# Chapter 9. Precision and Robustness in Geometric Computations

Stefan Schirra

Max-Planck-Institut für Informatik
Saarbrücken
Germany
stschirr@mpi-sb.mpg.de

## 1 Introduction

This part gives a concise overview of techniques that have been proposed and successfully used to attack precision problems in the implementation of geometric algorithms.

In reference to issues of quality of spatial data in GIS as well as in reference to implementation issues of geometric data structures and algorithms, the terms precision and accuracy are often used interchangeably. We adopt the terminology used in [47]. *Accuracy* refers to the relationship between reality and the measured data modelling it. *Precision* refers to the level of detail with which (numerical) data are represented in a model or in (arithmetic) calculations with the model.

Here, our attention is directed to precision, more precisely, to how to deal with the notorious problems that imprecise geometric calculations can cause. Inaccuracy in GIS data is not our main objective. Basically, we assume that the geometric data to be processed are accurate. Precision problems can make implementing geometric algorithms very unpleasant [27, 72] even under the assumption of perfectly accurate data, if no appropriate techniques are used to deal with imprecision. A quite sketchy discussion of dealing with inaccurate data is given in Section 5.2.

### 1.1 Precision and Correctness

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 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 (accumulated) 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. Figure 1 gives an example.

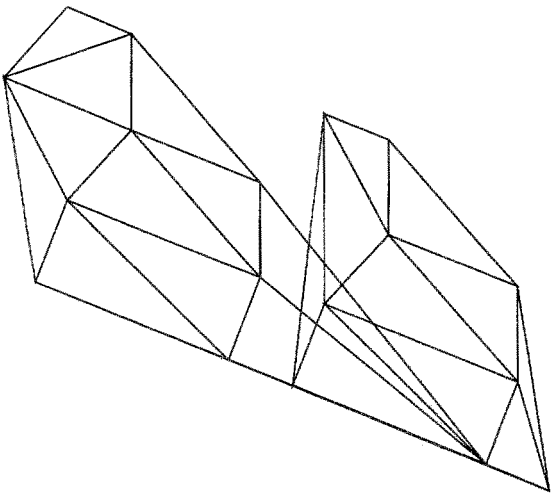

**Fig. 1.** Incorrect Delaunay triangulation. The error was caused by precision problems. The correct Delaunay triangulation is given in Figure 2. *Courtesy of J. R. Shewchuk* [102].

Conditional tests are critical parts of an implementation, because they determine the control flow. 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. In this case, the combinatorial part of the geometric output of the algorithm will be correct. Numerical data, however, computed by the algorithm might nevertheless be imprecise.

Rounding and cancellation errors may cause wrong decisions and hence lead to errors in the combinatorial part of the geometric output as well. Thereby imprecise calculations can destroy the correctness of the implementation of an otherwise correct algorithm.

### 1.2 Robustness and Stability

Along with the substitution of real arithmetic by floating-point arithmetic, correctness is often replaced by robustness. 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.

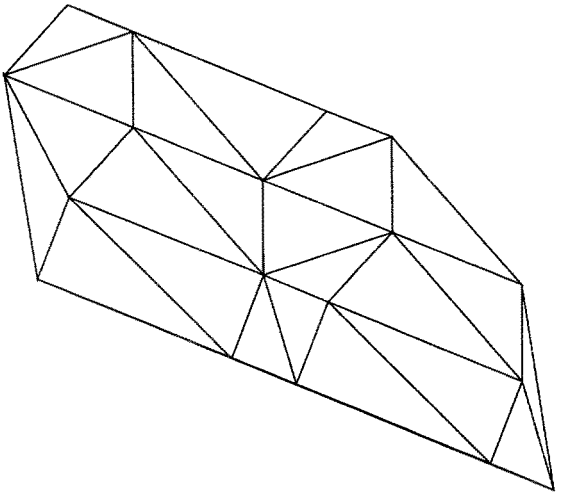

**Fig. 2.** Correct Delaunay triangulation. *Courtesy of J. R. Shewchuk*[102].

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. This is certainly true for most of the geometric problems in GIS.

### 1.3 Degeneracy

A related problem in the implementation of geometric algorithms is degeneracies. 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. Since precision problems are caused by degenerate and nearly degenerate configurations in the input, degeneracy is closely related to precision and robustness. Symbolic perturbation schemes [31, 32, 33, 112, 113] have been proposed to abolish the handling of degeneracies. Exact computation is a prerequisite for applying these techniques [111]. The handling of degeneracies and the use of symbolic perturbation schemes are a

point of controversy in the computational geometry literature [15, 99, 100]. For a discussion of degeneracy we refer the reader to [15] and [99].

Sometimes, the term robustness is also used with respect to degeneracies. Dey et al. [26] define robustness as the ability of a geometric algorithm to deal with degeneracies and "inaccuracies" during various numerical computations. The definition of robustness in [97] is similar.

## 1.4 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. 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 very promising, because it allows exact implementations of numerous geometric algorithms developed for real arithmetic without modifications of these algorithms.

## 1.5 Floating-Point Arithmetic

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` [59]. Since most geometric computations are executed with floating-point arithmetic, it is worth taking a closer look at floating-point computation. Goldberg [46] gives an excellent overview.

A finite-precision floating-point system has a base $\mathrm{B}$, a fixed mantissa length $l$ and and an exponent range $[e_{\min}..e_{\max}]$.

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

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

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

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 $1/2 \cdot \mathrm{B}^{-l}$, 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 accuracy of the floating-point machinery, seem be coming to an end. The IEEE standard 754 for binary floating-point computation [104] 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 $\infty$, and rounding toward -$\infty$ 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 $2^{-23}$ and $2^{-52}$. 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 $e_{\min}$. 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 $\infty$ 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 [83] gives some examples. Since the standard fixes the layout of bits for mantissa and exponent in the representation of floating-point numbers, bit-operations can be used to extract information.

