"Round-off errors arise because it is impossible to represent all real numbers exactly on a finite-state machine (which is what all practical digital computers are)."

— Numerical Analysis entry in Wikipedia

"The discrepancy between the true answer and the answer obtained in a practical calculation is called the truncation error*. Truncation error would persist even on a hypothetical, "perfect" computer that had an infinitely accurate representation and n roundoff error. As a general rule, there is not much that a programmer can do about roundouff error, other than to choose algorithms that do not magnify it unnecessarily (see discussion of "stability" below). Truncation error, on the other hand, is entirely under the programmer's control. In fact, it is only a slight exaggeration to say that clever minimization of truncation error is practically the entire content of the field of numerical analysis!"*

— Numerical Recipes in C: The Art of Scientific Computing (Chapter 1, p. 30)

Lecture 1
ON NUMERICAL NONROBUSTNESS

This chapter gives an initial orientation to some key issues that concern us. What is the nonrobustness phenomenon? Why does it appear so intractable? Of course, the prima facie *reason for nonrobustness is numerical errors. But this is not the full story: the key lies in understanding the underlying geometry.*

§1. What is Numerical Nonrobustness?

Nonrobustness of computer systems means different things to different people. Sometimes a computer solution is described as "nonrobust" if it is non-scalable (i.e., as the problem size gets larger, the solution becomes no longer practical) or non-extensible (i.e., when we make simple variations in the problem, the solution could not be similarly tweaked). In this book, we are only interested in nonrobustness that can be traced to errors in numerical approximations. For brevity, we simply say "(non)robustness" instead of "numerical (non)robustness".

This nonrobustness phenomenon is well-known and widespread. The fact is that most numerical quantities in computers are approximations. Hence there is **error**, which may be defined as the absolute difference $|x - \widetilde{x}|$ between the exact[1] value x and the computed value $\widetilde{x}$. Most of the time, such errors are **benign**, provided we understand that they are there and we account for them appropriately. The phenomenon that we call nonrobustness arises when benign errors leads a computation to commit **catastrophic errors**. There is no graceful recovery from such errors, and often the program crashes as a result. Instead of crashing dramatically, the program could also enter an infinite loop or silently produce an erroneous result. Note that crashing may be preferred over a silent error that may eventually lead to more serious consequences. We now illustrate how benign errors can turn catastrophic:

(A) Here is a robustness test that most current geometric libraries will fail: assume the library has primitives to intersect lines and planes in 3-dimensional Euclidean space, and predicates to test if a point lies on a plane. First, we construct a plane H and a line L. Then we form their intersection $H \cap L$ which is a point $q \in \mathbb{R}^3$. Finally, let us call the predicate to test if q lies on the plane H. The predicate ought to return "YES" meaning that q lies on H. But because of numerical errors, our predicate may return

[1] We initially assume that each quantity has a theoretically "exact" value. A more general setting allows values to be inexact but quantifiably so. We will return to this later. The nonrobustness issue is already non-trivial in the exact setting.

"NO", which is what we call a catastrophic error. Alternatively, if errors such as $|x - \widetilde{x}|$ in computing a number x are called **quantitative errors**, then catastrophic errors may be called **qualitative errors**.

For random choices of H and L, there is a high likelihood that the answer comes back as "NO". For instance, suppose H_0 is the plane $x + y + z = 1$ and $L_{i,j}$ is the line through the origin and through $(i, j, 1)$. Thus the parametric equation of the line is $L_{i,j}(t) = (it, jt, t)$. We can implement all these primitives using a straightforward implementation in IEEE arithmetic (see Exercise). For instance, H_0 will intersect $L_{1,1}$ in the point $p = \frac{1}{i+j+1}(i, j, 1)$. An experiment in [23] performed a variant of this test in 2-D for pairs of intersecting lines. With 2500 pairs of intersecting lines, this tests yields a failure rate of 37.5% when implemented in the straightforward way in IEEE arithmetic.

The related problem of "line segment intersection" is a well-known in the literature on robustness, and was first posed as a challenge by Forrest [14]. Although this challenge must have been taken up in every geometric library implementation, no universally accepted solution is known. See Knott and Jou [24] for a proposed solution.

(B) An important class of physical simulations involve tracking a "front", which is just a triangulated surface representing material boundary or the interface between two fluids such as oil and water. Such a simulation might be useful to understand how an oil spill evolves under different conditions. This front evolves over time, and a qualitative error arises when the surface becomes "tangled" (self-intersects), and thus no longer divides space into two parts.

(C) Trimmed algebraic patches are basic building blocks in nonlinear geometric modeling. Because of the high algebraic complexity, the bounding curves in such patches are usually approximated. This approximation leads to errors in queries such as whether a ray intersects a patch, resulting in topological inconsistencies and finally crashing the modeler. Visually, such errors are often seen as "cracks" in a supposedly continuous surface.

(D) In mesh generation, an important primitive operation is to classify a point as inside or outside of a cell. Typically a cell is a triangle in 2-D and a tetrahedron in 3-D. If we mis-classify a point, meshes can become inconsistent, ambiguous or deficient. Current CAD systems will routinely produce such "dirty meshes". This is a serious problem for all applications downstream.

In recent years, several research communities have been interested in numerical robustness and reliability in software: books, surveys, software, special journal issues, and special workshops have been devoted to this area. Some of this information is collected at the end of this chapter, under the Additional Notes Section. Connection between the viewpoints of different communities will be discussed. For instance, numerical analysts have know for a long time the various "pitfalls" in numerical computations, but the critical missing element in such discussions is the role of geometry as brought out by the above examples.

¶1. Book Overview. The purpose of this book is develop an approach to robust algorithms this is now dominant in the Computational Geometry community. This approach is called **Exact Geometric Computation** (EGC). It is now encoded in software such as `CGAL`, `LEDA` and `Core Library`. Most of these results are obtained within the last decade, and we feel it is timely to organize them into a book form. The book is written at an introductory level, assuming only a basic knowledge of algorithms. In brief, this is what our book cover:

- We begin by introducing and analyzing the fundamental problem of nonrobustness in geometric software.
- We briefly survey various approaches that have been proposed, including EGC.
- EGC can be seen as a new mode of numerical computation which need to be supported by critical computational techniques: filters, root bounds, root finding, etc. The necessary algebraic background will be developed.
- We will study in detail the application of EGC principles in several basic algorithms and geometric primitives. These simple examples come from linear geometry (i.e., geometry that does not go beyond basic linear algebra).

- We also consider the application of EGC principles in non-linear geometry, such as in algebraic curves and surfaces.
- We address three important topics related to EGC and robustness: data degeneracy, perturbation techniques, and geometric rounding.
- Applications of EGC in areas such as theorem proving and software certification will be illustrated.
- We touch on the subject of non-algebraic computation, which lay at the frontier of current knowledge.
- Finally we describe a new theory of real approximation which is motivated by the practical development in EGC. It points to an interesting complexity theory of real computation.

