Robustness and Precision Issues in Geometric Computation*

Stefan Schirra
Max-Planck-Institut für Informatik
Saarbrücken, Germany

Abstract

This is a preliminary version of a chapter that will appear in the *Handbook on Computational Geometry*, edited by J.R. Sack and J. Urrutia.
We give a survey on techniques that have been proposed and successfully used to attack robustness and precision problems in the implementation of geometric algorithms.

*Work on this survey was partially supported by the ESPRIT IV Long Term Research Project No. 21957 (CGAL).

1 Introduction

We give a survey[1] on techniques that have been proposed and successfully used to attack robustness problems in the implementation of geometric algorithms. Our attention is directed to precision[2], more precisely, on how to deal with the notorious problems that imprecise geometric calculations can cause in the implementation of geometric algorithms. Precision problems can make implementing geometric algorithms very unpleasant [36, 37, 49, 50, 94], if no appropriate techniques are used to deal with imprecision.

1.1 Precision, Correctness, and Robustness

Geometric algorithms are usually designed and proven to be correct in a computational model that assumes exact computation over the real numbers. In implementations of geometric algorithms, exact real arithmetic is mostly replaced by the fast finite precision floating-point arithmetic provided by the hardware of a computer system. For some problems and restricted sets of input data, this approach works well, but in many implementations the effects of squeezing the infinite set of real numbers into the finite set of floating-point numbers can cause catastrophic errors in practice. Due to rounding errors many implementations of geometric algorithms crash, loop forever, or in the best case, simply compute wrong results for some of the inputs for which they are supposed to work. Fig. 1 gives an example.

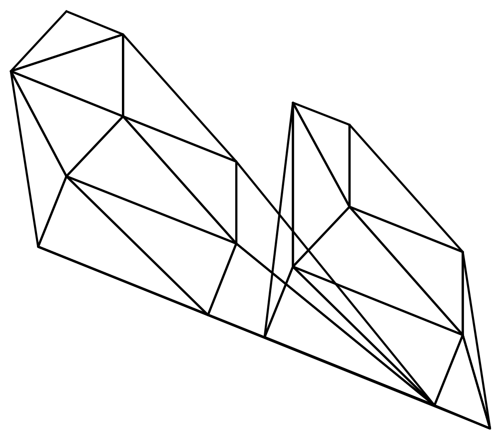

Figure 1: Incorrect Delaunay triangulation. The error was caused by precision problems, see [130] for more details. The correct Delaunay triangulation is given in Fig. 2. *Courtesy of J. R. Shewchuk* [130, 132].

The conditionals in a program are most critical because they determine the flow of control. If in every test the same decision is made as if all computations would have been done over the reals, the algorithm is always in a state equivalent to that of its theoretical counterpart.

[1] This survey is based on *Precision and Robustness in Geometric Computations*, Chapter 9 of *Algorithmic Foundations of Geographic Information Systems*, Lecture Notes in Computer Science 1340, Springer-Verlag, 1997.

[2] The terms precision and accuracy are often used interchangeably. We mainly adopt the terminology used in [63]. *Accuracy* refers to the relationship between reality and the data representing it. *Precision* refers to the level of detail with which (numerical) data is represented.

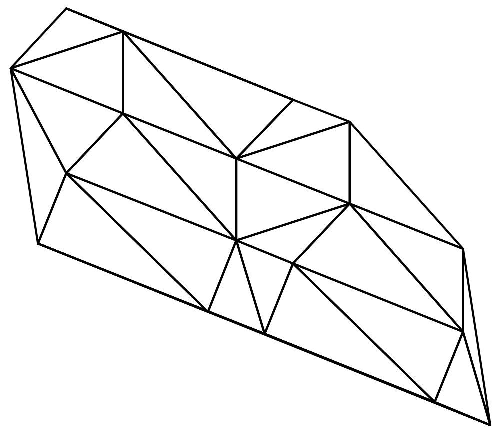

Figure 2: Correct Delaunay triangulation. *Courtesy of J. R. Shewchuk* [130, 132].

In this case, the combinatorial part of the geometric output of the algorithm will be correct. Numerical data, however, computed by the algorithm might still be imprecise. In a branching step of a geometric algorithm, numerical values are compared. Without loss of generality we can assume that one of the values is zero, i.e. that the branching is on the sign of the value of an arithmetic expression. In the theoretical model of computation a real-valued expression is evaluated correctly for all real input data, but in practice only an approximation is computed. Thus a wrong sign might be computed and hence the algorithm might branch incorrectly. Such a wrong decision has been made in the computation of the "triangulation" shown in Fig. 1.

An incorrect result is one possible consequence of an incorrect decision. Program crashing is the other possibility. Decisions made in branching steps are usually not independent. Mutually contradicting decisions violating basic laws of geometry may take the algorithm to a state which could never be reached with correct decisions. Since the algorithm was not designed for such states, it crashes. Therefore `segmentation fault`s and `bus error`s are more likely than incorrect results.

In general, robustness is a measure of the ability to recover from error conditions, e.g., tolerance of failures of internal components or errors in input data. Often an implementation of an algorithm is considered to be *robust* if it produces the correct result for some perturbation of the input. It is called *stable* if the perturbation is small. This terminology has been adopted from numerical analysis where backward error analysis is used to get bounds on the sizes of the perturbations. Geometric computation, however, goes beyond numerical computation. Since geometric problems involve not only numerical but also combinatorial data it is not always clear what perturbation of the input, especially of the combinatorial part, means. Perturbation of the input is justified by the fact that in many geometric problems the numerical data are real world data obtained by measuring and hence known to be inaccurate.

1.2 Attacks on the Precision Problem

There are two obvious approaches for solving the precision problem. The first is to change the model of computation: design algorithms that can deal with imprecise computation. For a small number of basic problems this approach has been applied successfully but a general theory of how to design algorithms with imprecise primitives or how to adopt algorithms designed for exact computation with real numbers is still a distant goal [67]. The second approach is exact computation: compute with a precision that is sufficient to keep the theoretical correctness of an algorithm designed for real arithmetic alive. This is basically possible, at least theoretically, in almost all cases arising in practical geometric computing. The second approach is promising, because it allows exact implementations of numerous geometric algorithms developed for real arithmetic without modifications of these algorithms. However, exact computation slows down the computation and the overhead in running time can be tremendous, especially in cascaded computations, where the output of one computation is used as input by the next.

2 Geometric Computation

A geometric problem can be seen as a mapping from a set of permitted input data, consisting of a combinatorial and a numerical part, to a set of valid output data, again consisting of a combinatorial and a numerical part. A geometric algorithm solves a problem if it computes the output specified by the problem mapping for a given input. For some geometric problems the numerical data of the output are a subset of the data of the input. Those geometric problems are called *selective*. In other geometric problems new geometric objects are created which involve new numerical data that have to be computed from the input data. Such problems are called *constructive*. Geometric problems might have various facets, even basic geometric problems appear in different variants.

We use two classical geometric problems for illustration, convex hull and intersection of line segments in two dimensions. In the two-dimensional *convex hull problem* the input is a set of points. The numerical part might consist of the coordinates of the input points; the combinatorial part is simply the assignment of the coordinate values to the points in the plane. The output might be the convex hull of the set of points, i.e., the smallest convex polygon containing all the input points. The combinatorial part of the output might be the sorted cyclic sequence of the points on the convex hull in counterclockwise order. The point coordinates form the numerical part of the output. In a variant of the problem only the extreme points among the input points have to be computed, where a point is called extreme if its deletion from the input set would change the convex hull. Note that the problem is selective according to our definition even if a convex polygon and hence a new geometric object is constructed.

In the *line segment intersection problem* the intersections among a set of line segments are computed. The numerical input data are the coordinates of the segment endpoints, the combinatorial part of the input just pairs them together. The combinatorial part of the output might be a combinatorial embedding of a graph whose vertices are the endpoints of the segments and the points of intersection between the segments. Edges connect two vertices if they belong to the same line segment l and no other vertex lies between them on l. Combinatorial embedding means that the set of edges incident to a vertex are given in cyclic order. The numerical part is formed by the coordinates of the points assigned to the vertices in the graph. Since the intersection points are in general not part of the input, the problem is constructive. A variant might ask only for all pairs of segments that have a point in common. This version is selective.

2.1 Geometric Predicates

Geometric primitives are the basic operations in geometric algorithms. There is a fairly small set of such basic operations that cover most of the computations in computational geometry algorithms. Geometric primitives subsume predicates and constructions of basic geometric objects, like line segments or circles. Geometric predicates test properties of basic geometric objects. They are used in conditional tests that direct the control flow in geometric algorithms. Well-known examples are: testing whether two line segments intersect, testing whether a sequence of points defines a right turn, or testing whether a point is inside or on the circle defined by three other points.

Geometric predicates involve the comparison of numbers which are given by arithmetic expressions. The operands of the expressions are numerical data of the geometric objects that are tested and constants, usually integers. Expressions differ by the operations used, but many geometric predicates involve arithmetic expressions over $+, -, *$ only, or can at least be reformulated in such a way.

2.2 Arithmetic Expressions in Geometric Predicates

One can think of an arithmetic expression as a labeled binary tree. Each inner node is labeled with a binary or unary operation. It has pointers to trees defining its operands. The pointers are ordered corresponding to the order of the operands. The leaves are labeled with constants or variables which are placeholders for numerical input values. Such a representation is called an *expression tree*.

The numerical data that form the operands in an expression evaluated in a geometric predicate in the execution of a a geometric algorithm might be again defined by previously evaluated expressions. Tracing these expressions backwards we finally get expressions on numerical input data whose values for concrete problem instances have to be compared in the predicates. Since intermediate results are used in several places in an expression we get a directed acyclic graph (dag) rather than a tree.

Without loss of generality we may assume that the comparison of numerical values in predicates is a comparison of the value of some arithmetic expression with zero. The *depth of an expression tree* is the length of the longest root-to-leaf path in the tree. For many geometric problems the depth of the expressions appearing in the predicates is bounded by some constant [151]. Expressions over input variables involving operations $+, -, *$ only are called *polynomial*, because they define multivariate polynomials in the variables. If all constants in the expression are integral, a polynomial expression is called *integral*. The *degree* of a polynomial expression is the total degree of the resulting multivariate polynomial. In [19, 91] the notion of the degree of an expression is extended to expressions involving square roots. An expression involving operations $+, -, *, /$ only is called *rational*.

2.3 Geometric Computation with Floating-Point Numbers

Floating-point numbers are the standard substitution for real numbers in scientific computation. In some programming languages the floating-point number type is even called `real` [81]. Since most geometric computations are executed with floating-point arithmetic, it is worth taking a closer look at floating-point computation. Goldberg [62] gives an excellent overview.

A finite-precision floating-point system has a base $\texttt{B}$, a fixed mantissa length l (also called significand length or precision), and an exponent range $[e_{\min}..e_{\max}]$.

$$\pm d_0.d_1d_2\cdots d_{l-1} * \mathrm{B}^e,$$

$0 \le d_i < \mathrm{B}$, represents the number

$$\pm(d_0 + d_1 \cdot \mathrm{B}^{-1} + d_2 \cdot \mathrm{B}^{-2} + \cdots + d_{l-1}\mathrm{B}^{-l+1}) \cdot \mathrm{B}^e.$$

A representation of a floating-point number is called normalized iff $d_0 \neq 0$. For example, the rational number 1/2 has representations $0.500 * 10^0$ or $5.000 * 10^{-1}$ in a floating-point system with base 10 and mantissa length 4 and normalized representation $1.00 * 2^{-1}$ in a floating-point system with base 2 and mantissa length 3.

Since an infinite set of numbers is represented by finitely many floating-point numbers, rounding errors occur. A real number is called representable if it is zero or its absolute value is in the interval $[\mathrm{B}^{e_{\min}}, \mathrm{B}^{e_{\max}+1}]$. Let r be some real number and f_r be a floating-point representation for r. Then $|r - f_r|$ is called *absolute error* and $|r - f_r|/|r|$ is called *relative error*. The relative error of rounding a representable real toward the nearest floating-point number in a floating-point system with base B and mantissa length l is bounded by $\frac{\mathrm{B}}{2} \cdot \mathrm{B}^{-l}$, which is called *machine epsilon*. Calculations can underflow or overflow, i.e., leave the range of representable numbers.