# 2 Geometric Computation

Geometric computing is a combination of numerical and combinatorial computation.

## 2.1 Geometric Problems

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, given 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 is 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.

Line simplification problems in cartography can be selective or constructive as well, depending on whether only input points are allowed as vertices of the simplified polyline or not.

## 2.2 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 a geometric algorithm. Geometric primitives subsume constructions of basic geometric objects, like line segments or circles, and predicates. 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 constants, in practical problems mainly integers, and numerical data of the geometric objects that are tested. 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.

## 2.3 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 [111]. 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 [11, 69] the notion of the degree of an expression is extended to expressions involving square roots. An expression involving operations $+, -, *, /$ only is called *rational.*

## 2.4 Geometric Computation with Floating-Point Numbers

In a branching step of a geometric algorithm, numerical values of some expression given by an expression dag are compared. 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. The accumulated error in the numerical calculation might be so large that the truth value of the predicate with the expressions evaluated with inherently imprecise floating-point computation is different from the truth value of the predicate with an exact evaluation of the predicate.

Naively applied floating-point arithmetic can set axioms of geometry out of order. A classical example is Ramshaw's braided lines (see Figure 3 and [83, 84]).

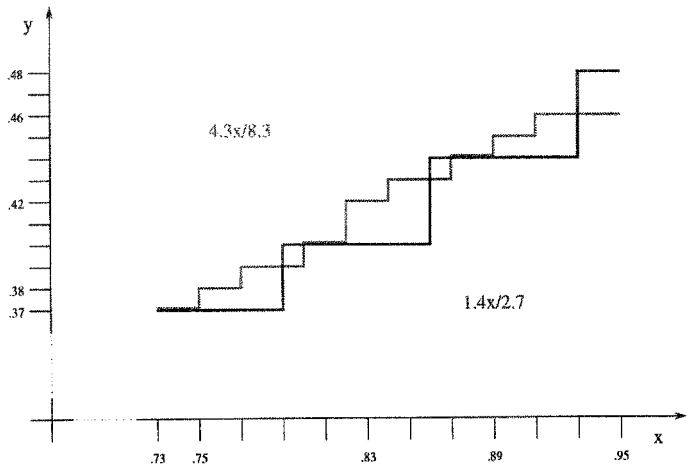


**Fig. 3.** Evaluation of the line equations $y = 4.3 \cdot x/8.3$ and $y = 1.4 \cdot x/2.7$ in a floating-point system with base 10 and mantissa length 2 and rounding to nearest suggests that the lines have several intersection points besides the true intersection point at the origin.

Rewriting an expression to an expression dag that leads to a numerically more stable evaluation order can help a lot. Goldberg [46] 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 precision [46].

As the example above shows, the way a numerical value is computed can highly influence its precision. Summation of floating-point numbers is another classical example. Rearranging the summands helps to reduce imprecision due to extinction.

### 2.5 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. Normally, 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 [114] 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. Figure 4 shows a polyline, where every three consecutive vertices are collinear under the "close to zero is zero" rule. In each

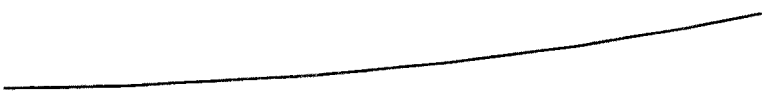

**Fig. 4.** A locally straight line

case, a fairly small perturbation 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 that might cause trouble, since the computed output and the exact output can be combinatorially very different [15].

## 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 mimiced 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.

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 [114] 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 $\alpha$ and an interval

that isolates $\alpha$ 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 flavours of exact geometric computation. Franklin's survey [44] already discusses the basics of many approaches to exact computation. Since the publication of his paper much progress has been made in improving the efficiency of exact computation (see [111] for an overview). Thus some of his conclusions have to be revisited.

### 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 [101], GNU MP [49], LiDIA [68], or the number type `integer` in LEDA [74]. Fortune and Van Wyk [41, 43] 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. Integer arithmetic with a fixed precision adjusted to the maximum possible integer size in the input and to 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 and if $d$ is the maximum degree and $m$ the maximum number of monomials of the integral polynomial expressions, then an integer arithmetic for integers with $db + \log m + O(1)$ bits suffices. Usually, $m$ is in $O(1)$. The degree of polynomial expressions in geometric predicates has recently gained more attention in the design of geometric algorithms. Liotta et al. [69] investigate the degree involved in some proximity problems in 2- and 3-dimensional space.

Many predicates include only expressions involving operations $+, -, *, /$. All the predicates arising in problems like map overlay in cartography and in most of the problems discussed in textbooks on computational geometry [70, 91, 30, 82, 86, 66, 62, 23, 8] are of this type. Such problems are called *rational* [111].

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 $(hx_0, hx_1, \ldots, hx_{d-1}, hx_d)$ such that $x_i = hx_i/hx_d$ for all $0 \leq i \leq d-1$. 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 the common denominator of the coordinates. 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. In principal, even sign tests for expressions involving square roots can be turned into a sequence of sign tests of polynomial expressions by repeated squaring [14, 69]. Therefore, arbitrary or multiple precision integer arithmetic is a powerful tool for exact geometric computation, but arbitrary precision integer arithmetic has to be supplied by software and is therefore much slower than the hardware-supported fixed precision 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 [61] 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 [60]. Numerator and denominator of the knick-points on a minimum link path can have superquadratic bitlength with respect to the number of polygon vertices [60]. 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 [41, 43] 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 [42], 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.

For integral polynomial expressions, modular arithmetic [1, 64] 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 $k$-tupels $(r_0, r_1, \ldots, r_{k-1})$ 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 [1], 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 [64]. 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 [41, 43] 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, 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 [9].

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 Lazy Evaluation

The LEA system [7] favors 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. Since it is hard to know in advance which precision will be needed in later decisions, numbers have to be presented in a way that allows for recomputation with higher precision if the currently available precision is not sufficient. In the LEA system, numbers are represented by intervals and expression dags that reflects their creation history. Initially only a low precision representation is calculated, representations with repeatedly increased precision are computed only if decisions can't be made with the current precision.

