# The Robustness Issue

D. Michelucci
Ecole des Mines, F-42023 Saint-Étienne 02
*micheluc@emse.fr*

**Abstract**

*This article first recalls with some examples the damages that numerical inaccuracy of floating point arithmetic can cause during geometric computations, in methods from Computational Geometry, Computer Graphics or CADCAM. Then it surveys the various approaches proposed to overcome inaccuracy difficulties. It seems that the only way to achieve robustness for existing methods from Computational Geometry is exact computation, it is the "Exact Computation Paradigm" of C.K. Yap and T. Dubé. In Computer Graphics or CADCAM, people prefer to abandon methods and data structures not robust enough against inaccuracy, namely Boundary Representations and related methods, this may be called the "Approximate Computation Paradigm".*

## 1 Introduction

### 1.1 The stakes

Geometric modellers provided by commercial CADCAM softwares or Computer Graphics packages, and methods from the more theoretical field of CG (*Computational Geometry*) all perform geometric computations. For instance, triangulating or meshing geometric domains for finite elements simulation, or computing intersections between geometric objects. Inaccuracy is a crucial issue for geometric computations. Not only numerical results can be inaccurate but geometric programs can crash, enter in infinite loops, or terminate with inconsistent results, the topology of which is the topology of no possible geometric objects. This lack of robustness is especially true with the most sophisticated methods, *i.e.* methods from CG.

The lack of robustness against inaccuracy, the fact that they do not take into account degeneracies (or if they do, it is with prohibitive cost, using an exact arithmetic and a perturbation technique) is one of the reasons why methods and data structures from CG are seldom used in Computer Graphics and CADCAM. For instance in the ACM Symposiums on Solid Modeling and Applications, less than 1 percent of the articles was the concern of CG, which is in cruel contrast with the pretensions of CG. Another reason is that CG says nothing about some basic geometric problems met every day in real applications (for example the computation of the boundary of semi algebraic objects). Another well-known reason was that first CG methods were too often difficult, if not impossible, to implement. Fortunately, randomized methods break off with this prejudicial habit of CG.

### 1.2 Exact Computation Paradigm *vs* Approximate Computation Paradigm

To avoid falling into disuse, solving the arithmetical issues is thus a crucial stake for the future of CG. This article emphasizes that exact computing (which do not need all operations to be performed in exact arithmetic) is the only way to achieve robustness for the existing methods in CG. This is in accordance with the "Exact Computation Paradigm" of Chee K. Yap and Thomas Dubé [DY95]. This idea is quite recent in the CG field, in contrast with the community of symbolic and algebraic computation, where the essential need for exact computations has always been found natural and obvious.

On the other side, the trend in CADCAM fields is to take another way, more practicable, to achieve robustness. The main idea is to stop considering the boundary of geometric objects (in complete opposition with CG), and to be content with conservative approximations for the interior or exterior of geometric objects. These approximations are computed from a reference definition, typically some non-evaluated description of a semi algebraic set called CSG (*Constructive Solid Geometry*) in CADCAM and Computer Graphics, using ray casting or marching methods, or methods combining Interval Analysis and recursive space subdivision. Such approach can be called the "Approximate Computation Paradigm", in opposition to the "Exact Computation Paradigm".

## 1.3 Other arithmetical issues

This article only deals with the inaccuracy problem, but the latter is not the only arithmetic issue for geometric computations: there are two others.

The first problem is due to the overwhelming number of degenerate geometric cases (alignment of more than two points, coplanarity of more than three points, cocircularity of more than three points, intersection of more than two lines in a point, parallelism between lines, etc) that geometric methods have to handle and programmers have to treat. It is not obvious that this is an arithmetic problem, but there is an arithmetic solution. It symbolically perturbs the data by infinitely small values to remove degeneracies [EM90, EC92, Mic95, Yap88], using a non-standard arithmetic (*i.e.* an arithmetic computing with non-standard, infinitely small numbers). However, this arithmetic solution has a drawback: it requires an exact arithmetic. Note that inaccuracies and degeneracies pose the so-called "robustness problem".

The second arithmetic issue for geometric computations is that data acquired from some sensors, or mechanical products machined by imperfect tools, are also known only up to some finite precision: some CADCAM applications need to take into account this other kind of inaccuracy [Jus92], for instance metrology. These two issues are beyond the scope of this article.

## 1.4 Outlines

- Section 2 explains the typical damages due to inaccuracy. Since the many decisions made in "If Then Else" tests by a geometric program are not all independent, wrong decisions can introduce fatal inconsistencies. The latter are more serious for the fastest methods than for the naïve ones: fastest methods run –and so depend– more on geometric properties, which are invalidated by inaccuracy.

- Section 3 presents heuristics approaches. Computer Scientists have first hoped that they would be sufficient. The $\epsilon$ heuristic (Section 3.1) declares strictly equal numbers equal up to a prescribed threshold, classically called $\epsilon$. The $\epsilon$ heuristic is based on a correct intuition (Section 3.2), which has later been used in exact "gap arithmetics". Probabilistic arithmetics (Section 3.3) typically resort to modular computations; if a rational number vanishes modulo some big enough prime and is small, one may take the risk to decide it is zero. Careful Programming (Section 3.4) uses a set of empirical rules to avoid the more obvious inconsistencies. It is a first step towards the Quest for Consistency (Section 3.5). The latter intends to take decisions which are always consistent though sometimes numerically wrong, unfortunately this approach stumbles over untractable problems. Last, the Fuzzy approach (Section 3.6) merges too close geometric entities: it is a recent variant of the $\epsilon$ heuristic. It does not seem CG methods can be reformulated according to this scheme.

- Section 4 discusses the Exact Computation Paradigm. Due to the flaws of heuristics approaches, it seems this solution is the only way to save classical CG methods from the ravages of inaccuracy. Typical exact rational arithmetics are first presented in Section 4.2: the poor man's exact arithmetic (Section 4.2.1), the LN library (Section 4.2.2), the lazy arithmetic (Section 4.2.3). These rational arithmetics have proved to be usable, but they are not always sufficient as some problems require exact algebraic arithmetics. The latter are a topic in its infancy in CG, Computer Graphics or CADCAM. The article presents two quadratic arithmetics (*i.e.* supplying the square root, and the usual arithmetic operations), the first is based on towers of quadratic extensions (Section 4.3.1), and the second on gap theorems

(Section 4.3.2). Possible algebraic arithmetics can be based on the $D_5$ representation (Section 4.3.3), on Gröbner's bases (Section 4.3.4), or on resultants (Section 4.3.5). This section concludes with the rounding problem (Section 4.5), the difficulty and importance of which have been under estimated so far. Rounding operations are needed to communicate between the floating point world and the exact one, and to stop the exponential growth of the algebraic complexity (Section 4.5.1). Rounding a geometric object while maintaining its topology turns out to be NP-complete (Section 4.5.2). A first consequence is that geometric methods and related data structures must be modified to withstand rounding operations which may modify the topology, for instance by introducing self intersections (Section 4.5.2). A second consequence of rounding (Section 4.5.3) is that CG basically deals with discrete problems and not continuous ones, but this discreteness has not been really exploited up to now.

- Section 5 discusses the Approximate Computation Paradigm. Geometric objects are first described by a semantic description, for instance a so-called *CSG* tree (Section 5.2). This description is only implicit. Basic informations like the object boundary or the number of connected components are not available. Then this description is evaluated, up to a prescribed accuracy, with robust methods: Interval Analysis and recursive subdivision of space (Section 5.3), marching method (Section 5.4), ray casting (Section 5.5), or ray representation (Section 5.6). These approximate methods are insensitive to inaccuracy and only require floating point arithmetic. An even more radical evaluation method is the discretization of the object (Section 5.7).

- Section 6 brings some clarifications. First, approximate representations are sufficient and are as relevant as exact ones, despite the pejorative connotation of the word "approximate". Second, the classification into: heuristic, approximate, exact methods is also discussed.

- Section 7 concludes.

# 2 The ravages of numerical inaccuracy

The reader already aware of the ravages of inaccuracy on geometric methods, especially from CG, can skip this section (partly already published in [Mic96]).

## 2.1 Notations

We will use the following notations :

1. Straight lines with equation $\alpha x + \beta y + \gamma = 0$ are represented by a triple of floating point numbers $(\alpha, \beta, \gamma)$.
2. The triple for the line through two distinct points $A$ and $B$ is
$$(y_B - y_A, x_A - x_B, x_B y_A - x_A y_B);$$
3. The intersection between two given lines $D : (\alpha, \beta, \gamma)$ and $D' : (\alpha', \beta', \gamma')$ is the point $(x_\Omega, y_\Omega)$ with

$$\Delta = \det \begin{vmatrix} \alpha & \beta \\ \alpha' & \beta' \end{vmatrix} = \alpha\beta' - \alpha'\beta \quad (1)$$

$$\Delta x = \det \begin{vmatrix} -\gamma & \beta \\ -\gamma' & \beta' \end{vmatrix} = \beta\gamma' - \beta'\gamma \quad (2)$$

$$\Delta y = \det \begin{vmatrix} \alpha & -\gamma \\ \alpha' & -\gamma' \end{vmatrix} = \alpha'\gamma - \alpha\gamma' \quad (3)$$

$$x_\Omega = \frac{\Delta x}{\Delta}, \quad y_\Omega = \frac{\Delta y}{\Delta} \quad (4)$$

4. Geometric algorithms often use the *lexicographical order* on coordinates, defined by:

$$(x, y) <_L (x', y') \Leftrightarrow x < x' \text{ or } (x = x' \text{ and } y < y').$$

A line with $\beta = 0$ (respectively $\alpha = 0$) *i.e.* parallel to the $Oy$ (respectively $Ox$) axis is said to be vertical (respectively horizontal).

The notation $a^*$ denotes the floating point approximation of $a$. *fp* is a shortcut for "floating point". $\lfloor x \rfloor$ stands for $x$ floor, and $\lceil x \rceil$ for $x$ ceil.

## 2.2 Inaccuracy in Computer Graphics

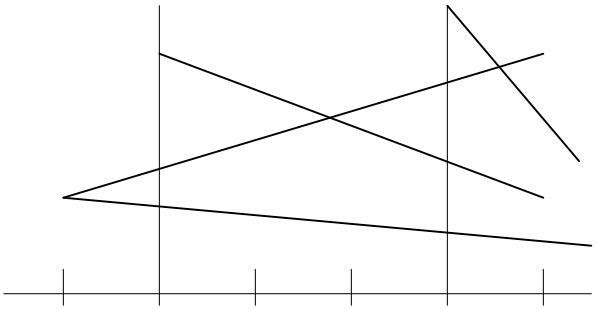

Figure 1: *A span, delimited by two vertical lines.*

This section studies the consequences of inaccuracy on a $2D$ algorithm. The data are segments. For all lines $L_x$ parallel to the $Oy$ axis and having integer abscissa: $x \in \{0, 1, 2 \ldots N\}$, the problem is to find the segments crossing $L_x$ and to sort them by increasing ordinate. This problem occurs in Computer Graphics as a sub-problem in Atherton's method, which computes images of $3D$ scenes defined by boolean combinations (intersection, union and difference) of polyhedra.

The standard method first determines the set of *spans*: a span is a maximal interval $[x_0, x_1]$ so that $]x_0, x_1[$ contains no initial vertex, but possibly contains intersection points. This stage may be achieved for instance by sorting all endpoints by increasing abscissa, then by scanning the resulting sorted list $V$ of vertices while maintaining the set $S$ of active segments: if $v_j = (x_j, y_j)$ is the left (respectively the right) vertex of the segment $s_j$, insert $s_j$ in $S$ (respectively remove $s_j$ from $S$) so that $S$ is still the set of the active segments in the interval $]x_j, x_{j+1}[$. It is also possible to use some bucket-sort scheme instead.

Let $S$ be the set of the active segments in the current span $]x_0, x_1[$. Let $S_0$ (respectively $S_1$) be the result of sorting $S$ by increasing ordinate in $x = x_0$ (respectively in $x = x_1$). If $S_0$ equals $S_1$ we are done and we study the next span, otherwise let $i$ be the lowest index so that $S_0[i] \neq S_1[i]$. Then these two segments $S_0[i]$ and $S_1[i]$ cut each other somewhere in $[x_0, x_1]$, in $(x_i, y_i)$. In such a case, recursively study spans $]x_0, \lfloor x_i \rfloor [$ and $]\lceil x_i \rceil, x_1[$. At the end, either there is no intersection points inside the span, or $x_0 = x_1$.

Though correct, this method infinitely loops because of inaccuracy, or at least it loops until the stack is filled up with recursive calls: It happens that the computed intersection point has *fp* abscissa $x_i$ outside the interval $[x_0, x_1]$, say between $x_1$ and $x_1 + 1$ (of course, this situation is absurd and cannot occur without inaccuracy). Thus the procedure `span`$(x_0, x_1)$ recursively calls `span`$(x_0, x_1)$: an infinite loop.

This problem is trivially solved: when $x_i$ is greater than $x_1$, cut $[x_0, x_1]$ into $[x_0, x_1 - 1]$. Symmetrically when $x_i$ is lower than $x_0$. It is worth noting that the reliability against inaccuracy of programs from Computer Graphics is much more easily achieved than for the ones from CG.

## 2.3 Inaccuracy in Computational Geometry

This section details the consequences of inaccuracy on a classical and representative algorithm from CG, the Bentley and Ottman's method.

