are making the choice to have strings as our basic data structure. The extra-logical powers typically come from oracles (perhaps countably many). To compute with real numbers, these oracles can represent real functions. Through these oracles, we can access countably many real constants (like π, e) via special string encodings that the oracle understands. Ko [27] uses such oracle Turing machines extensively in his work. Alternatively, in using RAMs as the base model, we are choosing the ability to have random access to storage locations containing integers. To compute with extra-logical objects such as real numbers, we allow the storage locations to store reals, and provide corresponding real predicates and operators.

¶7. Abstract Pointer Machines. Thus we may speak of "abstract Turing machines" or "abstract RAMs". But I am especially partial to **abstract pointer machines** that use Schönhage's pointer machines[8] [43] as the base model. Pointer machines directly encode and manipulate structures called **tagged graphs**, i.e., directed graphs whose edges (called **pointers**) have labels (called **tags**) taken from a finite set Δ of symbols. Its operations are instantly recognizable by computer scientists. In brief, its two main operations are

$$\begin{aligned} \text{Assignment} &: w \leftarrow v \\ \text{Test} &: \mathsf{if}\ (w \equiv v)\ \mathsf{goto}\ L \end{aligned}$$

where $w, v \in \Delta^*$ and L is a natural number (a label of an instruction). A pointer machine, in bare form, is a finite sequence of such instructions. If G is a tagged graph with a designated node called the origin, then a string $w \in \Delta^*$ yields a path that begins at the origin and ends at a node denoted $[w]_G$. The last pointer (edge) in the path w has a special role: execution of the assignment instruction "$w \leftarrow v$" will modify G by re-directing the last pointer of w to point at $[v]_G$. The test instruction "$\mathsf{if}\ (w \equiv v)\ \mathsf{goto}\ L$" is also easily understood: the indicated goto is executed if $[w]_G = [v]_G$.

Such machines are naturally extended to operate on algebraic entities such as real numbers, which are stored in the nodes of the graphs (see [56]). Abstract pointer machines are natural for geometric computation which calls for the juxtaposition of combinatorial structures with real numbers. Imagine trying to encode geometric structures into strings in an abstract Turing machine — a most unnatural thought.

In fact, the analogy to logic can be carried even further: just as modern logic does not require logical languages to be associated with any particular model, we can also view our computational models to be pure syntax, with certain syntactic rules. It is up to the application to provide models and "abstract interpretations". But this would take us beyond our immediate interest. *So in the following, I assume that each abstract computational model comes with a standard interpretation.*

¶8. Classification of Abstract Models. Once we have an abstract computational model, we can discuss computability and complexity. The simplest complexity model is to charge a unit cost for each operation. I will categorize important abstract computational models from the literature into three classes:

- **Analytic Models.** In the field of computable analysis, models based on Type-2 Theory of Effectivity (TTE) [53] or oracle TM's [27] have been studied. I will shortly discuss numerical models that fall under this classification.

[7] Also called the "non-logical" part.
[8] Also known as storage modification machines.

- **Algebraic Models.** The real RAM [1] is an example. The BSS model of Blum-Shub-Smale [5] can be seen as a real RAM with limited random access. Alternatively, it is a Turing machine whose tape cells can store arbitrary real numbers. The extra-logical powers here are the ring operations $(+, -, \times, 0, 1)$ and real comparison $(<, =)$. More generally, we call an abstract model "algebraic" if the extra-logical objects belong to algebraic structures like rings or fields, and the extra-logical functions or predicates take only these objects as arguments. Note that the extra-logical operation $\exp(x)$ or $\sin(x)$ count as "algebraic" in our sense. The Information-Based Complexity approach of Traub and Woźniakowski [50] focus on algorithms in such algebraic models.

- **Geometric Models.** It is tedious to design geometric algorithms directly in the algebraic or analytic models. So most geometric models use the real RAM as the base model, introduce higher level objects such as points or surfaces, and assume geometrically meaningful predicates and constructors such as those discussed earlier. For example, below we discuss a geometric constructor that shoots a ray to obtain a sample point on a surface.

Abstract models are important and useful, regardless of whether they are realistic or not. I stress this point because some have criticized algebraic models (e.g., BSS model) on account of unrealism. But it would be untenable to develop most of the algorithms of computational geometry in an analytic or algebraic model. The unique place[9] of the Turing model is never challenged by any of these abstract models. The abstract models serve other useful purposes, including providing a modular description of algorithms at various levels of abstraction [56]. Thus, the algebraic model, not the standard Turing machines, is most appropriate for describing Strassen's matrix multiplication algorithm.

4 Case Studies in Abstract Models

Computational models greatly influence the kinds of algorithms we design. They can hide or accentuate different computational issues. Of course, we know this. Nevertheless, we might gain some insights into potential pitfalls by looking at four case studies.

¶9. CASE 1: Meshing or, Watch your Implementation Gap. Most physical simulations require some kind of mesh. For our purposes, we may identify a mesh as a triangulation. I will consider a basic problem in meshing: generating topologically correct ε-close meshes for implicit surfaces (see the survey [6]). In case the surface is algebraic, a well-known algebraic approach to this problem uses some form of cylindrical algebraic decomposition. As these algebraic techniques are expensive and non-adaptive, I will focus on two adaptive approaches based on sampling and subdivision. They differ in their choice of abstract models. Let S be a non-singular surface given by $f(x, y, z) = 0$.

In the **sampling approach**, we construct a mesh T from a finite set P of sample points on S. Typically, $T = T(P, S)$ is a subset of the Delaunay triangles of P. We incrementally add sample points to P until the required ε-closeness criterion is achieved. Such algorithms are based on a geometric model that supports the classical primitive of "ray shooting". The primitive returns the first point p (if any) on S intersected by a given ray. We then add p to the set P. If S is an algebraic surface, the sample points would have algebraic number coordinates. Although it is possible to implement such a ray shooting model exactly (it reduces to computing the first positive root of a polynomial), this is expensive and nontrivial to implement. Should we implement *exact* ray shooting in order to *approximate* a surface? Probably not. Yet there is no known analysis of sampling algorithms based on approximate sample points. This leaves an **implementation gap** in what is otherwise a beautiful exact approach. The next section will expand on this example.