In LEA, interval arithmetic [7] with floating-point numbers is used to compute rough representations of a number. The interval is then repeatedly refined 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 trees is described by Yap and Dubé [29, 111, 114]. 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 [16] also stores the creation history in expression dags and uses floating-point approximations and errors bounds as first approximations. The strategy of repeatedly increasing the precision is similar to [29, 111, 114]. 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 C++-programming language is well suited for using number types that provide exact computation in a packed form like lazy numbers. Since arithmetic operators can be overloaded, software-based number types can be used exactly like `int` and `double`. Thereby lazy numbers can be used by a programmer exactly like the built-in number types. The user does not notice that his numbers are lazy-evaluated.

Lazy evaluation has to detect whether the precision of a computation is sufficient or not. How this can be done is described in the following subsections.

### 3.3 Floating-Point Filter

Replacing exact arithmetic, on which the correctness of a geometric algorithm was based, by imprecise finite-precision arithmetic works in practice for most 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. The idea

of floating-point filters is to filter out those branching steps where a floating-point computation gives the correct result. Only if it is not certified that the floating-point evaluation leads to a correct decision is the branching step reevaluated at a higher cost by calculating the exact value or a better approximation.

Filter techniques allow the use of high speed floating-point arithmetic. A filter simply computes a bound on the error of the floating-point computation and compares the absolute value of the computed result to the computed error bound. If the error bound is smaller, the computed approximation and the exact value have the same sign. 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, e.g., 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. Thus dynamic filters can let more floating-point calculations pass the filter but at the cost of the online error computation. 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.

Note the difference between static filters and heuristic epsilons. If the computed approximate value is larger than the error bound or $\varepsilon_{\text{magic}}$ respectively, the behavior is identical. The program continues based on the 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, which finally leads to a correct decision.

Mehlhorn and Näher use the following easily computable error bounds for integral expressions evaluated in floating-point arithmetic in their implementation of the Bentley-Ottmann plane sweep algorithm for computing the intersections among a set of line segments in the plane [71]. It assumes that neither overflow nor underflow occurs. Let $E$ be an integral expression. $E$ is also used to denote the value of $E$ while $\tilde{E}$ is used to denote the value of the expression when evaluated with floating point arithmetic, i.e., all operations are replaced by their floating point counterparts.

Mehlhorn and Näher [71] define the *measure* $mes(E)$ and the *index* $ind(E)$ of a polynomial expression $E$ such that

$$|\tilde{E} - E| \leq ind(E) \cdot \varepsilon_{\text{prec}} \cdot mes(E).$$

where $\varepsilon_{\text{prec}}$ is the machine precision of the floating-point system used. Both the index and the measure are easily computable by the following rules.

| | $mes(E)$ | $ind(E)$ |
|---|---|---|
| float $f \neq 0$ | $2^{\lfloor \log \lvert f \rvert \rfloor}$ | 0 |
| float 0 | 0 | 0 |
| $E_1 \pm E_2$ | $2 \cdot \max(mes(E_1), mes(E_2))$ | $(1 + ind(E_1) + ind(E_2))/2$ |
| $E_1 \cdot E_2$ | $mes(E_1) \cdot mes(E_2)$ | $1/2 + ind(E_1) + ind(E_2)$ |

If a filter fails, a refined filter can be used. A refined filter might compute a tighter error bound or use floating-point arithmetic with higher precision and thereby get better approximations and smaller error bounds. This step can be iterated. Composition of more and more refined filters leads to a lazy evaluation strategy. Finally, if necessary, exact arithmetic can be used. Such lazy evaluation strategies are called *adaptive*, because they do not compute more precisely than needed.

For orientation predicates and incircle tests in two- and three-dimensional space Shewchuk [102, 103] presents such a lazy evaluation strategy. It uses an (exact, if neither underflow nor overflow occurs) representation of sums and products of floating-point numbers as a symbolic sum of double precision floating-point numbers. Computation with numbers in this representation, called expanded doubles in [102], is based on the interesting results of Priest [92, 93] and Dekker [25] 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 filters to get best possible floating-point approximations, as suggested by Ottmann et al. [87].

### 3.4 Interval Arithmetic

Approximation and error bound define an interval that contains the exact value. Interval arithmetic [2, 79, 80] 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 $\infty$ and toward $-\infty$ are very useful. See, for example, [81, 105] 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 [61] to geometric computing.

A refinement of standard interval arithmetic is so-called affine arithmetic proposed by Comba and Stolfi [22]. 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 $[0, 0]$.

### 3.5 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 [21] 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 its determinant with fast hardware floating-point arithmetic.

Avnaim et al. [4] consider determinants of small matrices with integer entries, too. They present algorithms to compute the sign of $2 \times 2$ and $3 \times 3$ matrices with $b$-bit integer entries using precision $b$ and $b + 1$ only, respectively. Brönnimann and Yvinec [10] extend the method of [4] to $d \times d$ matrices and compare it with a variant of Clarkson's method.

### 3.6 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 either 0 or the separation bound are not 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 $E$. Mignotte discusses identification of algebraic numbers given by expressions involving square roots in [75].

The measure of a polynomial [76] can be used for automatic derivation 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 is easily computable but very weak [14].

Other recursive formulas for an expression involving square root operations leading to separation bounds are given in [111]. Here, a bound on the maximum absolute value of the coefficients of an integral polynomial is used. The rules are given in Table 2. By a result of Cauchy, $(h(E)+1)^{-1}$ is a separation bound, i.e., $\alpha(E) = 0$ or $\alpha(E) \geq (h(E)+1)^{-1}$.