### 2.3.1 Bentley and Ottman's algorithm

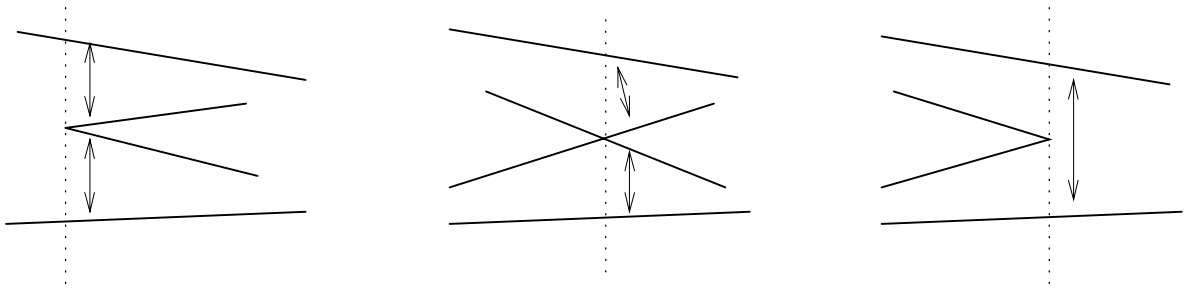

Figure 2: *The situations in which two segments become contiguous along the sweeping line. The vertical dotted line represents the sweeping line, the arrows show the couple of contiguous segments.*

In a nutshell, this algorithm computes all the $k$ intersection points between $n$ segments in the plane, in time $O((n+k)\log n)$. For sake of simplicity, this exposition will ignore degeneracies: intersection points common to more than two segments, vertical segments, intersection points confused with initial vertices, and so on. The principle is to sweep the plane by a vertical line from left to right, *i.e.* initial vertices are swept in lexicographic order, and to maintain the set of the segments crossing the sweeping line, ordered upwards. The method exploits the two following remarks. First, the order of the segments crossing the sweeping line obviously depends on the abscissa of the latter, but it changes only locally when passing an initial vertex or an intersection point: local changes are insertion or deletion of a segment when passing a vertex, or permutation of two intersecting segments when passing an intersection point. Secondly, two segments can cross each other only after they have been contiguous along the sweeping line (ignoring degeneracies). Thus, if each time two segments become contiguous along the sweeping line, the algorithm checks their possible intersection, then all intersection points will be found. The three situations in which two segments become contiguous along the vertical sweeping line are shown in Fig. 2: when a vertex is the beginning (*i.e.* the left vertex) of one or several segments; when a vertex is the end (*i.e.* the right vertex) of one or several segments, or when a vertex is simultaneously an end and a beginning, for instance an intersection point.

The algorithm is as follows. Set $X$ to all initial vertices: $X$ is ordered by the lexicographic order. Let $Y$ be the set of segments crossing the current sweeping line. $Y$ is ordered upwards. $X$ and $Y$ may be represented by balanced trees (actually, in Bentley and Ottman's paper, $X$ is represented with a priority queue). Initially, $Y$ is empty. While $X$ is not empty, do:

Let $p$ be the first point of $X$, and remove $p$ from $X$. Let $b$ and $a$ be the segment just below and just above $e$. If there are some segments with $p$ as right endpoint, remove them from $Y$. If $p$ is the left vertex of some segments $s_1 \dots s_t$ (ordered by increasing slope), insert them in $Y$, then compare $b$ with $s_1$ and $a$ with $s_t$, since $b$ and $s_1$ on the one hand and $a$ and $s_t$ on the other hand become contiguous in $Y$. Otherwise, when there are no segments with left vertex $p$, $a$ and $b$ become contiguous in $Y$ and thus are compared. Each time a new intersection point is found, insert it in $X$ and cut the corresponding segments.

### 2.3.2 A special configuration

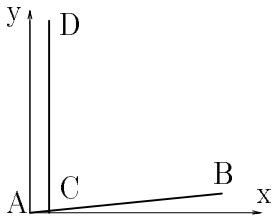


Figure 3: *A special configuration of two segments.*

M. Gangnet communicated me the configuration of segments in Fig. 3 in 1983. Assume for convenience that a *fp* arithmetic with base 10 is used, with 4 significant digits.

$A = (0,\ 0) = A^*$, $B = (10,\ 1) = B^*$, $C = (1,\ 0) = C^*$. Finally $D_u = (1 + u,\ 10) = D_u^*$, assuming there is no cancelation when computing $1.000 + u$. For instance, there is no cancelation for $u = 0.003 \Rightarrow D_u = 1.003 = D_u^*$, but there is one for $u = 0.0003 \Rightarrow D_u = 1.0003 \Rightarrow D_u^* = 1.000$ since there is only 4 significant digits. Afterward, $u$ is such that $D_u = D_u^*$.

The intersection point $I_u = AB \cap CD_u$ is computed as follows:

The coefficients triple for line $AB$ is: $(\alpha,\ \beta,\ \gamma) = (1,\ 10,\ 0)$. No inaccuracy.

The coefficients triple for line $CD_u$ is $(\alpha',\ \beta',\ \gamma') = (10,\ -u,\ -10)$. No inaccuracy.

$\Delta x = \beta\gamma' - \beta'\gamma = -100 = \Delta x^*$. No inaccuracy.

$\Delta y = \alpha'\gamma - \alpha\gamma' = 10 = \Delta x^*$. No inaccuracy.

$\Delta = \alpha\beta' - \alpha'\beta = -100 - u$. Here, $\Delta$ and $\Delta^*$ can be different due to cancelation: for instance, with $u = -0.03$, $\Delta = -100.03$ requires 5 digits to be exactly represented, but since the used *fp* arithmetic has only 4 significant digits, $-100.03$ is truncated to $-100.0$. Thus, $\Delta \neq \Delta^*$. Actually, $\Delta^* = -100.0$ for all $u$ in the interval $]-0.05, +0.05[$ (and maybe more, depending on the used rounding mode: with a rounding towards 0, $\Delta^* = -100.0$ for all $u \in ]-0.1, +0.1[$). Anyway, the points $I_u$ are equal for all $u$ in some interval.

As a first consequence, if a configuration contains segments say $CD_{-0.03}$, $CD_0$, $CD_{0.03}$ on the one hand and $AB$ on the other hand, the *fp* arithmetic cannot represent it in a consistent way – and no matter the algorithm used to compute the intersection points: points $I_{-0.03}$, $I_0$ and $I_{0.03}$ have the same *fp* coordinates, whereas $CI_{-0.03} \subset CD_{-0.03}$, $CI_0 \subset CD_0$, $CI_{0.03} \subset CD_{0.03}$ are different segments.

Of course, this *fp* arithmetic may seem unrealistic, it has been chosen only to simplify the statement. It is possible to find similar configurations for all *fp* arithmetic, no matter the used base and the length of the mantissa. We now follow the behavior of the Bentley-Ottman's method on such a configuration.

### 2.3.3 Inconsistencies due to inaccuracy

Consider the configuration of two segments $AB$ and $D_uC$, with $u < 0$ and $u$ in the inconsistent interval, for instance $u = -0.03$. From now on, we just use $D$ for $D_u$ and $I$ for $I_u$. To compute if point $I^* = (AB \cap CD)^* = (1., 0.1)$ belongs to the segment $DC$, two tests are possible.

First, $I^* \in CD$ iff $D \leq_L I \leq_L C$: with this test (mathematically correct when there is no inaccuracy), $I^*$ does not belong to $CD$. First mistake: the intersection point $I^*$ is forgotten.

A second test, mathematically equivalent to the first one when there is no inaccuracy, is: $I^* \in CD$ iff $x_D \leq x_I \leq x_C$ and $y_C \leq y_I \leq y_D$, taking into account that $DC$ has a negative slope. Here this test will correctly conclude that $I^*$ belongs to $CD$, in contradiction with the previous and theoretically equivalent test. Note it is possible to find other examples in which this last test fails.

• **If $I^*$ is forgotten**. Suppose first that the first test is used, or any test such that $I^*$ is forgotten. Thus, when the sweeping line passes $C$, the program believes that (or in less anthropomorphic words, the data structure $Y$ stores that) $AB$ is below $CD$, which is wrong. Since $C$ is the right endpoint of $DC$, the program has to remove $DC$ from $Y$. When $Y$ is the *only* data structure accessing segments, as it is the case in the original Bentley and Ottman's article, a classic binary search through $Y$ is needed to find $DC$ in $Y$, starting from the $Y$ root. We can suppose that the left (respectively the right) subtree stores the segments below (respectively above) the one of the root. Here the program wrongly believes that $AB$ is below $DC$ in $C$. Suppose for instance the root carries segment $AB$, and its right son carries segment $DC$. To find $DC$, the program compares the height of the searched $DC$ segment and the one of the root segment $AB$, *i.e.* it computes and compares the ordinate of point $DC \cap \{x = x_C\}$ and the one of $AB \cap \{x = x_C\}$. The heights are 0. for $DC$ and 0.1 for $AB$: thus $DC$ is below $AC$ in $C$, which is right, but contradictory with the wrong informations stored in the data structure. Thus the program searches $DC$ in the left subtree of $Y$, and it cannot find it since it is in the right one. This situation is theoretically impossible and a fatal failure occurs, like `Nil pointer dereferenced`.

• **If $I^*$ is not forgotten**. Suppose now that the second test is used, or any test such that $I^*$ is found, when the sweeping line passes $D$. So $DC$ is cut into $DI^*$ and $I^*C$, $AB$ into $AI^*$ and $I^*B$. $I^*$ is then inserted in $X$. Theoretically $I < C$ but for the program: $C = C^* < I^*$. Thus $I^*$ is inserted in $X$ just after $C = C^*$. Thus the next point to be swept is $C$. Here, the program has to remove the segment $I^*C$ from $Y$ (since $C$ is now the right endpoint of $I^*C$). But this segment has not yet been inserted in $Y$. Again, a fatal error occurs.

A possible solution is explained in section 4.2.1.

It is worth comparing the behavior of the naïve method in $O(n^2)$ which compares all couples of segments, and the one of Bentley and Ottman's method. Obviously, the naïve method goes slower (at least when the number of intersection points $k$ is less than $O(n^2)$), but it never crashes; it can output some wrong results, but the latter are not propagated, contrarily to Bentley and Ottman's method. The naïve method is much more robust against inaccuracy.

## 2.4 Inaccuracy and topology

In $2D$ let $P$ be a set of vertices and $S \in P \times P$ a set of non crossing segments, *i.e.* two distinct segments can only cut each other in a known common vertex belonging to $P$. For instance some method has been used to compute all intersection points between an initial set of intersecting segments.

Thus $S$ and $P$ define what is called a *planar map*: a combinatorial structure made of vertices, segments and faces, and supporting various topologic relations of incidence, contiguity or inclusion. Some applications (like Geographic Information Systems) need data structures for modelling such planar maps.

An important notion is the *half-edges* one. A segment is made of two half-edges, the left and the right. The left half-edge (respectively the right one) contains the left vertex (respectively the right one) of the segment, or in more general words its smaller (respectively its greater) vertex for the lexicographic order. Thus if the segment is vertical, the left (respectively the right) half-edge is the one below (respectively the one above). The two half-edges of the same segment are said to be *complementary*.

Moreover each half-edge $e$ having vertex $v$ is linked with its *neighbor*: it is an half-edge also incident to $v$, the first one which is met when turning counterclockwise around $v$ and starting from $e$. In this way all half-edges sharing the same vertex $v$ are cyclically linked around $v$, in the counterclockwise orientation. It is obvious that starting from any initial half-edge, and following the neighbor link will always yield to the starting half-edge, when the planar map is correct.

This *neighbor* link makes also possible to follow face contours: Starting from any half-edge $e_1$, take the complementary of its neighbor to get $e_2$, and then start again from $e_2$ to get $e_3$, and so on until the initial half-edge is reached: this way all half-edges $e_1$, $e_2 \ldots e_1$ of the same contour are listed. Here again,

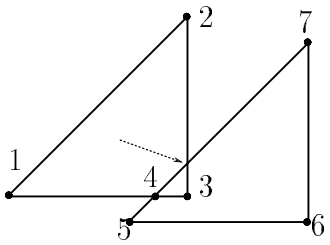


Figure 4: *A vertex has been forgotten: due to an inconsistent topology, following the contours will lead to trouble. Here only one contour will be found, which contains all half-edges:* ...(14), (21), (32), (43), (74), (67), (56), (45), (34), (23), (12), (41), (54), (65), (76), (47), (14)..., *where* $(ij)$ *is the half-edge containing the vertex* $i$*. Of course the correct planar map has one outer contour and three inner ones.*

starting from any half-edge, this travel will always yield to the initial half-edge, when the planar map is correct.

Now, these two properties cannot be guaranteed when the planar map is wrong, for instance when some intersection points have been forgotten (see Fig. 4), or when half-edges incident in the same vertex are too close to be correctly ordered by the neighbor link when using a *fp* arithmetic. In fact, the dual methods of following a contour and of turning around a vertex until the starting half-edge is reached, can enter in an infinite loop never going back to the initial half-edge (for instance with a half-edge sequence like: $e_1$, $e_2$, $e_3$, $e_4$, $e_5$, $e_6$, $e_3$ ...). The method can also terminate, but output inconsistent contours, for instance self intersecting ones (see Fig. 4) which will give trouble to a coloring method or a method locating points in the planar map, for example.

Remark: The previous discussion has used *half-edges*, but *winged-edges* or *quad-edges* or any other topological data structure could have been used as well.

## 2.5 Other examples of contradictions

Basically, inaccuracy invalidates geometric and mathematical properties. This section now gives several examples of such properties, often used by geometric algorithms, but ruined by inaccuracy. It would be too long to detail the consequences on geometric algorithms relying on such property, but they would now be obvious after the previous examples.

### 2.5.1 Example 1

The *power* of a point $M = (x, y)$ relatively to a line $D$ with triple $(\alpha, \beta, \gamma)$ is $\alpha x + \beta y + \gamma$. Suppose that $A$ belongs to the line $D$, because for instance $A$ has been defined as the intersection point between $D$ and another line. Theoretically, this power must be 0. Using *fp* arithmetic, it is unlikely to be true. So a data structure can store topological informations, like $A \in D$, that will be contradicted by numerical computations.

### 2.5.2 Example 2

Consider the three lines of figure number 5. $h$ is nearly horizontal, $m$ climbs up and has slope about 45 degrees, $d$ goes down and has slope about $-45$ degrees. Let the three intersection points be $h' = d \cap m$, $m' = d \cap h$, $d' = m \cap h$. If we exclude the degenerate case in which all intersection points coincide, then there are only two possible lexicographic orderings for them, either $d' <_L h' <_L m'$ or $m' <_L h' <_L d'$; in both cases, $h'$ is between $m'$ and $d'$. Now, with a small enough triangle, the coordinates of the three

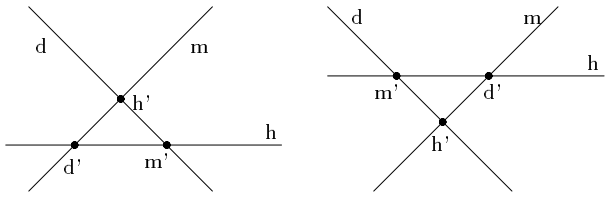


Figure 5: *The two possible lexicographic orders, in the generic case.*

points will differ only by their least significant bits that will likely be corrupted by rounding; so *fp* arithmetic will sometimes produce non consistent triangles impossible to draw.