We turn to the **subdivision approach**. Typically, we wish to construct a mesh for the part of the surface lying within some given box B_0. We construct a quadtree rooted at B_0 by repeated subdivision until each leaf box satisfies some criterion. The well-known marching cube algorithm falls under this approach. Subdivision methods are easy to implement and widely used in practice. I want to highlight an algorithm of Plantinga and Vegter (PV) [36] which represents the first complete purely numerical algorithm for the

[9] Although the standard base models are equivalent by Church's thesis, the Turing model captures complexity-theoretic concepts such as space, time, nondeterminism, etc. much better than most. The pointer model is close to the Turing model in this respect.

meshing of non-singular surfaces in $\mathbb{R}^3$. This is no mean achievement, considering that there were been several prior attempts (e.g., [49, 47, 40]) that fall short in one way or another. Unlike the sampling approach, this numerical algorithm suffers no implementation gap: it is easy to implement using just a bigFloat number package.

¶10. CASE 2: Transcendental Comparisons or, Can we really do this? The previous case study points out the hidden cost of implementing abstract real RAM operations. In our second case, we see an extreme example of this phenomenon. In 2003, while I was visiting the laboratory of Professor Doeksoo Kim in Hanyang University, Korea, he demonstrated his geometric software for computing shortest paths between any two points while avoiding a collection of n discs. In the real RAM model, it is an exercise to reduce this problem to Dijkstra's algorithm on a suitable graph. Of course, an effective implementation needs additional techniques such as the ability to cull away most of the irrelevant discs for any particular query, but that is another story. The implementation uses machine precision arithmetic and I casually suggested that to produce guaranteed results, they might look into tools like `LEDA` or `Core Library`. But upon reflection, I was greatly surprised to discover that I did not know how to solve it. That is because the shortest disc-avoiding path γ between two points consists of an alternating sequence of straightline segments (σ_i) and circular arcs (α_j):

$$\gamma = (\sigma_0, \alpha_1, \sigma_1, \alpha_2, \ldots, \alpha_m, \sigma_m).$$

This is illustrated by Figure 1 which shows two disc obstacles A, B, and two possible shortest paths from

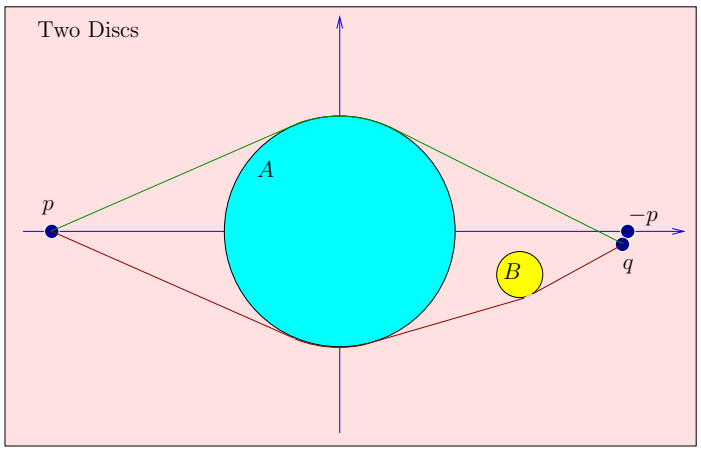

Figure 1: Shortest path from p to q

p to q. Note that q is close to $-p$, so it is not obvious which is shorter. The length of σ_i is algebraic but the length of α_j is non-algebraic. So the length of γ is an algebraic number plus a transcendental value. Dijkstra's algorithm requires the comparison of two such lengths, and there were no known decision methods here. Eventually, we were able to show the decidability of such comparisons [15] by appealing to Lindemann's theorem in transcendental number theory. Obtaining complexity bounds requires more work, depending on Baker's theory of linear form in logarithms. Although our story ended well, the initial fear that we might have unwittingly invoked an uncomputable form of the real RAM is a lesson not easily forgotten. For our next case, we turn to a problem where the computability remains open.

¶11. CASE 3: Discrete Morse Theory or, How to take the first step. A powerful research methodology in algorithms is to develop discrete analogues of continuous theories. In recent years, discrete forms of differential geometry, minimal surfaces, Ricci flows, etc. have been developed. Edelsbrunner, Harer, and Zomorodian [20] developed a discrete Morse theory for triangulated surfaces. Given a triangulated surface with a Morse function, we can compute its discrete Morse complex (which is a quadrangulation) in a purely combinatorial way. They further used discrete Morse theory to compute a simplification hierarchy that has many useful applications.

In some applications, we do not begin with a triangulation, but with a smooth surface S with associated height function. Suppose that we wish to compute its Morse complex. One approach is to first compute a triangulation T of S, then compute a discrete Morse complex using the algorithm of Edelsbrunner et al. Let us write $T \simeq S$ if the discrete Morse complex of T is combinatorially equivalent to the usual Morse complex of S. Unfortunately, we do not know how to compute a T such that $T \simeq S$. Another way to see this difficulty is to ask the simpler problem: given a non-degenerate saddle point, how do we connect it to its two maximas? The issue is to compute the integral lines correctly. No current (numerical) methods can guarantee this. Assuming S is algebraic, we see that the critical points are algebraic and can be located exactly. But the integral lines are probably nonalgebraic, and we have no a priori bounds on how close they can get to other critical points.

In general, the problem is to compute a discrete analogue T of a continuum S such that "$T \simeq S$", meaning that the topological invariants of T are equal to the corresponding invariants of S. Once we have T, the computability of its topological invariants is usually not in question. Computer scientists have gravitated naturally to this discrete computation, but I suggest that we also look at the more fundamental question of computing the transformation $S \mapsto T$ which is largely open.

This *first step*, the transition from continuous-to-discrete, amounts to solving an **explicitization problem** in the sense of ¶6. Surface meshing and computing the Morse complex are two examples. Such examples are easily multiplied: computing discrete representations of vector fields, the numerical solution of partial differential equations, etc. An interesting problem investigated by Nishida and Sugihara is the Voronoi diagram of points in a flow field [34].

¶12. CASE 4: Numerical Halting Problem or, How to be Adaptive. Behind each explicitization problem, you will find the zero problem, which I will now explain. For any set E of real expressions, we define a corresponding **zero problem**, $Zero(E)$: given $e \in E$, is the value of e equal to 0? Here, each $e \in E$ is an expression defined over some set Ω of partial functions on $\mathbb{R}$, and e either denotes a unique value $val(e) \in \mathbb{R}$, or $val(e)$ is undefined. Except in the case of algebraic expressions, the decidability of these zero problems is generally open.

Consider the "sum of square-roots" problem. This is the zero problem for the set E_0 comprising the expressions $e = \sum_{i=1}^{m} a_i\sqrt{b_i}$ where $a_i \in \mathbb{Z}$ and $b_i \in \mathbb{N}$. Its complexity is a famous problem in computational geometry (see Blömer [2, 3]). In this case study, I will illustrate the influence of abstract models in addressing the zero problem. If your abstract model is the real RAM with the ability to extract square roots, then $Zero(E_0)$ is trivial: explicitly evaluate the expression $e = \sum_{i=1}^{m} a_i\sqrt{b_i}$ in $3m - 1$ steps and perform the needed comparison to 0 in one more step. But the real RAM is unrealistic when discussing square-roots. So I turn to two other approaches, an algebraic one and an numerical one.