In [17] 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 $c$. Although Canny solves a much more general problem, his bounds can be used to

| | $M(E)$ | $deg(E)$ |
|---|---|---|
| integer $n$ | $\lvert n\rvert$ | 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 derivation of separation bounds for expressions involving square roots based on the measure of a polynomial

| | $h(E)$ | $d(E)$ |
|---|---|---|
| integer $n$ | $\lvert n\rvert$ | 1 |
| $E_1 + E_2$ | $(h(E_1)2^{1+d(E_1)})^{d(E_2)}(h(E_2)\sqrt{1+d(E_2)})^{d(E_1)}$ | $d(E_1) \cdot d(E_2)$ |
| $E_1 - E_2$ | $(h(E_1)2^{1+d(E_1)})^{d(E_2)}(h(E_2)\sqrt{1+d(E_2)})^{d(E_1)}$ | $d(E_1) \cdot d(E_2)$ |
| $E_1 \cdot E_2$ | $(h(E_1)\sqrt{1+d(E_1)})^{d(E_2)}(h(E_2)\sqrt{1+d(E_2)})^{d(E_1)}$ | $d(E_1) \cdot d(E_2)$ |
| $E_1/E_2$ | $(h(E_1)\sqrt{1+d(E_1)})^{d(E_2)}(h(E_2)\sqrt{1+d(E_2)})^{d(E_1)}$ | $d(E_1) \cdot d(E_2)$ |
| $\sqrt{E_1}$ | $h(E_1)$ | $2 \cdot d(E_1)$ |

**Table 2.** Recursive formulas for quantities $h(E)$ and $d(E)$ of an arithmetic expression involving square roots.

get fairly good separation bounds for expressions involving square roots. 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 $(3dc)^{-nd^n}$ [17, 18].

Based on the structure of an expression $E$ given by an expression tree, a system of polynomial equations can be built which has an isolated solution vector with $\alpha(E)$ as a component. The system of polynomial equations consists of a system $\mathcal{P}(E)$ in $n_E$ variables $X_1, \ldots, X_{n_E}$ and a distinct equation of the form $X_E = P_E(X_1, \ldots, X_{n_E})$. The variables correspond to subexpressions of $E$, the variable $X_E$ represents the value of $E$.

At the basis of recursion we have the distinct polynomial only. If $E = E_1 \pm E_2$ then $\mathcal{P}(E)$ is the union of the systems $\mathcal{P}(E_1)$ and $\mathcal{P}(E_2)$ and the distinct equation becomes $X_E = P_{E_1}(\ldots) \pm P_{E_2}(\ldots)$. Variables are renamed appropriately. The recursion step is completely analogous if $E = E_1 \cdot E_2$.

If $E = E_1/E_2$ the system $\mathcal{P}(E)$ contains the union of the systems $\mathcal{P}(E_1)$ and $\mathcal{P}(E_2)$. Furthermore the equation

$$X_{\text{new}} \cdot P_{E_2}(\ldots) = P_{E_1}(\ldots)$$

is added. It uses a new variable $X_{\text{new}}$ and is based on the distinct equations for the subexpressions. The new distinct equation becomes $X_E = X_{\text{new}}$. If $E = \sqrt{E}$

the procedure is similar. The new equation is

$$X_{\text{new}}^2 = P_E(\ldots).$$

The distinct equation is $X_E = X_{\text{new}}$ again.

If the system resulting from an expression tree has maximum degree $d_E$, maximum coefficient size $c_E$, and $n_E$ equations, $(3d_E c_E)^{-n_E d_E^{n_E}}$ is a separation bound for $E$. Note that $n_E - 1$ is the number of square root and division operations involved in $E$. There are alternative ways to derive a system of polynomial equations for an expression $E$. One could also introduce a new variable and a new equation for each operation. That would guarantee degree at most 2 but result in a system with more equations and variables.

Recently Burnikel et al.[12] 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 3. 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 $\alpha(E) \geq u(E)^{1-2^{k(E)-1}}$.

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

**Table 3.** Recursive formulas for quantities $u(E)$ and $l(E)$ of an arithmetic expression involving square roots.

This bound as well as the bound given in [111] involve square root operations. Hence they are not easily computable. In practice one computes ceilings of the results to get integers [111] or maintains integer bounds logarithmically [12, 16]. The `Real/Expr`-package [28, 88] and the number type `real` [16] in LEDA 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 `reals` in LEDA [74] uses the bounds given in [12].

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 $E_{\text{error}}$ 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 $|\tilde{E}| \geq sep(E) + E_{\text{error}}$.

# 4 Geometric Computation with Imprecision

In this section we briefly discuss the basic aspects of the design and implementation of geometric algorithms for calculations with imprecision.

## 4.1 Implementation with Imprecise Predicates

Imprecise arithmetic cannot guarantee correct evaluation of a geometric predicate. It can lead to wrong decisions and wrong results. But even if the result is not the exact result for the considered problem instance, it 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 are known not to be exact either. This observation motivates the definition of robustness and stability given in Section 1.2. 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 [37] requires in addition that the implementation of an algorithm would compute the exact result, if all computations were precise.

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 the requirements on the output are relaxed to allow "more general" perturbations of the input data. Robustness and stability are then defined with respect to the weaker problem formulation. For example, Fortune's and Milenkovic's line arrangement algorithm [40] computes a combinatorial arrangement that is realizable by pseudolines but not necessarily by straight lines. Shewchuk [102] 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 they might be dependent. 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. A relaxation of the problem sometimes 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 could be legal for arrangements of pseudolines. Although an inconsistent state cannot be reached from any legal input it can still contain useful information.

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* [37, 65].

It can be hard to achieve consistency with previous evaluations. For example, 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 [37].

For the incremental construction of Voronoi diagrams of points Sugihara et al. show how consistency with previous decisions can be forced [107, 108]. Their algorithm is extremely (quasi-)robust. Some "meaningful" output is computed even if the results of all numerical comparisons are chosen at random. Meaningful means that the computed result is guaranteed to have some topological properties of a Voronoi diagram.