### 2.5.3 Example 3

The intersection point between a vertical line $AB$ and an oblique one is a point $\Omega$ with the same abscissa as $A$, and $B$. But the evaluation of the previous formula with *fp* arithmetic will most of the time gives a slightly different point $\Omega^*$, such that either $x_A < x_{\Omega^*}$ or $x_{\Omega^*} < x_A$. Actually this also works with "almost vertical" line or segment, as previously seen.

This means that relying on the property : $\Omega \in [A, B] \Leftrightarrow A <_L \Omega <_L B$ (to test if $\Omega$ really belongs to the segment, not only to the straight line) is doomed to failure.

### 2.5.4 Example 4

If $\beta$ and $\beta'$ are both different from zero, the three following definitions for $y_\Omega$ are algebraically equivalent:

$$\frac{\alpha'\gamma - \alpha\gamma'}{\alpha\beta' - \alpha'\beta} \quad , \quad -\frac{\alpha x_\Omega + \gamma}{\beta} \quad \text{and} \quad -\frac{\alpha' x_\Omega + \gamma'}{\beta'}.$$

but evaluated with *fp* arithmetic, they will generally yield different results. This is quite a serious problem, as programmers frequently rely on such identities to detect the equality between two objects constructed in two different and legal ways.

### 2.5.5 Example 5

In the projective plane, if three distinct points $P_1, P_3$, and $P_5$ are collinear, and if the three distinct points $P_2, P_4$, and $P_6$ are also collinear, so are the three points $P_1P_2 \cap P_4P_5$, $P_2P_3 \cap P_5P_6$, and $P_3P_4 \cap P_6P_1$ after Pappus's theorem. Numerical inaccuracy likely prevents the detection of this property. Actually it works as well with all geometric theorems.

### 2.5.6 Example 6

This example will be reused in section 3.5. It intends to give an intuitive insight on the combinatorial structures underlying geometric objects. Consider $N$ points in the plane and for each triple $(A, B, C)$ of distinct points, define $|ABC|$ to be:

$$|ABC| = sign\left(\begin{vmatrix} 1 & 1 & 1 \\ x_A & x_B & x_C \\ y_A & y_B & y_C \end{vmatrix}\right)$$

This determinant is twice the signed area of the triangle $ABC$. As well-known, either $ABC$ turns on the left and $|ABC| = +1$, or $ABC$ turns on the right and $|ABC| = -1$, or $ABC$ are collinear and $|ABC| = 0$.

There are $3^{\binom{N}{3}}$ ways to assign signs for all $\binom{N}{3}$ triples. But of course very few of them are geometrically possible: triples signs are not independent. For instance they must verify $|ABC| = |BCA| = |CAB|$, and $|ABC| = -|BAC|$. Other less obvious rules are: $|ABC| = |BCD| = |CDA| = |DAB| \Rightarrow |ABD| = |BCA| = |CDB| = |ABC|$ (hint: $ABCD$ is convex), and $|ABP| = |BCP| = |CAP| \Rightarrow |ABC| = |ABP|$ (hint: $P$ is inside the triangle $ABC$). See [Knu92] for details.

When the $N$ points are given by their coordinates, it happens that computing $|ABC|$s with *fp* arithmetic yields inconsistent signs. Even when the inaccuracy is numerically negligible, a single error on a sign is sufficient to introduce a fatal inconsistency.

## 2.6 Inaccuracies and inconsistencies

The methods proposed in the CG field are theoretically fast because they exploit *properties of consistent geometric objects*, like properties of total orders, algebraic identities, geometric theorems or deep combinatorial structures underlying geometry. Unfortunately inaccuracy invalidates such properties. Maybe the simplest example is the use of the transitivity in total orders (used for instance in Bentley and Ottman's method): if it is known that $a < b$ and $b < c$, then it is deduced that $a < c$ without comparing $a$ and $c$.

Inconsistent decisions from numerical imprecision have two kinds of consequences on geometric algorithms.

• Either the algorithm crashes or enters in an infinite loop. It is typically the case for the most sophisticated and efficient methods proposed by CG, that rely on various geometrical or mathematical properties, like the transitivity in orders, algebraic identities or theorems, and that propagate (sometimes corrupted) intermediate results, for instance: $a < b$ and $b < c \Rightarrow a < c$. These methods enter in theoretically impossible situations: trying to delete an element which has not yet been inserted from some data structure, or ending up in an infinite loop while scanning a theoretically finite sequence.

• Or the algorithm terminates normally; it is generally the case for "brute-force" algorithms, too stupid to make the most of intermediate results and geometric consistency. However these algorithms yield inconsistent results, for instance a graph supposed to be planar will not be (see example in figure 4). Another algorithm, though mathematically correct and taking this result as its input, can crash, infinitely loop or deliver inconsistent results in turn.

Actual commercial geometric modellers cannot work without the active complicity and understanding of their users, who have learned to avoid the easiest traps, and who accept to slightly perturb their data until their geometric modeller works. Thus practitioners (programmers or users of geometric modellers) have been aware of the inaccuracy difficulties since the first geometric modellers, a roughly twenty-five years.

To overcome them, programmers and users have developed empirical tricks, described as below. These tricks do not always work. For a long time, people in CADCAM community have put up rather willingly with these limitations: first, engineers are used to the limitations of physical devices, and then they considered that, anyhow, it was impossible to do otherwise. However, today, CADCAM engineers are more and more exasperated by this robustness lack, and robustness has become a key issue in CADCAM.

From 1989 on, C. Hoffmann [Hof89] devoted in his book a comprehensive chapter to the robustness issue, studying examples of inaccuracy, pointing out the causes ("*The difficulty seems to be rooted in the interaction of approximate numerical and exact symbolic data*") and proposing approaches like the quest for consistency or the Exact Computation Paradigm – the main things learned since this overview are emphasized in the conclusion (Section 7).

In CG, awareness of the inaccuracy problem is much more recent, dating back from the late eighties, with Milenkovic's work [Mil88]: CG assumes a theoretic model for computers, in which each arithmetical operation is performed exactly in constant time and space. When known, the inaccuracy problem was

not considered worthy of any research, it was just a matter of programming. This is probably the reason why there is up to now so few available libraries solving problems in CG: this is in contrast with the situation in CADCAM, Computer Graphics, Numerical Analysis or Algebraic and Symbolic Algebra.

# 3 Heuristic approaches

## 3.1 The popular $\epsilon$ heuristic

To overcome inaccuracy, the most popular trick used in geometric modellers is the $\epsilon$ heuristic. When two *fp* numbers differ by less than a given threshold traditionally called $\epsilon$, they are considered to be the same. The test can be made in an absolute manner : $|a-b| < \epsilon$, or in a relative one : $|a-b| < \epsilon \times max(|a|,|b|)$. Some modellers use several $\epsilon$, say one for lengths, another for areas, another for angles.

This heuristic loses the equality transitivity: it is easy to find $a$, $b$ and $c$ so that $a =_\epsilon b$, $b =_\epsilon c$, but $a \neq_\epsilon c$ with $=_\epsilon$ meaning "equal for the $\epsilon$ heuristic": thus inconsistencies remain possible.

Moreover, finding the relevant value(s) for $\epsilon$(s) is much of a difficult task, depending on the usual range of numbers (it depends on the applications), and on the format of used *fp* numbers : it is common folklore in CADCAM community that the conversion from 32-bits *fp* numbers to 64-bits has required a not so easy $\epsilon$s updating. Of course the $\epsilon$ heuristic can fail, and sometimes it does. In practice, it seems to work not so bad and to improve the geometric modellers robustness, even though it owes a great deal to active complicity by the users.

## 3.2 Gap arithmetics

The $\epsilon$ heuristic is based on a right intuition, following Canny's gap theorem [Can88]:

**Canny's gap theorem :** *Let $x_1$, $x_2 \ldots x_n$ be the solutions of an algebraic system of $n$ equations and $n$ unknowns, having a finite number of solutions, with maximal total degree $d$, with relative integer coefficients smaller or equal to $M$ in absolute value. Then, for all $i \in [1,n]$, either $x_i = 0$ or $|x_i| > \epsilon_c$ where*

$$\epsilon_c = \frac{1}{(3Md)^{nd^n}}$$

This theorem gives a way to prove *numerically* that a number is zero : compute a (guaranteed) interval containing it, with width smaller than $\epsilon_c$. As soon as the interval does not contain 0, the number is clearly not 0 and its sign is known. Otherwise, if the interval contains 0 and has width less than $\epsilon_c$, the number can only be 0.

Alas, there are several problems. First, $\epsilon_c$ is far much smaller than the epsilon used in geometric modellers; actually $\epsilon_c$ is generally much smaller than the smallest positive *fp* number, even in simple examples. So, an extended arithmetic supplying bigfloats is required. Second, even if such an arithmetic is available, such a computational scheme will have an exponential cost : an exponential number of digits is needed to prove the nullity of a number, because of the term $nd^n$ in Canny's theorem. Now, there is no hope to significantly widen the Canny's gap in the worst case, because it is already sharp; it is almost reached in the following simple case : $x_1(Mx_1 - 1) = 0$, $Mx_2 - x_1^2 = 0 \ldots Mx_n - x_{n-1}^2 = 0$. A possibility will be to find a more convenient $\epsilon$ depending on the system at hand, and not only on $d$, $M$ and $n$. More on gap arithmetics in section 4.3.2, more on gap theorems in Yap's book [Yap96].

## 3.3 Probabilistic arithmetics

Another more practicable but only probabilistic method is to compute with some bigfloat library, and to use the $\epsilon$ heuristic, with say $10^{-200}$, and hope that life will not be so bad to produce a counterexample. Some people currently investigate such an approach, but I don't know any publication.

Another probabilistic trick stems from modular arithmetic. The idea is to perform all computations modulo one (or several) finite field $\mathbb{F}_n$ (for instance $\mathbb{Z}/n\mathbb{Z}$ in which $n$ is a prime integer, about $2 \times 10^9$). Clearly, if $a \bmod \mathbb{F}_n$ does not vanish, $a$ cannot be 0, even when $a^*$ is very small: we have to precise $a^*$ in some way to reliably find its sign, for instance using some bigfloat library or some on-line arithmetic. If $a^*$ is small, and if $a \bmod \mathbb{F}_n$ vanishes, one can take the risk to assume $a = 0$. This scheme is straightforward to implement when only rational operations ($+$, $-$, $\times$, $:$) and numbers are used, because each rational number has only one homomorphic element in the finite field. The only difficulty arises when a division by 0 occurs in the finite field, which is very unlikely. Such a scheme has been investigated by A. Agrawal and A. Requicha [AR94], as well as by M. Benouamer and his colleagues [BJMM94] (see section 4.2.3 below about hash coding lazy numbers).

However this approach becomes more problematic when algebraic non rational operations and numbers get involved: for instance each quadratic number have two homomorphic images in the finite field (or in its closure) and it becomes impossible to discriminate them, for example to distinguish the positive from the negative square root in $\mathbb{F}_n$. Thus, to know if an algebraic number vanishes, all its homomorphic images in $\mathbb{F}_n$ or its closure must be tested [MG94].

## 3.4 Careful programming

Some computer scientists prefer to avoid the $\epsilon$ heuristic and have settled a set of tricks:

• Check first in the data structures before computing; for instance, before computing the power of a vertex relatively to some line, verify first if the line is topologically incident to the vertex from the data structures at hand. This avoid the inconsistency of which in 2.5.1.

• Handle in a special way some particular cases, for instance the intersection point between a vertical line and an oblique one. In this case, assign the abscissa with the abscissa of the vertical line, not with the expression $\Delta x/\Delta$. This avoid the inconsistency of which in 2.5.3.

• Do not use several distinct formulas for the same value (though it seems to contradict the previous rule). This avoid the inconsistency of which in 2.5.4.

• The computation of $ab \cap cd$, $ab \cap dc$, $ba \cap cd$, $ba \cap dc$ (they are, for instance, segments in $2D$) generally give slightly different results. Before computing an intersection, it is worth systematically orienting segments at hand, so that $a <_L b$ and $c <_L d$ and so on, thus by exchanging vertices in order to obtain: $ab \cap cd = ab \cap dc = ba \cap cd = ba \cap dc$.

• Use numerical *input* data rather than *derived* (thus corrupted) numerical data.

• Prefer non redundant data structures, to limit the probability of contradictions.

Quoting C. Hoffmann [Hof89]: "*Conceptually we view these heuristics as attempts to reduce the logical interdependence of decisions that are based on numerical computations.*"

M. Iri and K. Sugihara [IS89] have used this kind of approach for computing Voronoï's diagrams. They ensure that their program will never crash because of inaccuracy, that the resulting graph is correct when there is no numerical difficulty, and otherwise that the graph is connected with all vertices having degree 3, like a correct Voronoï's diagram. It is impressive. However remains a great problem: there is strictly no guaranty that another program, mathematically correct, and using this Voronoï's diagram as an input, will not crash.

To conclude, all these stratagems sometimes show wits, but they can only avoid the more obvious inconsistencies. Avoiding more convoluted ones (for instance the non respect of Pappus's theorem, see 2.5.5) and avoiding contradictions between several pieces of software written by different programmers using different conventions, notations and formulas, seems an impossible task. The next section explains why.

## 3.5 Respecting Consistency: the Quest

The aim at careful programming, in Iri and Sugihara's work [IS89], and in some V. Milenkovic's work [Mil88], can be paraphrased as:

In Numerical Analysis, it is possible to find an approximation in the solution of the problem at stake, that is to say the exact solution of another but close problem of the same kind. It is even possible to measure in some way these distances. We would like to do the same in geometric algorithms, *i.e.* to find an approximation in the solution that happens to be the exact solution of another close problem.

It means that, at least, the solution that has been found out must be "geometrically realizable". As for an example, let us read over to the problem in section 2.5.6: let $P_1, P_2 \dots P_n$ be $n$ points in the plane. We want to compute with *fp* arithmetic any of the $|ABC|$; we accept that some triple sign be wrong (relatively to exact arithmetic), but we want to get a consistent signs set, which is the signs set of another set of points $Q_1, Q_2 \dots Q_n$, close to $P_1, P_2 \dots P_n$.