Fortunately, the times where the results of floating-point computations could drastically differ from one machine to another, depending on the precision of the floating-point machinery, seem to be coming to an end. The IEEE standard 754 for binary floating-point computation [133] is becoming widely accepted by hardware-manufacturers. The IEEE standard 754 requires that the results of $+, -, \cdot, /$ and $\sqrt{\ }$ are exactly rounded, i.e., the result is the exact result rounded according to the chosen rounding mode. The default rounding mode is round to nearest. Ties in round to nearest are broken such that the least significant bit becomes 0. Besides rounding toward nearest, rounding toward zero, rounding toward ∞, and rounding toward -∞ are rounding modes that have to be supported according to IEEE standard 754.

The standard makes reasoning about correctness of a floating-point computation machine-independent. The result of the basic operations will be the same on different machines if both support IEEE standard and the same precision is used. Thereby code becomes portable.

The IEEE standard 754 specifies floating-point computation in single, single extended, double, and double extended precision. Single precision is specified for a 32 bit word, double precision for two consecutive 32 bit words. In single precision the mantissa length is $l = 24$ and the exponent range is $[-126..127]$. Double precision has mantissa length $l = 53$ and exponent range $[-1022..1023]$. Hence the relative errors are bounded by 2^{-24} and 2^{-53}. The single and double precision formats usually correspond to the number types `float` and `double` in C++.

Floating-point numbers are represented in normalized representation. Since the zeroth bit is always 1 in normalized representation with base 2, it is not stored. There are exceptions to this rule. *Denormalized* numbers are added to let the floating-point numbers underflow nicely and preserve the property "$x - y = 0$ iff $x = y$". Zero and the denormalized numbers are represented with exponent $e_{\min} - 1$. Besides these floating-point numbers there are special quantities $+\infty$, $-\infty$ and `NaN` (`Not a Number`) to handle exceptional situations. For example $-1.0/0.0 = -\infty$, `NaN` is the result of $\sqrt{-1}$, and ∞ is the result of overflow in positive range.

Due to the unavoidable rounding errors, floating-point arithmetic is inherently imprecise. Basic laws of arithmetic like associativity and distributivity are not satisfied by floating-point arithmetic. Section 13.2 in [108] gives some examples. Since the standard (almost) fixes

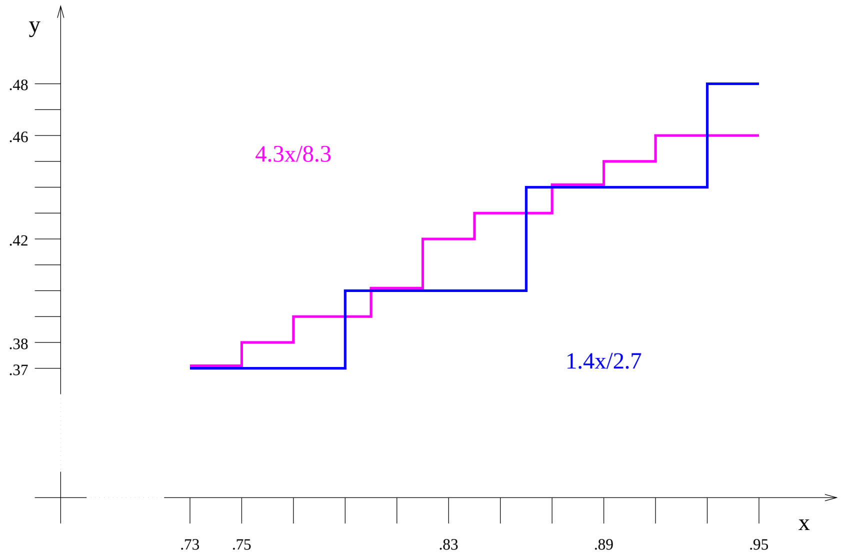

Figure 3: Evaluation of the line equations $y = 4.3 \cdot x/8.3$ and $y = 1.4 \cdot x/2.7$ in a floating-point system with base 10 and mantissa length 2 and rounding to nearest suggests that the lines have several intersection points besides the true intersection point at the origin.

the layout of bits for mantissa and exponent in the representation of floating-point numbers, bit-operations can be used to extract information.

Naively applied floating-point arithmetic can set axioms of geometry out of order. A classical example is Ramshaw's braided lines (see Fig. 3 and [108, 109]).

Rewriting an expression to get a numerically more stable evaluation order can already help a lot: Goldberg [62] gives the following example due to Kahan. Consider a triangle with sides of length $a \geq b \geq c$ respectively. The area of a such a triangle is

$$\sqrt{s(s-a)(s-b)(s-c)}$$

where $s = (a+b+c)/2$. For $a = 9.0$, $b = c = 4.53$ the correct value of s in a floating-point system with base 10, mantissa length 3 and exact rounding is 9.03 while the computed value $\tilde{s}$ is 9.05. The area is 2.34, the computed area, however, is 3.04, an error of nearly 30%. Using the expression

$$\sqrt{(a+(b+c)) \cdot (c-(a-b)) \cdot (c+(a-b)) \cdot (a+(b-c))}/4$$

one gets 2.35, an error of less than 1%. For a less needle-like triangle with $a = 6.9$, $b = 3.68$, and $c = 3.48$ the improvement is not so drastic. Using the first expression, the result computed by a floating-point system with base 10, mantissa length 3 and exact rounding is 3.36. The second expression gives 3.3. The exact area is approximately 3.11. One can show that the relative error of the second expression is at most 11 times machine epsilon [62].

Rewriting also helps with the braided lines. If the abscissae are computed as $(4.3/8.3) \cdot x$ and $(1.4/2.7) \cdot x$, there is no braiding anymore. The lines still do have more than one point in common, but besides the crossing at the origin there are no further crossings anymore. As the examples above show, the way a numerical value is computed influences its precision. Summation of floating-point numbers is another classical example for such effects. Rearranging the summands can helps to reduce imprecision due to extinction.

2.4 Heuristic Epsilons

A widely used method to deal with numerical inaccuracies is based on the rule of thumb

If something is close to zero it is zero.

Some trigger-value $\varepsilon_{\text{magic}}$ is added to a conditional test where a numerical value is compared to zero. If the computed approximation is smaller than $\varepsilon_{\text{magic}}$ it is treated as zero. Adding such epsilons is popular folklore. What should the $\varepsilon_{\text{magic}}$ be? In practice, $\varepsilon_{\text{magic}}$ is usually chosen as some fixed tiny constant and hence not sensitive to the actual sizes of the operands in a concrete expression. Furthermore, the same epsilon is often taken for all comparisons, no matter which expression or which predicate is being evaluated. Usually, no proof is given that the chosen $\varepsilon_{\text{magic}}$ makes sense. $\varepsilon_{\text{magic}}$ is guessed and adjusted by trial and error until the current value works for the considered inputs, i.e., until no catastrophic errors occur anymore. Yap [150] suggests calling this procedure *epsilon-tweaking*.

Adding epsilon is justified by the following reasoning: If something is so close to zero, then a small modification of the input, i.e., a perturbation of the numerical data by a small amount, would lead to value zero in the evaluated expression. There are, however, severe problems with that reasoning. The size of the perturbation causes a problem. The justification for adding epsilons assumes that the perturbation of the (numerical) input is small. Even if such a small perturbation exists for each predicate, the existence of a global small perturbation of the input data is not guaranteed. Fig. 4 shows a polyline, where every three consecutive vertices are collinear under the "close to zero is zero" rule. In each case, a fairly small perturbation

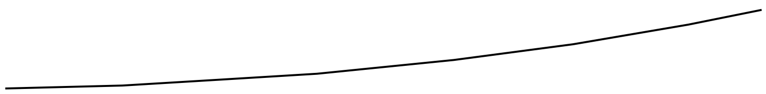

Figure 4: A locally straight line

of the points exists that makes them collinear. There is, however, no small perturbation that makes the whole polyline straight. The example indicates that collinearity is not transitive. Generally, equality is not transitive under epsilon-tweaking. This might be the most serious problem with this approach. Another problem is that different tests might require different perturbations, e.g., predicate P_1 might require a larger value for input variable x_{56} while test P_2 requires a smaller value, such that both expressions evaluate to zero. There might be no perturbation of the input data that leads to the decisions made by the "close to zero is zero" rule. Finally, a result computed with "close to zero is zero" is not the exact result for the input data but only for a perturbation of it. For some geometric problems this might cause trouble, since the computed output and the exact output can be combinatorially very different [22].

3 Exact Geometric Computation

An obvious approach to the precision problem is to compute "exactly". In this approach the computation model over the reals is mimicked in order to preserve the theoretical correctness proof. Exact computation means to ensure that all decisions made by the algorithm are correct decisions for the actual input, not only for some perturbation of it. As we shall see, it does not mean that in all calculations exact representations for all numerical values have to be computed. Approximations that are sufficiently close to the exact value can often be used to guarantee the correctness of a decision. Empirically it turns out to be true for most of the decisions made by a geometric algorithm that approximations are sufficient. Only degenerate and nearly degenerate situations cause problems. That is why most implementations based on floating-point numbers work very well for the majority of the considered problem instances and fail only occasionally. This is made possible by the fact that the numerical input data for geometric algorithms are hardly arbitrary real numbers. In almost all cases the numerical input data are rationals given as floating-point numbers or even integers.

If an implementation of an algorithm does all branchings the same way as its theoretical counterpart, the control flow in the implementation corresponds to the control flow of the algorithm proved to be correct under the assumption of exact computation over the reals, and hence the validity of the combinatorial part of the computed output follows. Thus, for selective geometric problems, it is sufficient to guarantee correct decisions, since all numerical data are already part of the input.

For constructive geometric problems, new numerical data have to be computed "exactly". A representation of a real number r should be called exact only if it allows one to compute an approximation of r to whatever precision, i.e. no information has been lost. According to Yap [150] a representation of a subset of the reals is exact if it allows the exact comparison of any two real numbers in that representation. This reflects the necessity for correct comparisons in branchings steps in the exact geometric computation approach. Examples of exact representations are the representation of rationals by numerator and denominator, where both are arbitrary precision integers, and the representation of algebraic numbers by an integral polynomial P having root α and an interval that isolates α from the other roots of P. Further examples are symbolic and implicit representations. For example, rather than compute the coordinates of an intersection point of line segments explicitly, one can represent them implicitly by maintaining the intersecting segments. Another similar example is the representation of a number by an expression dag, which reflects the computation history. Allowing symbolic or implicit representation can be seen as turning a constructive geometric problem into a selective one.

As suggested in the discussion above, there are different flavors of exact geometric computation. In the last decade, much progress has been made in improving the efficiency of exact geometric computation (see also [151] and [150] for an overview).

3.1 Exact Integer and Rational Arithmetic

A number of geometric predicates in basic geometric problems include only integral expressions in their tests. Thus, if all numerical input data are integers, the evaluation of these predicates involves integers only. With the integer arithmetic provided by the hardware only overflow may occur, but no rounding errors. The problem with overflow in integral computation is abolished if arbitrary precision integer arithmetic is used. There are several software packages for arbitrary or multiple precision integers, e.g., BigNum [129], GNU MP [65], LiDIA [90], or the number type `integer` in LEDA [95, 96]. Fortune and Van Wyk [57, 58] report on experiments with such packages.

Since the integral input data are usually bounded in size, e.g., by the maximal representable `int`, there is not really a need for *arbitrary* precision integers. *Multiple* precision integer arithmetic with a fixed precision adjusted to the maximum possible integer size in the input and the degree of the integral polynomial expression arising in the computation is adequate. If the input integers have binary representation with at most b-bits, then an integer arithmetic for integers with $db + \log m + O(1)$ bits suffices to evaluate an integral polynomial expression with m monomials of degree at most d, where we assume that the coefficients of the monomials are bounded by a constant. If v is the number of numerical input data involved in a polynomial expression, then m is bounded by $(v+1)^d$.

The degree of polynomial expressions in geometric predicates has recently gained attention as an additional measure of algorithmic complexity in the design of geometric algorithms. Liotta et al. [91] investigate the degree involved in some proximity problems in 2- and 3-dimensional space, Boissonnat and Preparata [15] investigate the degree involved in line segment intersection.

Many predicates include only expressions involving operations $+, -, *, /$. In most of the problems discussed in textbooks on computational geometry [16, 31, 40, 85, 88, 92, 107, 112, 117] all predicates are of this type. Such problems are called *rational* [151].

A rational number can be exactly stored as a pair of arbitrary precision integers representing numerator and denominator respectively. Let us call this *exact rational arithmetic*. The intermediate values computed in rational problems are often solutions to systems of linear equations like the coordinates of the intersection point of two straight lines.