§1.1. Primitives for Points and Lines

We will now flesh out the 2-D version of example (A) from the introduction. Consider a simple geometry program which uses a library with primitives for defining points and lines in the Euclidean plane. The library also has primitives to intersect two lines, and to test if a point is on a line. Mathematically, a point p is a pair (x, y) of real numbers, and a line ℓ is given by an equation

$$\ell : ax + by + c = 0$$

where (a, b, c) are the real parameters defining ℓ. We write $\ell := \mathsf{Line}(a, b, c)$ to indicated that the line ℓ is defined by (a, b, c). Note that $\mathsf{Line}(a, b, c)$ is well-defined only if $ab \neq 0$; moreover, $\mathsf{Line}(a, b, c) = \mathsf{Line}(da, db, dc)$ for all $d \neq 0$. Thus each line can be viewed[2] as a point in a peculiar homogeneous space. Similar, $p := \mathsf{Point}(x, y)$ is a definition of the point p. If $\ell' = \mathsf{Line}(a', b', c')$ is another line, then the intersection of ℓ and ℓ' is the $\mathsf{Point}(x, y)$ that satisfies the linear system

$$\begin{bmatrix} a & b \\ a' & b' \end{bmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = -\begin{pmatrix} c \\ c' \end{pmatrix}$$

which yields

$$\begin{pmatrix} x \\ y \end{pmatrix} = \frac{1}{\Delta} \begin{bmatrix} -b' & b \\ a' & -a \end{bmatrix} \begin{pmatrix} c \\ c' \end{pmatrix} \tag{1}$$

where $\Delta = ab' - a'b$ is the determinant. In two exceptional cases, the intersection is not a point: (a) when $\ell = \ell'$ (so there are infinitely many points of intersection) or, (b) when $\ell \neq \ell'$ and $\ell \| \ell'$ (the lines are distinct and parallel, so there are no intersection points). In these cases, we assume that the intersection is an "undefined value" (denoted $\uparrow$). So we define the **operation** Intersect(ℓ, ℓ') to return the intersection point (possibly $\uparrow$) of lines ℓ and ℓ'. This operation is easily implemented in any conventional programming language using the formula (1). Similarly, it is easy to program the **predicate** OnLine(p, ℓ) which returns True iff p lies on line ℓ, otherwise returns False. Note that $p = \mathsf{Point}(x_0, y_0)$ lies on $\ell = \mathsf{Line}(a, b, c)$ iff $ax_0 + by_0 + c_0 = 0$.

> *CONVENTION: In general, geometric primitives (operations or predicates or constructors) are partial functions, capable of returning the undefined value, $\uparrow$. In particular, if any input argument is undefined, then the result is undefined.*

Consider the following expression:

$$\text{OnLine}(\text{Intersect}(\ell, \ell'), \ell) \tag{2}$$

[2] A homogeneous point (a, b, c) is defined up to multiplication by a non-zero constant, and is often written as a proportionality $(a : b : c)$. In standard homogeneous space, we exclude the point $(0 : 0 : 0)$. But here, we exclude an additional point, $(0 : 0 : c)$, $c \neq 0$.

Mathematically, the value of this expression is True unless Intersect(ℓ, ℓ') is undefined. Yet, any simple implementation of these primitives on a computer will fail this property. That is to say, for "most choices" of ℓ and ℓ', the expression will not evaluate to True. The reason for this is clear – computer arithmetic are approximate. For most inputs, the expression (2) is sensitive to the slightest error in its evaluation.

An exercise below will attempt to quantify "most choices" in the above remark.

Operations such as line intersection form the basis of more complex operations in geometry libraries. If such basic operations are nonrobust, applications built on top of the libraries will face the same nonrobustness problem but with unpredictable consequences. So it is no surprise that applications such as geometric modelers are highly susceptible to catastrophic errors.

The property of failing to return True when it should have, is a qualitative error, and this is a result of some quantitative error. Numerical errors are, by nature, a quantitative phenomenon and they are normally benign. But under certain conditions, they become a qualitative phenomenon. We need to understand when such transitions take place, and to find measures either to detect or to avoid them. This is the essence of the approach called Exact Geometric Computation.

§1.2. Epsilon Tweaking

A common response of programmers to this example is to say that the comparison "is $ax+by+c = 0$?" is too much to ask of approximate quantities. Instead, we should modify the predicate OnLine(Point(x, y), Line(a, b, c)) to the following

$$\text{"Is } |ax + bx + c| < \varepsilon\text{?"} \tag{3}$$

where $\varepsilon > 0$ is a small constant. This simple idea is general and widely used: every time we need to compare something to 0, we will compare it to some small "epsilon" constant. There could be many different epsilons in different parts of a large program. It is somewhat of an art to pick these epsilons constants, perhaps chosen to avoid errors in a battery of test cases. In the programming world, the empirical task of choosing such constants is called "epsilon tweaking". We now try to analyze what epsilon tweaking amounts to.

Note that in our expression (2) above, if we make ε large enough, the predicate will surely return True for all pairs of intersecting lines. This avoids the problem of false negatives, clearly it introduces many false positives. We should understand that the introduction of ε changes the underlying geometry – we may call this **epsilon geometry**. But what is the nature of this geometry? One interpretation is that we are now dealing with lines or points that are "fat". A point $p = \mathsf{Point}(x, y)$ is really a disk centered about (x, y) with radius $\varepsilon_1 \geq 0$. Similar, a line $\ell = \mathsf{Line}(a, b, c)$ really represents a strip S of region with the mathematical line as the axis of S. The strip has some width $\varepsilon_2 \geq 0$. For simplicity, suppose $\varepsilon_1 = \varepsilon_2 = \varepsilon/2$. Then the OnLine$(p, \ell)$ predicate is really asking if the fat point p intersects the fat line ℓ. This is illustrated in figure 1.

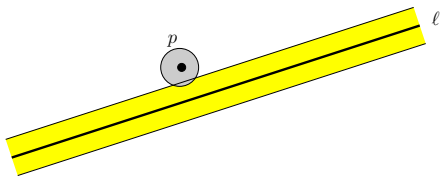


Figure 1: A fat point p intersecting a fat line ℓ.

This interpretation of equation (3) is justifiable after we normalize the equation of the line $ax+by+c = 0$ so that $a^2 + b^2 = 1$. To normalize the coefficients (a, b, c), we simply divide[3] each coefficient by $\sqrt{a^2 + b^2}$. The distance of any point (x, y) from the normalized $Line(a, b, c)$ is given by $|ax + by + c|$. Thus the epsilon test (3) amounts to checking if p is within distance ε from the line ℓ.

More important, we observe that by resorting to epsilon-tweaking, we have exchanged "plain old Euclidean geometry" (POEG) for some yet-to-be-determined geometry. In a later chapter, we will look at some attempts

[3]

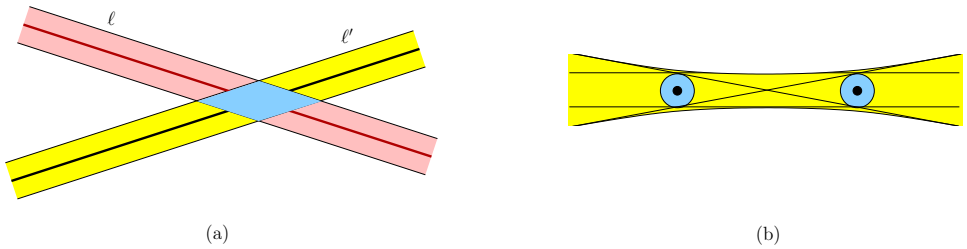


Figure 2: (a) Intersecting two fat lines, (b) Fat line through two fat points.

to capture such kinds of novel geometries. Even if a consistent interpretation can be obtained, it is hard to reason correctly with such geometries. For instance, what is the intersection of two fat lines? As we see from Figure 2(a), it is not obvious how to interpret the diamond-shaped intersection as a fat point, unless we further generalize the concept of a point to include diamond shapes. Again, we expect to be able to define a fat line that passes through two fat points. In Figure 2(b), we see that fat lines might have to be generalized to a double-wedge with non-linear boundaries.