This algorithm will be something like this: all triple signs $ABC$ (yes, there are $O(n^3)$ of such triples, but our problem here is not to obtain an efficient algorithm, but a robust one) are straightforwardly computed with *fp* arithmetic, by the devoted formula, and with an error bound. The value of the majority of signs will be clearly positive, or clearly negative. For remaining ambiguous signs, we want to resort to some oracle who will give us a set of missing signs that is geometrically realizable.

The question now becomes: is it possible to decide if a partial sign system implies such other missing sign, or equivalently, if a given sign system is consistent (geometrically realizable) or not?

This problem is not only a toy problem, because a lot of methods (convex hull computations, intersection between polygons, for example) proposed by CG in $2D$ can be reformulated in order to use only triple signs in geometric tests. For instance, two segments $pq$ and $rs$ intersect each other (with no degeneracy) iff $|pqr| \times |pqs| = -1$ and $|rsp| \times |rsq| = -1$. Thus, for applications in which all geometric tests can be reformulated only with triple sign test, the robustness problem will be theoretically solved. It is worth remarking that this approach can be extended to $3D$ and beyond.

The combinatorial properties of such triple sign systems in $2D$ have been studied. They are partly characterized, notably in D. Knuth's CC (*CounterClockwise*) axioms (see Fig. 6), or equivalently in uniform acyclic oriented matroids of rank 3: see [Knu92] for details. Though illuminating, this axiomatic system and the corresponding combinatorial structure are too weak to capture all properties of the "true" geometry: for instance, Pappus's theorem is not a consequence of Knuth's axioms, and one can find sign systems that, though verifying Knuth's axioms, reveal not to be geometrically realizable.

Axiom 1 (cyclic symmetry). $pqr \Rightarrow qrp$
Axiom 2 (antisymmetry). $pqr \Rightarrow \neg prq$
Axiom 3 (nondegeneracy). $pqr \vee prq$
Axiom 4 (interiority). $tqr \wedge ptr \wedge pqt \Rightarrow pqr$
Axiom 5 (transitivity). $tsp \wedge tsq \wedge tsr \wedge tpq \wedge tqr \Rightarrow tpr$

Figure 6: *The five Knuth's axioms, to explore the combinatorial properties of the generic sign systems of triple of points. $pqr$ means: $|pqr| = 1$ and $\neg pqr$ means $|pqr| = -1$.*

Up to now a complete combinatorial characterization is not known, and perhaps there can be no finite set of purely combinatorial axioms (in Knuth's style, *i.e.* not using continuity, or coordinalization) that characterizes geometrically realizable systems: see [Knu92] pp 96 for more. Another very bad news is that the decision problem (is this given set of triple-signs consistent?) for Knuth's sign systems is NP-complete.

Actually, it seems that the problem is even more complicated. D. Knuth only considers generic situations, in which triples have sign $+1$ or $-1$, but never 0. Now some degenerate configurations are realizable in

some fields like the real algebraic closure of $\mathbb{Q}$, but not in others like $\mathbb{Q}$. Such a configuration is given in B. Grünbaum's book [Grü67] and in C. Hoffmann's one [Hof89], and illustrated in Fig. 7. Place nine points $A$, $B$, $C$, $D$, $E$, $F$, $G$, $H$, $I$ so that the following subsets of points (and only these subsets) are collinear: $ABEF$, $ADG$, $AHI$, $BCH$, $BGI$, $CFI$, $DEI$, $DFH$, as well as every couple of points, obviously. All solutions are projectively equivalent to the one in the figure, in which a regular pentagon occurs. As a consequence, this configuration is not realizable in $\mathbb{Q}^2$; it is in $\mathbb{R}^2$, actually in $\mathbb{Q}[\sqrt{5}]^2$.

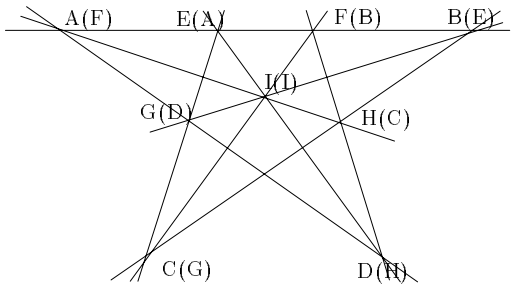


Figure 7: *This configuration above is not realizable in $\mathbb{Q}^2$; it is in $\mathbb{R}^2$.*

In conclusion, the approach of Computing with *fp* arithmetic while respecting geometric consistency is not so easy to achieve. Actually, I think this approach is untractable, as it goes for C.K. Yap and T. Dubé [DY95]. From 1989 on, C. Hoffmann has had the same intuition.

### 3.6 Fuzzy boundaries

The $\epsilon$ heuristic loses the order transitivity (it is possible to have $a =_\epsilon b$, $b =_\epsilon c$ and $a \neq_\epsilon c$), so inconsistencies remain possible. In such a case, a solution is to give up the distinction between $a$, $b$ and $c$, and to merge them into another larger entity, actually an interval. Computations are performed with an interval arithmetic or another equivalent method supplying error bounds; as soon as two entities overlap, they are merged in a third larger entity that contains both previous ones.

One can remark that two close but non overlapping entities have to be merged when gets introduced a third entity that overlaps both former ones. One can deplore this information loss (the distinction between the first two entities has been lost though they are not modified), and fear that existing geometric algorithms will not spontaneously withstand such a non monotonic logic. But this is the spirit of this approach.

The example in figure 5 will become something like in figure 8. This approach has been investigated by

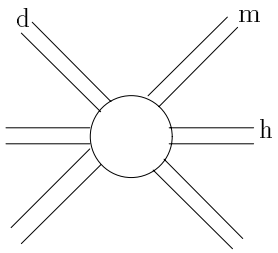


Figure 8: *Three lines with their halos, incident to a fuzzy point (the circle).*

M. Segal [Seg90] and by D. Jackson [Jac95] in solid modelling. In $3D$, geometric elements (vertices, edges or arcs, surfaces) are surrounded by a thin halo of imprecision; two distinct and not adjacent elements must not have overlapping halos. During say the computation of some boolean set operation (intersection or union or difference between two solid geometric objects), two elements the halos of which overlap must be cut or merged to restore the data structure consistency. In 1995, D. Jackson has implemented this way a robust algorithm to compute boolean set operations between $3D$ geometric objects with curved surfaces. Patrikalakis's team [HPY96] has also used this method.

The main advantages of this approach are that it applies not only to "linear" problems but also to algebraic ones, and that it does not rely on an exact arithmetic; so it is fast. Moreover, it is intuitive. Finally, it can handle inaccurate data in a natural way: either these data are obtained from some sensors and thus are known only up to some precision, or at the other end, the modelling stage has taken into account the fact that mechanical objects can be manufactured only within some tolerance. Up to now, this is the only approach that can represent fuzzy data.

A drawback is that all classical algorithms (for instance for computing boolean operations between solids) must be reformulated: it cannot be implemented only at the arithmetic level, since some classical methods cannot withstand the non monotonic logic of such approach, as pointed out before.

More generally, no one can assert this approach really solves the inaccuracy issue. For instance, when we want to know if the halos of two geometric entities overlap, their distance can be computed in several but algebraically equivalent ways; with a first formula, one may find that the elements do not overlap, but they will with another formula: maybe some contradictions remain possible.

This approach has not been proved yet. For this reason, this paper classifies it is an heuristic approach. However this choice is debatable, and people promoting this kind of method would probably classify it in the Approximate Computation Paradigm.

# 4 The "Exact Computation Paradigm"

## 4.1 Outlines

An obvious solution against inaccuracy is the use of an exact arithmetic. Actually, for existing CG methods and more generally for geometric methods dealing with Boundary Representation or relying on some other geometric consistency, there is more and more agreement that *the only way for consistency is exactness*: there is no definitive proof for this statement, but the other approach (respecting consistency while using approximate computations) comes up against NP-complete problems (see section 3.5).

Alas, even when an exact arithmetic on big integers or big rational is sufficient, the straightforward implementation is far too slow. From an experiment by M. Karasick, D. Lieber and L.R. Nackmann [KLN91], the Voronoï's triangulation of 10 random points in $2D$ takes 0.1 second; that of 10 random points with rational coordinates (2 digits for the numerator, 3 for the denominator, with radix $2^{16}$) takes 1200 seconds with a standard rational library and generated intermediate values up to 81 digits long. Of course computers now work faster, but the order of the ratio magnitude between both running times is still the same. It is easy to understand why exact arithmetics are so seldom used in geometric modellers.

Rational arithmetics (Section 4.2) are sufficient for a large class of classical methods from CG. In some cases, the machine numbers are even sufficient to achieve exact computations, with some tricks. Section 4.2.1 presents this fast but limited solution. When this solution does not apply, one can contemplate capitalizing on the fact that the *fp* arithmetic (or some interval arithmetic to have an upper bound of errors) is very often sufficient to decide the sign of an expression, and to use an exact arithmetic only when the *fp* arithmetic is not reliable. In practice, this idea is implemented in several ways. This paper only presents the $LN$ library due to S. Fortune and C. Van Wyk [FVW93] in section 4.2.2 and the lazy exact arithmetic due to M. O. Benouamer, P. Jaillon, J-M. Moreau and the author [BJMM93, MM] in

section 4.2.3. Due to lack of space, other approaches in the same tendency cannot be detailed out but are worth mentioning: [OTC87, KLN91, Yam87, GT91, NSTY93].

These improved rational arithmetics have proved to be usable (up to the rounding problem, see lower), but they are not sufficient for many geometric problems met in real life, problems involving for instance intersection between algebraic curves or surfaces. Rotations by $k\pi$ with $k \in \mathbb{Q}$ also introduce algebraic numbers. Idem for some square lengths: It is an old story.

Thus algebraic arithmetics (Section 4.3) are required. In CG or CADCAM, they are just a topic in its infancy. As far as I know, only quadratic arithmetics have been experimented in CG up to now (though *LINETOOL*, the geometric editor by L.W. Ericson and C.K. Yap [EY88], may perhaps be considered as a counterexample?). Section 4.3.1 presents the repeated squaring method (in fact, a natural extension which gives a true quadratic arithmetic). Section 4.3.2 presents the gap quadratic arithmetic proposed by C.K. Yap and T. Dubé [DY95].

In Symbolic Computing, Algebraic arithmetics are now classical and many methods from this field can be useful for the Exact Computation Paradigm: effective Elimination Theory with Gröbner's bases and resultants, and Real Algebraic Geometry with classic Sturm's sequences, Collins's Cylindric Algebraic Decomposition and its recent developments [BPR96]. Discussing all these methods is impossible here, but fortunately they have already been detailed in a number of papers or books. A special chapter in Hoffmann's book [Hof89] gives nonspecialists a nice introduction to Gröbner's bases. Resultants are presented in Wee and Goldman's survey [WG95], and they have already been used in robotics, and in CADCAM, notably by D. Manocha's team [KM96]. A comprehensive treatment of the mathematical background and algorithms of Elimination theory and Real Geometry may be found in B. Mishra's book [Mis93] or C.K. Yap's one [Yap96].

Section 4.3.3 presents the $D_5$ representation of algebraic numbers (not only quadratic ones). It is only used in the Algebraic and Symbolic Computation field for the moment, but seems to be a good candidate for geometric computations, in conjunction with Gröbner's bases, which are discussed in section 4.3.4. Another exact algebraic arithmetic has been recently used in CADCAM by D. Manocha and some of his students [KKM97]: see Section 4.3.5.

Exact Computation Paradigm is confronted with the following problems:

• What is the best exact arithmetic? The next step is obviously to experiment, improve and compare these exact arithmetics, and perhaps to imagine new ones.

• Merging exact arithmetics with non standard arithmetics would solve the robustness issue (inaccuracy and degeneracies) at the lowest level, the arithmetical one. It is briefly discussed in Section 4.4.

• Unfortunately, programs using Symbolic Computing tools (Gröbner's bases, resultants, $D_5$) in the straightforward way, *i.e.* as black box libraries, are extremely slow. Usable programs are very intricate. This sophistication makes programs difficult to implement, enrich and modify, as noticed by Manocha's team [KKM97] among others. It seems there is a need for packages efficiently handling arithmetic expressions (the DAGs in Sections 4.2.3 and 4.3.2), which arise as a basic data structure. It turns out that Symbolic Computing does not provide such tools, likely because it deals with complex but few arithmetic expressions. In opposition, Geometric Computing deals with a lot of these arithmetic expressions, generally simpler.

• Unfortunately, rounding operations (tackled in [Hof90, MN90, BMP94, For95]) are sooner or later unavoidable, first for communication with the outside *fp* world, and overall to break the exponential degree growth. In other words, exact computations can only be used *temporarily* to protect some methods from inaccuracy, and not in a lasting way. *Above all, accounting for rounding implies drastic modifications in methods and data structures, even with Exact Computation Paradigm.* See Section 4.5.

People promoting Approximate Computation Paradigm (Section 5) use these difficulties as arguments.

## 4.2 Rational Arithmetics

### 4.2.1 The poor man's exact arithmetic

In some restricted cases, it is possible to use an exact arithmetic which is as fast as the *fp* one. This section describes the various tricks I used in 1982–1983 to implement such an arithmetic, for a $2D$ graphic editor [GM84, Mic87, GHPT89]. Probably many people confronted with inaccuracy problems have used similar tricks at this time, but very few of them were published, if any.