(1) Suppose you use the standard RAM that allows ring operations on arbitrary integers. To solve $Zero(E_0)$, you can use a well-known algebraic method known as "repeated squaring": in each stage of this process, if you arrange so that one side has exactly those terms involving a given square root, then you can eliminate this square root by squaring both sides of the two-sided equation. Unless you are extremely lucky, you will need to perform m such stages. After the first $\log_2 m$ stages, we expect to see terms which are products of any subset of the original square roots (there are 2^m such terms). So the complexity is at least single exponential in m.

(2) Suppose you use some numerical model (the next section will provide one such) that supports approximations of square roots to any desired precision. For simplicity, assume that approximations are given by enclosing intervals. We can compute a potentially infinite sequence $I_i = [u_i, v_i]$ $(i = 0, 1, 2, \ldots)$ of improving approximations to $val(e)$, where $v_i - u_i \leq 2^{-i}$. If $u_i > 0$ or $v_i < 0$ for any i, we can stop and conclude that $val(e) \neq 0$. Thus, if $|val(e)| \neq 0$, this will stop within $1 - \log_2 \min\{1, |val(e)|\}$ steps. What if $val(e) = 0$? In general, we have no method of stopping. But for $e \in E_0$, we can compute an a priori **zero bound** $\beta(e) \geq 0$ with the property that if $val(e) \neq 0$ then $|val(e)| > \beta(e)$. In this case, if you have not concluded that $val(e) \neq 0$ after $-\log_2 \min\{1, \beta(e)\}$ steps, you can declare $val(e) = 0$. Known bounds for $\beta(e)$ imply that after at most an exponential number of steps, we can declare that $val(e) = 0$. It is not known if this is necessary.

This numerical approach gives rise to the **numerical halting problem**, the problem deciding when to stop computing a potentially infinite sequence I_i $(i = 0, 1, 2, \ldots)$ of approximations. Like the classic halting problem for Turing machines, this decision problem is asymmetrical: one case is easy, and the other is hard.

If $val(e) \neq 0$, it is trivial to halt. If $val(e) = 0$, then it is highly non-trivial to halt.

Which method should we prefer? The algebraic method is non-adaptive (all-or-nothing) because, informally, with a measure-zero exception, it requires the worst case complexity bound. The numerical method is adaptive because, again with a measure-zero exception, its complexity depends on $|val(e)|$. Some years ago, I noted two other advantages of the numerical method:
(a) Typically, the zero problem is only a subproblem of the more general **sign problem**: given $e \in E$, we want to know the sign of $val(e)$, assuming $val(e)$ is defined. The algebraic method of repeated squaring requires nontrivial modifications in order to decide sign (there are numerous cases to consider) but signs come for free with the numerical method.
(b) Suppose you need to perform $n \log n$ comparisons of the form $e_i : e_j$ where $1 \leq i < j \leq n$. This problem arises in Fortune's sweepline algorithm. This reduces to the sign problem for the expression $e_i - e_j$. Using the algebraic approach, one must do repeated squaring for each comparison. Using the numerical approach, we have a better option. Assume we have a bound B such that $B \leq \beta(e_i - e_j)$ for all $i < j$. Then you just approximate each expression e_i by some numerical value $\widetilde{e}_i$ with error less than $B/2$. It turns out that B is not too large so that the difficulty of this approximation is comparable to performing a repeated squaring comparison. Now the comparison $e_i : e_j$ is easily decided by comparing the approximations $\widetilde{e}_i : \widetilde{e}_j$ (declare $val(e_i) = val(e_j)$ if $|\widetilde{e}_i - \widetilde{e}_j| < B$). So the algebraic approach requires $n \log n$ difficult computations, but the numerical method only requires n difficult computations (to approximate each $\widetilde{e}_i$).

5 The Exact Numerical Model

Numbers are the fountainhead of analytic geometry. But we are trained to design algorithms exclusively in abstract models that are devoid of numbers. This is the source of the implementation gaps we saw in our case studies. The goal in this section is to develop a numerical computational model that avoids such pitfalls while remaining useful for geometric algorithms.

Smale has observed that numerical analysis has no abstract computational models to investigate the fundamental properties of numerical computation. The BSS model has been offered as a candidate for this purpose [4] (see [5, Chap. 1]). For error analysis, numerical analysts use the standard arithmetic model (see below) which falls short of a full-scale computational model. Perhaps numerical analysts see no need for a general model because most of their problems do not involve geometry or complex combinatorial structures. In the following exercise, I hope you will see some merit in taking up Smale's challenge.

¶13. Duality in Numbers. Numbers in $\mathbb{R}$ are dual citizens: they belong to an algebraic structure (a field), as well as to an analytic structure (a metric space). As in the particle-wave duality of light, numbers seem to vacillate between its particle-like (algebraic/discrete) and wave-like (analytic/continuous) natures. In many computations, we treat them exclusively as citizens of one or the other kingdom. But in order to address the central problems of continuous computation, we need a representation of numbers which expresses the dual nature of numbers.

Consider a concrete example: the number $\alpha = \sqrt{15 - \sqrt{224}}$ can be represented directly by the indicated radical expression. This is exact, but for the purposes of locating its proximity on the number line (e.g., is α in the range $[0.01, 0.02]$?), this representation alone is unsatisfactory. An approximation such as $\alpha = 0.0223$ would be useful for proximity queries. But no single approximation is universally adequate. If necessary, we should be able to improve the approximation to $\alpha = 0.02230498$, and so on. Thus, the analytic nature of α is captured by the *potential* to give arbitrarily good approximations for the locus of α. This is only a potential because we cannot reach its limit in finite time. In the analytic approach to real computation, this is the central concept [53, 27]. For us, this potential exists because we maintain an exact representation of α. Thus, we need a **dual representation** of α, comprising the exact expression plus a dynamic approximation process. Computationally, this is very interesting because iteration at run-time becomes necessary.

Of course, we have seen dual representations earlier in `Leda Real`. This representation can be generalized to geometric objects. For instance, to represent an algebraic curve C in $\mathbb{R}^3$ exactly, we store a pair of polynomials (f, g) in case the curve is defined by $f = g = 0$. To approximate the locus of C, we use any suitable explicitization. A simple solution is a polygonal line P that is ε-close to C (for any desired ε).