Most of our intuitions and algorithms are guided by Euclidean geometry, and we will find all kinds of surprises in trying to design algorithms for such geometries.

¶2. Interval Geometry. There is an alternative interpretation of the test (3). Here, we assume that each point is only known up to some ball of radius ε. Then the test (3) is only useful for telling us if the point definitely does not lie on the line. But the conditions for knowing if a point is definitely on the line can only be determined with additional assumptions. We then proceed to a three-valued logic in our programs, in which "partial predicates" gives answers of yes/no/maybe. This is basically the geometric analog of the well-known idea of **interval arithmetic**. This is not unreasonable in some applications, but a common problem is that the "maybe" answers might rapidly proliferate throughout a computation and lead to no useful results. Alternatively, we might think of the "geometric intervals" (unknown region about a point or a line) growing rapidly in an iterated computation.

Exercises

Exercise 1.1: (a) Write a simple planar geometry program which handles points and lines. An object oriented programming language such as C++ would be ideal, but not essential. Provide the operations to define points and lines, to intersect two lines and to test if a point is on a line. You must provide for undefined values, and must not attempt any epsilon-tweaking.

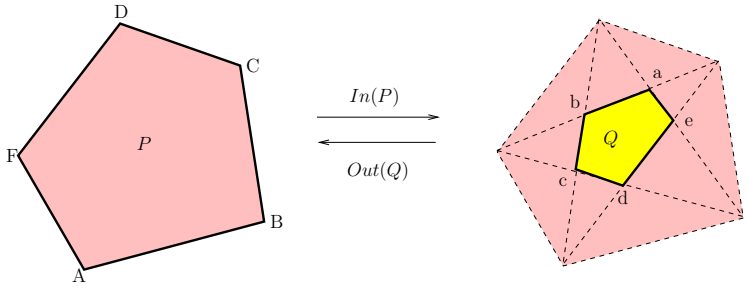
(b) Conduct a series of experiments to quantify our assertion that the expression (2) will not evaluate to True in a "large number" of cases. Assume that we implement the primitives in the straightforward way, as outlined in the text, using IEEE double precision arithmetic. Assume that $\ell = \mathsf{Line}(1, 1, -1)$ and $\ell' = \mathsf{Line}(i, j, 0)$ where $i, j = 1, \ldots, 20$. Estimate the percentage of times when the expression (2) fails (i.e., the expression did not evaluate to the True). ♢

Exercise 1.2: (a) Extend the previous exercise to 3-D, so that you can also define planes.

(b) Repeat the experiment explained in Example (A) of §1: let H_0 be the plane $x + y + z = 1$ and $L_{i,j}$ denote the parametrized line $L_{i,j}(t) = (it, jt, t)$. Compute the intersection p_{ij} of $L_{i,j}$ with H_0, and then check of p_{ij} lies on the plane H_0. Assume $i, j = 1, \ldots, 50$. How many percent of the 2500 tests fail? Does your experiment achieve the reported 37.5% failure rate as noted in the text? ♢

Exercise 1.3: Verify that for a normalized line $\ell = \mathsf{Line}(a, b, c)$, the *signed distance* of $p = \mathsf{Point}(x, y)$ from is $ax + by + c$. Interpret this sign. ♢

Exercise 1.4: We want to define the $In()$ and $Out()$ operations on convex pentagons. Let P be the convex pentagon in Figure 3(a). The 5 diagonals of P intersect pairwise in 5 points. These points determine



(a) Pentagon $P = Out(Q)$ (b) Pentagon $Q = In(P)$

Figure 3: Convex pentagon P and Q and their connection via In, Out

a convex polygon Q inside P, as in Figure 3(b). We define $In(P) := Q$. Similarly, let $Out(Q)$ denote the convex pentagon P obtained by extending the edges of Q until they intersect; there are exactly five intersection points, and these define P. It is easy to see that the following identity holds

$$P = In^m(Out^m(P)) = Out^m(In^m(P)) \tag{4}$$

for any $m \geq 0$, where $In^m(\cdots)$ and $Out^m(\cdots)$ means applying the indicated operations m times. But due to numerical errors, the identity Equation (4) is unlikely to hold. In this exercise, we ask you to implement $In(\cdots)$ and $Out(\cdots)$, and check the identity (4) (for various values of m). You could try more exotic identities like $In^m(Out^{2m}(In^m(P))) = P$ for $m \geq 0$. ♢

Exercise 1.5: If P is non-convex, and perhaps even self-intersecting, is there an interpretation of $In(P)$ and $Out(P)$? ♢

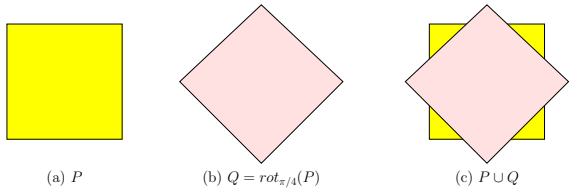
End Exercises

§2. Nonrobustness In Action

The previous section gives a hint of the potential problems with non-robust geometric primitives such as line intersection. We now illustrate consequences of such problems at a macroscopic level.

§2.1. Nonrobustness of CAD Software

Most commercial CAD software can perform Boolean operation on planar polygons, and apply rigid transformations on polygons. Boolean operations such as union, intersection and complement are defined on polygons viewed as sets. By definition, rigid transformations preserve distance; examples are translations and rotations. See Figure 4 where we rotate a square P about its center and then union this with P.



(a) P (b) $Q = rot_{\pi/4}(P)$ (c) $P \cup Q$

Figure 4: Rigid transformation and Boolean operations on polygons.

SYSTEM	n	α	TIME	OUTPUT
ACIS	1000	1.0e-4	5 min	correct
ACIS	1000	1.0e-5	4.5 min	correct
ACIS	1000	1.0e-6	30 min	too difficult!
Microstation95	100	1.0e-2	2 sec	correct
Microstation95	100	0.5e-2	3 sec	incorrect!
Rhino3D	200	1.0e-2	15 sec	correct
Rhino3D	400	1.0e-2	–	crash!
CGAL/LEDA	5000	6.175e-6	30 sec	correct
CGAL/LEDA	5000	1.581e-9	34 sec	correct
CGAL/LEDA	20000	9.88e-7	141 sec	correct

Table 1: Boolean Operations on CAD Software

Consider the following test[4] which begins by constructing a regular n-gon P, and rotating P by α degrees about its center to form Q, and finally forming the union R of P and Q. For a general α, R is now a $4n$-gon. The following table shows the results from several commercial CAD systems: ACIS, Microstation95 and Rhino3D.