The $2D$ graphic editor used Bentley and Ottman's method to compute the intersection points between the data segments. First the coordinates of the initial vertices were rounded on integers in the range 0 to $G = 30,000$: it was not a problem for the application. Thus equations of straight lines could also be stored in 3 `int` (machine integer): $(\alpha, \beta, \gamma)$ such that $|\alpha| \leq G$, $|\beta| \leq G$ and $|\gamma| \leq 2G^2$, a trivial consequence of formula 1 in section 2.1. The intersection points between segments $(x = \frac{\Delta x}{\Delta}, y = \frac{\Delta y}{\Delta})$ could be represented (assuming w.l.o.g. that $\Delta > 0$) by an `int` tuple $(x_e = \left\lfloor \frac{\Delta x}{\Delta} \right\rfloor, x_r = \Delta x \bmod \Delta, y_e = \left\lfloor \frac{\Delta y}{\Delta} \right\rfloor, y_r = \Delta y \bmod \Delta, \Delta)$: it is easy to see that $0 \leq x_e, y_e \leq G$, and that $0 \leq x_r, y_r < \Delta \leq 2G^2$. Some temporarily required values, such as $\Delta x$ or $\Delta y$, could exceed the maximum `int` value, but these computations were exactly performed using `double` numbers: their mantissa is long enough to store the met integers. A final trick was used to compare and sort these coordinates: the comparison of the two rational numbers $\frac{a}{b}$ and $\frac{c}{d}$, with $0 \leq a < b$ and $0 \leq c < d$, cannot reduce to the comparison of $ac$ and $bd$, since in some cases, these values were too large to be exactly represented by `int`, or even by the mantissa of `double` numbers. Using a simultaneous continuous fraction expansion of $\frac{a}{b}$ and $\frac{c}{d}$, it is possible to say that ([Mic87] pp 38):

$$\text{order}(\frac{a}{b}, \frac{c}{d}) = \text{order}(\frac{d}{c}, \frac{b}{a}) = \text{order}(\left\lfloor \frac{d}{c} \right\rfloor + \frac{d \bmod c}{c}, \left\lfloor \frac{b}{a} \right\rfloor + \frac{b \bmod a}{a})$$

If $\left\lfloor \frac{d}{c} \right\rfloor \neq \left\lfloor \frac{b}{a} \right\rfloor$, then:

$$\text{order}(\frac{a}{b}, \frac{c}{d}) = \text{order}(\left\lfloor \frac{d}{c} \right\rfloor, \left\lfloor \frac{b}{a} \right\rfloor)$$

otherwise:

$$\text{order}(\frac{a}{b}, \frac{c}{d}) = \text{order}(\frac{d \bmod c}{c}, \frac{b \bmod a}{a})$$

Since $c < d$ and $a < b$, the recursion eventually terminates. For instance,

$$\begin{aligned} \text{order}(\frac{2}{7}, \frac{3}{10}) &= \text{order}(\frac{10}{3}, \frac{7}{2}) = \text{order}(3 + \frac{1}{3}, 3 + \frac{1}{2}) = \text{order}(\frac{1}{3}, \frac{1}{2}) \\ &= \text{order}(\frac{2}{1}, \frac{3}{1}) = \text{order}(2 + \frac{0}{1}, 3 + \frac{0}{1}) = \text{order}(2, 3) = \text{smaller} \end{aligned}$$

Despite its interests, the limitations of such tricks are obvious. It cannot work on $3D$ or beyond because the computation depth increases: involved numbers become too big to be exactly representable by machine numbers. For the same reason, algorithms cannot be reentrant.

Remark: obviously, the trick for the comparison can be used to compute the sign of the determinant $\begin{vmatrix} a & c \\ b & d \end{vmatrix}$. This idea has since be used and extended to 3 by 3 determinants with integer entries by F. Avnaim, J-D. Boissonnat, O. Devillers, F.P. Preparata and M. Yvinec [ABD+95]. K.L. Clarkson [Cla92] has also used ideas from lattice reduction for computing the sign of a $n$ by $n$ determinant. The possibility to compute the sign of a determinant significantly more quickly than its exact value has remained unclear so far. Note a lot of CG methods use only tests which may be formulated as a determinant sign in a straightforward way (since all polynomial expression can be reformulated as a determinant, anyway!).

### 4.2.2 The LN library

S. Fortune and C. van Wyk proceed in two steps: First the program is pre-compiled and the minimum number of digits needed for the exact arithmetic (the longest integer generated by the algorithm, knowing the data range and the arithmetic expressions in the program) is determined. For each test in the program, they automatically generate *C++* code:

1. to compute the test in standard *fp* arithmetic, using references to original data only;
2. to test if *fp* value is greater than the maximum possible error for the expression;
3. finally, to call the exact, long integer library to evaluate the expression.

Second, the program is then compiled and linked with the exact library. Note that every test must be made with reference to original data. This permits a static (*i.e.* before running time) computation of the maximum possible error for each expression when evaluated in *fp* arithmetic; so the error bound has not to be computed at run time with intervals or whatever method. It speeds up execution, in a remarkable fashion, but it is not always very convenient for the user [CM93]; it forbids on-line and reentrant algorithms, in which computation depth is not *a priori* known, and it causes a proliferation of types: for instance input points and intersection points cannot be of the same type; this proliferation is a programmer's burden, and sometimes a compiler's one.

### 4.2.3 The lazy arithmetic

The lazy arithmetic computes with lazy rational numbers. A lazy rational number is first represented by an interval of two *fp* numbers, guaranteed to bracket the rational number, be it known (exactly evaluated) or not; and then by a symbolic definition, to permit recovering the exact value of the underlying rational number, if need be. The definition is either a standard representation of a rational number (for example 2 arrays or lists of digits in some basis, for numerator and denominator), or the sum or the product of two other lazy numbers, or the reciprocal or opposite of another lazy number. Thus each lazy number is the root of a tree, whose nodes are binary (sum or product) or unary (opposite or reciprocal) operators, and whose leaves are usual rational numbers; actually, lazy numbers form a directed acyclic graph rather than a tree, since any node or leaf may be shared. Each operation is generally performed in constant time and space : a new cell is allocated for the number, its interval is computed from the intervals of the operand(s), and the definition field is filled (operation type, and pointers to the operand(s)). Intervals are more often than not sufficient during computations; the only cases in which they become insufficient and thus the definition has to be "evaluated" (*i.e.* with rational arithmetic) are : when one wants to compare two lazy numbers the intervals of which overlap, when one wants a lazy number sign or reciprocal the interval of which contains 0. A possible evaluation method is the natural and recursive one. Using such a lazy library is transparent: classical geometric methods need not to be modified.

The lazy library also provides hashing of lazy numbers. Hashing techniques typically permit to recover topologic data from numerical ones, for instance vertices from coordinates. Obviously this technique needs to compute hash codes from numbers. Here we face a difficulty since the exact value of lazy numbers is unknown, and approximations are not relevant to reliably compute hash keys. The solution stems from modular arithmetic [MM].

Contrarily to LN, the lazy library is fully dynamic and so equally applies to on-line and reentrant algorithms : the computation depth needs not to be known *a priori*. In compensation, LN when usable should be faster than the lazy library.

My colleague J.M. Moreau would not forgive me if I did not mention that his constraint Delaunay triangulation software (used in Geographic Information System), using a reluctant arithmetic (a variant of the lazy arithmetic, but in which laziness is managed by the programmer, not by a library) is less than 2 times slower than the pure *fp* version, when the latter works of course.

## 4.3 Algebraic Arithmetics

### 4.3.1 Quadratic Arithmetic, with towers of extensions

In Fortune's method [For87], one has to compare numbers of the form $\frac{a+\sqrt{b}}{c}$, where $a$, $b$, $c$ are integers. It is possible to use repeated squarings, for this restricted case. This section presents a more general *quadratic arithmetic*, which provides exact comparisons and operations: $+$, $-$, $\div$, $\times$ and $\sqrt{}$ on non negative numbers, starting from $\mathbb{Q}$. As for an example, such an arithmetic[1] can be used to compute the $2D$ arrangement of a set of circles and lines, or the $3D$ arrangement of a set of spheres and planes.

The idea is to compute in a tower of $\mathbb{R}$eal quadratic extensions $K_0 = \mathbb{Q}, \ldots K_i = K_{i-1}(\sqrt{\alpha_{i-1}})$ where $K_i$ is an algebraic (and quadratic) extension over $K_{i-1}$, and $\alpha_{i-1} \in K_{i-1}$ is $\mathbb{R}$eal and positive and has no square roots in $K_{i-1}$. It means $K_i = K_{i-1}(\sqrt{\alpha_{i-1}})$ is the set of the numbers $u + v\sqrt{\alpha_{i-1}}$, with $u$ and $v$ two elements of $K_{i-1}$: in other words, these numbers are represented by a vector of two components: $u$, $v \in K_{i-1}$, and $K_i$ is represented by $\alpha_{i-1} \in K_{i-1}$ (which we already know how to represent, by induction) and by some reference to $K_{i-1}$. Operations in $K_i$ straightforwardly reduce to operations in $K_{i-1}$:

$$
\begin{aligned}
&(u + v\sqrt{\alpha_{i-1}}) + (u' + v'\sqrt{\alpha_{i-1}}) = (u + u') + (v + v')\sqrt{\alpha_{i-1}} \\
&(u + v\sqrt{\alpha_{i-1}}) \times (u' + v' \times \sqrt{\alpha_{i-1}}) = (u \times u' + v \times v' \times \alpha_{i-1}) + (u \times v' + u' \times v)\sqrt{\alpha_{i-1}} \\
&-(u + v\sqrt{\alpha_{i-1}}) = (-u) + (-v)\sqrt{\alpha_{i-1}} \\
&1/(u + v\sqrt{\alpha_{i-1}}) = (u/[u^2 - \alpha_{i-1} \times v^2]) - (v/[u^2 - \alpha_{i-1} \times v^2])\sqrt{\alpha_{i-1}}
\end{aligned}
$$

Computing the sign of $w = u + v\sqrt{\alpha_{i-1}} \in K_i$ also boils down to computations in $K_{i-1}$:

$$
\begin{aligned}
&u = 0 \text{ or } v = 0 : \text{ trivial} \\
&u > 0 \text{ and } v \geq 0 \Rightarrow w > 0 \\
&u > 0 \text{ and } v < 0 \Rightarrow \text{sign}(w) = \text{sign}(u^2 - v^2\alpha_{i-1}) \\
&u < 0 \Rightarrow \text{sign}(w) = -\text{sign}(-w)
\end{aligned}
$$

and in the end $K_0 = \mathbb{Q}$, where we know how to compute a sign, so the recursion eventually stops.

The last required operation is the square root in $K_i$. Assume $w = u + v\sqrt{\alpha_{i-1}} \in K_i$ is positive. The first thing is to test if $w$ is a square in $K_i$, say the square of $z \in K_i$ with $z = x + y\sqrt{\alpha_{i-1}} > 0$ with $x, y \in K_{i-1}$. We suppose $u$ and $v$ do not vanish, because this case trivially reduces to the same problem in $K_{i-1}$.

$$
\begin{aligned}
&w = u + v\sqrt{\alpha_{i-1}} = (x + y\sqrt{\alpha_{i-1}})^2 \\
&\Leftrightarrow u = x^2 + \alpha_{i-1} \times v^2 \text{ and } v = 2 \times x \times y \\
&\Leftrightarrow x^2 = \frac{1}{2}\left[u \pm \sqrt{u^2 - \alpha_{i-1} \times v^2}\right] \text{ and } v = 2 \times x \times y
\end{aligned}
$$

Thus $w \in K_i$ is a square in $K_i$ iff $u^2 - \alpha_{i-1} \times v^2$ is a square in $K_{i-1}$ and if $\frac{1}{2}\left[u + \sqrt{u^2 - \alpha_{i-1} \times v^2}\right]$ or $\frac{1}{2}\left[u - \sqrt{u^2 - \alpha_{i-1} \times v^2}\right]$ is a square in $K_{i-1}$ (Note that they cannot be both squares in $K_{i-1}$ because their product: $\frac{\alpha_{i-1}v^2}{4}$ is not a square in $K_{i-1}$).

Thus testing if $w \in K_i$ is a square in $K_i$ reduces to computations in $K_{i-1}$: in the end, testing if $w \in K_0 = \mathbb{Q}$ is a square in $\mathbb{Q}$ is trivial. If $w$ is a square in $K_i$, the method also gives its positive square root $x + y\sqrt{\alpha_{i-1}}$. When $w \in K_i$ is not a square in $K_i$, we have to define the quadratic extension of $K_i$ which contains the square root of $w$: call this extension $K_{i+1} = K_i(\sqrt{w})$. In particular, the coordinates of $\sqrt{w}$ in $K_{i+1}$ are: $(0 \in K_i, 1 \in K_i)$.

[1] I have implemented this arithmetic in Lisp and for fun, but not tested it inside a geometric algorithm.

For conciseness, this section has only presented the exact part. However several optimizations are possible: postponing exact computations in the lazy way by using intervals bracketing numbers, managing several towers (to reduce their depth and thus the complexity of computation) of extensions and waiting lazily for a collision before merging them, etc. Of course, in the worst case, the complexity is still exponential: a number in $K_n$ is represented by $2^n$ rational numbers.

### 4.3.2 A gap quadratic arithmetic

Gap arithmetics have been proposed by J.W. Hong [Hon86] and by J. Canny [Can88] (see section 3.2), in the Symbolic and Algebraic Computation field. Then T. Dubé and C.K. Yap [DY95] have used this scheme for a $\mathbb{R}$eal *quadratic arithmetic*, which supplies the usual rational operations ($+$, $-$, $\times$, :) and comparisons, plus the square root of non negative numbers, starting from $\mathbb{Q}$.

The main idea of gap arithmetic is to maintain an upper bound of the size of an *exact* and *virtual* representation for each number, *virtual* meaning that this exact representation is not computed. From this bound for any number $z$, it must be possible to effectively deduce a gap $\epsilon_z$ such that: $|z| < \epsilon_z \Rightarrow z = 0$. Thus an accurate enough approximation of the number $z$, *i.e.* a braketing interval $[a, b]$ with $-\epsilon_z < a \leq 0 \leq b < \epsilon_z$, proves $z$ is zero. Otherwise, when the interval does not contain 0, the sign of $z$ is trivially known.