¶14. Some Virtues of Numerics. Let us next focus on numerics. I shall speak of "numerics" when I want to view numbers only as analytical objects, and ignoring their algebraic nature. In numerical analysis, they are fixed-precision floating point numbers, but for exact computation, we must transpose them to BigFloats. **BigFloats** (or dyadic numbers) form the set $\mathbb{F} := \{m2^n : m, n \in \mathbb{Z}\} = \mathbb{Z}\left[\frac{1}{2}\right]$. Practitioners instinctively know the virtues of numerics but it is easy for theoreticians to miss them. Implicit in our discussion of virtues is a comparison with numerical computing that is based on other number systems with more algebraic properties, for example $\mathbb{Q}$ or algebraic numbers. I do not claim to say anything new, but it is useful to collect these thoughts in one place. What is perhaps new is the audience, since I am talking numerics in the context of exact computation.

- Numerics is useful in exact computation. More precisely, approximations can often lead to the correct decisions, and when combined with zero bounds, such approximations will eventually lead to the right decisions. This is the *sine qua non* for exact computation.
- Numerics is relatively easy to implement. There is only one number type, the "real" numbers (which in computing is translated into floating point numbers). If you compute with algebraic numbers, the traditional approaches require data structures for manipulating polynomials and algorithms for manipulating polynomials. Most implementers avoid this if they could.

  Even the use of rational numbers $\mathbb{Q}$ for their analytical properties will introduce irrelevant algebraic properties that are expensive to maintain. Trefethen gave a striking example of this from Newton iteration [51]. There is a canonical reduced representation for elements of $\mathbb{F}$ and $\mathbb{Q}$: the numerator and denominator must be relatively prime. While performing a sequence of ring operations on a number, it is necessary to reduce its representation periodically in order to avoid exponential growth. This is computationally easy for $\mathbb{F}$, but not for $\mathbb{Q}$.
- Numerics are efficient. We know this for machine numerics, but even BigFloats are efficient. Essentially, BigFloats are as efficient as BigIntegers, and we regard the complexity of BigInteger arithmetic as the base line for exact computation.
- Numerics are easy to understand. This is an important consideration for implementations. For the most part, the analytical properties we need are the metric properties and total ordering of real numbers. In contrast, algebraic properties of numbers can be highly nontrivial (try simplifying nested radical expressions).
- Numerical computation has adaptive complexity. We saw this in CASE 4. There are applications in which the only possibility of obtaining any solution at all relies on adaptivity.
- Numerical approaches have wider applicability. Many problems of CS&E have no closed form solutions. In such situations, numerical solutions remain viable. But even when closed form solutions exist, the numerical solution might be preferred.
- In CS&E, the numerics may be an essential part of the solution. Such is the case with explicitization problems (¶6). An answer of the form "$\alpha \approx 0.022$" might be acceptable, but the form "$\alpha = \sqrt{15 - \sqrt{224}}$" is unacceptable (even though it is exact). This is a blind spot if you exclusively think in algebraic computational models.

¶15. Standard Numerical Model. Having established a place for numerics in exact computation, I will now discuss how we can incorporate it into our computational model. The numerical analysts are experts in this domain, so we first look at their treatment of numerics. The **standard arithmetic model** [26, p. 44] of numerical analysis is the following: if $\circ \in \{+, -, \times, \div\}$ is any arithmetic operation and x, y are floating point numbers, then the corresponding machine operation $\widetilde{\circ}$ satisfies the following property:

$$x \widetilde{\circ} y = (x \circ y)(1 \pm \mathbf{u})$$

where[10] $\mathbf{u}$ is the unit round-off error, provided $x \circ y \neq 0$. In the usual understanding, $\mathbf{u}$ is fixed. If we allow $\mathbf{u}$ to vary, we essentially obtain the multiprecision arithmetic model of Brent [9, p. 242-3].

[10] In our error notation, any appearance of "$\pm$" should be replaced by the sequence "$+\theta$" for some variable θ satisfying $-1 \leq \theta \leq 1$. E.g., $1 \pm \mathbf{u}$ translates to $1 + \theta\mathbf{u}$. Note that θ is an implicit variable.

More generally, I assume that for each operation $\widetilde{\circ}$, an arbitrary non-zero relative error $\mathbf{u}$ can be explicitly given as an argument. Notice that the numerical analysts' model is only about arithmetic. It is agnostic about the nature of the base model. But to convert it into an abstract computational model, I choose pointer machines as the base model. We thereby obtain the **standard numerical model**. This can be classified as an analytic model.

¶16. Computational Ring. The standard numerical model is wonderful for developing the algorithms of numerical analysis, and especially for performing backwards error analysis. But this model is problematic for exact numerical computation (ENC) — it lacks the critical ability to decide zero. You can never be sure that any computed quantity is exactly zero. Zero as an algebraic object has been abolished. We have noted [56] that computing in the continuum puts a big "stress" on our computational models because we are trying to simulate an uncountable set $\mathbb{R}$ using only a countable domain ($\mathbb{N}$ or finite strings). The algebraic models [5] cope by making the zero problem trivial. The analytic models [53, 27] cope by making the zero problem undecidable. The standard numerical model represents a third solution, by making the zero problem meaningless.

To restore the place of zero, we must view BigFloats as an algebraic structure. In fact, it is useful to generalize BigFloats by an axiomatic treatment: let $D \subseteq \mathbb{R}$ be a countable set that is a ring extension of $\mathbb{Z}$, and which is closed under division by 2. Further, there is a representation[11] for D, viz., an onto partial function $\rho : \{0,1\}^* \dashrightarrow D$ relative to which there are algorithms to perform the ring operations, division by 2, and exact comparisons in D (see [56]). Call D a **computational ring**. We note that D is dense in $\mathbb{R}$, and we have mandated a minimal amount of algebraic properties in D. Computational rings provide our answer to the standard arithmetic model.

The smallest computational ring is the set of BigFloats $\mathbb{F}$. In practice, an important computational ring is $\mathbb{Z}[\frac{1}{2}, \frac{1}{5}]$ (see [35]). But $\mathbb{Q}$ or real algebraic numbers are also examples. We now construct an abstract pointer model in which elements of D are directly represented and the operations on D are available. The fundamental objects manipulated by our pointer machines are **numerical graphs**, i.e., tagged graphs in which each node stores an element of D. This constitutes our **basic numerical model**. Numerical graphs can directly represent $n \times n$ matrices $D^{n \times n}$, or polynomials with coefficients in D, etc. Under our classification scheme ¶8, this is both an analytic and an algebraic model.