For some basic geometric problems there are stable, robust, or quasirobust implementations of geometric algorithms. Li and Milenkovic [67], Guibas et al. [53, 52], and Kawaguchi et al. [19] consider the convex hull problem in two dimensions, Barber [6] considers convex hulls and related problems, Hopcraft, Hoffmann, and Karasick [57] and Hopcroft and Kahn [58] consider intersection of polygons and convex polyhedra respectively. Fortune and Milenkovic [40] and Milenkovic [77] consider line arrangements. Fortune [39] considers the Delauney triangulation of point sets in two-dimensional space and Dey et al. [26] in three-dimensional space. For modelling polygonal regions in the plane Milenkovic [77] uses a technique called data normalization to modify the input such that it can be processed with imprecise arithmetic. Pullar [94] describes possible applications of these techniques to GIS. Sugihara and Iri present a solid modelling system free from topological errors [109].

The techniques used in these algorithms are fairly special and it seems unlikely that they can be easily transferred to other geometric problems. A general theory showing how to implement geometric algorithms with imprecise predicates is still a distant goal.

### 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 [52]. Instead of a boolean value, an epsilon predicates returns a real number that gives some information "how much" the input satisfies the predicate. In epsilon geometry the size of a perturbation is measured by a non-negative real number. 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 $\varrho$ if the predicate is still satisfied after perturbing with any perturbations of size at most $-\varrho$. In [52] 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 [52, 53]. Reasoning in the epsilon geometry framework seems to be difficult.

### 4.3 Axiomatic Approach

In [97, 98] 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 [97] is computing a closest pair of a set of points $S$ by plane sweep [54]. Instead of a closest pair, the distance $\delta_S$ 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 [97] are sufficient to prove that for the $\delta$ computed by Schorn's plane sweep implementation

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

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 could be 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 $\delta_S$, where $\varepsilon_{\text{prec}}$ is machine precision.

Further geometric problems to which the axiomatic approach is applied in [97] 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, which is also discussed in [36].

# 5 Related Issues

## 5.1 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 piping 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 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 [106]. However, rounding geometric data is more complicated than rounding numbers and can be very difficult [78], because combinatorial and numerical data have to be kept consistent.

An intensively studied example is rounding an arrangement of line segments, the underlying geometric structure of cartographic maps. Greene and Yao [50] 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, introduced by 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 are snapped to the pixel center, cf. Fig. 5. Snap-rounding is used in [55, 51, 48]. 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 [51] and [48].

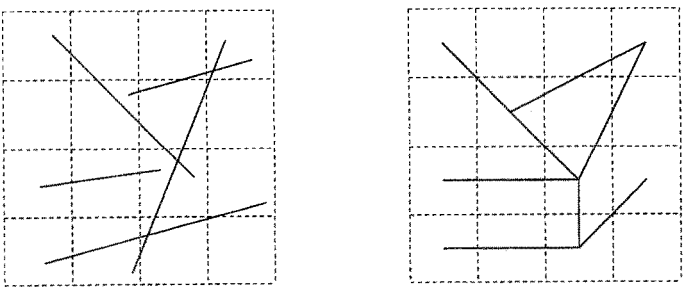

**Fig. 5.** Snap-rounding line segments

## 5.2 Inaccurate Data

Cartographic data are inherently inaccurate. Sometimes, they can nevertheless be treated as exact. In preprocessing and postprocessing steps, input and output data respectively might have to be "cleaned up". For example, in map overlay, spurious or sliver polygons [20] have to be removed that result from the overlay of objects which are identical in the real world but not in the overlaid maps. Treating inaccurate data as exact 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, 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.

The alternative to treating possibly inaccurate data as exact is to incorporate uncertainty into the problem statement and to develop and use algorithms solving the resulting problems (exactly). Goodchild [47] gives an overview on approaches to incorporate inaccuracy and uncertainty in cartographic data in GIS. For example, tolerance regions can be added to geometric objects to model inaccuracy and uncertainty in the data, see e.g. [35]. Inaccuracies in the position of points can be modelled by epsilon circles, inaccuracies in lines by a Perkal epsilon band [90].

Pullar discusses consequences of using tolerance circles to point coincidence and point clustering problems [95]. Similar to point coincidence under the "close to zero is zero" rule, transitivity is a problem, cf. Fig. 6, if points are considered as coincident if their tolerance regions overlap. In [95] clustering of points is considered to solve the coincidence problem.

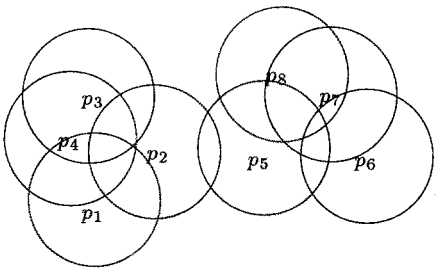


**Fig. 6.** Points with circular tolerance regions. An obvious clustering would be $\{\{p_1, \ldots, p_4\}, \{p_5, \ldots, p_8\}\}$.

Enhancing tolerance regions with a probability distribution leads to a better model of uncertainty. An example of this approach is modelling coordinates $x_1, \ldots, x_d$ of a point position which is known to be possibly inaccurate by probability distributions $X_1, \ldots, X_d$ such that the mean of $X_i$ is at $x_i$. As with computation with imprecision, a lot of research on modelling and handling uncertainty in geometric data is still needed.

### 5.3 Geometric Algorithms in a Library

The purpose of a library is to provide reusable software components. Reusability requires generality. The components must be usable in or adaptable to various applications. Generality as such is not sufficient. The components must not only be adaptable, they must also lead to efficient solutions.

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. Clearly, correctness in the sense of reliability should be beyond question for geometric algorithms and primitives in a library. However, by far not all implementations of geometric algorithms are correct. Many implementations of geometric algorithms pretend to solve a geometric problem, but for a not-clearly-specified set of problems instances they don't. Due to missing or improper handling of special cases or just incorrect coding of complicated parts and especially to precision problems, many implementations of geometric algorithms disappoint the user occasionally by unexpected failures, break downs, or computing garbage.