The last three rows are results from the CGAL/LEDA software. In contrast to the other nonrobust software, the CGAL/LEDA output is always correct. We shall describe CGAL/LEDA and similar software later, as they embody the kind of solution developed in this book.

This last column in this table illustrates three forms of failures: program crashing, program looping (or taking excessive time, interpreted as too difficult) or silent error. The last kind of failure is actually the most insidious, as it may ultimately cost most in terms of programmer effort.

This table indicates just the tip of the iceberg. CAD software is widely used in automotive, aerospace and manufacturing industry. The cost of software unreliability to the US automobile industry (alone) is estimated at one billion dollars per year, according to a 2002 report [36] from National Institute of Standards and Technology (NIST). Of course, this estimate combines all sources of software errors; it is a difficult task to assign such costs to any particular source. But from a technical viewpoint, there is a general agreement among researchers. In a 1999 MSRI Workshop on Mathematical Foundations of CAD, the consensus opinion of the CAD researchers stated that "the single greatest cause of poor reliability of CAD systems is lack of topologically consistent surface intersection algorithms".

Nonrobust software has a recurring cost in terms of programmer productivity, or may lead to defective products which have less predictable future cost.

§2.2. Nonrobustness of Meshing Software

A mesh is a data structure which represents some physical volume or the surface of such a volume using a collection of interconnected "elements". We call it a **volume mesh** or a **surface mesh**, respectively. The mesh elements may be connected together regularly or irregularly. An example of a regular mesh is a square grid. See Figure 5 for an example of irregular surface meshes.

Mesh generation is a fundamental operation in all of computational sciences and engineering, and in many physical simulation areas. A typical operation is to create a surface meshes from, say, an implicit description of a surface as an algebraic equation. For instance, the mesher might[5] break down when one tries to create surface mesh for the cylinder of unit radius centered about the z-axis.

[4] Mehlhorn and Hart, Invited Talk at ACM Symp. of Solid Modeling, Ann Arbor, June 2001.

[5] This is an actual experience with a meshing software that is part of a fairly widely distributed CFD system. The fix, it turns out, was to perturb the axis slightly away from the z-axis.

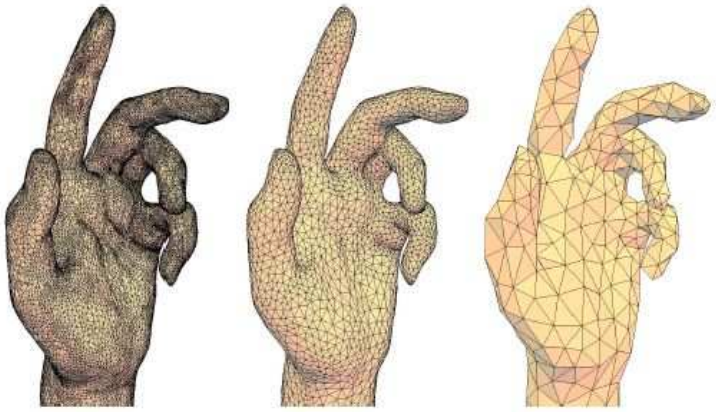


Figure 5: Surface Meshes from Dyer (SGP'07)

Quoting[6] industry experts in the field of computational fluid dynamics (CFD) on a typical scenario: *in a CFD aircraft analysis with 50 million elements, we spend 10-20 minutes for surface mesh generation, 3-4 hours for volume meshing, 1 hour for actual flow analysis, and finally 2-4 weeks for debugging and geometry repair.*

Some explanation of this scenario is in order. The first phase (surface mesh) typically involves constructing a triangulated closed manifold representing, say an aircraft wing. The second phase (volume meshing) involves partitioning the space exterior to the triangulated closed manifold. This volume is of interest for the study of air flow around the body. In other applications such as heat flows, we might be more interested in the volume internal to closed manifold. The third phase (simulation) amounts to solving partial differential equations in the volume. The final phase (debugging) arises when the simulation breaks down due to inconsistencies in the data or model. Note that geometry repair logically belongs to the surface and volume meshing phases. But in practice, errors are not immediately detected and these are caught when problems develop further downstream, and are traced back to meshing errors.

The proportion of time spent for three phases of two such applications are shown in the next table, also taken from Boeing Company:

	Fly Back Vehicle	Target for Computational Electromagnetics
Geometry Fixup	66%	60%
Gridding	17%	25%
Solving	17%	15%

Meshing ought to be a routine step for many applications. But lacking robust meshing, an engineer be in the loop to manually fix any problems that arise. In other words, this step cannot be fully automated.

Nonrobustness is a barrier to full automation in many industrial processes. This has a negative impact on productivity.

§2.3. Testing Numerical Accuracy of Software

[6]Tom Peters (University of Connecticutt) and Dave Ferguson (The Boeing Company), talk at the DARPA/NSF CARGO Kickoff Workshop, Newport RI, May 20-23, 2002.

Mehlhorn [27] pose the "Reliable Algorithmic Software Challenge" (RASC). A critical component in the solution to RASC is a new mode of numerical computation in which we can demand any *á priori* accuracy of a numerical variable. We call this **guaranteed accuracy computation** [47]. Here we explore one class of applications for this mode. In many areas of numerical computation, there are many commercial software available. To evaluate their individual performance, as well as to compare these software, standardized test suites have been collected. Very often test suites only contain input data (called benchmarks), and software are compared in terms of their speed on these benchmarks (under a variety of platforms). But in numerical software, the correctness of these software cannot be taken for granted – we need to also know the accuracy of the software. To do this, each benchmark problem ought to come with model answers to some precision. This would allow us to measure deviation from the model answers. But which software do we trust to produce such model answers? This is one clear application for guaranteed accuracy computation. We look at two such examples.

For our first example, consider linear programming, a major contender for the shortlist of most useful algorithms. Linear programming can be defined as follows. Given vectors $c \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$ $(m > n)$, and a matrix $A \in \mathbb{R}^{m \times n}$, to find a point $x \in \mathbb{R}^n$ which maximizes the dot product $c^T x$ subject to $Ax \leq b$ Geometrically, the point x can be taken to be a vertex of the convex polyhedron defined by $Ax \leq b$ whose projection onto the direction c is maximum. Because of its importance in the industry, many commercial software standardized test suites are available. One of the most well-known software is called `CPLEX`, and `netlib` is a popular collection of benchmark problems. The following table taken from [27] shows the performance of `CPLEX` on 5 benchmark problems.

PROBLEM				CPLEX Solution		CPLEX Solution		
Name	m	n	# non-zeros	RelObjErr	ComputeTime	Violations	Opt?	VerifyTime
degen3	1504	1818	26230	$6.91e-16$	8.08s	0	opt	8.79s
etamacro	401	688	2489	$1.50e-16$	0.13s	10	feas	1.11s
fffff800	525	854	6235	$0.00e+00$	0.09s	0	opt	4.41
pilot.we	737	2789	9218	$2.93e-11$	3.8s	0	opt	1654.64
scsd6	148	1350	5666	$0.00e+00$	0.1s	13	feas	0.52s

For each problem, in this table, we not only give the dimension $m \times n$ of the matrix A, but column 4 also gives the number of non-zeros in the matrix A, as a better indication of the input size. The solution from `CPLEX` is a basis of A. The relative error in the objective value of this basis is indicated. Using guaranteed precision methods, the solution returned by `CPLEX` is checked. Thus, we find that two of these solutions are actually non-optimal (etamacro and scsd6). The column labeled 'Violations' counts the number of constraints violated by the basis returned by `CPLEX`. The time to check (last column) is usually modest compared to that of `CPLEX`, but the benchmark problem 'pilot.we' is an obvious exception. does not always produce the correct optimal answer. Such errors are qualitative, not merely quantitative.