Trefethen observed [52, Appendix] that numerical analysis has an undeserved reputation of being "the study of rounding errors", when its true subject matter is "the study of algorithms for continuous mathematics". I think this reputation is partly a function of the standard numerical model. What I found interesting [56] is that numerical analysts inevitably design algorithms in some exact algebraic model (check any numerical analysis text book). But they go on to address the implementation gap (¶9) between the exact model and the standard numerical model. This is the error analysis.

¶17. Exact Numerical Machines. We could design algorithms directly in the basic numerical model, but that would be programming in assembly language. So we explore some extensions of the basic numerical model. My goal is to introduce the capabilities needed to implement the algorithm of Plantinga-Vegter naturally. These capabilities will not affect computability, though they might affect complexity.

Functions will be the key abstraction for our model. Here, we see a major difference between algebraic and analytic thinking. In algebraic thinking, functions are seen as holistic objects within an algebraic structure, e.g., polynomials as elements of a ring. But in analytic thinking, functions are more versatile: they are objects which we can evaluate (query) at run-time, compute approximations of, compose with other functions, numerically differentiate, etc. In analytic complexity theory, functions viewed in this way are modeled by oracles [27].

In the following definitions, let $f : \mathbb{R}^n \to \mathbb{R}$ be a real function.

- We say f is **sign computable** if the function $\mathtt{sign}(f) : D^n \to \{-1, 0, 1\}$ where $\mathtt{sign}(f)(\mathbf{x}) = \mathtt{sign}(f(\mathbf{x}))$ is computable by a basic numerical machine.

[11] Here, $\{0,1\}^*$ is the set of binary strings. As ρ is a partial function, $\rho(w)$ may be undefined for some $w \in \{0,1\}^*$. If $\rho(w)$ is defined, then w is a "name" for the element $\rho(w) \in D$. Since ρ is onto, each element in D has at least one name. Our algorithms on D must directly operates names. See Weihrauch [53] for the theory of representations.

- Consider the approximations of functions. Any function of the form $\widetilde{f} : D^n \times \mathbb{N} \to D$ where $\widetilde{f}(\mathbf{x}, p) = f(\mathbf{x}) \pm 2^{-p}$ is called an **absolute approximation** of f. We say f is **absolutely approximable** if there is a basic numerical machine that computes such an $\widetilde{f}$.

- We need interval functions: let $\mathbb{I}(D)$ denote the set of intervals with endpoints in D. For $n \geq 1$, let $\mathbb{I}^n(D)$ denote the n-fold Cartesian product of $\mathbb{I}(D)$. Each $B \in \mathbb{I}^n(D)$ is called an n-box.

- We say $\square f : \mathbb{I}^n(D) \to \mathbb{I}(D)$ is a **box function** for f if it is an inclusion function (i.e., $f(B) \subseteq \square f(B)$) and whenever $\{B_i : i \in \mathbb{N}\}$ is a strictly monotone sequence of n-boxes with B_i properly containing B_{i+1}, and $\cap_i B_i$ is a point p, then $\cap_i \square f(B_i) = f(p)$. We say that f is **box computable** if there is a basic numerical machine that computes such an $\square f$. It is easy to see that box computable functions are (1) continuous and (2) absolutely approximable.

- We say f belongs to the class $\square C^k$ ($k \geq 0$) if each partial derivative of f up to order k exists and is box computable. Thus $\square C^0$ are just the box computable functions. Following [13], we say f is in the **class** PV if $f \in \square C^1$ and f is sign computable.

The above notions of computability are all relative to the basic numerical model. This avoids issues of computability (cf. CASES 2 and 3). There are deeper issues which we do not take up, such as the dependence of these notions on D. Our goal is to incorporate such functions as $\texttt{sign}(f)$, $\square f$, and $\widetilde{f}$ as first class programming objects in our model. Recall that the basic numerical model operates on numerical graphs. A function whose input and output are numerical graphs is called a **semi-numerical function**.

Our **exact numerical model** (ENM) extends the basic numerical model by having extra-logical objects that are semi-numerical functions, and whose tagged graphs have nodes that can store either a semi-numerical function or an element of D. We have a built-in predicate to test nodes for the type of its stored value (the type is either D or a semi-numerical function). Suppose $u, v, w \in \Delta^*$ and G is a tagged graph (¶7). If a semi-numerical function F is stored in node $[u]_G$ and its argument is accessed through node $[v]_G$, then we can invoke an evaluation of F on this argument by executing the following instruction:

$$w \leftarrow EVALUATE(u, v)$$

See [56] for similar details. If we like, we could provide functors to construct semi-numerical functions from scratch, functors to compose two semi-numerical functions, etc. But for our simple needs here, we may assume the semi-numerical functions are simply available (passed as arguments to our numerical machines, like oracles).

¶18. From Smooth Surfaces to Singular Ones The preceding development was a build-up to state the following result:

THEOREM 1 (Plantinga-Vegter). *There is an ENM algorithm which, given $\varepsilon > 0$, a box function and sign function for some function $f : \mathbb{R}^3 \to \mathbb{R}$, and $B_0 \in \mathbb{I}^3(D)$, will compute an isotopic ε-close mesh for the surface $S : f = 0$ provided S is non-singular and $S \subseteq B_0$.*

The statement of this theorem does not reveal the beauty and naturalness of the PV algorithm; to see this, refer to their original paper [36]. It suffices to say that their method uses standard subdivision of the box B_0 and cleverly exploits isotopy. This result can be extended in several ways: First, the surface S need not be confined within the box B_0, but we must slightly relax the correctness statement on the boundary of B_0. Box B_0 can be replaced by more complicated regions which need not be connected or simply-connected. The function f is allowed to have singularities outside B_0. See [13, 28] for these extensions in the plane. Extensions of the PV algorithm to higher than 3 dimensions are currently unknown, but recently Galehouse [24] introduced a new approach that is applicable in every dimension. All these extensions stayed within the ENM framework.

What if the surface S is singular? We can use the PV algorithm as a subroutine to locate and determine the singularities. This was done for the planar case in [13]. Let me sketch the basic idea: say $S : f = 0$ is a curve with only isolated singularities (if $f(X, Y)$ is a square free polynomial, this will be the case). Now apply the PV algorithm to $F = f^2 + f_x^2 + f_y^2 - \delta$ where $\delta > 0$. For sufficiently small δ, the curve $S_\delta : F = 0$

will be a nonsingular curve, i.e., a collection of ovals or infinite curves. Moreover, if the oval is sufficiently small, we know that it isolates a singularity. Once we have isolated singularities in sufficiently small boxes, we can run the PV algorithm on the original curve S but on a region that excludes these small boxes. We can determine the degree of each singularity (i.e., how many open arcs of S have endpoints in the singularity) by considering an annulus around its small box. All these can be carried out in the algebraic case, because we have computable zero bounds.