Division can be avoided in rational predicates, e.g., exact rational arithmetic postpones division. With exact rational arithmetic, numerator and denominator of the result of the evaluation of a rational expression are integral polynomial expressions in the numerators and denominators of the rational operands. A sign test for a rational expression can be done by two sign tests for integral polynomial expressions. Hence rational expressions in conditional tests in geometric predicates can be replaced by tests involving integral polynomial expressions.

Homogeneous coordinates known from projective geometry and computer graphics can be used to avoid division, too. In homogeneous representation, a point in d-dimensional affine space with Cartesian coordinates $(x_0, x_1, \ldots, x_{d-1})$ is represented by a vector $(hx_0, hx_1, \ldots$ $\ldots, hx_{d-1}, hx_d)$ such that $x_i = hx_i/hx_d$ for all $0 \le i \le d-1$. Note that the homogeneous representation of a point is not unique; multiplication of the homogeneous representation vector with any $\lambda \neq 0$ gives a representation of the same point. The homogenizing coordinate hx_d is a common denominator of the coordinates. For example, homogeneous representation allows division-free representation of the intersection point of two straight lines given by $a \cdot X + b \cdot Y - c = 0$ and $d \cdot X + e \cdot Y + f = 0$. The intersection point can be represented by homogeneous coordinates $(b \cdot f - c \cdot e, a \cdot f - c \cdot d, a \cdot e - b \cdot d)$.

A test including rational expressions in Cartesian coordinates transforms into a test including only polynomial expressions in homogeneous coordinates after multiplication with an appropriate product of homogenizing coordinates. Since all monomials appearing in the resulting expressions have the same degree in the homogeneous coordinates, the resulting polynomial is a homogeneous polynomial. For example, the test "$a \cdot x_0 + b \cdot x_1 + c = 0$?", which tests whether point (x_0, x_1) is on the line given by the equation $a \cdot X + b \cdot Y + c = 0$, transforms into "$a \cdot hx_0 + b \cdot hx_1 + c \cdot hx_2 = 0$?".

Many geometric predicates that do not obviously involve only integral polynomial expressions can be rewritten so that they do. Above, we have illustrated this for rational problems. Even sign tests for expressions involving square roots can be turned into a sequence of sign tests of polynomial expressions by repeated squaring [21, 91]. Therefore, multiple or arbitrary precision integer arithmetic is a powerful tool for exact geometric computation, but such integer arithmetic has to be supplied by software and is therefore much slower than the hardware-supported integer arithmetic. The actual cost of an operation on arbitrary precision integers depends on the size of the operands, more precisely on the length of their binary representation. If expressions of large depth are involved in the geometric calculations the size of the operands can increase drastically. In the literature huge slow down factors are reported if floating-point arithmetic is simply replaced by exact rational arithmetic. Karasick, Lieber, and Nackman [84] report slow-down factors of about 10 000.

While in most rational problems the depth of the involved rational expressions is a small constant, there are problems where the size of the numbers has a linear dependence on the problem size. An example is computing minimum link paths inside simple polygons [82]. Numerator and denominator of the knick-points on a minimum link path can have super-quadratic bitlength with respect to the number of polygon vertices [82]. This is by the way a good example of how strange the assumption of constant time arithmetic operations in theory may be in practice.

Fortune and Van Wyk [57, 58] noticed that in geometric computations the sizes of the integers are small to medium compared to those arising in computer algebra and number theory. Multiple precision integer packages are mainly used in these areas and hence tuned for good performance with larger integers. Consequently Fortune and Van Wyk developed LN [56], a system that generates efficient code for integer arithmetic with fairly "little" numbers. LN takes an expression and a bound on the size of the integral operands as input. The generated code is very efficient if all operands are of the same order of magnitude as the bound. For much smaller operands the generated code is clearly not optimal. LN can be used to trim integer arithmetic in an implementation of a geometric algorithm for special applications. On the other hand, LN is not useful for generating general code. Chang and Milenkovic report on the use of LN in [27].

For integral polynomial expressions, modular arithmetic [2, 86] is an alternative to arbitrary precision integer arithmetic. Let $p_0, p_1, \ldots, p_{k-1}$ be a set of integers that are pairwise relatively prime and let p be the product of the p_i. By the Chinese remainder theorem there is a one-to-one correspondence between the integers r with $-\lfloor \frac{p}{2} \rfloor \leq r < \lceil \frac{p}{2} \rceil$ and the k-tuples $(r_0, r_1, \ldots, r_{k-1})$ with $-\lfloor \frac{p_i}{2} \rfloor \leq r_i < \lceil \frac{p_i}{2} \rceil$. By the integer analog of the Lagrangian interpolation formula for polynomials [2], we have

$$r = \sum_{i=0}^{k-1} r_i s_i q_i \mod p$$

where $r_i = r \mod p_i$, $q_i = p/p_i$, and $s_i = q_i^{-1} \mod p_i$. Note that s_i exists because of the relative primality and can be computed with an extended Euclidean gcd algorithm [86]. To evaluate an expression, a set of relatively prime integers is chosen such that the product of the primes is at least twice the absolute value of the integral value of the expression. Then the expression is evaluated modulo each p_i. Finally Chinese remaindering is used to reconstruct the value of the expression.

Modular arithmetic is frequently used in number theory, but not much is known about its application to exact geometric computation. Fortune and Van Wyk [57, 58] compared modular arithmetic with multiple precision integers provided by software packages for a few basic geometric problems without observing much of a difference in the performance. Recently, however, Brönnimann et al. reported on promising results concerning the use of modular arithmetic in combination with single precision floating-point arithmetic for sign evaluation of determinants [17] and Emiris reported on the use of modular arithmetic in the computation of general dimensional convex hulls [42].

Modular arithmetic is particularly useful if intermediate results can be very large, but the final result is known to be relatively small. The drawback is that a good bound on the size of the final result must be known in order to choose sufficiently many relatively prime integers, but not too many.

3.2 Adaptive Evaluation

Replacing exact arithmetic, on which the correctness of a geometric algorithm is based, by imprecise finite-precision arithmetic usually works in practice for many of the given input data and fails only occasionally. Thus always computing exact values would put a burden on the algorithm that is rarely really needed.

Adaptive evaluation (also called lazy evaluation) is guided by the rule

Why compute something that is never used,

so why compute numbers to high precision, before you know that this precision is actually needed.

The simplest form of adaptive evaluation is a *floating-point filter*. The idea of floating-point filters is to filter out those computations where floating-point computation gives the correct result. This technique has been successfully used in exact geometric computation [34, 57, 58, 84, 93, 94]. Floating-point filters make use of the fast hardware-supported floating-point arithmetic. A filter simply takes a bound on the error of the floating-point computation and compares the absolute value of the computed numerical value to the error bound. If the error bound is smaller, the computed approximation and the exact value have the same sign. Only if it is not certified by the error bound that the floating-point evaluation has led to a correct decision, the expression considered in the branching step is reevaluated, for instance, with exact arithmetic.

Error bounds can be computed a priori if specific information on the input data is available, e.g., if all input data are integers from a bounded range, for instance, the range of integers representable in a computer word. Such so-called static filters require only little additional effort at run time, just one additional test per branching, plus the refined reevaluation in the worst case. Dynamic filters compute an error bound on the fly parallel to the evaluation in floating-point arithmetic. Since they take the actual values of the operands into account and not only bounds derived from the bounds on the input data, the estimates for the error involved in the floating-point computation can be much tighter than in a static filter. In the error computation one can put emphasis on speed or on precision. The former makes arithmetic operations more efficient while the latter lets more floating-point computations pass a test. Semi-dynamic filters partially precompute the error bound a priori. Mehlhorn and Näher [93] use such semi-dynamic filters in their implementation of the Bentley-Ottmann plane sweep algorithm [13] for computing the intersections among a set of line segments in the plane.

Note the difference between static filters and heuristic epsilons. In both cases approximations to a numerical values are compared to some small values. If the computed approximation is larger than the error bound or $\varepsilon_{\text{magic}}$, respectively, the behavior is identical. The program continues based on the (in the former case verified) assumption that the computed floating-point value has the correct sign. If, however, the computed approximate value is too small, the behavior is completely different. Epsilon-tweaking assumes that the actual value is zero, which might be wrong, while a floating-point filter invokes a more expensive computation finally leading to a correct decision.

Using only error bounds, a floating-point filter rarely works for expressions whose value is actually zero, because both the computed approximation and the error bound have to be zero to certify sign zero. To detect sign zero, one can use "certified epsilons" described in Section 3.5, or use a special procedure to test an expression for zero, e.g. [14], or one might use exact arithmetic.

If a filter fails, a refined filter can be used. A refined filter might compute a tighter error bound or use a floating-point arithmetic with larger mantissa and thereby get better approximations and smaller error bounds. This step can be iterated. Composition of more and more refined filters leads to an adaptive evaluation scheme. Such schemes are called adaptive, because they adapt the used precision to the size of the value of the expression to be evaluated.

For orientation predicates and incircle tests in two- and three-dimensional space Shewchuk [130, 131] presents such an adaptive (or lazy) evaluation scheme. It uses an exact[3] representation of values resulting from expressions over floating-point numbers involving only additions, subtractions, and multiplications as a symbolic sum of floating-point numbers. Computation with numbers in this representation, called *expanded doubles* in [130], is based on the interesting results of Priest [118, 119] and Dekker [33] on extending the precision of floating-point computation. An adapted combination of these techniques allows one to reuse values computed in previous filtering steps in later filtering steps.

For integral expressions scalar products delivering exactly rounded results can be used in floating-point filters to get best possible floating-point approximations. Ottmann et al. [113] first used exactly rounding scalar products to solve precision problems in geometric computation.

Number representations supporting recomputation with higher precision are very useful for adaptive evaluation. The LEA[4] system [12] provides lazy evaluation for rational computation. In this system, numbers are represented by intervals and expression dags that reflect their creation history. Initially only a low precision representation is calculated using interval arithmetic, cf. Section 3.3. Only if decisions can't be made with the current precision, repeatedly representations with increased precision are computed by redoing the computation along the expression dag with refined intervals for the operands. If the interval representation can't be refined anymore with floating-point evaluation, exact rational arithmetic is used to solve the decision problem.

Another approach based on expression dags is described by Yap and Dubé [39, 150, 151]. In this approach the precision used to evaluate the operands is not systematically increased, but the increase is demanded by the intended increase in the precision of the result. The data type `real` in LEDA [23] stores the creation history in expression dags, too, and uses floating-

[3] if neither underflow nor overflow occurs

[4] LEA [12] should not to be confused with LEDA [95, 96]

point approximations and errors bounds as first approximations. The strategy of repeatedly increasing the precision is similar to [39, 150, 151]. In both approaches software-based multiple precision floating-point arithmetic with a mantissa length that can be arbitrarily chosen and an unbounded exponent is used to compute representations with higher precision. Furthermore, both approaches include square root operations besides $+, -, *, /$. The `reals` now provide k-th root operations as well [96].

3.3 Interval Arithmetic

Approximation and error bound define an interval that contains the exact value. Interval arithmetic [3, 104, 105] is another method to get an interval with this property. In interval arithmetic real numbers are represented by intervals, whose endpoints are floating-point numbers. The interval representing the result of an operation is computed by floating-point operations on the endpoints of the intervals representing the operands. For example, the lower endpoint of the interval representing the result of an addition is the sum of the lower endpoints of the intervals of the summands. Since this floating-point addition might be inexact, either the rounding mode is changed to rounding toward $-\infty$ before addition or a correction term is subtracted. For interval arithmetic, rounding modes toward ∞ and toward $-\infty$ are very useful. See, for example, [106, 137] for applications of interval methods to geometric computing. The combination of exact rational arithmetic with interval arithmetic based on fast floating-point computation has been pioneered by Karasick, Lieber and Nackman [84] to geometric computing.

A refinement of standard interval arithmetic is the so-called affine arithmetic proposed by Comba and Stolfi [30]. While standard interval arithmetic assumes that the unknown values of operands and subexpressions can vary independently, affine arithmetic keeps track of first-order dependencies and takes these into account. Thereby error explosion can often be avoided and tighter bounds on the computed quantities can be achieved. An extreme example is computing $x - x$ where for x some interval $[x.lo, x.hi]$ is given. Standard interval arithmetic would compute the interval $[x.lo - x.hi, x.hi - x.lo]$, while affine arithmetic gives the true range $[0, 0]$.

3.4 Exact Sign of Determinant

Many geometric primitives can be formulated as sign computations of determinants. The classical example of such a primitive is the orientation test, which in two-dimensional space determines whether a given sequence of three points is a clockwise or a counterclockwise turn or whether they are collinear. Another example is the incircle test used in the construction of Voronoi diagrams of points.