As an example, a gap arithmetic is possible in the rational case, *i.e.* when only rational numbers and operations are used. The idea is to maintain for each met rational number $x$ upper bounds for the digit number in of its denominator: $d(x)$, and numerator: $n(x)$, for a given base, possibly $B = 2$. These upper bounds are known for the input numbers, and they are computed as follow for $x + y$, $xy$, $1/x$ and $-x$, without explicitly computing the exact rational form:

$$
\begin{array}{l}
n(x+y) = 1 + max(n(x) + d(y),\ n(y) + d(x)),\ d(x+y) = d(x) + d(y) \\
n(x \times y) = n(x) + n(y),\ d(x \times y) = d(x) + d(y) \\
n(1/x) = d(x),\ d(1/x) = n(x) \\
n(-x) = n(x),\ d(-x) = d(x)
\end{array}
$$

Now, to know if a number $x$ vanishes, just compute a good enough approximation $x^*$ of $x$, by using some (possibly on-line) bigfloat library: the smallest (in absolute value) non-zero rational number, having denominator with at most $d(x)$ digits in base $B$, is $\pm\epsilon_x$ with $\epsilon_x = \frac{1}{B^{d(x)}-1}$. Thus $x$ vanishes iff $x \in ]-\epsilon_x, \epsilon_x[$, or more conveniently when $x \in [-B^{-d(x)}, B^{-d(x)}]$. Optimization: when a number appears to be 0, its fields $n$ and $d$, and the ones of its dependent numbers can be strengthened on-the-fly.

Dubé and Yap's gap quadratic arithmetic uses Cauchy's bound, which says that:

$$|\alpha| > \frac{1}{1+h}$$

in which $h$ is the height, or a height bound, of the (non vanishing) algebraic number $\alpha$. This theorem gives the required gap. The height of $\alpha$ is the biggest absolute value in the coefficients of $\alpha$ minimal polynomial: a characteristic polynomial of $\alpha$ has $\alpha$ as a root; the monic characteristic polynomial with lowest degree is the minimal polynomial: it gives $\alpha$ height. Characteristic but non minimal polynomials give upper bounds of the height. Characteristic polynomials are the exact and virtual representation used. Note Loos [Loo83] also uses in a concrete way this representation: he effectively computes these characteristic polynomials.

Thus Dubé and Yap's arithmetic has to maintain a bound for each number height; it turns out that bounding the degree of the characteristic polynomials is required as well. Polynomials characterizing $\alpha \pm \beta$, $\alpha \times \beta$ are computable (they are not computed: it is the *virtual* representation) from the polynomials of $\alpha$ and of $\beta$, using Sylvester's resultant: the resultant degree is bounded by the product of the degrees in both polynomials, and the magnitude of its coefficients are bounded using a generalized Hadamard's

bound: of course, the heights exponentially increase. See [DY95] for details. Bounding degree and height is easy for unary operations: $\frac{1}{\alpha}$ (height and degree are unchanged) and $\sqrt{\alpha}$ (height is unchanged, degree is multiplied by 2), and for the rational numbers of the basic case.

As usual, numbers are represented by a definition tree, like in the lazy arithmetic, except that a new kind of node is used for the square root. Each number comes associated with a bound of its height and degree, and a bracketing interval. The leaves carry rational numbers. This DAG arises as a basic data structure in Exact Computation Paradigm, it is also more and more used in Symbolic Computation, where it is called a "Straight Line Program".

To compute the sign of a number $\alpha$ with height $h_\alpha$, when the available interval is insufficient, an approximation accurate to $\epsilon_\alpha = \frac{1}{1+h_\alpha}$ has to be computed. Several ways are possible. Yap and Dubé have chosen to propagate the required accuracy from the root down to the leaves. Then the computations are performed upwards using a bigfloat library. Another way would be to use an on-line arithmetic, which represent numbers by a (potentially infinite) stream of digits and in a redundant way (*i.e.* digits can take negative value) [Vui90, BCRO86, Wie80, MM94, Sch89]. This has the advantage that computations can be stopped as soon as the sign is known. However, on-line arithmetic performances have been disappointing so far (conclusion in [MM94]), this field still being in its infancy.

Dubé and Yap have experimented their gap arithmetic on Fortune's algorithm [For87] with rather encouraging results, better than the repeated squaring method. However the depth of computation is very low in this case, which hide the effects of the bound overestimation in the gap arithmetic. For instance, the latter is constrained to suppose the degree of a product, a sum or a difference is always the product of the degrees of the operands, which is the worst case. In other words, a gap arithmetic cannot detect simplification (except when a number turns out to be zero). The previous representation (see section 4.3.1), or the $D_5$ representation (see below) have not this drawback.

### 4.3.3 The $D_5$ representation of algebraic numbers

The name $D_5$ [Duv89] stems from the names of its authors: J. Della Dora, C. Dicrescenzo and Dominique Duval. $D_5$ represents algebraic numbers in the unordered case, *i.e.* it does not permit to sort numbers: it can decide equality, but not the $<$, $>$, $\leq$ and $\geq$ comparisons. In other words, it does not distinct between conjugate algebraic numbers, like $\sqrt{2}$ and $-\sqrt{2}$. Using isolating intervals and interval arithmetic are the most obvious and easy way to overcome this limitation.

Let $K$ be our "groundfield", *i.e.* a computable subfield of $\mathbb{C}$: computable means that we know how to perform the operations: $+$, $-$, $\times$, $\div$, and $=$. Initially, and typically, $K = \mathbb{Q}$. Let $\alpha$ be a complex number characterized as a root of $\phi(x) = 0$, in which $\phi \in K[X]$, *i.e.* $\phi$ is a polynomial with coefficients in $K$. $\phi$ does not have to be the minimal polynomial of $\alpha$; actually $\alpha$ can even belong to $K$. It turns out to be the main feature of $D_5$. However it is a convenient and cheap way to ensure $\phi$ is square-free (a polynomial is square-free iff it has no multiple root, *i.e.* $\gcd(\phi, \phi')$ has degree 0).

Then $K(\alpha)$ is also computable: $K(\alpha)$ is the set of the numbers $a = A(\alpha)$, where $A \in K[\alpha]$ are polynomials in $\alpha$ with coefficients in $K$. Their degree can be made strictly smaller than the degree of $\phi$: if by the euclidean division $A(x) = Q(x)\phi(x) + R(x)$ —here degree$(R)<$degree$(\phi)$— then $a = A(\alpha) = Q(\alpha)\phi(\alpha) + R(\alpha) = R(\alpha)$. From now on, assume the polynomials of $K(\alpha)$ are in this reduced form. Assuming $a = A(\alpha)$, $b = B(\alpha)$, the sum, product and inverse in $K(\alpha)$ are computed as follows ($P$ modulo $\phi$ is the remainder of $P$ in the euclidean division of $P$ by $\phi$):

$$
\begin{aligned}
a + b &= A(\alpha) + B(\alpha) = (A + B)(\alpha) \\
a \times b &= A(\alpha) \times B(\alpha) = (AB \text{ modulo } \phi)(\alpha) \\
\frac{1}{a = A(\alpha)} &= \frac{U(\alpha)A(\alpha) + V(\alpha)\phi(\alpha)}{A(\alpha)} = \frac{U(\alpha)A(\alpha)}{A(\alpha)} = U(\alpha) \text{ when } a \neq 0
\end{aligned}
$$

in which $U$ and $V$ in the last rule follow from Bezout's equality: since $a \neq 0$, $A$ and $\phi$ are coprime, as proved just below (*). Then after Bezout's theorem, there exist $U$ and $V$ such that $U(x)A(x) + V(x)\phi(x) = 1$. The subresultant PRS algorithm [Knu81] is a good way to compute $U$ and $V$.

The last thing to prove $K(\alpha)$ is computable is the computability of the equality test between two numbers $b = B(\alpha)$ and $c = C(\alpha)$, which boils down to the nullity test of their difference $a = b - c = A(\alpha)$. If $\phi$ was irreducible, $A(\alpha) = 0$ would be equivalent for $A$ to be identically null, but it is not always the case, and $a$ can be zero without $A$ being identically null.

But $a = A(\alpha) = 0$ and $\phi(\alpha) = 0$ imply that $\alpha$ is a common root of $A$ and $\phi$, *i.e.* a root of the greatest common divisor of $A$ and $\phi$: namely $G = gcd(A, \phi)$, which is computed only with operations in $K$ (and the groundfield $K$ is computable). Let $\phi = G \times \phi_2$. If the degree of $G$ is zero, then $A(\alpha) \neq 0$. Otherwise, since $\phi(\alpha) = G(\alpha)\phi_2(\alpha) = 0$, either $\alpha$ is a root of $G$, so as $a = A(\alpha)$ vanishes, or $\alpha$ is a root of $\phi_2$ and $a$ does not vanish. There is no way for $D_5$ to choose between these two possibilities, thus it makes a "splitting" and continues the forthcoming computations in both branches of the splitting: in the first branch, $a = 0$ and $\alpha$ is now characterized by the polynomial $G$; in the second, $a \neq 0$, $\alpha$ is now characterized by $\phi_2$, and $a$ is represented by $a = (A \text{ modulo } \phi_2)(\alpha)$. Note that $\phi_2$ (the new $\phi$) and $A$ are coprime, thus $\phi_2$ and $A$ modulo $\phi_2$ (the new $A$ representing $a$: $a = (A \text{ modulo } \phi_2)(\alpha)$) are also coprime, as previously promised in (*).

Here, we will differ from $D_5$ not using a splitting: we can distinct between the two cases by using separating intervals: $a = 0$ iff degree($G$)$> 0$ and the interval that isolates $a$ contains 0. Intervals that isolate roots are classically computed with well-known Sturm's sequences, requiring only operations in the groundfield $K$.

We have proved that $K(\alpha)$ is computable. A last thing we have to manage is the occurrence of new numbers, say $\beta$, defined, this time, by an equation: $\psi(\beta) = 0$ with coefficients in $K(\alpha)$. All we have to do is to compute in $(K(\alpha))\ (\beta)$, in the same way as before: in other words, $K(\alpha)$, which is computable, is our new groundfield. Thus if an algebraic system is available in a triangulated form: $f_1(\alpha_1) = 0$, $f_2(\alpha_1, \alpha_2) = 0$, $\ldots f_n(\alpha_1, \alpha_2, \ldots \alpha_n) = 0$, $D_5$ permits exact computations with its roots. We will see in Section 4.3.4 that Gröbner's bases can provide such a triangulated form. GCD computations and resultants are other possible methods.

The main superiority of $D_5$ over other representations (for instance, Loos' representation [Loo83]), and its elegance, is that $K(\alpha)$ need not be irreducible, which avoids very costly factorizations. Instead, $D_5$ quietly and lazily waits for some simple gcd computation to detect the simplification, if need be.

Remark: $D_5$ is not only a representation for algebraic numbers but a way to "compute with parameters" submitted to algebraic constraints [DD93, GD94], for instance to compute with $\alpha$ and $\beta$ such that $\alpha^2 + \beta^2 - 1 = 0$: here there is not a finite set of possible values for $\alpha$ and $\beta$. Finally, it is possible to extend $D_5$ for the $\mathbb{R}$eal ordered case, *i.e.* to provide comparisons: $<$, $>$, $\leq$, $\geq$. However this extension (using for instance Thom's lemma and the like) will be too complex and slow for our restricted need of representing algebraic numbers: isolating intervals seem much simpler and faster.

Up to now, $D_5$ has not been used for CG. Using $D_5$ just as a representation for algebraic numbers permits several optimizations: using bracketing intervals to postpone exact $D_5$ computations like in lazy rational arithmetic, handling several towers of computable fields to decrease the computation depth (since in CG, we generally face a lot of small computation trees, rather than a few high trees), and the like.

### 4.3.4 Gröbner's bases

In a nutshell, Gröbner's bases permit to exactly solve systems of algebraic equations. When there is a finite number of solutions, this method can express the system $g_1(x_1, \ldots x_n) = g_2(x_1, \ldots x_n) = \ldots = g_n(x_1, \ldots x_n) = 0$ in a triangulated form: $f_1(x_1) = f_2(x_1, x_2) = \ldots f_n(x_1, x_2, \ldots x_n) = 0$ in time roughly $d^{O(n)}$ (this counts the number of operations in the coefficients field, typically $\mathbb{Q}$) for a system of $n$ unknowns and $n$ equations with degree not greater than $d$. From the triangulated form, it is then

possible to compute exactly with the roots, using $D_5$ (there are other solutions not relying on $D_5$, but refer to the specialized literature), and to locate real roots in isolating intervals with Sturm's sequences. It is also possible to detect if one of the points $(x_1, \ldots x_n)$ fulfills or not another given equation.

Applications in geometry are obvious: for instance it is possible to compute exactly the intersection points between three real algebraic surfaces with known implicit equation, and to detect if some of these points also lie on a fourth surface. Thus it is possible to compute exactly the $3D$ arrangement of a set of algebraic implicit surfaces.

The complexity for computing a Gröbner's basis: roughly $d^{O(n)}$ is high but it is in some way almost optimal, since a dense algebraic system (in $n$ unknowns and $n$ equations with degree not greater than $d$) has $O(d^n)$ coefficients, and since the output may also have this complexity. Anyway, the complexity gets worse when there is an infinite number of solutions – roughly $d^{O(n^2)}$.

In practice, Gröbner's bases are practicable for small systems with low degrees. To give an idea, computing Gröbner's basis (with total degree ordering) of three polynomials in three variables with degree $d \leq 3$ and coefficient magnitude smaller than 1000 takes a fraction of second with Mupad, on a standard workstation. Gröbner's bases have already been used outside the Symbolic Computing field, in CADCAM world, especially to solve little systems of geometric constraints [Kon92]. It seems that until now they have been used only as a library, a black box: this approach has the advantage of modularity and simplicity, but its drawback: very poor performance, like naïvely replacing a *fp* arithmetic by an exact rational one.

The exponential complexity of computing Gröbner's bases remains, which strongly and definitively restricts the size of tractable problems. Thus the need for rounding (section 4.5).

### 4.3.5 Exact Algebraic Arithmetic with Resultants

Very recently in CADCAM field, D. Manocha and some of his students [KKM97] have used an exact algebraic arithmetic to reliably compute intersections between algebraic parameterized surfaces of low degree (2-4: *e.g.* quadrics and torii). It is probably the very first time that an exact algebraic arithmetic has been used for this problem in CADCAM domain. They use Dixon's resultants instead of Gröbner's bases for Elimination, with Milne's multivariate Sturm's sequences in order to locate roots. A number of improvements makes this approach practicable: for instance modular arithmetic speeds up the calculation of resultant coefficients, and intervals with rational endpoints that isolate roots are computed as lazily as possible. For the moment, it is not clear that extensions to implicit surfaces and to higher standard degrees are possible, due to the intrinsic exponential cost of the involved symbolic computations.