Let us return to the problem of generating model answers for benchmark problems. As in linear programming, the primary goal here is qualitative correctness. Next, we want to ensure certain bounds on the quantitative error. We need software that can guarantee their accuracies in both these senses. In this book, we will see techniques which produce such software. See [10] for the issues of certifying and repairing approximate solutions in linear programming.

Another example is from numerical statistical computation [25]. Here, McCullough [9] address the problem of evaluating the accuracy of statistical packages. At the National Institute of Standards and Technology (NIST), such model answers must have 15 digits of accuracy, and these are generated by running the program at 500 bits of accuracy. It is unclear how 500 bits is sufficient; but it is clear that 500 bits will often be unnecessary. We would like software like `LEDA Real` [7] or `Core Library` [23] that can automatically generate the necessary accuracy needed.

Essentially the same problem has been addressed by Lefévre et al [26], who posed the **Table Maker's Dilemma**, namely how to compute correctly rounded results from evaluation of transcendental functions.

The correct rounding problem is part of a larger, industry-wide concern about the reliability of computer hardware arithmetic. Prior to the 1980's there was widespread concern about the non-portability and unpredictability of computer arithmetic across hardware vendors. The upshot of this is the IEEE standard

on arithmetic which is now widely accepted. The paper [49] provides the foundations of exact rounding.

> *Software testing is a meta application where guaranteed quality of numerical output is not an option.*

§2.4. Accuracy as a Contractual Issue

Geometric tolerancing is one cornerstone of the manufacturing sciences. It provides the language of specifying geometric shapes for the purposes of manufacture. As all manufacturing processes are inherently inexact, part and parcel of this specification includes the acceptable deviation from the ideal shape.

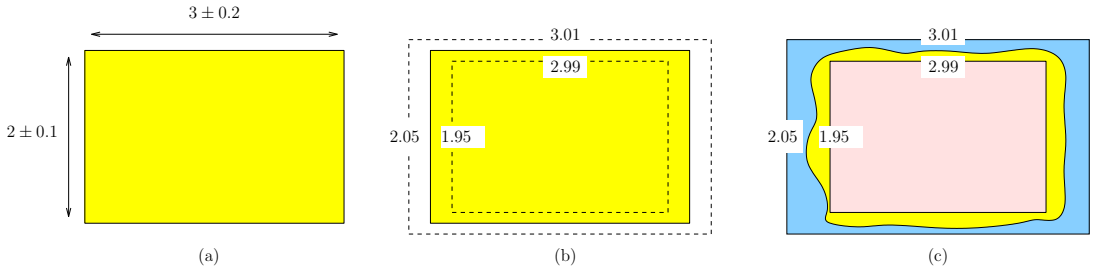


Figure 6: Tolerance Geometry: (a) ideal, (b) zone, (c) actual.

For instance, suppose we wish to manufacture a $3'' \times 2''$ piece of metal plate (see Figure 6). We might tolerance this as $3 \pm 0.2'' \times 2 \pm 0.1''$. The naive and highly intuitive understanding of this tolerance is that any rectangular plate with dimension $L'' \times W''$ is acceptable as long as $|L - 3| \leq 0.2$ and $|W - 2| \leq 0.1$. The problem is that no manufactured object is really a rectangle – it may not even have any straight edge at all. The modern understanding is that the specification describes a **tolerance zone** (Figure 6(b)) and an object is within tolerance if there is some placement of the object whose boundary lies inside this zone (Figure 6(c)). Note that by "placement", we mean a position of the object obtained by rotation and/or translation.

The language and semantics of dimensioning and tolerance is encoded into industry standards (e.g., ANSI standard Y14.5M-1982 [45, 31]); its mathematical basis is similarly encoded [43]. There is a parallel development at the International Standards Organization (ISO).

Beyond having a language to specify shape tolerances, we must have some attendant "assessment methodology" to measure the deviation of physically manufactured shapes, and to verify conformance. Such methodology can be regarded as a subarea of the much larger subject of *metrology*. We use the term *dimensional metrology* to refer to its application in geometric tolerancing. Dimensional metrology is much less understood than tolerancing itself [2]. In fact, this has precipitated a minor crisis [42]. The background of this crisis comes from the proliferation of the **coordinate measurement machine** (CMM) technology in the 1980's. A CMM is a physical device that can be programmed to take measurements of a physical object *and* to compute tolerance characteristics of the object using these measurements. Such machines represent a major advancement over, say, traditional hard gauges (or calipers). While hard gauges give only a Boolean value (go or no-go), the CMM computation returns "characteristic values" that may be used in other applications of metrology such as process monitoring. Thus CMM is regarded as state-of-the-art in dimensional metrology. Note that CMM's are inherently coupled to computational power via embedded software. The problem is that that there is no standard governing this software – different vendor software can give widely divergent answers. This uncertainty could in principle force all government procurement services to come a grinding halt! Imagine a scenario where a critical component of a jet engine is to be toleranced to within 0.001mm but because of inaccurate software, it is really 0.01mm. This could give rise to contractual disputes, premature mechanical failures, or worse. This was the situation uncovered by Walker [42] in 1988.

A basic shape for dimensional tolerancing is the circular disc. We might tolerance such a disc to have radius 2 ± 0.05 cm. The "tolerance zone" semantics in this case is clear: we want the manufactured disc to

have its boundary within an annular region whose inner and outer radii are $1.95 - -2.05$ cm. But what kind of assessment method is available for this? The standard practice is to uniformly measure $5 - 10$ points on the boundary of the manufactured disc, and algorithmically decide if these points lie within the tolerance zone. The justification of such a procedure is far from obvious (see [28]).

In summary, a major component of dimensional metrology revolves around the computational issues, or what Hopp has termed *computational metrology* [21]. To be sure, the uncertainty in CMM technology is grounded in the measurement hardware as well as the software. Modeling and compensating for hardware biases is a major research area. But often, software errors can be orders of magnitude worse than hardware uncertainty.

Dimensional metrology have many economic consequences that may not be obvious. Imagine that we want to tolerance the roundness of a cylinder for a jet engine. If we over-tolerance, this may lead to premature wear-and-tear, and possibly disastrous breakdown. If we under-tolerance, this can greatly add to the cost and number of machining hours, without producing any appreciable benefits. There are limits to how tightly we can tolerance parts, because of the inherent uncertainty in various processes, measurements and computations. If we tolerance too close to this limit, the results are probably unreliable. Moreover, manufacturing costs can increase dramatically as our tolerance approach this limit: the number of machining hours will greatly increase, and The slightest mistake[7] can lead to a rejection of expensive parts made of special alloys.