Recently some effort has been focused on exact sign determination. Clarkson [29] gives an algorithm to evaluate the sign of a determinant of a $d \times d$ matrix with integer entries using floating-point arithmetic. His algorithm is a variant of the modified Graham-Schmidt orthogonalization. In his variant, scaling is used to improve the conditioning of the matrix. Since only positive scaling factors are used, the sign of the determinant does not change. Clarkson shows that only $b + O(d)$ bits are required, if all entries are b-bit integers. Hence, for small dimensional matrices his algorithm can be used to evaluate the sign of the determinant with fast hardware floating-point arithmetic.

Avnaim et al. [5] consider determinants of small matrices with integer entries, too. They present algorithms to compute the sign of 2×2 and 3×3 matrices with b-bit integer entries using precision b and $b+1$ only, respectively. Brönnimann and Yvinec [18] extend the method of [5] to $d \times d$ matrices and present a variant of Clarkson's method. The new version of Clarkson's method allows for a simplified analysis. Furthermore, Shewchuk's work on adaptive evaluation [131] is focussed on predicates evaluated by sign of determinant computation. We already mentioned the use of modular arithmetic combined with floating-point arithmetic to compute the sign of determinants of integer matrices [17].

3.5 Certified Epsilons

While the order of two different numbers can be found by computing sufficiently close approximations, it is not so straightforward to determine whether two numbers are equal or, equivalently, whether the value of an expression is zero. From a theoretical point of view arithmetic expressions arising in geometric predicates are expressions over the reals. Hence the value of an expression can in general get arbitrarily close to zero if the variable operands are replaced by arbitrary real numbers. In practice the numerical input data originate from a finite, discrete subset of the reals, namely a finite subset of the integers or a finite set of floating-point numbers, i.e., a finite subset of the rational numbers. The finiteness of such input excludes arbitrarily small absolute non-zero values for expressions of bounded depth. There is a gap between zero and other values that a parameterized expression can take on. A separation bound for an arithmetic expression E is a lower bound on the size of this gap. Besides the finiteness of the number of possible numerical inputs, the coarseness of the input data can generate a gap between zero and other values taken on. A straightforward example is integral expressions. If all operands are integers the number 1 is clearly a separation bound.

Once a separation bound is available it is clear how to decide whether the value of an expression is zero or not. Representations with repeatedly increased precision are computed until either the error bound on the current approximation is less than the absolute value of the approximation or their sum is less than the separation bound. In the phrasing of interval arithmetic, it means to refine the interval until neither zero nor the separation bound nor its negative are contained in the interval.

How can we get separation bounds without computing the exact value or an approximation and an error bound? Most geometric computations are on linear objects and involve only basic arithmetic operations over the rational numbers. In distance computations and operations on nonlinear objects like circles and parabolas, square root operations are used as well. For the rational numerical input data arising in practice, expressions over the operations $+, -, *, /, \sqrt{\ }$ take on only algebraic values.

Let E be an expression involving square roots. Furthermore we assume that all operands are integers. We use $\alpha(E)$ to denote the algebraic value of expression E. Computer algebra provides bounds for the size of the roots of polynomials with integral coefficients. These bounds involve quantities used to describe the complexity of an integral polynomial, e.g., degree, maximum coefficient size, or less well-known quantities like height or measure of a polynomial. Once an integral polynomial with root $\alpha(E)$ is known the root bounds from computer algebra give us separation bounds. In general, however, we don't have a polynomial having root $\alpha(E)$ at hand. Fortunately, all we need to apply the root bounds are bounds on the quantities involved in the root bounds. Upper bounds on these quantities for some polynomial having root $\alpha(E)$ can be derived automatically from an expression E.

Recursive formulas leading to separation bounds for an expression involving square root

	$M(E)$	$deg(E)$
integer n	$\|n\|$	1
$E_1 + E_2$	$2^{deg(E_1)deg(E_2)} M(E_1)^{deg(E_2)} M(E_2)^{deg(E_1)}$	$deg(E_1) \cdot deg(E_2)$
$E_1 - E_2$	$2^{deg(E_1)deg(E_2)} M(E_1)^{deg(E_2)} M(E_2)^{deg(E_1)}$	$deg(E_1) \cdot deg(E_2)$
$E_1 \cdot E_2$	$M(E_1)^{deg(E_2)} M(E_2)^{deg(E_1)}$	$deg(E_1) \cdot deg(E_2)$
E_1 / E_2	$M(E_1)^{deg(E_2)} M(E_2)^{deg(E_1)}$	$deg(E_1) \cdot deg(E_2)$
$\sqrt{E_1}$	$M(E_1)$	$2 \cdot deg(E_1)$

Table 1: Automatic computation of separation bounds for expressions involving square roots based on the measure of a polynomial

operations are given in [151]. The formulas deliver a bound on the maximum absolute value of the coefficients of an integral polynomial having root $\alpha(E)$. By a result of Cauchy, this gives a separation bound. In [151], this bound is called *height-degree bound*.

Mignotte discusses identification of algebraic numbers given by expressions involving square roots in [97]. The measure of a polynomial [98] can also be used for automatic computation of a root bound. Table 1 gives the rules for (over)estimating measure and degree of an integral polynomial having root $\alpha(E)$. We have $\alpha(E) = 0$ or $|\alpha(E)| \geq M(E)^{-1}$. This bound, called *degree-measure bound*, is never worse than the height-degree bound.

In [24] Canny considers isolated solutions of systems of polynomial equations in several variables with integral coefficients. He gives bounds on the absolute values of the non-zero components of an isolated solution vector. The bound depends on the number of variables, the maximum total degree d of the multivariate integral polynomials in the system and their maximum coefficient size c. Canny shows that the absolute value of a component of an isolated solution of a system of n integral polynomial equations in n variables is either zero or at least $(3dc)^{-nd^n}$ [24, 25]. Although Canny solves a much more general problem, his bounds can be used to get fairly good separation bounds for expressions involving square roots, cf. [20].

Burnikel et al. [20] have shown that

$$\alpha(E) \geq \left(u(E)^{2^{2k(E)-1}} l(E) \right)^{-1}$$

where $k(E)$ is the number of (distinct) square root operations in E and the quantities $u(E)$ and $l(E)$ are defined as given in Table 2. Note that $u(E)$ and $l(E)$ are simply the numerator and denominator of an expression obtained by replacing in E all $+$ by $-$ and all integers by their absolute value. If E is division-free and $\alpha(E)$ is non-zero, then $\alpha(E) \geq u(E)^{1-2^{k(E)-1}}$. It is shown in [20] that this bound is never worse than the degree-measure bound and the polynomial system bound for division-free expressions.

The bound given in [20] as well as the bound given in [151] involve square root operations. Hence they are not easily computable. In practice one computes ceilings of the results to get integers [151] or maintains integer bounds logarithmically [20, 23].

The number type `real` [23, 96] in LEDA and the `Real/Expr`-package [38, 114] provide exact computation (in C++) for expressions with operations $+$, $-$, $\cdot$, $/$ and $\sqrt{\ }$ and initially integral operands, using techniques described above. In particular, the recent version of the `real`s in LEDA [96] uses the bounds given in [20].

	$u(E)$	$l(E)$
integer n	$\lvert n \rvert$	1
$E_1 + E_2$	$u(E_1) \cdot l(E_2) + l(E_1) \cdot u(E_2)$	$l(E_1) \cdot l(E_2)$
$E_1 - E_2$	$u(E_1) \cdot l(E_2) + l(E_1) \cdot u(E_2)$	$l(E_1) \cdot l(E_2)$
$E_1 \cdot E_2$	$u(E_1) \cdot u(E_2)$	$l(E_1) \cdot l(E_2)$
E_1 / E_2	$u(E_1) \cdot l(E_2)$	$l(E_1) \cdot u(E_2)$
$\sqrt{E_1}$	$\sqrt{u(E_1)}$	$\sqrt{l(E_1)}$

Table 2: Recursive formulas for quantities $u(E)$ and $l(E)$ of an arithmetic expression involving square roots.

Note the difference between separation bounds and $\varepsilon_{\text{magic}}$s in epsilon tweaking. In epsilon-tweaking a test for zero is replaced by the test "$|\tilde{E}| < \varepsilon_{\text{magic}}$?". With separation bounds it becomes "$|\tilde{E}| < sep(E) - E_{\text{error}}$?" where $sep(E)$ is a separation bound and E_{error} is a bound on the error accumulated in the evaluation of E. The difference is that the latter term is self-adjusting, it is based on an error bound, and justified; it is guaranteed that the result is zero, if the condition is satisfied. While $\varepsilon_{\text{magic}}$ is always positive, it might happen that the accumulated error is so large that $sep(E) - E_{\text{error}}$ is negative. Last but not least, the conclusion is different if the test is not satisfied. Epsilon-tweaking concludes that the number is non-zero if it is larger than $\varepsilon_{\text{magic}}$ while the use of separation bounds allows this conclusion only if $|\tilde{E}| \geq E_{\text{error}}$.

4 Geometric Computation with Imprecision

In this section we look at the design and implementation of geometric algorithms with imprecision calculations. With potentially imprecise computations we cannot hope to always get the exact result. But even if the result is not the exact result for the considered problem instance, it still can be meaningful. An algorithm that computes the exact result for a very similar problem instance can be sufficient for an application, since the input data might be known not to be accurate either. This observation motivates the definition of robustness given in Section 1.1 and below Section 4.1. In addition to the existence of a perturbation of the input data, for which the computed result is correct, Fortune's definition of robustness and stability [51] requires that the implementation of an algorithm would compute the exact result, if all computations were precise. His definition reflects the attempt to save the (theoretical) correctness proof. It implies that all degenerate cases have to be handled. In contrast to this, Sugihara [140] avoids handling degenerate cases at all, see also Section 4.3. Even if a degeneracy is detected it is treated like a non-degeneracy by changing the sign from zero to positive or negative. The justification for this approach is again inaccuracy of the input data.

The output of an algorithm might be useful although it is not a correct output for any perturbation of the input. In some situations it might be feasible to allow perturbation of the output as well. For example, for some applications it might be sufficient that the output of a two-dimensional convex hull algorithm is a nearly convex polygon while other applications require convexity. Sometimes requirements are relaxed to allow "more general" perturbations of the input data. Robustness and stability are then defined with respect to

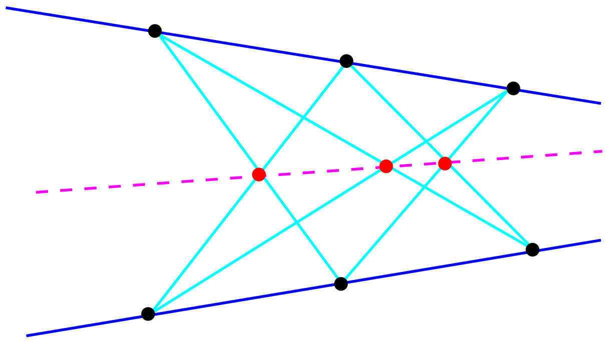

Figure 5: Pappus theorem is an example where the result of some orientation tests for points in the plane is determined by the result of other orientation tests. Collinearity of the points on the top line and the bottom line implies collinearity of the three intersection points in the middle.

the weaker problem formulation, cf. Section 4.1. For example, Fortune's and Milenkovic's line arrangement algorithm [55] computes a combinatorial arrangement that is realizable by pseudolines but not necessarily by straight lines. Shewchuk [130] suggests calling an algorithm *quasi-robust* if it computes useful information but not a correct output for any perturbation of the input.

For many implementations of geometric primitives it is easy to show that the computed result is correct for some perturbation of the input. The major problem in the implementation with imprecise predicates is their combination. The basic predicates evaluated in an execution of an algorithm operate on the same set of data and and hence might be dependent. Furthermore, the results of dependent geometric predicates might be mutually exclusive, i.e., there might be no small perturbation leading to correctness for all predicates. Hence an algorithm might get into an inconsistent state, a state that could not be reached from any input with correct evaluation. That is where a relaxation of the problem helps. An illegal state can be a legal state for a similar problem with weaker restrictions, e.g., a state illegal for an algorithm computing an arrangement of straight lines can be legal for arrangements of pseudolines.

Avoiding inconsistencies among the decisions is a primary goal in achieving robustness in implementations with imprecise predicates. Consistency is a non-issue if an algorithm never evaluates a basic predicate whose outcome is implied by the results of previous evaluations of basic predicates. Such an algorithm is called *parsimonious* [51, 87].