A solution is perhaps the sparse resultant (see [Emi94] for instance), which exploits the sparsity of algebraic systems. It is funny to see that computation of sparse resultant is a CG problem, involving Minkowski's sums about Newton's polytopes.

## 4.4 Merging exact and non standard arithmetics

Another issue is the merging of exact arithmetics with non standard arithmetics, which compute with infinitely small or big numbers: they permit symbolic perturbations on the data, and remove degeneracies. These non standard arithmetics are rare: people usually put forward efficiency arguments to use only a very restricted perturbation scheme, and not a general non standard arithmetic: however the latter permits to treat degeneracies on the lowest level, that is to say the arithmetical level. This is the most convenient for the programmer. Moreover, it works for *all* methods, generically. In other words it definitively solves the robustness problem. But for the moment, the only general non-standard arithmetic I am aware of [Mic95] uses only a usual (*i.e.* non lazy) rational arithmetic.

## 4.5 The rounding problem

The rounding problem has already been tackled [Hof90, MN90, BMP94, For95], but it seems people have not yet realized all its possible consequences, notably that it may jeopardize classical methods or data structures in CG, even in Exact Computation Paradigm.

### 4.5.1 Rounding is unavoidable

For CG methods to be used, we must provide interfaces between the "exact world" and the *fp* world, *i.e.* *fp* geometric objects (possibly inconsistent) have to be converted into exact geometric objects, and exact results have to be rounded on *fp* objects. It is the first reason why rounding operations are required. There is a second one:

Scenes in Computer Graphics, and mechanical parts in CAGD are finalized after many incremental modifications, for instance: copy and paste, rotations, deformations, blending, boolean operations. In an algebraic framework, these operations may theoretically be performed exactly. However such an approach is untractable, because intrinsically exponential. First the number of incremental modifications in the design process cannot be bounded *a priori*. Then, even trivial but essential operations like rotations by $\theta = k\pi$, $k \in \mathbb{Q}$ introduce algebraic numbers $\cos\theta$ and $\sin\theta$. Moreover the involved polynomial degree grows as an exponential function of computation depth (height of the DAG in sections 4.2.3 and 4.3.2). For instance the degree of the sum or the product of two algebraic numbers is already the product of the operand degrees, though additions and multiplications seem so trivial that we take them for granted. The size of the polynomial coefficients also breaks through.

Rounding numerical data (vertices coordinates and coefficients in surface equations) on integer, rational or *fp* numbers is *the only way* to avoid the degree growth.

Most of the time, geometric modellers only use the *fp* arithmetic, which performs the rounding. Some geometric modellers occasionally use exact computations and representations, typically to protect some routine and data structures from the ravages of inaccuracy (say a routine computing Boolean Operations over Boundary representations), but they are obliged to use rounding, sooner or later. It means geometric modellers do not, and cannot, use exact representations in a lasting way.

### 4.5.2 Rounding and consistency

Unfortunately, it is not a trivial task to round a geometric object while maintaining its topology, or at least the consistency of its representation. For simplicity this article considers only rounding polyhedra, say rounding algebraic polyhedra (*i.e.* with algebraic coordinates) to rational ones, or rational polyhedra to *fp* ones.

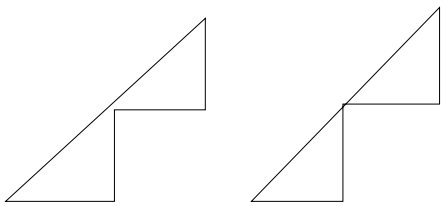

Figure 9: *Rounding vertices may introduce self intersections.*

Suppose we choose to round vertices (below we consider rounding face equations). Vertices may be easily rounded: just round each coordinate by using continued fraction expansions or some transformations like: $x' = \frac{\lfloor 2^n x \rceil}{2^n}$. But rounding independently each vertex destroys the geometric consistency. For instance

planar faces with more than 3 vertices are no longer planar after rounding. A solution is to handle only triangular faces: consistency of faces is saved, though rounding still loses coplanarities, cocircularities (of more than 3 vertices), and collinearities (of more than 2 vertices). Such losses are anyway unavoidable, since some algebraic configurations are not realizable in the rational field, as the one in Fig. 7. Of course, the equation of the rounded planes must be re-computed from the coordinates of the rounded vertices, and not by rounding plane equations, in order to preserve vertex-face incidences. The last difficulty is that the rounding may introduce self-intersections: see Fig. 9. Actually, V.J. Milenkovic and L.R. Nackmann [MN90] have proved that rounding polygons and polyhedra is NP-complete. Fortune's solution [For95] is to accept self-intersecting polyhedra. As a consequence, the localization test is not based on parity but on the winding number. *Thus accounting for robustness substantially modifies algorithms and data structures.*

Another approach rounds equations of the polyhedron faces, vertices being re-computed from the rounded face planes. But the dual from the previous problem arises: rounding independently the equations of $n > 3$ faces incident to a common vertex destroys the co-incidence, thus the topology is altered. Self intersections may also be introduced.

The last approach represents polyhedra by a boolean formula on half linear spaces, *i.e.* by a CSG tree with linear inequalities at leaves. This representation is implicit: vertices, incidence and neighborhood relations are not explicitly known, but they can be computed (with a given accuracy) if need be. Since this representation is not redundant, rounding is trivially achieved by rounding each inequality independently. Of course, rounding may still modify the topology of the polyhedron, *but it cannot introduce inconsistencies in the representation.* In fact, it is nothing else than Approximate Computation Paradigm.

#### 4.5.3 Exploiting discreteness

This section concludes with a more positive consequence of rounding. Since data are rounded on integers (or rationals), *CG basically deals with discrete problems, not with continuous ones.* CGers have exploited this discreteness on the arithmetical level, for instance when computing the sign of determinants (see Section 4.2.1), or when counting the required number of digits to exactly perform some computations. Paradoxically, apart from some exceptions like interval trees and related methods, CGers have not really exploited discreteness on the algorithmic level to compute say convex hulls, Voronoï's diagrams, Delaunay's triangulations, or intersections between polytopes.

This CG attitude with discreteness contrasts with Computer Graphics and CADCAM, where bucketing techniques or spatial subdivision methods (see Section 5.3 for an example) are widely used. CGers do not appreciate space subdivision methods, because they still assume a continuous model of space: with this model, the subdivision depth cannot be bounded *a priori*, nor can be time and space requirements for space subdivision methods. The CGers' attitude is a bit contradictory, since they exploit discreteness on the arithmetic level, and at the algorithmic one, they assume continuity of space. Exploiting discreteness on the algorithmic level is an open research area for CG, maybe a fertile one.

# 5 "Approximate Computation Paradigm"

## 5.1 Outlines: Boundaries are a disease of intelligence

CADCAM community faces and is aware of the robustness problem since its birth. Engineers have first believed that some tricks (the $\epsilon$ heuristic, and careful programming) would be sufficient to solve the problem. Now, a new tendency appears, since say the conferences CSG94 and CSG96 [csg94, csg96], that promotes a more radical approach. I call it "Approximate Computation Paradigm", in opposition to Yap & Dubé's "Exact Computation Paradigm". The word "Approximate" has unfortunately a pejorative connotation that Section 6 tries to correct.

Approximate Computation Paradigm states: algorithms or data structures that do not withstand inaccuracy are in some way paranoiac and must be rejected. Thus methods from CG must be rejected: they rely and thus depend on geometric consistencies and exact arithmetic. One must also get rid of topology-based data structures, like Boundary Representations (BRep for short). In a nutshell, BReps explicitly handle representations for vertices, edges and surface patches, and all the topologic incidence relations between them; they are very explicit but they rely on geometric consistencies and their redundancy exposes them to inconsistencies and failures (does this vertex numerically lie on this surface though it topologically does?). In particular, we know that robustness is exceedingly difficult to achieve when performing boolean set operations between geometric objects represented by BReps. Note that CG typically uses BReps: CG methods compute on boundaries.

This tendency promotes to base the geometric modellers on a "semantic" description of geometric objects, a reference definition: the aim at this definition is not to speed up this or that geometric algorithm. Typically this definition is a CSG representation (Section 5.2) or some variant of it.

Then this CSG representation is "evaluated" (to use the accepted word) when need be, with a given –but finite– accuracy, to produce a more explicit geometric representation. For instance interval analysis and a spatial recursive subdivision (Section 5.3 ) can enumerate the set of boxes or voxels (*volume element*) which are strictly inside the object, or strictly outside, or which are cut by (or close to) the boundary. Marching methods (Section 5.4) can approximately triangulate the object boundary. The ray casting method (Section 5.5) displays the object defined by CSG representation, or samples it with an array or parallel rays (Section 5.6) to obtain a "ray representation". This last method can also account for non pure CSG objects, like free form objects the boundary of which is made of sewn parameterized surface patches.

After an evaluation, the designer corrects the CSG definition, and re-evaluates it, and so on, until completion of the design process. Note CSG definition does not refer explicitly data structures produced during the evaluations: thus possible inconsistencies in these data structures cannot contaminate CSG definition. An additional guarantee.

## 5.2 CSG representation

CSG representations (*Constructive Solid Geometry*) describe objects in only an implicit way, by CSG trees. A leaf of a CSG tree carries a *primitive* object: the set of points $(x, y, z)$ verifying some (typically algebraic) inequality $f(x, y, z) < 0$, for instance a quadric, a torus or a more complicated object. A node is either the union, the intersection or the difference between other CSG trees.

Mathematically speaking, a CSG is a semi-algebraic set, up to some regularization problems (is it $f(x, y, z) < 0$ or $f(x, y, z) \leq 0$ ?) which are not relevant here. Note in passing that Symbolic Computing has proposed several methods to evaluate semi-algebraic sets, like Collins's Cylindric Algebraic Decomposition or more recent variants, but their cost is prohibitive (worse than exponential).

Thus the contour of the object represented by a CSG tree is not explicitly described, and it is not obvious that a CSG tree does not describe only the empty set, contrarily to BReps. But there exist very robust methods to display objects defined by CSG trees, to approximately triangulate them, or to discretize them: they are presented in the next sections.

Two remarks:

• A "syntaxically correct" CSG tree is a consistent one, in opposition to BReps, the consistency of which is difficult to prove and achieve.

• The analogy between CSG trees, DAGs, and Straight Line Programs (more and more used in Symbolic Computing) is noteworthy.

## 5.3 Interval Analysis and recursive subdivision of space

Interval Analysis [Kea96] can compute conservative bounds for a function range on an interval, for instance for $f(x \in [x_0, x_1], y \in [y_0, y_1], z \in [z_0, z_1])$ in which $f$ is a CSG leaf. When this bound does not contain 0, one knows whether the box $[x_0, x_1] \times [y_0, y_1] \times [z_0, z_1]$ is inside or outside the primitive object. Other more sophisticated tests [Sny92] from Interval Analysis detect if the box is cut by the boundary (the surface having equation: $f(x, y, z) = 0$), and if the latter is simple enough. This is so if, for instance, for any given point $x \in [x_0, x_1]$ and $y \in [y_0, y_1]$, there is at most one $z \in [z_0, z_1]$ so that $(x, y, z)$ belongs to the surface having equation $f(x, y, z) = 0$. It is also possible to detect if a box contains a single intersection curve between two surfaces, simple enough, or a regular intersection point between three surfaces. In a cell containing a single surface (respectively a single intersection curve between two surfaces), it is also possible to bracket it between two (respectively four) planes.

Otherwise, but if the box is too small according to an *a priori* threshold, the box is divided in 2 or 8 depending on the implementations, and the sub-boxes are studied the same way. Filiations between boxes may be stored in an octree. Such a method find boxes strictly inside CSG object, strictly outside, cut by a boundary in a simple way, or residual. Such residual boxes have smaller size than the prescribed threshold, and they usually contain or are very close to singularities or near-singularities. The robustness of this method may be obvious. SVLIS modeller [Bow95] uses such a method.

These methods can be used beyond $\mathbb{R}^3$: this "dimensionality paradigm" (the name is due to C.M. Hoffmann [Hof90]) has been exploited by J. Woodwark for Feature Recognition, by K.D. Wise and A. Bowyer for Spatial Planning [WB96], by C.M. Hoffmann for Surface Interrogations [Hof90].

To be a bit provocative, note this method can be used when the inequalities at the leaves of CSG tree are linear. Moreover, for simplicity, we can impose never are there 4 planes crossing in the same point, to avoid degeneracies. So we have a CG problem and we have a resolution method, with we can even remove the condition restricting the boxes smallness. No doubt CGers will be very reluctant in front of such a method. They will find it shocking that depth of the octree depends on the numeric complexity of the plane equations, for a given structure of CSG tree: they consider coordinates belong to $\mathbb{R}$, so it is possible to have arbitrarily short or long distances between vertices: the ratios cannot be bounded *a priori*, nor can be the depth and size of the octree, nor can be the memory space and time requirements of the algorithm. In fact, as already pointed out in Section 4.5.3, this assumed continuity of space is in contradiction with the exploitation of discreteness on the arithmetic level which is made by Exact Computation Paradigm.

## 5.4 Marching methods

To approximately triangulate objects defined by CSG trees within a given tolerance $\mu$ (see [PA94] in [csg94], [TGP96] in [csg96]), the space $\mathbb{R}^3$ is first partitioned with a regular cubic lattice, sided $\mu$. Each cube is then partitioned into tetrahedra; for all vertices $v = (x, y, z)$ of the lattice, the value of CSG tree at $v$ is computed: for a primitive described by an inequality $f(x, y, z) < 0$, it is $f(v)$; for nodes $A \cap B$ and $A \cup B$, it is respectively $max(A(v), B(v))$ and $min(A(v), B(v))$ where $A(v)$ and $B(v)$ recursively stand for the value of $A$ and $B$ CSG trees in point $v$. The object surface cut a given tetrahedron when the values in the 4 vertices have opposite signs. These 4 values define, by linear interpolation, a unique linear map $l(x, y, z)$ from $\mathbb{R}^3$ to $\mathbb{R}$, and the plane $l(x, y, z) = 0$ is considered as a good enough approximation of the object contour inside the tetrahedron: it gives a triangle or a quadrilateral. The same is done for all tetrahedra. This technique is illustrated in $2D$ in Fig. 10.