¶19. Simple Real Root Isolation, or How to avoid zero. The PV algorithm makes the fairly strong assumption that f is sign computable. For instance, we do not know whether this property holds for the class of hypergeometric functions (with rational parameters). But such hypergeometric functions can be shown to be box computable (cf. [19]). So it is desirable remove the sign computability condition all together.

I will now show this for the 1-D case (it will be the only technical result of this paper). In 1-D, meshing amounts to real root isolation and root refinement for a function $f : \mathbb{R} \to \mathbb{R}$. It is not hard to devise such a root isolation algorithm, which we call EVAL (see [14], but the original algorithm is from Mitchell [33]). EVAL depends on two interval predicates which we call C_0 and C_1:

$$\begin{aligned} C_0(I): &\quad 0 \notin \square f(I), \\ C_1(I): &\quad 0 \notin \square f'(I). \end{aligned}$$

Clearly, these predicates are computable if $f \in PV$. Given a BigFloat interval I_0 in which f has only simple roots, EVAL will return a set of isolating intervals for each of the roots of f in I_0. We use a queue Q for processing the intervals:

```
EVAL(I_0):
    Q ← {I_0}
    While Q is non-empty
        Remove I from Q
        If C_0(I) holds, discard I
        Else if C_1(I) holds,
(*)         If f has different non-zero signs at end points of I, output I
(*)         Else, discard I
        Else
(**)        If f(m) = 0, output [m, m] where m is the midpoint of I
(**)        Split I into I', I'' at m, and put both intervals into Q
```

Termination and correctness are easy to see. We now modify EVAL so that f does not have to be sign computable. There is a small price to pay, as there will be some indeterminacy at the boundary of the input interval.

LEMMA 2. *There is an ENM algorithm which, given $\varepsilon > 0$, a box function $\square f$ for $f : \mathbb{R} \to \mathbb{R}$, and $[a, b] \in \mathbb{I}(D)$, will isolate all the real roots of f in some interval J where*

$$[a, b] \subseteq J \leq [a - \varepsilon, b + \varepsilon]$$

provided f has only simple roots in $[a, b]$. Moreover, there is at most one output isolating interval that overlaps $[a - \varepsilon, a]$ and at most one that overlaps $[b, b + \varepsilon]$.

Proof. Observe that from the box function $\square f$, we can easily construct an absolute approximation function $\widetilde{f}$ for f. Thus, for each $x \in D$ and $p \in \mathbb{N}$, we have $\widetilde{f}(x, p) = f(x) \pm 2^{-p}$. If $|\widetilde{f}(x, p)| > 2^{-p}$, then we know the sign of f at x.

We modify the EVAL algorithm by omitting two lines marked (**) as we can no longer compute the sign of $f(m)$. We also replace the two lines marked (*) by the following subroutine: assume $I = [a, b]$ is the input to our subroutine. So $C_1(I)$ holds, and I has at most one root of f. The following subroutine will either decide that I has no root or some $J \subseteq [a - \varepsilon, b + \varepsilon]$ is an isolating interval:

1. We dovetail the absolute approximations of $f(a)$ and $f(b)$
   with increasing precision until we see a non-zero sign of $f(a)$ or of $f(b)$.
   ◁ *This must halt because $C_1(I)$ holds.*
2. Wlog, say we see the sign of $f(a)$, and it is negative.
   If $f'(I) < 0$, then there are no roots in I. So assume $f'(I) > 0$.
3. For $i = 0, 1, 2, \ldots$, we check the predicate $C_1(J_i)$ where $J_i = [b, b + \varepsilon 2^{-i}]$.
   Halt at the first $k \geq 0$ where $C_1(J_k)$ holds.
   ◁ *This must halt because $f'(b) > 0$.*
4. This means $f'(b) > 0$ and $f'(b + \varepsilon 2^{-k}) > 0$.
   As before, do dovetailing to determine the sign of either $f(b)$ or $f(b + \varepsilon 2^{-k})$.
5. If we know the sign of $f(b)$, then I contains a root iff $f(a)f(b) < 0$.
   The other case of knowing the sign of $f(b + \varepsilon 2^{-k})$ is similar.

One final detail: the isolating intervals which this modified algorithm outputs might be overlapping. To clean up the intervals so that there is no ambiguity, observe that $C_1(I)$ holds at each output interval I. Therefore, if two outputs I and J overlap, we see that $I \cup J$ has a unique root which is found in $I \cap J$. So we may replace I, J by $I \cap J$. **Q.E.D.**

We should be able to extend the PV algorithm in 2- and 3-D by a similar relaxation of the conditions on f. So what have we learned from this? It is (not surprisingly) that you can avoid the zero problem if there are no singularities. So you could have developed this algorithm in the standard numerical model. But should you have singularities (multiple roots in the 1-D case) this option is not available.

¶20. Towards Numerical Computational Geometry. Our exact numerical model satisfies the need for higher level abstractions in designing algorithms. Such algorithms will have adaptive complexity because of the use of numerics. Iterations is completely natural. As we saw in the PV Algorithm, domain subdivision will be useful in such algorithms. Another feature is that, unlike standard numerical algorithms, we can actively control the precision of individual operations. This can lead to a speedup [45, 44]. Another direction is in producing numerical algorithms that are "complete", i.e., do not have exceptional inputs for which the algorithm fails. E.g., see [55, 13, 16]. Currently, most geometric algorithms based on numerical primitives are "incomplete" because they are based on the standard numerical model.

Such algorithms represent a marked departure from the typical algorithms seen in computational geometry, and suggests the name "numerical computational geometry" for such activities. In fact, other researchers in interval computation are also producing similar kinds of algorithms. See particularly the work of Ratschek and Rokne [38, 39]. I think both lines of work may eventually converge, but the main gap between their view and ours is located in the difference between using the standard numerical model and our exact numerical model (cf. [40] and [28])

¶21. What about Complexity Theory? The most serious challenge for numerical computational geometry is the development of a complexity analysis of adaptive and iterative algorithms. Of course, the lack of analysis does not hamper the usefulness of such algorithms, but it discourages theoreticians from looking at this class of algorithms. Previous work on adaptive complexity analysis has stemmed from analysis of simplex algorithms in the 1980's [8]. All such analyses have depended on probabilistic assumptions. The acclaimed smoothed analysis of Spielman and Teng [48] tries to minimize such objections by "localizing" the probabilistic assumptions to each input instance.