In general it can be very hard to achieve consistency with previous decisions by detecting whether the outcome of a predicate can be deduced from previously evaluated predicates. A well known illustration for this fact is Pappus theorem, cf. Fig. 5. Indeed, checking whether the outcome of an orientation test is implied by previous tests on the given set of points is as hard as the existential theory of the reals [51, 67].

The following sections present some design principles for robustness under computation with imprecision.

4.1 Representation and Model Approach

The representation and model view formalizes the "compute the correct solution for a related input" idea. It distinguishes real mathematical objects, the *models*, and their computer *representations*. A geometric problem $\mathcal{P}$ defines a mapping between models, while a computer program A leads to a mapping between representations. For instance, subtraction maps a pair of real numbers to a real number while its counterpart on a computer maps a pair of computer representations of the mathematical object real number, namely floating-point numbers, to a representation, a floating-point number.

For the ideal one-to-one correspondence between representations and models a computer algorithm is correct if the model corresponding to the computed output representation is the solution to the problem for the model corresponding to the input representation. As with real numbers and floating-point numbers, the correspondence between mathematical models and computer representations is normally not one-to-one because of the finite nature of computer representations. To take this approximation behavior into account correctness is replaced by robustness as follows: A computer algorithm $A : \mathcal{I}_{\mathrm{rep}} \to \mathcal{O}_{\mathrm{rep}}$ for a geometric problem $\mathcal{P} : \mathcal{I} \to \mathcal{O}$ is called robust, if for every computer representation x_{rep} in the set of inputs $\mathcal{I}_{\mathrm{rep}}$, there is a model x in $\mathcal{I}$ corresponding to x_{rep}, such that $\mathcal{P}(x)$ is among the models in $\mathcal{O}$ corresponding to the computed output $A(x_{\mathrm{rep}})$, see Fig. 6. The obvious way to prove

$$\begin{tikzcd} \mathcal{I} \arrow[r, "\mathcal{P}"] & \mathcal{O} \arrow[d] \\ \mathcal{I}_{\mathrm{rep}} \arrow[u] \arrow[r, "A"] & \mathcal{O}_{\mathrm{rep}} \end{tikzcd}$$

Figure 6: A geometric problem is defined on models, while a computer algorithm works on representations.

robustness of a computer algorithm in the sense above is to show that there is always a model for which the computer algorithm takes the correct decisions. But this is often a highly non-trivial task.

Of course, this definition of robustness depends to a large extent on the interpretation of "correspondence" between representations and models for the input and the output part. Generous definitions of correspondence in the output part make it easier to prove "robustness" of an algorithm. Following Shewchuk's suggestion, algorithms with a fairly generous interpretation of robustness should rather be called quasi-robust, because the output they compute might be less useful than expected.

Hoffmann, Hopcroft, and Karasick introduced the "representation and model" formalization in [74], our exposition follows Stewart [136]. Hoffmann, Hopcroft, and Karasick gave an algorithm for intersection of polygons and proved its robustness. However, the underlying correspondence between computer representations and models of polygons was fairly loose. The edges of a model need not be close to the edges of the representation. Furthermore, both simple and non-simple polygons could model a representation. Thus for simple polygons the computed intersection polygon(s) need not be simple. In [77] Hopcroft and Kahn consider robust intersection of a convex polyhedron with a halfspace. Again, the computed output can be arbitrarily far away from the real intersection polyhedron.

Milenkovic's *hidden variable method* [99] fits into the representation and model scheme as well. In the hidden variable method the representation provides a structure with certain topological properties (plus finite precision approximations of the numerical values). A corresponding model provides the hidden (infinite precision) numerical data and has the same topological structure as the representation. In [99], Milenkovic applies the hidden variable method to the computation of line arrangements. An arrangement representation in $\mathcal{O}_{\text{rep}}$ consists of combinatorial data describing the topology of the arrangement and approximate representations for the vertices of the arrangement. A model has the same topology as the corresponding representation, but may have different vertex locations. Since the computed topology might not be realizable by straight lines, the lines in a model need not be straight, but they must have certain monotonicity properties and be close to the straight lines in $\mathcal{I}_{\text{rep}}$.

In [136] Stewart proposes *local robustness* as an alternative for problems for which robustness (in the representation and model sense) is inherently difficult to achieve. Local robustness no longer requires that an algorithm is robust with respect to all problem instances. An algorithm is called locally robust for a set of features, if it is robust for all inputs consisting of exactly those features. Stewart claims, that appropriate feature sets can be chosen such that algorithms which are locally robust algorithms with respect to these feature sets are very unlikely to fail in practice. He presents locally robust algorithms for polyhedral intersection and polyhedral arrangements.

4.2 Epsilon Geometry

An interesting theoretical framework for the investigation of imprecision in geometric computation is *epsilon geometry* introduced by Guibas, Salesin, and Stolfi [69]. Instead of a Boolean value, an epsilon predicate returns a real number that gives some information "how much" the input satisfies the predicate. Epsilon geometry assumes that the size of a perturbation can be measured by a non-negative real number and that only the identity has size zero.

If an input does not satisfy a predicate, the "truth value" of an epsilon predicate is the size of the smallest perturbation producing a perturbed input that satisfies the predicate. If the input satisfies a predicate, the "truth value" is the non-positive number ϱ if the predicate is still satisfied after applying any perturbations of size at most $-\varrho$. In [69] epsilon predicates are combined with interval arithmetic. Imprecise evaluations of epsilon predicates compute a lower and an upper bound on the "truth value" of an epsilon predicate. Guibas, Salesin, and Stolfi compose basic epsilon predicates to less simple predicates. Unfortunately epsilon geometry has been applied successfully only to a few basic geometric primitives [69] and the computation of planar convex hulls [70]. Reasoning in the epsilon geometry framework seems to be difficult.

4.3 Topology-Oriented Approach

In order to avoid inconsistent decisions the topology-oriented approach places higher priority on topological and combinatorial data than on numerical values. Whenever numerical computations would lead to decisions violating topology, the decision is replaced by a topology-conforming decision. Usually, violation of topology is not tested directly, but a set of rules is given and it is shown that following these rules ensures the desired topological properties. This approach guarantees topologically consistent output, i.e. valid combinatorial data of the output, but the computed numerical values of the output might not be corresponding to the combinatorial data. For instance, in [144] the computed graph structure representing the Voronoi diagram will always be planar, but the computed coordinates of the vertices might not give a planar embedding.

Typically topology-oriented approaches do not treat degeneracies explicitly. They assume sign computations not to produce sign zero. If the numerical value computed in a sign computation is zero, it is replaced by a positive or a negative value, whatever is consistent with the current topology.

The topology-oriented approach can lead to amazingly robust algorithms. The algorithms never crash or loop for ever and they compute output having essential combinatorial properties. For instance, the Voronoi diagram algorithm presented in [144] produces some planar graph even if in all decision steps involving sign computations the sign is chosen at random! Of course, "closeness" of the computed output to the correct solution is not guaranteed in this case. Usually it is argued that the computed output comes closer to the correct one if higher precision is used, and, furthermore, that it is the correct one, if the precision is sufficiently high and there are no degeneracies.

Sugihara et al. used the topological-oriented approach in several algorithms for computing Voronoi diagrams [79, 111, 144, 145, 146], polyhedral modeling problems [141, 142, 143], and 3-dimensional convex hull [103].

Results on computation with imprecision are usually not unequivocally classifiable under the set of design principles described in Sections 4.1 to 4.5. For example, Milenkovic's hidden variable method can be seen as an topology-oriented approach, too, because the topological structure of the output representation has to be respected by every model corresponding to this representation. Thereby, topology gets priority over numerical data, which is characteristic for the topology-oriented approach as well.

4.4 Axiomatic Approach

In [122, 123] Schorn proposes what he calls the *axiomatic approach*. The idea is to investigate which properties of primitive operations are essential for a correctness proof of an algorithm and to find algorithm invariants that are based on these properties only.

One of the algorithms considered in [122] is computing a closest pair of a set of points S by plane sweep [72]. Instead of a closest pair, the distance δ_S of a closest pair is computed. In his implementation Schorn uses distance functions $d(p,q)$, $d_x(p,q)$, $d_y(p,q)$, and $d'_y(p,q)$ on points $p=(p_x,p_y)$ and $q=(q_x,q_y)$ in the plane. In an exact implementation these functions would compute $\sqrt{(p_x-q_x)^2+(p_y-q_y)^2}$, p_x-q_x, p_y-q_y, and q_y-p_y, respectively. Schorn lists properties for these functions that are essential for a correctness proof: First, they must have some monotonicity properties. d_x must be monotone with respect to the x-coordinate of its first argument, i.e., $[p_x \geq p'_x \Rightarrow d_x(p,q) \geq d_x(p',q)]$ holds, and inverse monotone in the x-coordinate of its second argument, i.e. $[q_x \leq q'_x \Rightarrow d_x(p,q) \geq d_x(p,q')]$ holds. Similarly, $[q_y \leq q'_y \Rightarrow d_y(p,q) \geq d_y(p,q')]$ and $[q_y \geq q'_y \Rightarrow d'_y(p,q) \geq d'_y(p,q')]$ must hold for d_y and d'_y, respectively. Second, d_x, d_y, and d'_y must be "bounded by d", more precisely, $[p_x \geq q_x \Rightarrow d(p,q) \geq d_x(p,q)]$, $[p_y \geq q_y \Rightarrow d(p,q) \geq d_y(p,q)]$, and $[p_y \leq q_y \Rightarrow d(p,q) \geq d'_y(p,q)]$ must hold. Finally, d must be symmetric, i.e., $d(p,q)=d(q,p)$. These properties, called axioms in [122], are sufficient to prove that for the δ computed by Schorn's plane sweep implementation

$$\delta = \min_{s,t \in S} d(s,t)$$

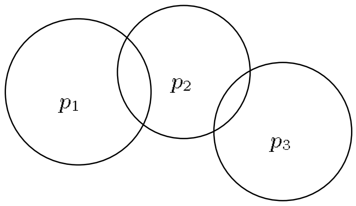

Figure 7: Coincidence inconsistency of points with tolerance regions. If points are considered to be coincident if there tolerance regions overlap, then p_1 and p_2 are coincident and so are p_2 and p_3, but p_1 and p_3 are not.

holds. No matter what d, d_x, d_y, and d'_y are, as long as they satisfy all axioms, $\min_{s,t \in S} d(s,t)$ is computed by the sweep. In particular, if exact distance functions are used, the correct distance of a closest pair would be computed. Schorn uses floating-point implementations of the distance functions d, d_x, d_y, and d'_y. He shows that they have the desired properties and that they guarantee a relative error of at most $8\varepsilon_{\text{prec}}$ in the computed approximation for δ_S, where $\varepsilon_{\text{prec}}$ is machine epsilon.

Further geometric problems to which the axiomatic approach is applied in [122, 123] to achieve robustness are: finding pairs of intersecting line segments and computing the winding number of a point with respect to a not necessarily simple polygon. The latter involves point in polygon testing as a special case.

4.5 Tolerance-Based Approach

This approach associates tolerances to geometric objects in order to represent uncertainties. This is a generalization of the representation of a numerical value by an approximation and an error bound or an interval. Tolerance-based approaches can be seen as a special variant of the representation and model design principle. The tolerances associated with geometric objects restrict the correspondence between representation and model. A model can correspond to a representation only if it satisfies the tolerance constraints associated to the representation.

A goal in processing geometric data with a tolerance-based approach is to keep the data in a consistent state in order to ensure the existence of a model. For example, points with associated tolerance regions should have a coincidence relation that is reflexive and transitive, see Fig. 7. If inconsistencies arise, the tolerance regions have to be adjusted, either by shrinking them through recomputation of the relevant data with higher precision, or by splitting or merging objects and their tolerance regions. Tolerance-based approaches usually maintain additional neighborhood information on the location of the objects to enable consistency checking. In the example given in Fig. 8 one has to detect that after merging points p_2 and p_3 into one point with an enlarged tolerance region an inconsistency with p_1 arises.

Pullar [120] discusses consequences of using tolerance circles to point coincidence and point clustering problems. Segal [125] uses a tolerance-based approach in the boundary evaluation in constructive solid geometry. Fang and Brüderlin [48] consider polyhedral modeling as well. They present two versions, a more strict version called the *linear model* where the models corresponding to a representation must be linear as well, and the less strict *curve model* that allows for curved models as well and hence requires less efforts to ensure consistency.

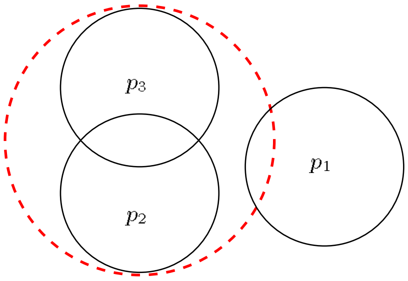

Figure 8: Processing points with tolerance regions requires backtracking if points p_2 and p_3 are merged after p_1 has been processed.