We describe an anecdote from Hopp: in the aluminum can industry, the thickness of the top of the pop-top cans has some physical limitations. It must be (say) not be thicker than 0.4mm, or else the pop-top mechanism may not work (many of us knows this from first hand experience). It must not be thinner than 0.3mm, or else the internal pressure may cause spontaneous explosion. Hence, we might tolerance this thickness as 0.35 ± 0.05mm. Now, if manufacturing processes were more reliable, and our ability to assess tolerance were more accurate, we might be able to tolerance this thickness as 0.32 ± 0.02mm. Such a change would save the aluminum can industry multi-millions of dollars in material cost each year.

For a survey of the computational problems in dimensional metrology, see [13] or [1]; for a perspective on current dimensional theory, see Voelcker [41]. An NSF-sponsored workshop on manufacturing and computational geometry addresses some of these issues [46].

§2.5. For Lack of a Bit

On 4 June 1996, the maiden flight of the[8] Ariane 5 launcher in French Guiana ended in self-destruction. About 40 seconds after initiation of the flight sequence, at an altitude of about 3700 m, the launcher veered off its flight path, broke up and exploded. A report from the Inquiry Board located the critical events: "At approximately 30 seconds after lift-off, the computer within the back-up inertial reference system, which was working on stand-by for guidance and attitude control, became inoperative. This was caused by an internal variable related to the horizontal velocity of the launcher exceeding a limit which existed in the software of this computer. Approximately 0.05 seconds later the active inertial reference system, identical to the back-up system in hardware and software, failed for the same reason. Since the back-up inertial system was already inoperative, correct guidance and attitude information could no longer be obtained and loss of the mission was inevitable."

The software error began with an overflow problem: "The internal SRI software exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer. This resulted in an Operand Error. The data conversion instructions (in Ada code) were not protected from causing an Operand Error, although other conversions of comparable variables in the same place in the code were protected."

There is a similar episode involving the Patriot Missiles in the 1990 Gulf War. We refer to Higham [18, p. ???] for details.

Naturally, any major incident could be attributed to failures at several levels in the system. For instance, the Inquiry Board in the Ariane incident also pointed at failure at the organizational level: "The extensive

[7] Such parts end up as $10,000 paper weights in some offices.

[8] Ariane is the French version of the Greek name Ariadne (in English!). In Greek mythology, Ariadne was the daughter of King Minos of Crete.

reviews and tests ... did not include adequate analysis and testing of the inertial reference system or of the complete flight control system, which could have detected the potential failure." Kahan and Darcy [22] responded: "What software failure could not be blamed upon inadequate testing? The disaster can be blamed just as well upon a programming language (Ada) that disregarded the default exception-handling specifications in IEEE Standard 754." The Ariane failure lost a satellite payload worth over half a billion dollars. Other costs, including the potential loss of lives, are harder to quantify.

In mission-critical applications, less than fully robust software is not an option.

§2.6. Human Failures

There are many other examples of the drastic consequences of unreliability of numerical computation:

- Intel Pentium chip floating point error (1994-5)
- North Sea Sleipner Class oil rig collapse (1991) due inaccurate finite element analysis,
- Among major engineering structures, bridges seem particularly prone to unpredictable errors. Three modern examples are Tacoma Bridge, Washington, Auckland Bridge, and London Millennium Bridge. Better computer simulations with improved accuracy in numerical simulation may have been able to predict the problems.
- An incidence at the Vancouver Stock Exchange index, Jan 1982 to Nov 1983.

It should be stressed that although numerical errors are at the root of these examples, one could also point to various human errors in these episodes. For instance, in the Ariane 5 failure, one can also point a finger at imperfect software testing, at management lapses, etc. On the other hand, these failings of human institutions would be irrelevant if the source of these problems (numerical errors) could be eliminated.

§3. Responses to Numerical Nonrobustness

The ubiquitous nature of numerical nonrobustness, would seem to cry out for a proper resolution. As computers become more integrated into the fabric of modern society, this problem is expected[9] to worsen. The fact that we do not see a general declaration of war on nonrobustness (in the same way that computer security has marshaled support in recent years) calls for explanation. Here are some common (non)responses to nonrobustness:

¶3. Nonrobustness is inevitable. It is often stated in textbooks and popular science articles that computer arithmetic is inherently inexact, and so exact computation is impossible. Whatever such a statement means, it is wrong to conclude that numerical nonrobustness is unavoidable in software.

¶4. Nonrobustness is unimportant because it is a rare event. It is true that, for any given software, the inputs that causes nonrobust behavior are in a certain sense "rare". The problem is that these rare events occur at a non-negligible frequency.

¶5. Nonrobust software is a mere inconvenience. When the productivity of scientists and engineers is negatively impacted, this "inconvenience" becomes a major issue. A study from NIST [36] estimates the cost of software unreliability in several industries including transportation (aircraft and automobile manufacture) and financial services.

[9] Aided by the relentless progress of Moore's law, we may experience nonrobustness more frequently in the following sense: what used to fail once a month may now fail every day, simply because the same code is executed faster, perhaps on larger data sets.

¶6. Avoid ill-conditioned inputs. It is well-known that we can lose significance very quickly with ill-conditioned inputs. One response of numerical analysts is to suggest that we should avoid ill-conditioned inputs. For instance, if the input is ill-conditioned, we should first perform some random perturbation. Unfortunately, we note that ill-conditioned inputs are often a deliberate feature of the input data. For example, in engineering and architectural designs, multiple collinear points is probably a design feature rather than an accident. So a random perturbation destroys important symmetries and features in the data.

¶7. Use stable algorithms. Stability is an important concept in numerical computation. The definition[10] of what this means is usually informal, but see a definition in [40]. Higham [18, p.30] gives guidelines for designing numerically stable algorithm. The problem is that stable algorithms only serves to reduce the incidence of non-robustness, not eliminate it.

¶8. No one else can solve it either. This might be the response of software vendor, but it seems that many users also buy into this argument. We believe that this response[11] is increasingly weak in many domains, including meshing software.

§4. Additional Notes

An early effort to address nonrobustness geometric algorithms resulted in a 2-volume special issue of *ACM Trans. on Graphics*, Vol.3, 1984, edited by R. Forrest, L.Guibas and J.Nievergelt. The editorial deplores the disconnect between the graphics community and computational geometry, and points out the difficulties in implementing robust algorithms: "A Bugbear ... is the plethora of special cases ... glossed over in theoretical algorithms". "In practice, even the simplest operations such as line segment intersection... are difficult, if not impossible, to implement reliably and robustly". "All geometric modeling systems run into numerical problems and are, on occasion, inaccurate, inconsistent, or simply break...". This was in the early days of computational geometry where the field has a bibliography with just over 300 papers at that time, and the book of Shamos and Preparata was just out. See also [14].

In the literature, what we call "robustness" also goes under the name of **reliability**. The interest in robust or reliable software appears in different forms in several communities.

- The area called **validated computing** or **certified computing** aims to achieve robustness by providing guaranteed error bounds. The techniques comes from interval arithmetic. There is active software and hardware development, aimed to make interval (or enclosure) methods easily accessible to programmers. There are several key books in this area; in particular, [35] addresses problems of computational geometry. The journal[12] **Reliable Computing** from Kluwer Academic Publishers devoted to this field; see also the special issue [34].