Recently, we introduced the concept of **continuous amortization** [14]. This yields an analysis of adaptive complexity *without probabilistic assumptions*. The key idea is to bound the subdivision tree size in terms of an integral. If the input domain is a box B, the number of subdivisions can be bounded by an integral of the form $I = \int_{\mathbf{x} \in B} \phi(\mathbf{x}) d\mathbf{x}$. Amortization is a well-known computational paradigm and analysis technique in discrete algorithms [18]. We can view the integral approach as a "continuous" form of amortization. We applied this analysis to the EVAL algorithm ¶19, proving that the tree size is polynomial in the worst case depth. This is a mark of adaptivity since in the worst case, tree size is exponential in depth. We believe similar analyses are applicable to other subdivision algorithms.

6 Conclusion

This essay began with the accomplishments of Kurt in experimental computational geometry, and the significance of LEDA in the history of computing. I extracted from this work a unique mode of computation (exact numerical computation), and extrapolated it to general computing, and to computational geometry in particular. My motivation is to equip ourselves to address the host of interesting continuum problems in CS&E. But none of us plan to turn into applied mathematicians or numerical analysts to address these problems. Our strength is in exact/discrete thinking. We celebrate this, and rightly so. You probably agree with me that our discrete/exact views can bring something new to the problems of CS&E. But to do this, we need an analytic model of computation in which the exact views are captured. The clue lies in the zero problem, but more generally the "explicitization problems" of continuous-to-discrete computation. I described an exact numerical model that has many of the desired properties. This article (it turned out) spent much time discussing computational models because, as our case studies show, the wrong model can lead us astray. As a computer scientist, I have found extreme satisfaction in designing geometric algorithms in the exact numerical model. Some of these algorithms also seem quite practical. Perhaps you will find the same satisfaction.

Acknowledgments

I am grateful for the warm hospitality of Professor Subir Ghosh at Tata Institute of Fundamental Research, Mumbai and Professor Subhash Nandy at the Indian Statistical Institute (ISI), Kolkata. Discussions with them at WALCOM 2009 have provoked many of the thoughts expressed in this paper, and it was on the campus of ISI in which the term "numerical computational geometry" was first suggested. This paper has greatly benefited from insightful comments from Helmut Alt, David Bindel, Michael Burr, Richard Cole, Ker-I Ko, and Lloyd Trefethen.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachusetts, 1974.

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

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

[4] L. Blum, F. Cucker, M. Shub, and S. Smale. Complexity and real computation: A manifesto. *Int. J. of Bifurcation and Chaos*, 6(1):3–26, 1996.

[5] L. Blum, F. Cucker, M. Shub, and S. Smale. *Complexity and Real Computation*. Springer-Verlag, New York, 1998.

[6] J.-D. Boissonnat, D. Cohen-Steiner, B. Mourrain, G. Rote, and G. Vegter. Meshing of surfaces. In J.-D. Boissonnat and M. Teillaud, editors, *Effective Computational Geometry for Curves and Surfaces*. Springer, 2006. Chapter 5.

[7] J. Bokowski and B. Sturmfels. *Computational Synthetic Geometry*, volume 1355 of *Lecture Notes in Mathematics*. Springer, 1989.

[8] K. H. Borgwardt. Probabilistic analysis of the simplex method. In J. Lagarias and M. Todds, editors, *Mathematical Developments Arising from Linear Programmng*, volume 114, pages 21–34. AMS, 1990. This volume also has papers by Karmarkar, Megiddo.

[9] R. P. Brent. Fast multiple-precision evaluation of elementary functions. *J. of the ACM*, 23:242–251, 1976.

[10] B. Buchberger, G. E. Collins, and R. Loos, editors. *Computer Algebra*. Springer-Verlag, Berlin, 2nd edition, 1983.

[11] C. Burnikel, S. Funke, K. Mehlhorn, S. Schirra, and S. Schmitt. A separation bound for real algebraic expressions. In *9th ESA*, volume 2161 of *Lecture Notes in Computer Science*, pages 254–265. Springer, 2001. To appear, Algorithmica.

[12] C. Burnikel, S. Funke, and M. Seel. Exact geometric computation using cascading. *Int'l. J. Comput. Geometry and Appl.*, 11(3):245–266, 2001. Special Issue.

[13] M. Burr, S. Choi, B. Galehouse, and C. Yap. Complete subdivision algorithms, II: Isotopic meshing of singular algebraic curves. In *Proc. Int'l Symp. Symbolic and Algebraic Computation (ISSAC'08)*, pages 87–94, 2008. Hagenberg, Austria. Jul 20-23, 2008.

[14] M. Burr, F. Krahmer, and C. Yap. Integral analysis of evaluation-based real root isolation, Feb. 2009. Submitted, 2009.

[15] E.-C. Chang, S. W. Choi, D. Kwon, H. Park, and C. Yap. Shortest paths for disc obstacles is computable. *Int'l. J. Comput. Geometry and Appl.*, 16(5-6):567–590, 2006. Special Issue of IJCGA on Geometric Constraints. (Eds. X.S. Gao and D. Michelucci). Also: Proc.21st SoCG, 2005, pp.116–125.

[16] J.-S. Cheng, X.-S. Gao, and C. K. Yap. Complete numerical isolation of real zeros in general triangular systems. In *Proc. Int'l Symp. Symbolic and Algebraic Comp. (ISSAC'07)*, pages 92–99, 2007. Waterloo, Canada, Jul 29-Aug 1, 2007. DOI: http://doi.acm.org/10.1145/1277548.1277562. In press, Journal of Symbolic Computation.

[17] H. Cohen. *A Course in Computational Algebraic Number Theory*. Springer, 1993.

[18] T. H. Corman, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, Cambridge, Massachusetts and New York, second edition, 2001.

[19] Z. Du and C. Yap. Uniform complexity of approximating hypergeometric functions with absolute error. In S. Pae and H. Park, editors, *Proc. 7th Asian Symp. on Computer Math. (ASCM 2005)*, pages 246–249, 2006.

[20] H. Edelsbrunner, J. Harer, and A. Zomorodian. Hierarchical Morse complexes for piecewise linear 2-manifolds. *Discrete and Computational Geometry*, 30(1):87 – 107, 2003.

[21] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schoenherr. The CGAL kernel: a basis for geometric computation. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, pages 191–202, Berlin, 1996. Springer. Lecture Notes in Computer Science No. 1148; Proc. 1st ACM Workshop on Applied Computational Geometry (WACG), Federated Computing Research Conference 1996, Philadelphia, USA.