4.6 Further and More Specific Approaches

For modeling polygonal regions in the plane Milenkovic [99] uses a technique called *data normalization* to modify the input such that it can be processed with imprecise arithmetic, more precisely such that all finite precision operations on the normalized data give the correct result. The permitted modification operations are vertex shifting (given a polygon P and a vertex v, move all vertices of P with distance less than a certain ε onto v) and edge cracking (given a segment $s = AB$ and a set V of points, each point with distance at most a certain ε to s, replace $s = AB$ by a polyline from A to B whose vertex set is $V \cup \{A, B\}$).

For some basic geometric problems there are stable, robust, or quasi-robust computer algorithms. In Table 3 we group results on robustness with imprecise computation in a problem-oriented way.

Convex Hull:	
2-dimensional	[28] [59] [70] [80] [89]
3-dimensional	[103]
d-dimensional	[9, 10]
Operations on Polygonal Objects:	
line arrangements	[55] [99] [100]
intersection of polygons	[74]
intersection of polyhedra	[75] [77] [136] [142]
2-d modeling	[48] [99] [101]
3-d modeling	[83] [125] [127] [135] [141] [143]
polyhedral decomposition	[6, 7] [126]
point location	[11] [49] [122] [134]
line segment intersection	[66] [100] [113] [122] [139]
triangulation	[51]
Delaunay and Voronoi Diagrams:	
points in 2-d	[53] [78] [79] [111] [140] [144, 145] [146]
points in 3-d	[35] [79]

Table 3: Some robustness results for basic geometric problems with imprecise computation. Note that exact methods are not listed here.

The techniques used in the algorithms cited in this section and the reasoning processes used to prove robustness are fairly problem specific and it seems unlikely that they can be easily transferred to other geometric problems.

5 Related Issues

In this section we first look at some issues that are closely related to precision and robustness: degeneracies, inaccurate data, and rounding. Finally, we briefly address precision and robustness in computational geometry libraries.

5.1 Degeneracy

Degeneracy is closely related to precision and robustness, since precision problems are caused by degenerate and nearly degenerate configurations in the input. Typical cases of degeneracy are four cocircular points, three collinear points, or two points with the same ordinate. Theoretical papers on computational geometry often assume the input in general position and leave the "straightforward" handling of special cases to the reader. This might make the presentation of an algorithm more readable, but it can put a huge burden on the implementor, because the handling of degeneracies is often less straightforward than claimed.

In Section 2 we viewed a geometric problem as a mapping from a set of permitted input data, consisting of a combinatorial and a numerical part, to a set of valid output data, consisting of a combinatorial and a numerical part. We now assume that the combinatorial part of the input is trivial, i.e. just a sequencing of data or so, such that we can view a geometric problem $\mathcal{P}$ as a function from $\mathbb{R}^{nd}$ to $C_{\text{out}} \times \mathbb{R}^m$, where n, m, and d are integers and C_{out} is some discrete space, modeling the combinatorial part of the output, e.g., a planar graph or a face incidence lattice. A problem instance $x \in \mathbb{R}^{nd}$, which for concreteness we view as n points in d-dimensional space, is called *degenerate* if $\mathcal{P}$ is discontinuous at x. For example, if $d = 2$ and $\mathcal{P}(x)$ is the Voronoi diagram of x, i.e., a straight-line planar graph together with coordinates for its vertices, then x is degenerate iff x contains four cocircular points. An instance x is called *degenerate with respect to some algorithm A* if the computation of A on input x contains a sign test with outcome zero. Clearly[5], if A solves $\mathcal{P}$ and x is a degenerate problem instance then x is also degenerate for A.

Symbolic perturbation schemes introduced to computational geometry by Edelsbrunner and Mücke [41], refined by Emiris and Canny [44, 43] and Emiris, Canny, and Seidel [45] and extended by Yap [147, 148], have been proposed to abolish the handling of degeneracies, see also [128]. With these schemes, the input is perturbed symbolically, e.g., Emiris and Canny [43] propose to replace the j-th coordinate x_{ij} of the i-th input point by $x_{ij} + \varepsilon \cdot i^j$, where ε is a positive infinitesimal, and the computation is carried out on the perturbed input. All intermediate results are now polynomials in ε. It can be shown that the Emiris and Canny scheme removes many geometric degeneracies, e.g., collinearity of three points, at only a constant factor increase in running time. The same statement holds for the other perturbation schemes, although with a larger constant of proportionality. Exact computation is a prerequisite for applying these techniques [151].

The handling of degeneracies and the use of symbolic perturbation schemes are a point of controversy in the computational geometry literature, see [22, 124, 128]. Symbolic perturba-

[5] This assumes all functions evaluated in sign tests to be continuous functions of the inputs.

tion is a fairly general technique that abolishes the handling of degenerate cases and it can be very useful [45]. However, for degenerate input x, not $\mathcal{P}(x)$, but the limit of $\mathcal{P}(x(\varepsilon))$ for $\epsilon \to 0$ is computed. This may or may not be sufficient. The complexity of the postprocessing required to retrieve the answer $\mathcal{P}(x)$ for a degenerate input x from the answer $\mathcal{P}(x(\varepsilon))$ to the perturbed input $x(\varepsilon)$ can be significant. Burnikel et al. claim in [22] that for many geometric problems algorithms handling degeneracies directly are only moderately more complex than algorithms assuming non-degenerate inputs. Furthermore, they show that perturbation schemes may incur a significant loss in efficiency, since the computed output for the symbolically perturbed input may be significantly larger than the actual solution. Burnikel et al. use line segment intersection and convex hull (in arbitrary dimensions) as examples.

Halperin and Shelton [71] combine a (non-symbolic) perturbation scheme with floating-point arithmetic to compute an arrangement of circles on a sphere, where the circles on the sphere result from intersection of the sphere with other spheres. They use their algorithm in molecular modeling. Since the given sphere locations are not accurate anyway in the molecular modeling application, perturbation doesn't harm.

Sometimes, the term robustness is also used with respect to degeneracies. Dey et al. [35] define robustness as the ability of a geometric algorithm to deal with degeneracies and "inaccuracies" during various numerical computations. The definition of robustness in [122] is similar.

5.2 Inaccurate Data

In practice, many geometric data is known to be inaccurate, for instance geometric data obtained by measuring real world data. Since imprecise arithmetic also introduces uncertainty, processing geometric objects computed with imprecise computation and processing of real world data known to be potentially inaccurate are highly related issues.

Treating inaccurate data as exact data works with exact geometric computation as long as the input data are consistent. If not, we are in a situation similar to computation with imprecision. An algorithm might get into states it was not supposed to get in and which it therefore cannot handle. This similarity has led researchers to advocate imprecise computation and to attack both inconsistencies arising from imprecise computation and inconsistencies due to inaccurate data uniformly. In this approach, however, it is not clear whether errors in the output are caused by precision problems during computation or inaccuracies in the data. Source errors and processing errors become indistinguishable. Exact computation, on the other hand, only assures that inconsistencies are due to faulty data. But knowing that an error was caused by a source error does not at all tell you how to proceed. Tolerance-based approaches discussed in Section 4.5 are a natural choice to deal with inaccurate data. As with computation with imprecision, a lot of research on modeling and handling uncertainty in geometric data is still needed.

5.3 Rounding

The complexity, e.g., the bit-length of integers, of numerical data in the output of algorithms for constructive geometric problems is usually higher than that of the input data. Thus cascading geometric computations can result in expensive arithmetic operations. If the cost caused by increased precision resulting from cascaded computation is not tolerable, precision must be decreased by rounding the geometric output data. The goal in rounding is not to

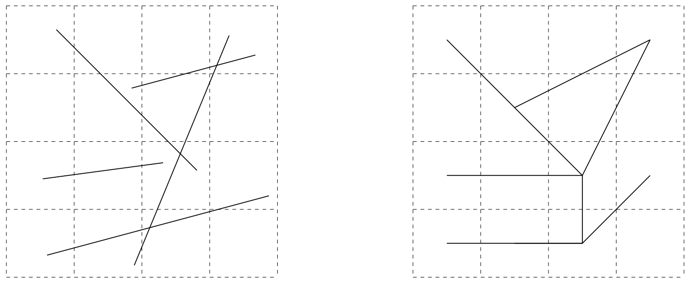

Figure 9: Snap-rounding line segments

deviate too much from the original data both with respect to geometry and topology while reducing the precision. Rounding geometric objects is related to simultaneous approximation of reals by rationals [138]. However, rounding geometric data is more complicated than rounding numbers and can be very difficult [102], because combinatorial and numerical data have to be kept consistent.

An intensively studied example is rounding an arrangement of line segments. Greene and Yao [66] were the first to investigate rounding line segments consistently to a regular grid. Note that simply rounding each segment endpoint to its nearest grid point can introduce new intersections and hence significantly violate the original topology. Greene and Yao break line segments into polylines such that all endpoints lie on the grid and the topology is largely preserved. Largely means, incidences not present in the original arrangement might arise, but it can be shown that no additional crossings are generated. Currently the most promising structure is *"snap-rounding"*, also called *"hot-pixel"* rounding, usually attributed to Greene and Hobby. A pixel in the regular grid is called hot if it contains an endpoint of an original line segment or an intersection point of the original segments. In the rounding process all line segments intersecting a hot pixel are snapped to the pixel center, cf. Fig. 9. Snap-rounding is used in [64, 68, 73]. Rounding can be done as a postprocessing step after exact computation, but it can also be seen as part of the problem and be incorporated into the algorithmic solution, as e.g. in [64] and [68].

5.4 Robustness in Geometric Algorithms Libraries

Library components should come with a precise description what they compute and for which inputs they are guaranteed to work. Correctness means that a component behaves according to such a specification. Exactness should not be confused with correctness in the sense of reliability. There is nothing wrong with approximation algorithms or approximate solutions as long as they do what they profess to do. Correctness can have unlike appearances: An algorithm handling only non-degenerate cases can be correct in the above sense. Also, an algorithm that guarantees to compute the exact result only if the numerical input data are integral and smaller than some given bound can be correct as well as an algorithm that computes an approximation to the exact result with a guaranteed error bound. Correctness in the sense of reliability is a must for (re)usability and hence for a geometric algorithms library.

Among the library and workbench efforts in computational geometry [4, 32, 61, 46, 96, 110] the XYZ-Geobench and LEDA deserve special attention concerning precision and robustness.

In XYZ-Geobench [110, 121] the axiomatic approach to robustness, described in section 4.4, is used. In LEDA [95, 96] arbitrary precision integer arithmetic is combined with the floating-point filter technique to yield efficient exact components for rational problems. Recently, in Europe and the US, new library projects called CGAL (Computational Geometry Algorithms Library) [26, 47, 115] and GeomLib [1, 8] have been started. The goal of both projects is to enhance the technology transfer from theory to practice in geometric computing by providing reliable, reusable implementations of geometric algorithms.

6 Conclusion

Over the past decade much progress has been made on the precision and robust problem, but no satisfactory general-purpose solution has been found. If exact predicates or exact number types are available, exact geometric computation is the more convenient approach. Algorithms designed for the real RAM model [117] can be implemented in a straightforward way; a redesign to deal with imprecision is not necessary. Moreover, exact computation is a prerequisite for the use of symbolic perturbation schemes. However, even with adaptive evaluation, exact geometric computation has its costs. Concerning efficiency, practitioners often ask for the impossible. Reliable algorithms based on exact geometric computation are requested to be competitive in performance to algorithms that sometimes crash or exhibit otherwise unexpected behavior. It should be clear, however, that one has to pay for the detection of degenerate and nearly degenerate situations in order to get reliability.

Exact geometric computation is not a panacea; it has limits. For cascaded computations with large depth, i.e. computations where the result of an arithmetic operation is an operand in another arithmetic operation many times in a row, the increase on required precision with the depth of computation makes exact geometric computation less suited. In this case, rounding intermediate results becomes important. Next, there are applications where speed is much more an issue than accuracy. As long as the computed outputs are useful, a fast robust algorithm dealing with imprecise computation will be more appropriate. Unfortunately, implementation with imprecision is much less straightforward. There is no general, widely applicable theory on how to deal with imprecision.