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.

A good library is more than just a collection of reusable software. It provides reliable, reusable components that can be combined in a fairly seamless way. Due to the composition problem with imprecise predicates described in Section 4.1, even stable, imprecise predicates are not very useful as library components. Building a library upon exact geometric predicates is much easier. With exact predicates, algorithms developed under the real computation model can be implemented in a straightforward way. A redesign that deals with imprecision in the predicates is not necessary. Exact basic predicates can simplify the task of implementing approximation algorithms as well. For input data that are known to be inaccurate, exactness is not so important. Correctness in the sense of reliability is then the primary goal, not exactness, but currently exact geometric computation seems to be the safest way to reach it.

Among the library and workbench efforts in computational geometry [3, 45, 63, 24, 74, 85] the XYZ-Geobench and LEDA deserve special attention concerning precision and robustness. In XYZ-Geobench [85, 96] the axiomatic approach to robustness, described in section 4.3, is used. In LEDA [73, 74] 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 projects called CGAL (Computational Geometry Algorithms Library) [34, 89] and GeomLib [5] 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

In his book on randomization and geometry [82] Mulmuley writes

> *Dealing with the finite nature of actual computers is an art that requires infinite patience.*

Nevertheless, the precision problem is almost ignored and left to the implementor in the textbooks on computational geometry, for sake of simplicity and readability of presentation. The emphasis is on understanding an algorithm and their correctness over the reals rather than on implementation issues of these algorithms. More than a half page description of the precision problems is hardly given.

Despite a lot of research having been done on the precision and robust problem, no satisfactory general-purpose solution has been found. There is no consensus in the geometry literature on how to deal with precision problems. Some researchers want to use fast floating-point arithmetic exclusively and hence investigate design and implementation of algorithms with imprecise predicates. Others prefer exact geometric computation, because it allows fairly straightforward implementation of geometric algorithms designed for the real RAM model [91] and sometimes because they want to use perturbation schemes. Exact geometric computation seems to be the more practical approach to reach reliability, especially if number packages supporting exact geometric computation [29, 13] are available. However, there need not be a consensus. Both approaches have their merits.

Practitioners often ask for the impossible. Algorithms computing exact or at least highly accurate results are requested to be competitive in performance to algorithms that sometimes crash or exhibit otherwise unexpected behavior. Efficiency is compared for inputs that all of them handle. That is somewhat unfair. It should be clear that one has to pay for the detection of degenerate and nearly degenerate situations, but it should also be clear, that one gets much more.

Surely, this survey is incomplete and biased. Most of the presentation is devoted to exact geometric computation. Implementation with imprecise primitives has gained less attention here, because it lacks generality and its application is

much less straightforward. Related surveys on the problem of precision and robustness in geometric computation are given by Fortune [38], Hoffmann [56], and Yap [110]. Franklin [44] especially discusses cartographic errors caused by precision problems.

### Acknowledgements

Work on these notes was partially supported by the ESPRIT IV LTR Project No. 21957 (CGAL).

## References

1. A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.
2. G. Alefeld and J. Herzberger. *Introduction to Interval Computation.* Academic Press, New York, 1983.
3. F. Avnaim. *C++GAL: A C++ Library for Geometric Algorithms*, 1994.
4. F. Avnaim, J.D. Boissonnat, O. Devillers, F.P. Preparata, and M. Yvinec. Evaluating signs of determinants using single precision arithmetic. Technical Report 2306, INRIA Sophia-Antipolis, 1994.
5. J.E. Baker, R. Tamassia, and L. Vismara. GeomLib: Algorithm engineering for a geometric computing library, 1997. (Preliminary report).
6. J.L. Barber. Computational geometry with imprecise data and arithmetic : Phd Thesis. Technical Report CS-TR-377-92, Princeton University, 1992.
7. M.O. Benouamer, P. Jaillon, D. Michelucci, and J-M. Moreau. A "lazy" solution to imprecision in computational geometry. In *Proc. of the 5th Canad. Conf. on Comp. Geom.*, pages 73–78, 1993.
8. J.D. Boissonnat and M. Yvinec. *Algorithmic Geometry.* Cambridge University Press, Cambridge, UK, 1997.
9. 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.
10. H. Brönnimann and M. Yvinec. Efficient exact evaluation of signs of determinants. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 166–173, 1997.
11. C. Burnikel. *Exact Computation of Voronoi Diagrams and Line Segment Intersections.* PhD Thesis, Universität des Saarlandes, Saarbrücken, Germany, 1996.
12. 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.
13. C. Burnikel, J. Könemann, K. Mehlhorn, S. Näher, S. Schirra, and C. Uhrig. Exact geometric computation in LEDA. In *Proceedings of the 11th ACM Symposium on Computational Geometry*, pages C18–C19, 1995.
14. C. Burnikel, K. Mehlhorn, and S. Schirra. How to compute the Voronoi diagram of line segments: Theoretical and experimental results. In *ESA94*, pages 227–239, 1994.
15. C. Burnikel, K. Mehlhorn, and S. Schirra. On degeneracy in geometric computations. In *Proc. of the 5th ACM-SIAM Symp. on Discrete Algorithms*, pages 16–23, 1994.