[22] S. Fortune. Editorial: Special issue on implementation of geometric algorithms. *Algorithmica*, 27(1):1–4, 2000.

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

[24] B. Galehouse. *Topologically Accurate Meshing Using Spatial Subdivision Techniques*. Ph.D. thesis, New York University, Department of Mathematics, Courant Institute, May 2009. From `http://cs.nyu.edu/exact/doc/`.

[25] G. Hardy. What is Geometry? (presidential address to the mathematical association, 1925). *The Mathematical Gazette*, 12(175):309–316, 1925.

[26] N. J. Higham. *Accuracy and stability of numerical algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, 1996.

[27] K.-I. Ko. *Complexity Theory of Real Functions*. Progress in Theoretical Computer Science. Birkhäuser, Boston, 1991.

[28] L. Lin and C. Yap. Adaptive isotopic approximation of nonsingular curves: the parametrizability and non-local isotopy approach. In *Proc. 25th ACM Symp. on Comp. Geometry*, page to appear, June 2009. Aarhus, Denmark, Jun 8-10, 2009.

[29] K. Mehlhorn. The reliable algorithmic software challenge (RASC). In *Computer Science in Perspective*, volume 2598 of *LNCS*, pages 255–263, 2003.

[30] K. Mehlhorn and S. Näher. LEDA – a library of efficient data types and algorithms. In *Lecture Notes in Computer Science*, volume 379, pages 88–106. Springer-Verlag, 1989. Proc. 14th Symp. on MFCS, to appear in CACM.

[31] K. Mehlhorn and S. Näher. LEDA: a platform for combinatorial and geometric computing. *CACM*, 38:96–102, 1995.

[32] K. Mehlhorn and S. Schirra. Exact computation with `leda_real` – theory and geometric applications. In G. Alefeld, J. Rohn, S. Rump, and T. Yamamoto, editors, *Symbolic Algebraic Methods and Verification Methods*, pages 163–172, Vienna, 2001. Springer-Verlag.

[33] D. P. Mitchell. Robust ray intersection with interval arithmetic. In *Graphics Interface'90*, pages 68–74, 1990.

[34] T. Nishida and K. Sugihara. Voronoi diagram in the flow field. In N. K. T. Ibaraki and H. Ono, editors, *Algorithms and Computation, 14th ISAAC 2003*, number 2906 in Lecture Notes in Computer Science, pages 26–35, 2003.

[35] S. Pion and C. Yap. Constructive root bound method for k-ary rational input numbers. *Theor. Computer Science*, 369(1-3):361–376, 2006.

[36] S. Plantinga and G. Vegter. Isotopic approximation of implicit curves and surfaces. In *Proc. Eurographics Symposium on Geometry Processing*, pages 245–254, New York, 2004. ACM Press.

[37] M. Pohst and H. Zassenhaus. *Algorithmic Algebraic Number Theory*. Cambridge University Press, Cambridge, 1997.

[38] H. Ratschek and J. G. R. G. Editors). Editorial: What can one learn from box-plane intersections? *Reliable Computing*, 6(1):1–8, 2000. Special Issue on Reliable Geometric Computations.

[39] H. Ratschek and J. Rokne. *Geometric Computations with Interval and New Robust Methods: With Applications in Computer Graphics, GIS and Computational Geometry*. Horwood Publishing Limited, UK, 2003.

[40] H. Ratschek and J. G. Rokne. SCCI-hybrid methods for 2d curve tracing. *Int'l J. Image Graphics*, 5(3):447–480, 2005.

[41] Research Triangle Park (RTI). Planning Report 02-3: The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology (NIST), U.S. Department of Commerce, May 2002.

[42] D. Richardson. Zero tests for constants in simple scientific computation. *Mathematics in Computer Science*, 1(1):21–38, 2007. Inaugural issue on Complexity of Continous Computation.

[43] A. Schönhage. Storage modification machines. *SIAM J. Computing*, 9:490–508, 1980.

[44] V. Sharma. Robust approximate zeros in banach space. *Mathematics in Computer Science*, 1(1):71–109, 2007. Based on his thesis.

[45] V. Sharma, Z. Du, and C. Yap. Robust approximate zeros. In G. S. Brodal and S. Leonardi, editors, *Proc. 13th European Symp. on Algorithms (ESA)*, volume 3669 of *Lecture Notes in Computer Science*, pages 874–887. Springer-Verlag, Apr. 2005. Palma de Mallorca, Spain, Oct 3-6, 2005.

[46] J. R. Shoenfield. *Mathematical Logic*. Addison-Wesley, Boston, 1967.

[47] J. M. Snyder. Interval analysis for computer graphics. *SIGGRAPH Comput.Graphics*, 26(2):121–130, 1992.

[48] D. A. Spielman and S.-H. Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *J. of the ACM*, 51(3):385–463, 2004.

[49] B. T. Stander and J. C. Hart. Guaranteeing the topology of an implicit surface polygonalization for interactive meshing. In *Proc. 24th Computer Graphics and Interactive Techniques*, pages 279–286, 1997.

[50] J. Traub, G. Wasilkowski, and H. Woźniakowski. *Information-Based Complexity*. Academic Press, Inc, 1988.

[51] L. N. Trefethen. Computing with functions instead of numbers. *Mathematics in Computer Science*, 1(1):9–19, 2007. Inaugural issue on Complexity of Continous Computation. Based on talk presented at the Brent's 60th Birthday Symposium, Weierstrass Institute, Berlin 2006.

[52] L. N. Trefethen and D. Bau. *Numerical linear algebra*. Society for Industrial and Applied Mathematics, Philadelphia, 1997.

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

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

[55] C. K. Yap. Complete subdivision algorithms, I: Intersection of Bezier curves. In *22nd ACM Symp. on Comp. Geometry*, pages 217–226, July 2006.

[56] C. K. Yap. Theory of real computation according to EGC. In P. Hertling, C. Hoffmann, W. Luther, and N.Revol, editors, *Reliable Implementation of Real Number Algorithms: Theory and Practice*, number 5045 in Lecture Notes in Computer Science, pages 193–237. Springer, 2008.

[57] C. K. Yap and T. Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, pages 452–492. World Scientific Press, Singapore, 2nd edition, 1995.

[58] C. K. Yap and J. Yu. Foundations of exact rounding. In S. Das and R. Uehara, editors, *Proc. WALCOM 2009*, volume 5431 of *Lecture Notes in Computer Science*, pages 15–31, Heidelberg, 2009. Springer-Verlag. Invited talk, 3rd Workshop on Algorithms and Computation, Kolkata, India.

[59] 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.