Related surveys on the problem of precision and robustness in geometric computation are given by Fortune [52], Hoffmann [76], and Yap [149]. Yap [150] and Yap and Dubé [151] address exact geometric computation. Franklin [60] especially discusses cartographic errors caused by precision problems. Dobkin and Silver [36] illustrate the effect of cascading geometric computation on the numerical accuracy of the computed result. Furthermore, robustness and precision issues were discussed at the ACM Workshop on Applied Computational Geometry at FCRC'96 in Philadelphia, see [54, 67, 116].

Acknowledgment: The author would like to thank Christoph Burnikel, Kurt Mehlhorn, Greg Perkins, and Michael Seel for their comments on earlier versions of this survey.

References

[1] P. K. Agarwal, M. T. Goodrich, S. R. Kosaraju, F. P. Preparata, R. Tamassia, and J. S. Vitter. Applicable and robust geometric computing, 1995. see `http://www.cs.brown.edu/cgc/`.

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

[3] G. Alefeld and J. Herzberger. *Introduction to Interval Computation.* Academic Press, New York, 1983.

[4] F. Avnaim. *C++GAL: A C++ Library for Geometric Algorithms.* INRIA Sophia-Antipolis, 1994.

[5] F. Avnaim, J.-D. Boissonnat, O. Devillers, F. Preparata, and M. Yvinec. Evaluating signs of determinants using single-precision arithmetic. *Algorithmica*, 17:111–132, 1997.

[6] C. L. Bajaj and T. K. Dey. Robust decompositions of polyhedra. In *Proc. 9th FSTTCS*, volume 405 of *Lecture Notes Comput. Sci.*, pages 267–279. Springer Verlag, 1989.

[7] C. L. Bajaj and T. K. Dey. Convex decomposition of polyhedra and robustness. *SIAM J. Comput.*, 21:339–364, 1992.

[8] J. E. Baker, R. Tamassia, and L. Vismara. GeomLib: Algorithm engineering for a geometric computing library, 1997. (Preliminary report).

[9] C. B. Barber. Computational geometry with imprecise data and arithmetic : Phd thesis. Technical Report CS-TR-377-92, Princeton University, 1992.

[10] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. The Quickhull algorithm for convex hulls. *ACM Trans. Math. Software*, 22(4):469–483, Dec. 1996.

[11] C. B. Barber and M. Hirsch. A robust algorithm for point in polyhedron. In *Proc. 5th Canad. Conf. Comput. Geom.*, pages 479–484, 1993.

[12] M. Benouamer, P. Jaillon, D. Michelucci, and J.-M. Moreau. A lazy solution to imprecision in computational geometry. In *Proc. 5th Canad. Conf. Comput. Geom.*, pages 73–78, 1993.

[13] J.L. Bentley and T.A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, C-28:643–647, 1979.

[14] J. Blömer. Computing sums of radicals in polynomial time. In *Proc. 32nd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 670–677, 1991.

[15] J.-D. Boissonnat and F. Preparata. Robust plane sweep for intersecting segments. Technical Report 3270, INRIA, Sophia-Antipolis, France, September 1997.

[16] J.-D. Boissonnat and M. Yvinec. *Algorithmic Geometry.* Cambridge University Press, Cambridge, UK, 1997.

[17] H. Brönnimann, I.Z. Emiris, V.Y. Pan, and S. Pion. Computing exact geometric predicates using modular arithmetic with single precision. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 174–182, 1997.

[18] H. Brönnimann and M. Yvinec. Efficient exact evaluation of signs of determinants. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 166–173, 1997.

[19] C. Burnikel. *Exact Computation of Voronoi Diagrams and Line Segment Intersections.* Ph.D thesis, Universität des Saarlandes, March 1996.

[20] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. A strong and easily computable separation bound for arithmetic expressions involving square roots. In *Proc. of the 8th ACM-SIAM Symp. on Discrete Algorithms*, pages 702–709, 1997.

[21] C. Burnikel, K. Mehlhorn, and S. Schirra. How to compute the Voronoi diagram of line segments: Theoretical and experimental results. In *Proc. 2nd Annu. European Sympos. Algorithms*, volume 855 of *Lecture Notes Comput. Sci.*, pages 227–239. Springer-Verlag, 1994.

[22] C. Burnikel, K. Mehlhorn, and S. Schirra. On degeneracy in geometric computations. In *Proc. 5th ACM-SIAM Sympos. Discrete Algorithms*, pages 16–23, 1994.

[23] C. Burnikel, K. Mehlhorn, and S. Schirra. The LEDA class `real` number. Technical Report MPI-I-96-1-001, Max-Planck-Institut für Informatik, 1996.

[24] J. F. Canny. *The Complexity of Robot Motion Planning*. ACM Doctoral Dissertation Award 1987. MIT Press, 1987. PhD thesis.

[25] J. F. Canny. Generalised characteristic polynomials. *J. Symbolic Computation*, 9:241–250, 1990.

[26] CGAL project. *see* `http://www.cs.ruu.nl/CGAL/`.

[27] J. Chang and V. Milenkovic. An experiment using LN for exact geometric computations. In *Proc. 5th Canad. Conf. Comput. Geom.*, pages 67–72, 1993.

[28] W. Chen, K. Wada, and K. Kawaguchi. Parallel robust algorithms for constructing strongly convex hulls. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 133–140, 1996.

[29] K. L. Clarkson. Safe and effective determinant evaluation. In *Proc. 33rd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 387–395, 1992.

[30] J. L. D. Comba and J. Stolfi. Affine arithmetic and its applications to computer graphics, 1993. Presented at SIBGRAPI'93, Recife (Brazil), October 20-22.

[31] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry*. Springer Verlag, 1997.

[32] P. de Rezende and W. Jacometti. Geolab: An environment for development of algorithms in computational geometry. In *Proc. 5th Canad. Conf. Comput. Geom.*, pages 175–180, Waterloo, Canada, 1993.

[33] T. J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18:224 – 242, 1971.

[34] O. Devillers and F. P. Preparata. A probabilistic analysis of the power of arithmetic filters. Technical Report CS-96-27, Center for Geometric Computing, Dept. Computer Science, Brown Univ., 1996.

[35] T. K. Dey, K. Sugihara, and C. L. Bajaj. Delaunay triangulations in three dimensions with finite precision arithmetic. *Comput. Aided Geom. Design*, 9:457–470, 1992.

[36] D. P. Dobkin and D. Silver. Applied computational geometry: Towards robust solutions of basic problems. *J. Comput. Syst. Sci.*, 40:70–87, 1989.

[37] D. Douglas. It makes me so CROSS. In D.J. Peuquet and D.F. Marble, editors, *Introductory Readings in Geographic Information Systems*, pages 303–307. Taylor & Francis, London, 1990.

[38] T. Dubé, K. Ouchi, and C. K. Yap. *Tutorial for* `Real/Expr` *Package*, 1996.

[39] T. Dubé and C. K. Yap. A basis for implementing exact computational geometry. extended abstract, 1993.

[40] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer Verlag, 1986.

[41] H. Edelsbrunner and E. Mücke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Trans. on Graphics*, 9:66–104, 1990.

[42] I. Emiris. A complete implementation for computing general dimensional convex hulls. Research Report 2551, INRIA, Sophia-Antipolis, France, 1996.

[43] I. Emiris and J. Canny. An efficient approach to removing geometric degeneracies. In *Proc. of the 8th ACM Symp. on Computational Geometry*, pages 74–82, 1992.

[44] I. Emiris and J. Canny. A general approach to removing degeneracies. *SIAM J. Comput.*, 24:650–664, 1995.

[45] I. Z. Emiris, J. F. Canny, and R. Seidel. Efficient perturbations for handling geometric degeneracies. *Algorithmica*, 19(1–2):219–242, September 1997.

[46] P. Epstein, J. Kavanagh, A. Knight, J. May, T. Nguyen, and J.-R. Sack. A workbench for computational geometry. *Algorithmica*, 11:404–428, 1994.

[47] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. The CGAL kernel : a basis for geometric computation. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry : Towards Geometric Engineering (WACG96)*, pages 191–202. Springer LNCS 1148, 1996.

[48] S. Fang and B. Brüderlin. Robustness in geometric modeling — tolerance-based methods. In *Computational Geometry — Methods, Algorithms and Applications: Proc. Internat. Workshop Comput. Geom. CG '91*, volume 553 of *Lecture Notes Comput. Sci.*, pages 85–101. Springer-Verlag, 1991.

[49] A. R. Forrest. Computational geometry in practice. In R. A. Earnshaw, editor, *Fundamental Algorithms for Computer Graphics*, volume F17 of *NATO ASI*, pages 707–724. Springer-Verlag, 1985.

[50] 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.

[51] S. Fortune. Stable maintenance of point set triangulations in two dimensions. In *Proc. 30th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 494–505, 1989.

[52] S. Fortune. Progress in computational geometry. In R. Martin, editor, *Directions in Geometric Computing*, pages 81 – 128. Information Geometers Ltd., 1993.

[53] S. Fortune. Numerical stability of algorithms for 2-d Delaunay triangulations. *Internat. J. Comput. Geom. Appl.*, 5(1):193–213, 1995.

[54] S. Fortune. Robustness issues in geometric algorithms. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry : Towards Geometric Engineering (WACG96)*, pages 9–14. Springer LNCS 1148, 1996.

[55] S. Fortune and V. Milenkovic. Numerical stability of algorithms for line arrangements. In *Proc. 7th Annu. ACM Sympos. Comput. Geom.*, pages 334–341, 1991.

[56] S. Fortune and C. van Wyk. *LN user manual*, 1993.

[57] S. Fortune and C. J. Van Wyk. Efficient exact arithmetic for computational geometry. In *Proc. 9th Annu. ACM Sympos. Comput. Geom.*, pages 163–172, 1993.

[58] S. Fortune and C. J. van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Trans. Graph.*, 15(3):223–248, July 1996.

[59] P. G. Franciosa, C. Gaibisso, G. Gambosi, and M. Talamo. A convex hull algorithm for points with approximately known positions. *Internat. J. Comput. Geom. Appl.*, 4(2):153–163, 1994.

[60] W. R. Franklin. Cartographic errors symptomatic of underlying algebra problems. In *Proc. Internat. Sympos. Spatial Data Handling*, volume 1, pages 190–208, 20–24 August 1984.

[61] G.-J. Giezeman. *PlaGeo, a library for planar geometry, and SpaGeo, a library for spatial geometry.* Utrecht University, 1994.

[62] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 32(1):5–48, March 1991.

[63] M. F. Goodchild. Issues of quality and uncertainty. In J.C. Muller, editor, *Advances in Cartography*, pages 113–139. Elsevier Applied Science, London, 1991.

[64] M. Goodrich, L. Guibas, J. Hershberger, and P. Tanenbaum. Snap rounding line segments efficiently in two and three dimensions. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 284–293, 1997.

[65] T. Granlund. *GNU MP, The GNU Multiple Precision Arithmetic Library*, 2.0.2 edition, June 1996.

[66] D. H. Greene and F. F. Yao. Finite-resolution computational geometry. In *Proc. 27th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 143–152, 1986.

[67] L. Guibas. Implementing geometric algorithms robustly. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry : Towards Geometric Engineering (WACG96)*, pages 15–22. Springer LNCS 1148, 1996.

[68] L. Guibas and D. Marimont. Rounding arrangements dynamically. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 190–199, 1995.

[69] L. Guibas, D. Salesin, and J. Stolfi. Epsilon geometry: building robust algorithms from imprecise computations. In *Proc. 5th Annu. ACM Sympos. Comput. Geom.*, pages 208–217, 1989.

[70] L. Guibas, D. Salesin, and J. Stolfi. Constructing strongly convex approximate hulls with inaccurate primitives. In *Proc. 1st Annu. SIGAL Internat. Sympos. Algorithms*, volume 450 of *Lecture Notes Comput. Sci.*, pages 261–270. Springer-Verlag, 1990.

[71] D. Halperin and C. Shelton. A perturbation scheme for spherical arrangements with application to molecular modeling. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 183–192, 1997.

[72] K. Hinrichs, J. Nievergelt, and P. Schorn. An all-round sweep algorithm for 2-dimensional nearest-neighbor problems. *Acta Informatica*, 29:383–394, 1992.

[73] J.D. Hobby. Practical line segment interscetion with finite precision output. Technical Report 93/2-27, Bell Laboratories (Lucent Technologies), 1993.

[74] C. M. Hoffmann, J. E. Hopcroft, and M. Karasick. Towards implementing robust geometric computations. In *Proc. 4th Annu. ACM Sympos. Comput. Geom.*, pages 106–117, 1988.

[75] C. M. Hoffmann, J. E. Hopcroft, and M. T. Karasick. Robust set operations on polyhedral solids. *IEEE Comput. Graph. Appl.*, 9(6):50–59, November 1989.