- The geometric design community is acutely aware of nonrobustness in CAD software. Hoffman [20, 19] has been an advocate of efforts to address this problem. Farouki [12] looks at numerical stability issues, especially in connection to the favorable properties of the Bezier representation. Patrikalakis [32] surveys robustness issues in geometric modelling software. One of the key open problems in this field is the development of fully reliable numerical algorithms for the surface-surface intersection (SSI) problem.

- Nonrobustness in computer graphics might appear to be a non-issue if we think the main computational issue is deciding how to turn on a pixel on the screen. But in reality, there is much geometric processing before making this final decision. The book of Christer Ericson [11, Chap. 11] is devoted to numerical robustness, presenting many of the issues that implementor of computer games face.

- There has been long-standing interest in theory of real computation. We mention two schools of thought: one is the "Type Two Theory of Effectivity" (TTE Theory) [44], with roots in the classical

[10] We are indebted to Professor Michael Overton for this remark.

[11] We think CAD software vendors ought to be extremely interested in the current research development when confronted with the data in Table 1.

[12] From 1991 to 1994, it was known as "J. of Interval Computations".

constructive analysis. The other school (BSS or Algebraic Theory) [6] is based on the uniform algebraic models. The annual workshop *Computability and Complexity in Analysis* is representative of the TTE approach. The Algebraic School is initiated by the work of Smale and his co-workers. Many interesting connections with standard complexity theory has been discovered.

- Over the years, there has been several software developed to support real computation: these are usually based on the lazy-evaluation paradigm, which provides more and more bits of accuracy on demand. A recent system along this line is Mueller's `iRRAM` [30]. The annual workshop "Real Numbers and Computers" is representative of this community.

- Numerical analysts have long recognized the pitfalls associated with numerical computing [15, 38, 17]. Key topics here are numerical stability, accuracy and conditioning. A modern treatment of these issues is Higham [18]. The book of Chaitin-Chatelin and Valérie Frayssé [8] addresses stability of finite-precision computation. The key role of condition numbers have also been studied in the Smale school.

- The meteorologist Edward N. Lorenz (1917-2008) accidentally discovered computational chaos in the winter of 1961. He was running a weather simulation for a second time, but for a longer time period. He started the second simulation in the middle, by typing in numbers that were printed from the first run. He was surprised to see the second weather trajectory quickly diverge from the first. He eventually realized that the computer stored numbers to an accuracy of 6 decimal places (e.g., 0.123456), but the printout was shorted to 3 decimal places (e.g., 0.123). This discrepancy led to the diverging simulation results. He published this finding in 1963. In a 1964 paper, he described how a small change in parameters in a weather model can transform a regular periodic behavior into chaotic pattern. But it was the title of his 1972 talk at the American Association for the Advancement of Science that entered the popular imagination: "Predictability: Does the Flap of a Butterfly's Wings in Brazil Set Off a Tornado in Texas?" This story in recounted in James Gleick's book "Chaos".

- Various workshops been devoted to robust computation, including: SIGGRAPH Panel on Robust Geometry (1998), NSF/SIAM Workshop on Integration of CAD and CFD (1999), MSRI Workshop on Foundations of CAD (1999), Minisymposium Robust Geometric Computation at Geometric Design and Computing (2001), DIMACS Workshop on Implementation of Geometric Algorithms (2002).

- Reports in computing fields emphasize the challenges of robust computation: NSF Report on Emerging Challenges in Computational Topology [3], Computational Geometry Task Force Report [5, 4], ACM Strategic Directions Report [39].

- Special journal issues have address this topic. E.g., [16], [29], [33]. Two surveys on nonrobustness are [37, 48]. Major software such as `LEDA` and `CGAL` have been highly successful in solving nonrobustness in many basic problems of geometry.

§5. APPENDIX: General Notations

We gather some common notations that will be used in this book.

The set of natural numbers is denoted by $\mathbb{N} = \{0, 1, 2, \ldots\}$. The other systems of numbers are integers $\mathbb{Z} = \{0, \pm 1, \pm 2, \ldots\}$, rational numbers $\mathbb{Q} = \{n/m : n, m \in \mathbb{Z}, n \neq 0\}$, real numbers $\mathbb{R}$ and complex numbers $\mathbb{C}$.

For any positive real number x, we write $\lg x$ and $\ln x$ for their logarithms to base 2 and the natural logarithm, $\log_2 x$ and $\log_e x$.

The set of $m \times n$ matrices with entries over a ring R is denoted $R^{m\times n}$. Let $M \in R^{m\times n}$. If the (i, j)th entry of M is x_{ij}, we may write $M = [x_{ij}]_{i,j=1}^{m,n}$ (or simply, $M = [x_{ij}]_{i,j}$). The (i, j)th entry of M is denoted $M(i; j)$. More generally, if $i_1, i_2, \ldots, i_k$ are indices of rows and $j_1, \ldots, j_\ell$ are indices of columns,

$$M(i_1..i_k; j_1..j_\ell) \tag{5}$$

denotes the submatrix obtained by intersecting the indicated rows and columns. In case $k = \ell = 1$, we often prefer to write $(M)_{i,j}$ or $(M)_{ij}$ instead of $M(i; j)$. If we delete the ith row and jth column of M,

the resulting matrix is denoted $M[i;j]$. Again, this notation can be generalized to deleting more rows and columns. E.g., $M[i_1, i_2; j_1, j_2, j_3]$ or $[M]_{i_1,i_2;j_1,j_2,j_3}$. The **transpose** of M is the $n \times m$ matrix, denoted M^T, such that $(M^T)_{i,j} = (M)_{j,i}$.

References

[1] G. Anthony, H. Anthony, B. Bittner, B. Butler, M. Cox, R. Drieschner, R. Elligsen, A. B. Forbes, H. Groß, S. Hannaby, P. Harris, and J. Kok. Chebyshev best-fit geometric elements. NPL Technical Report DITC 221/93, National Physical Laboratory, Division of Information Technology and Computing, NPL, Teddington, Middlesex, U.K. TW11 0LW, October, 1992.

[2] W. Beckwith and F. G. Parsons. Measurement methods and the new standard B89.3.2. In *Proc. 1993 International Forum on Dimensional Tolerancing and Metrology*, pages 31–36, New York, NY, 1993. The American Society of Mechanical Engineers. CRTD-Vol.27.

[3] M. Bern, D. Eppstein, P. K. Agarwal, N. Amenta, P. Chew, T. Dey, D. P. Dobkin, H. Edelsbrunner, C. Grimm, L. J. Guibas, J. Harer, J. Hass, A. Hicks, C. K. Johnson, G. Lerman, D. Letscher, P. Plassmann, E. Sedgwick, J. Snoeyink, J. Weeks, C. Yap, and D. Zorin. Emerging challenges in computational topology, 1999. Invited NSF Workshop on Computational Topology, (organizers: M.Bern and D.Eppstein), Miami Beach, Florida, June 11–12, 1999. Report available from Computing Research Repository (CoRR), http://xxx.lanl.gov/abs/cs.CG/9909001.

[4] Bernard Chazelle, et al. The computational geometry impact task force report: an executive summary. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, pages 59–66. Springer, 1996. Lecture Notes in Computer Science No. 1148.

[5] Bernard Chazelle, et al. Application challenges to computational geometry: CG impact task force report. Tr-521-96, Princeton Univ., April, 1996. See also URL www.cs.princeton.edu/~chazelle/.