16. 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.
17. J.F. Canny. *The Complexity of Robot Motion Planning.* PhD Thesis, 1987.
18. J.F. Canny. Generalised characteristic polynomials. *J. Symbolic Computation*, 9:241–250, 1990.
19. Wei Chen, Koichi Wada, and Kimio Kawaguchi. Parallel robust algorithms for constructing strongly convex hulls. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 133–140, 1996.
20. N.R. Chrisman. The accuracy of map overlays: a reassessment. In D.J. Peuquet and D.F. Marble, editors, *Introductory Readings in Geographic Information Systems*, pages 308–320. Taylor & Francis, London, 1990.
21. K. L. Clarkson. Safe and effective determinant evaluation. In *Proc. 33rd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 387–395, 1992.
22. 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.
23. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry.* Springer Verlag, 1997.
24. 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.
25. T.J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18:224 – 242, 1971.
26. T.K. Dey, K. Sugihara, and C.L. Bajaj. Delaunay triangulations in three dimensions with finite precision arithmetic. *Computer Aided Geometric Design*, 9:457–470, 1992.
27. 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.
28. T. Dubé, K. Ouchi, and C.K. Yap. Tutorial for `Real/Expr` package. 1996.
29. T. Dubé and C.K. Yap. A basis for implementing exact computational geometry. extended abstract, 1993.
30. H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer Verlag, 1986.
31. 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.
32. I. Emiris and J. Canny. A general approach to removing degeneracies. In *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Sience*, pages 405–413, 1991.
33. 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.
34. A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. The CGAL kernel : a basis for geometric computation. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry : Towards Geometric Engineering (WACG96)*, pages 191–202. Springer LNCS 1148, 1996.
35. S. Fang and B. Brüderlin. Robustness in geometric modeling - tolerance based methods. In *Proc. Workshop on Computational Geometry CG'91*, pages 85–102. Springer Verlag LNCS 553, 1991.
36. 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.

37. S. Fortune. Stable maintenance of point-set triangulations in two dimensions. In *Proceedings of the 30th IEEE Symposium on Foundations of Computer Sience*, pages 494–499, 1989.
38. S. Fortune. Progress in computational geometry. In R. Martin, editor, *Directions in Geometric Computing*, pages 81 – 128. Information Geometers Ltd., 1993.
39. S. Fortune. Numerical stability of algorithms for 2D Delaunay triangulations and Voronoi diagrams. *Int. J. Computational Geometry and Applications*, 5:193–213, 1995.
40. S. Fortune and V. Milenkovic. Numerical stability of algorithms for line arrangements. In *Proc. of the 7th ACM Symp. on Computational Geometry*, pages 334–341, 1991.
41. S. Fortune and C. van Wyk. Efficient exact arithmetic for computational geometry. In *Proc. of the 9th ACM Symp. on Computational Geometry*, pages 163–172, 1993.
42. S. Fortune and C. van Wyk. *LN user manual*, 1993.
43. S. Fortune and C. Van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Transactions on Graphics*, 15(3):223–248, 1996.
44. W.R. Franklin. Cartographic errors symptomatic of underlying algebra problems. In *Proc. International Symposium on Spatial Data Handling*, volume 1, pages 190–208, Zürich, 20–24 August 1984.
45. G.-J. Giezeman. *PlaGeo, a library for planar geometry, and SpaGeo, a library for spatial geometry*, 1994.
46. D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, pages 5–48, 1991.
47. M.F. Goodchild. Issues of quality and uncertainty. In J.C. Muller, editor, *Advances in Cartography*, pages 113–139. Elsevier Applied Science, London, 1991.
48. 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.
49. T. Granlund. *GNU MP, The GNU Multiple Precision Arithmetic Library*, 2.0.2 edition, June 1996.
50. D. Greene and F. Yao. Finite resolution computational geometry. In *Proc. of the 27th IEEE Symposium on Foundations of Computer Science*, pages 143–152, 1986.
51. L. Guibas and D. Marimont. Rounding arrangements dynamically. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 190–199, 1995.
52. L. Guibas, D. Salesin, and J. Stolfi. Epsilon geometry: Building robust algorithms from imprecise computations. In *Proc. of the 5th ACM Symp. on Computational Geometry*, pages 208–217, 1989.
53. L. Guibas, D. Salesin, and J. Stolfi. Constructing strongly convex approximate hulls with inaccurate primitives. In *Proc. SIGAL Symp. on Algorithms*, pages 261–270, Tokyo, 1990.
54. K. Hinrichs, J. Nievergelt, and P. Schorn. An all-round sweep algorithm for 2-dimensional nearest-neighbor problems. *Acta Informatica*, 29:383–394, 1992.
55. J.D. Hobby. Practical line segment interscetion with finite precision output. Technical Report 93/2-27, Bell Laboratories (Lucent Technologies), 1993.
56. C.M. Hoffmann. The problem of accuracy and robustness in geometric computation. *IEEE Computer*, pages 31–41, March 1989.

57. C.M. Hoffmann, J.E. Hopcroft, and M.S. Karasick. Towards implementing robust geometric computations. In *Proc. of the 4th ACM Symp. on Computational Geometry*, pages 106–117, 1988.
58. J.E. Hopcroft and P.J. Kahn. A paradigm for robust geometric algorithms. *Algorithmica*, 7:339–380, 1992.
59. K. Jensen and N. Wirth. *PASCAL- User Manual and Report. Revised for the ISO Pascal Standard.* Springer Verlag, 3rd edition, 1985.
60. 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.
61. M. Karasick, D. Lieber, and L.R. Nackman. Efficient Delaunay triangulation using rational arithmetic. *ACM Transactions on Graphics*, 10(1):71–91, 1991.
62. R. Klein. *Algorithmische Geometrie.* Addison-Wesley, 1997. (in German).
63. A. Knight, J. May, M. McAffer, T. Nguyen, and J.-R. Sack. A computational geometry workbench. In *Proc. 6th Annu. ACM Sympos. Comput. Geom.*, page 370, 1990.
64. D.E. Knuth. *The Art of Computer Programming Vol. 2: Seminumerical Algorithms.* Addison-Wesley, 2nd edition, 1981.
65. Donald E. Knuth. *Axioms and Hulls*, volume 606 of *Lecture Notes in Computer Science.* Springer-Verlag, Heidelberg, Germany, 1992.
66. M.J. Laszlo. *Computational geometry and computer graphics in C++.* Prentice Hall, Upper Saddle River, NJ, 1996.
67. Z. Li and V. Milenkovic. Constructing strongly convex hulls using exact or rounded arithmetic. *Algorithmica*, 8:345–364, 1992.
68. LiDIA -Group, Fachbereich Informatik Institut für Theoretische Informatik TH Darmstadt. *LiDIA Manual A library for computational number theory*, 1.3 edition, April 1997.
69. 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.
70. K. Mehlhorn. *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry.* Springer Verlag, 1984.
71. K. Mehlhorn and S. Näher. Implementation of a sweep line algorithm for the straight line segment intersection problem. Technical Report MPI-I-94-160, Max-Planck-Institut für Informatik, 1994.
72. K. Mehlhorn and S. Näher. The implementation of geometric algorithms. In *13th World Computer Congress IFIP94*, volume 1, pages 223–231. Elsevier Science B.V. North-Holland, Amsterdam, 1994.
73. K. Mehlhorn and S. Näher. LEDA, a platform for combinatorial and geometric computing. *Communications of the ACM*, 38:96–102, 1995.
74. K. Mehlhorn, S. Näher, and C. Uhrig. *The LEDA User manual*, 3.5 edition, 1997. cf. `http://www.mpi-sb.mpg.de/LEDA/leda.html`.
75. M. Mignotte. Identification of algebraic numbers. *Journal of Algorithms*, 3:197–204, 1982.
76. M. Mignotte. *Mathematics for Computer Algebra.* Springer Verlag, 1992.
77. V. Milenkovic. Verifiable implementations of geometric algorithms using finite precision arithmetic. *Artificial Intelligence*, 37:377–401, 1988.
78. 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.