[76] C.M. Hoffmann. The problem of accuracy and robustness in geometric computation. *IEEE Computer*, pages 31–41, March 1989.

[77] J. E. Hopcroft and P. J. Kahn. A paradigm for robust geometric algorithms. *Algorithmica*, 7:339–380, 1992.

[78] H. Inagaki and K. Sugihara. Numerically robust algorithm for constructing constrained Delaunay triangulation. In *Proc. 6th Canad. Conf. Comput. Geom.*, pages 171–176, 1994.

[79] H. Inagaki, K. Sugihara, and N. Sugie. Numerically robust incremental algorithm for constructing three-dimensional Voronoi diagrams. In *Proc. 4th Canad. Conf. Comput. Geom.*, pages 334–339, 1992.

[80] J. W. Jaromczyk and G. W. Wasilkowski. Computing convex hull in a floating point arithmetic. *Comput. Geom. Theory Appl.*, 4:283–292, 1994.

[81] K. Jensen and N. Wirth. *PASCAL- User Manual and Report. Revised for the ISO Pascal Standard.* Springer Verlag, 3rd edition, 1985.

[82] S. Kahan and J. Snoeyink. On the bit complexity of minimum link paths: Superquadratic algorithms for problems solvable in linear time. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 151–158, 1996.

[83] M. Karasick. *On the Representation and Manipulation of Rigid Solids.* Ph.D. thesis, Dept. Comput. Sci., McGill Univ., Montreal, 1989.

[84] M. Karasick, D. Lieber, and L. R. Nackman. Efficient Delaunay triangulations using rational arithmetic. *ACM Trans. Graph.*, 10:71–91, 1991.

[85] R. Klein. *Algorithmische Geometrie.* Addison-Wesley, 1997. (in German).

[86] D. E. Knuth. *The Art of Computer Programming Vol. 2: Seminumerical Algorithms.* Addison-Wesley, 2nd edition, 1981.

[87] D. E. Knuth. *Axioms and Hulls*, volume 606 of *Lecture Notes in Computer Science.* Springer-Verlag, Heidelberg, Germany, 1992.

[88] M. J. Laszlo. *Computational geometry and computer graphics in C++.* Prentice Hall, Upper Saddle River, NJ, 1996.

[89] Z. Li and V. Milenkovic. Constructing strongly convex hulls using exact or rounded arithmetic. *Algorithmica*, 8:345–364, 1992.

[90] LiDIA -Group, Fachbereich Informatik Institut für Theoretische Informatik TH Darmstadt. *LiDIA Manual A library for computational number theory*, 1.3 edition, April 1997.

[91] G. Liotta, F. Preparata, and R. Tamassia. Robust proximity queries: An illustration of degree-driven algorithm design. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 156–165, 1997.

[92] K. Mehlhorn. *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry.* Springer Verlag, 1984.

[93] K. Mehlhorn and S. Näher. Implementation of a sweep line algorithm for the straight line segment intersection problem. Report MPI-I-94-160, Max-Planck-Institut Inform., Saarbrücken, Germany, 1994.

[94] K. Mehlhorn and S. Näher. The implementation of geometric algorithms. In *Proc. 13th World Computer Congress IFIP94*, volume 1, pages 223–231, 1994.

[95] K. Mehlhorn and S. Näher. LEDA, a platform for combinatorial and geometric computing. *Communications of the ACM*, 38:96–102, 1995.

[96] K. Mehlhorn, S. Näher, and C. Uhrig. *The LEDA User manual*, 3.5 edition, 1997. see `http://www.mpi-sb.mpg.de/LEDA/leda.html`.

[97] M. Mignotte. Identification of algebraic numbers. *Journal of Algorithms*, 3:197–204, 1982.

[98] M. Mignotte. *Mathematics for Computer Algebra.* Springer Verlag, 1992.

[99] V. Milenkovic. Verifiable implementations of geometric algorithms using finite precision arithmetic. *Artif. Intell.*, 37:377–401, 1988.

[100] V. Milenkovic. Double precision geometry: a general technique for calculating line and segment intersections using rounded arithmetic. In *Proc. 30th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 500–505, 1989.

[101] V. Milenkovic. Robust polygon modeling. *Comput. Aided Design*, 25(9), 1993. (special issue on Uncertainties in Geometric Design).

[102] V. Milenkovic and L. R. Nackman. Finding compact coordinate representations for polygons and polyhedra. In *Proc. 6th Annu. ACM Sympos. Comput. Geom.*, pages 244–252, 1990.

[103] T. Minakawa and K. Sugihara. Topology oriented vs. exact arithmetic - experience in implementing the three-dimensional convex hull algorithm. In *ISAAC97*, 1997.

[104] R. E. Moore. *Interval Analysis.* Prentice-Hall, Englewood Cliffs, NJ, 1966.

[105] R. E. Moore. *Methods and Applications of Interval Analysis.* SIAM, Philadelphia, 1979.

[106] S. P. Mudur and P. A. Koparkar. Interval methods for processing geometric objects. *IEEE Computer Graphics and Applications*, 4(2):7–17, 1984.

[107] K. Mulmuley. *Computational Geometry : An Introduction through Randomized Algorithms.* Prentice Hall, Englewood Cliffs, NJ, 1994.

[108] J. Nievergelt and K. H. Hinrichs. *Algorithms and Data Structures: With Applications to Graphics and Geometry.* Prentice Hall, Englewood Cliffs, NJ, 1993.

[109] J. Nievergelt and P. Schorn. Das Rätsel der verzopften Geraden. *Informatik Spektrum*, (11):163–165, 1988. (in German).

[110] J. Nievergelt, P. Schorn, M. de Lorenzi, C. Ammann, and A. Brüngger. XYZ: Software for geometric computation. Technical Report 163, Institut für Theorische Informatik, ETH, Zürich, Switzerland, 1991.

[111] Y. Oishi and K. Sugihara. Topology oriented divide and conquer algorithm for Voronoi diagrams. *Graphical Models and Image Processing*, 57(4):303–314, 1995.

[112] J. O'Rourke. *Computational geometry in C.* Cambridge University Press, Cambridge, 1994.

[113] T. Ottmann, G. Thiemt, and C. Ullrich. Numerical stability of geometric algorithms. In *Proc. of the 3rd ACM Symp. on Computational Geometry*, pages 119–125, 1987.

[114] K. Ouchi. Real/Expr: Implementation of exact computation. Courant Institute, New York University, 1997. Master thesis.

[115] M. Overmars. Designing the computational geometry algorithms library CGAL. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry : Towards Geometric Engineering (WACG96)*, pages 53–58. Springer LNCS 1148, 1996.

[116] F. Preparata. Robustness in geometric algorithms. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry : Towards Geometric Engineering (WACG96)*, pages 23–24. Springer LNCS 1148, 1996.

[117] F. Preparata and M.I. Shamos. *Computational Geometry.* Springer Verlag, 1985.

[118] D. M. Priest. Algorithms for arbitrary precision floating point arithmetic. In *10th Symposium on Computer Arithmetic*, pages 132 – 143. IEEE Computer Society Press, 1991.

[119] D. M. Priest. *On Properties of Floating-Point Arithmetic: Numerical Stability and the Cost of Accurate Computations.* PhD thesis, Department of Mathematics, University of California at Berkeley, 1992.

[120] D. Pullar. Consequences of using a tolerance paradigm in spatial overlay. In *Proc. of Auto-Carto 11*, pages 288–296, 1993.

[121] P. Schorn. An object-oriented workbench for experimental geometric computation. In *Proc. 2nd Canad. Conf. Comput. Geom.*, pages 172–175, 1990.

[122] P. Schorn. *Robust Algorithms in a Program Library for Geometric Computation*, volume 32 of *Informatik-Dissertationen ETH Zürich.* Verlag der Fachvereine, Zürich, 1991.

[123] P. Schorn. An axiomatic approach to robust geometric programs. *J. Symbolic Computation*, 16:155–165, 1993.

[124] P. Schorn. Degeneracy in geometric computation and the perturbation approach. *The Computer Journal*, 37(1):35–42, 1994.

[125] M. Segal. Using tolerances to guarantee valid polyhedral modeling results. *Comput. Graph.*, 24(4):105–114, August 1990.

[126] M. Segal and C. H. Sequin. Partitioning polyhedral objects into nonintersecting parts. *IEEE Comput. Graph. Appl.*, 8(1):53–67, January 1988.

[127] M. G. Segal and C. H. Sequin. Consistent calculations for solids modelling. In *Proc. 1st Annu. ACM Sympos. Comput. Geom.*, pages 29–38, 1985.

[128] R. Seidel. The nature and meaning of perturbations in geometric computations. In *STACS94*, 1994.

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

[130] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. Technical Report CMU-CS-96-140, School of Computer Science, Carnegie Mellon University, 1996.

[131] J. R. Shewchuk. Robust adaptive floating-point geometric predicates. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 141–150, 1996.

[132] J. R. Shewchuk. Triangle: Engineering a 2D quality mesh generator and delaunay triangulator. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry : Towards Geometric Engineering (WACG96)*, pages 203–222. Springer LNCS 1148, 1996.

[133] IEEE Standard. 754-1985 for binary floating-point arithmetic. *SIGPLAN*, 22:9–25, 1987.

[134] A. J. Stewart. Robust point location in approximate polygons. In *Proc. 3rd Canad. Conf. Comput. Geom.*, pages 179–182, 1991.

[135] A. J. Stewart. *The theory and practice of robust geometric computation, or, how to build robust solid modelers.* Ph.D. thesis, Dept. Comput. Sci., Cornell Univ., Ithaca, NY, August 1991. Technical Report TR 91-1229.

[136] A. J. Stewart. Local robustness and its application to polyhedral intersection. *Internat. J. Comput. Geom. Appl.*, 4(1):87–118, 1994.

[137] K. G. Suffern and E. D. Fackerell. Interval methods in computer graphics. *Computers & Graphics*, 15(3):331–340, 1991.

[138] K. Sugihara. On finite-precision representations of geometric objects. *J. Comput. Syst. Sci.*, 39:236–247, 1989.

[139] K. Sugihara. An intersection algorithm based on Delaunay triangulation. *IEEE Comput. Graph. Appl.*, 12(2):59–67, March 1992.

[140] K. Sugihara. A simple method for avoiding numerical errors and degeneracy in Voronoi diagram construction. *IEICE Trans. Fundamentals*, E75-A(4):468–477, April 1992.

[141] K. Sugihara. Topologically consistent algorithms related to convex polyhedra. In *Proc. 3rd Annu. Internat. Sympos. Algorithms Comput.*, volume 650 of *Lecture Notes Comput. Sci.*, pages 209–218. Springer-Verlag, 1992.

[142] K. Sugihara. A robust and consistent algorithm for intersecting convex polyhedra. *Comput. Graph. Forum*, 13(3):45–54, 1994. Proc. EUROGRAPHICS '94.

[143] K. Sugihara and M. Iri. A solid modelling system free from topological inconsistency. *J. Inform. Proc.*, 12(4):380–393, 1989.

[144] K. Sugihara and M. Iri. Construction of the Voronoi diagram for 'one million' generators in single-precision arithmetic. *Proc. IEEE*, 80(9):1471–1484, September 1992.

[145] K. Sugihara and M. Iri. A robust topology-oriented incremental algorithm for Voronoi diagrams. *Internat. J. Comput. Geom. Appl.*, 4:179–228, 1994.

[146] K. Sugihara, Y. Ooishi, and T. Imai. Topology-oriented approach to robustness and its applications to several Voronoi-diagram algorithms. In *Proc. 2nd Canad. Conf. Comput. Geom.*, pages 36–39, 1990.

[147] C. K. Yap. A geometric consistency theorem for a symbolic perturbation scheme. In *Proc. of the 4th ACM Symp. on Computational Geometry*, pages 134–141, 1988.

[148] C. K. Yap. Symbolic treatment of geometric degeneracies. *J. Symbolic Comput.*, 10:349–370, 1990.

[149] C. K. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *CRC Handbook in Computational Geometry*, pages 653–668. CRC Press, 1997.

[150] C. K. Yap. Towards exact geometric computation. *Computational Geometry: Theory and Applications*, 7(1-2):3–23, 1997. Preliminary version appeared in Proc. of the 5th Canad. Conf. on Comp. Geom., pages 405-419, (1993).

[151] C. K. Yap and T. Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, volume 1 of *Lecture Notes Series on Computing*, pages 452–492. World Scientific Press, Singapore, 2nd edition, 1995.