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

[7] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. Exact efficient geometric computation made easy. In *Proc. 15th ACM Symp. Comp. Geom.*, pages 341–450, New York, 1999. ACM Press.

[8] F. Chaitin-Chatelin and V. Frayssé. *Lectures on Finite Precision Computations*. Society for Industrial and Applied Mathematics, Philadelphia, 1996.

[9] B. M. Cullough. Assessing the reliability of statistical software: Part II. *The American Statistician*, 53:149–159, 1999.

[10] M. Dhiflaoui, S. Funke, C. Kwappik, K. Mehlhorn, M. Seel, E. Schömer, R. Schulte, , and D. Weber. Certifying and repairing solutions to large LPs, how good are LP-solvers? In *SODA*, pages 255–56, 2003.

[11] C. Ericson. *Real-Time Collision Detection*. Morgan Kaufmann, 2005.

[12] R. T. Farouki. Numerical stability in geometric algorithms and representation. In D. C. Handscomb, editor, *The Mathematics of Surfaces III*, pages 83–114. Clarendon Press, Oxford, 1989.

[13] S. C. Feng and T. H. Hopp. A review of current geometric tolerancing theories and inspection data analysis algorithms. Technical Report NISTIR-4509, National Institute of Standards and Technology, U.S. Department of Commerce. Factory Automation Systems Division, Gaithersburg, MD 20899, February 1991.

[14] A. R. Forrest. Computational geometry and software engineering: Towards a geometric computing environment. In D. F. Rogers and R. A. Earnshaw, editors, *Techniques for Computer Graphics*, pages 23–37. Springer-Verlag, 1987.

[15] G. E. Forsythe. Pitfalls in computation, or why a math book isn't enough. *Amer. Math. Monthly*, 77:931–956, 1970.

[16] S. Fortune. Editorial: Special issue on implementation of geometric algorithms, 2000.

[17] L. Fox. How to get meaningless answers in scientific computation (and what to do about it). *IMA Bulletin*, 7(10):296–302, 1971.

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

[19] C. M. Hoffmann. *Geometric and Solid Modeling: an Introduction*. Morgan Kaufmann Publishers, Inc., San Mateo, California 94403, 1989.

[20] C. M. Hoffmann. The problems of accuracy and robustness in geometric computation. *IEEE Computer*, 22(3):31–42, March 1989.

[21] T. H. Hopp. Computational metrology. In *Proc. 1993 International Forum on Dimensional Tolerancing and Metrology*, pages 207–217, New York, NY, 1993. The American Society of Mechanical Engineers. CRTD-Vol.27.

[22] W. Kahan and J. D. Darcy. How Javas floating-point hurts everyone everywhere, March 1998. Paper presented at ACM 1998 Workshop on Java for High Performance Network Computing, Stanford University.

[23] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A Core library for robust numerical and geometric computation. In *15th ACM Symp. Computational Geometry*, pages 351–359, 1999.

[24] G. Knott and E. Jou. A program to determine whether two line segments intersect. Technical Report CAR-TR-306, CS-TR-1884, Computer Science Department, University of Maryland, College Park, August 1987.

[25] K. Lange. *Numerical Analysis for Statisticians*. Springer, New York, 1999.

[26] V. Lefèvre, J.-M. Muller, and A. Tisserand. Towards correctly rounded transcendentals. *IEEE Trans. Computers*, 47(11):1235–1243, 1998.

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

[28] K. Mehlhorn, T. Shermer, and C. Yap. A complete roundness classification procedure. In *13th ACM Symp. on Comp. Geometry*, pages 129–138, 1997.

[29] N. Mueller, M. Escardo, and P. Zimmermann. Guest editor's introduction: Practical development of exact real number computation. *J. of Logic and Algebraic Programming*, 64(1), 2004. Special Issue.

[30] N. T. Müller. The iRRAM: Exact arithmetic in C++. In J. Blank, V. Brattka, and P. Hertling, editors, *Computability and Complexity in Analysis*, pages 222–252. Springer, 2000. 4th International Workshop, CCA 2000, Swansea, UK, September 17-19, 2000, Selected Papers, Lecture Notes in Computer Science, No. 2064.

[31] A. G. Neumann. The new ASME Y14.5M standard on dimensioning and tolerancing. *Manufacturing Review*, 7(1):16–23, March 1994.

[32] N. M. Patrikalakis, W. Cho, C.-Y. Hu, T. Maekawa, E. C. Sherbrooke, and J. Zhou. Towards robust geometric modelers, 1994 progress report. In *Proc. 1995 NSF Design and Manufacturing Grantees Conference*, pages 139–140, 1995.

[33] S. Pion and C. Y. G. Editors). Reliable, 2004.

[34] H. Ratschek and J. G. R. G. Editors). Editorial: What can one learn from box-plane intersections?, 2000. Special Issue on Reliable Geometric Computations.

[35] 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.

[36] 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.

[37] S. Schirra. Robustness and precision issues in geometric computation. In J. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*. Elsevier Science Publishers, B.V. North-Holland, Amsterdam, 1999.

[38] I. A. Stegun and M. Abramowitz. Pitfalls in computation. *J. Soc. Indust. Appl. Math.*, 4(4):207–219, 1956.

[39] R. Tamassia, P. Agarwal, N. Amato, D. Chen, D. Dobkin, R. Drysdal, S. Fortune, M. Doorich, J. Hershberger, J. O'Rourke, F. Preparata, J.-R. Sack, S. Suri, I. Tollis, J. Vitter, and S. Whitesides. Strategic directions in computational geometry working group report. *ACM Computing Surveys*, 28(4), Dec. 1996.

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

[41] H. B. Voelcker. A current perspective on tolerancing and metrology. In *Proc. 1993 International Forum on Dimensional Tolerancing and Metrology*, pages 49–60, New York, NY, 1993. The American Society of Mechanical Engineers. CRTD-Vol.27.

[42] R. K. Walker. CMM form tolerance algorithm testing. GIDEP Alert X1-A1-88-01 and X1-A1-88-01a, Government-Industry Data Exchange Program, Pomona, CA, 1988.

[43] R. K. Walker and V. Srinivasan. Creation and evolution of the ASME Y14.5.1M standard. *Manufacturing Review*, 7(1):16–23, March 1994.

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

[45] A. Y14.5M-1982. *Dimensioning and tolerancing*. American Society of Mechanical Engineers, New York, NY, 1982.

[46] C. K. Yap. Report on NSF Workshop on Manufacturing and Computational Geometry. *IEEE Computational Science & Engineering*, 2(2):82–84, 1995. Video Proceedings. Workshop at the Courant Institute, New York University, April 1–2, 1994.

[47] C. K. Yap. On guaranteed accuracy computation. In F. Chen and D. Wang, editors, *Geometric Computation*, chapter 12, pages 322–373. World Scientific Publishing Co., Singapore, 2004.

[48] 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.

[49] C. K. Yap and J. Yu. Foundations of exact rounding. In *Proc. WALCOM 2009*, volume 5431 of *Lecture Notes in Computer Science*, 2009. To appear. Invited talk, 3rd Workshop on Algorithms and Computation, Kolkata, India.