79. R.E. Moore. *Interval Analysis.* Prentice-Hall, Englewood Cliffs, NJ, 1966.
80. R.E. Moore. *Methods and Applications of Interval Analysis.* SIAM, Philadelphia, 1979.
81. S.P. Mudur and P.A. Koparkar. Interval methods for processing geometric objects. *IEEE Computer Graphics and Applications*, 4(2):7–17, 1984.
82. K. Mulmuley. *Computational Geometry : An Introduction through Randomized Algorithms.* Prentice Hall, Englewood Cliffs, NJ, 1994.
83. J. Nievergelt and K. H. Hinrichs. *Algorithms and Data Structures : with Applications to Graphics and Geometry.* Prentice Hall, Englewood Cliffs, NJ, 1993.
84. J. Nievergelt and P. Schorn. Das Rätsel der verzopften Geraden. *Informatik Spektrum*, (11):163–165, 1988. (in German).
85. 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.
86. J. O'Rourke. *Computational geometry in C.* Cambridge University Press, Cambridge, 1994.
87. 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.
88. K. Ouchi. Real/Expr: Implementation of exact computation, 1997.
89. M. Overmars. Designing the computational geometry algorithms library CGAL. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry : Towards Geometric Engineering (WACG96)*, pages 53–58. Springer LNCS 1148, 1996.
90. J. Perkal. On epsilon length. *Bulletin de l'Académie Polonaise des Sciences*, 4:399–403, 1956.
91. F. Preparata and M.I. Shamos. *Computational Geometry.* Springer Verlag, 1985.
92. D.M. Priest. Algorithms for arbitrary precision floating point arithmetic. In *10th Symposium on Computer Arithmetic*, pages 132 – 143. IEEE Computer Society Press, 1991.
93. 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.
94. D. Pullar. Spatial overlay with inexact numerical data. In *Proc. of Auto-Carto 10*, pages 313–329, 1991.
95. D. Pullar. Consequences of using a tolerance paradigm in spatial overlay. In *Proc. of Auto-Carto 11*, pages 288–296, 1993.
96. P. Schorn. An object-oriented workbench for experimental geometric computation. In *Proc. 2nd Canad. Conf. Comput. Geom.*, pages 172–175, 1990.
97. P. Schorn. *Robust Algorithms in a Program Library for Geometric Algorithms.* PhD Thesis, Informatik-Dissertationen ETH Zürich, 1991.
98. P. Schorn. An axiomatic approach to robust geometric programs. *J. Symbolic Computation*, 16:155–165, 1993.
99. P. Schorn. Degeneracy in geometric computation and the perturbation approach. *The Computer Journal*, 37(1):35–42, 1994.
100. R. Seidel. The nature and meaning of perturbations in geometric computations. In *STACS94*, 1994.
101. 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.

102. 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.
103. J. R. Shewchuk. Triangle: Engineering a 2D quality mesh generator and delaunay triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry : Towards Geometric Engineering (WACG96)*, pages 203–222, 1996.
104. IEEE Standard. 754-1985 for binary floating-point arithmetic. *SIGPLAN*, 22:9–25, 1987.
105. K.G. Suffern and E.D. Fackerell. Interval methods in computer graphics. *Computers & Graphics*, 15(3):331–340, 1991.
106. K. Sugihara. On finite-precision representations of geometric objects. *J. Comput. Syst. Sci.*, 39:236–247, 1989.
107. K. Sugihara. A simple method for avoiding numerical errors and degeneracies in Voronoi diagram construction. *IEICE Trans. Fundamentals*, E75-A(4):468–477, 1992.
108. K. Sugihara and M. Iri. Construction of the Voronoi diagram for over $10^5$ generators in single-precision arithmetic. In *Abstracts 1st Canad. Conf. Comput. Geom.*, page 42, 1989.
109. K. Sugihara and M. Iri. A solid modelling system free from topological inconsistency. *Journal of Information Processing*, 12(4):380–393, 1989.
110. C. K. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *CRC Handbook in Computational Geometry*. CRC Press. (to appear).
111. C. K. Yap and T. Dubé. The exact computation paradigm. In D.Z. Du and F. Hwang, editors, *Computing in Euclidean Geometry*, pages 452–492. World Scientific Press, 1995. 2nd edition.
112. 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.
113. C.K. Yap. Symbolic treatment of geometric degeneracies. *J. Symbolic Comput.*, 10:349–370, 1990.
114. 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).