Marching methods are not sensitive to inaccuracy: in the worst cases, a vertex value is close to 0, and *fp* evaluations may yield a wrong sign for the value, but the only and immaterial consequence will be to move the approximation surface a little.

The true object topology and the one of its linear piecewise approximation may be different. Small components, with size less than the threshold, can be missed. In the vicinity of singularities and quasi-singularities of the true object boundary, the approximation remains manifold. CADCAM engineers

consider this filtering as an advantage, a simplification. Geometrically (in opposition to topologically), the object and its approximation are close, up to $\mu$.

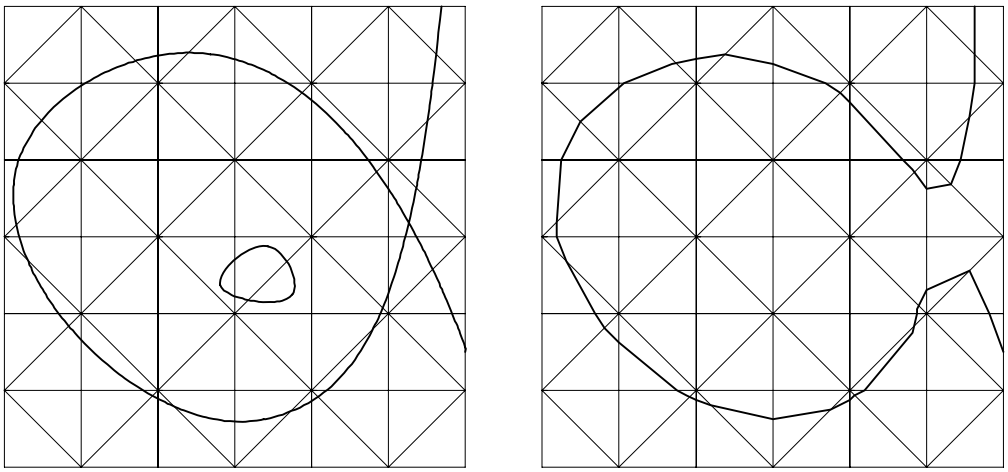

Figure 10: *A $2D$ curve and its piecewise linear approximation. The topology may be different, and some small components of the $\mathbb{R}$eal curve may be forgotten. But this technique is perfectly robust.*

Of course, it is better to use some optimizations not to consider all lattice cells, like some interval computations [dFS95, Tau93], or like using continuity: once a starting tetrahedron crossed by the surface is known, the sides by which the contour surface leaves the tetrahedron are easily computed and the contour surface is then followed in the neighboring tetrahedron. It is also possible to approximate better the intersection curve between two surfaces in a cell. All the variants and optimizations are beyond the scope of this article, the main thing being marching methods reliability is preserved.

Thus an approximate BRep (and all its precious informations) can be obtained from a CSG tree, without having to perform boolean set operations on BReps, which is a very unreliable process.

Remark: it is possible to go further and to question the need for a BRep: why not stopping at the discretization step?, as Section 5.7 argues.

## 5.5 Ray casting methods

Pictures are described in Computer Graphics with $2D$ arrays of points, the so-called "pixels", a shortcut for "picture elements". To compute such a picture of an object described by a *CSG* tree, ray casting methods compute which one of the objects is seen in each pixel. The eye location and the point to be computed define a half straight line: the ray, whose intersection with the scene has to be computed. When the object is a primitive $f(x, y, z) < 0$, in which $f$ is typically a polynomial in $x$, $y$, $z$, this problem boils down to the resolution of an algebraic equation in $t$: just replace $x$, $y$, $z$ in $f(x, y, z) = 0$ by $x = x_e + at$, $y = y_e + bt$, $z = z_e + ct$, $(x_e, y_e, z_e)$ being the eye location and $(a, b, c)$ the ray support vector. The numeric resolution of the resulting equation: $F(t) = 0$ yields the intersection, a set of intervals $[t_0, t_1]$, $[t_2, t_3] \ldots$ along the ray, with $0 \leq t_0 \leq t_1 \leq t_2 \leq t_3 \ldots$. When the object is a boolean combination, $A \cap B$ for instance, just recursively compute the ray intersection with subtrees $A$ and $B$, so to give two resulting sets of intervals $A_{[]}$ and $B_{[]}$, then calculate $A_{[]} \cap B_{[]}$: a trivial merge.

The difficult part is the numeric resolution of $F(t) = 0$, by interval analysis [Kea96] or whatever numerical methods. Obviously *fp* and interval arithmetics cannot reliably decide in some ambiguous intervals: for instance they cannot distinct between the 3 cases in figure 11. Idem for the 3 cases in figure 12. However,

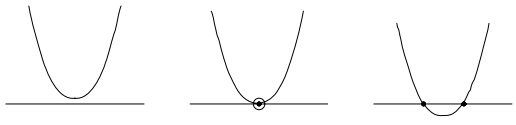

Figure 11: *fp or interval arithmetic cannot distinct these 3 cases, when the polynomial is tangent, or almost tangent, to the x axis. But a confusion does not matter, since in all these cases, the number of roots is even.*

the main thing is not to make a mistake on the parity of the number of roots in such ambiguous intervals, that is to say not to confuse a case in figure 11 (even parity) with one in figure 12 (odd parity). It is easily achieved.

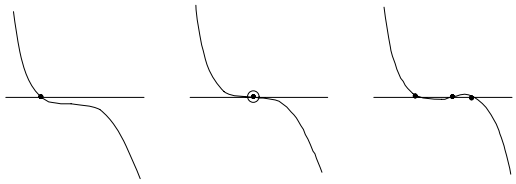

Figure 12: *fp or interval arithmetic cannot distinct these 3 cases, when the polynomial is tangent, or almost tangent, to the x axis. But a confusion does not matter, since in all these cases, the number of roots is odd.*

Assuming the parity is correct, mistakes have immaterial consequences on the final picture since they occur only when the ray is tangent or almost tangent to a surface. Thus the only effect is to move slightly and locally the object outline. Useless to indicate, a ray tracer never crashes due to these numerical errors, and mistakes are not propagated from pixels to pixels. This robustness against errors contrasts with the CG methods behaviour.

When the list of intervals is stored for each pixel, one gets the "ray representation": the object has been sampled by an array of parallel lines. This ray representation permits to measure approximately the mass, the inerty, the area, the volume, the gravity center and so on [Pri96].

## 5.6 Ray-representations

For a couple of years, ray representations (rayreps for short) are fashionable data structures in CADCAM [MMZ94, MV95, Pri96, BM97] due to their simplicity, versatility and robustness. A rayrep can be computed by any visualization method, for instance ray tracing or the well-known Z-buffer [FvDF$^+$94], or by merging two other rayreps with the same family of rays. It is possible to compute this way boolean operations between two rayreps (which have possibly been computed with different methods). Finally, to account for a new kind of geometric object, it suffices to implement the corresponding visualization routine.

An inconvenient of rayreps is its anisotropy: surfaces parallel or nearly parallel to the ray direction are less sampled than the ones perpendicular or nearly perpendicular to the ray direction. The obvious solution is to use a triple rayrep, *i.e.* three rayreps with three orthogonal ray directions, like $Ox$, $Oy$ and $Oz$. A triple rayrep induces a regular cubic lattice in which a marching method can then built an approximate triangulation of the boundary. Here again, an approximate BRep can be safely obtained

from a CSG tree, without unreliable computations of boolean operations over BReps. See [BM97] for more.

Moreover, triple rayreps make it possible to account for "sculptured solids" (see next paragraph), contrarily to marching and recursive space subdivision methods presented in Section 5.4 and 5.3.

The major part of commercial CADCAM softwares are based on free form surfaces: these surfaces are described by parameterized patches: $x = X(u,v)$, $y = Y(u,v)$, $z = Z(u,v)$ in which $u,v \in [0,1]^2$ and $X$, $Y$, $Z$ are polynomials or rational functions; not by the implicit form: $f(x,y,z) < 0$ in CSG representation. Note parameterized formulation describes only a surface, not a solid, thus the designer has to sew together these surfaces very carefully to obtain the boundary of a consistent solid which is called a "sculptured solid". Sculptured solids cannot be described by CSG trees (in the original version) and they are not compatible with the marching and recursive space subdivision methods, since the implicit form $f(x,y,z) < 0$ is not available.

Fortunately, triple rayreps solve the problem [BM97]. The idea is to accept at the leaves of CSG trees a new kind of primitives: sculptured solids. Then all you need to built a triple rayrep is a method to display such sculptured solids: ray casting is a possible one, though it is slow. A second is the well-known and fast Z-buffer method.

## 5.7 Discretization

Boundary representations are basically used to "evaluate" more or less accurately the boundary of a *CSG* object. However they are not the only possible way, just the usual one, due to the CADCAM history. Discretization is another solution: the space is represented by a $3D$ array of points, the so-called "voxels", a shortcut for "volume elements". This discrete representation makes trivial the most frequent geometric problems (estimating mass properties, interference detection, boolean operation, etc) and it virtually removes the inaccuracy problem.

Nowadays, Computer Tomography and Magnetic Resonance Imaging make it possible to acquire such image data in $3D$. At the other end, from such a voxel-based representation, Rapid Prototyping [SBE95] can produce real tactile plastic prototypes for manufacturers, chemists or biologists with "printing in $3D$", *i.e.* with stereolithography. The stereolithography apparatus builds the prototype slice by slice, laying down a thin layer (between 0.1 and 0.5 millimeters) of liquid resin on the previous slice, instantly curing it into solid plastic, and starting again. Moreover, at this level of precision, the voxel-based representation is also the most precise one: this is in contrast with the not that old reluctance of some theorists for this discrete representation, which they considered as a trivial and very rough approximation of "exact" CSG models. Last, the voxel-based representation is always the simplest one, obviously.

It is worth comparing the history of space representation with the one of pictures. In the beginning of Computer Graphics and CADCAM, more than twenty years ago, pictures were usually not represented by discrete representations, *i.e.* $2D$ arrays of pixels, but by BReps, because discrete representations were too cumbersome at this time, and available devices only provided wire frame display for which BReps are best suited. Related algorithms, for removing hidden parts for instance, already had trouble with inaccuracy. Nowaday, pictures are represented by discrete representations, and everybody has forgotten these algorithms and their inaccuracy problems. One can wonder if, similarly, the time has not come for discrete representations of space to supplant boundary representations of solids, and to remove the inaccuracy problem in geometric computations.

For the moment, some technical facts or habits delay this mutation: for instance, graphic workstations are mainly built for displaying polygons in real time, not array of voxels or CSG trees; Finite Element methods are often based on BReps for their geometrical part.

# 6 Some clarifications

## 6.1 Accuracy *vs* Exactness

The word "Approximate" sounds unfortunately pejorative and this clarification may be useful. If purely mathematical problems like automatic theorem proving require exactness, applications in CADCAM, Computer Graphics, Medicine, Physics only need accurate enough approximations. In such fields, data themselves are approximate, and at the other end the computed results are materialized only up to some tolerance: for instance in the CADCAM world, milling machines are not perfectly exact. Thus it makes no sense to compute results more precise than data, and *a fortifiori* to compute exact results. Here exactness is mostly illusory.

Symmetrically, with Exact Computation Paradigm, exactness does not mean data structures in the computer are an exact representation of the corresponding object in the real world. For instance, "exact" polytopes of CG are only approximations for geometric objects in the real world. In the same way, algebraic surfaces used in Computer Graphics are only approximations for natural objects or livings (what could be the algebraic surface describing a pear boundary?). Even algebraic surfaces used in CADCAM are only approximations (accurate and convenient, of course) for machined mechanical parts – it is the basis of the so-called tolerancing problem. In Exact Computation Paradigm, exactness only means that results are exactly computed from data, which are anyway approximate: it is just a way to achieve consistency. Moreover, there is the rounding problem.

As an aside, F. Chaitin-Chatelin [CC96] recently argues that, in Physics (the inaccuracy problem touches almost every scientific computing field), the exact solution of a single simulation may be less significant than samples of wrong *fp* ones. This is the "Qualitative Computing Paradigm". To quote F. Chatin-Chatelin: "*Because no equation is exact in the real world, computer simulations can be closer to the physical reality of unstable processes than exact computation.*"

## 6.2 About the classification

This paper has classified the existing approaches to achieve robustness into three groups: heuristic, exact, and approximate approaches. Roughly, the first class: heuristic approaches, groups together the empirical techniques engineers have found to avoid the more frequent inconsistencies. The second class: the Exact Computation Paradigm, groups together methods using (requiring) exact computations in order to work, *i.e.* roughly CG methods and methods handling Boundary Representations: this scheme is probably the only solution to save these methods from inconsistency. The third class: the Approximate Computation Paradigm, groups together the reliable methods which computer scientists, especially in the CADCAM field, have recently proposed to bypass the inaccuracy problem: these methods don't need exact computations. This classification is convenient but things are not so simple.

As already pointed out, classifying the "Fuzzy Boundary" approach as an heuristic method and not as an approximate one is questionable. Moreover, a lot of hybrid methods mixing exactness and approximations can be imagined. For instance to approximately evaluate the boundary of a CSG object, it is possible to first approximate the CSG primitives with polyhedra, then to compute the boolean operations on these polyhedra. Robustness is then achieved thanks to Exact Computation Paradigm: polyhedra are rounded into say rational polyhedra (with triangular faces and rational coordinates for vertices), then exact computations avoid inconsistencies. Another hybrid method is to use exact computation in a marching method (Section 5.4): the values at the lattice vertices are rounded on rational numbers, then the Exact Computation Paradigm is used to obtain a consistent and "exact" triangulation.

# 7 Conclusions

This paper has shown how crucial for geometric computations the inaccuracy issue is. Some examples have shown the specificity of geometric computations, the fact that below geometry lay deeper combina-

torial structures, the non-respect of which lead to topological inconsistencies and running time crashes. This paper has surveyed the most typical proposed approaches to overcome inaccuracy problems. Main conclusions are:

• It is sure that arithmetic issues (inaccuracies and degeneracies) are the current crucial challenge for CG field. As long as the robustness problem is not solved, CG will not apply to problems in the real world, and it will stay a theoretical field: its algorithms and data structures will not be used.

• Though there is no formal proof, there is more and more agreement that the classical algorithms from CG, and geometric methods working on Boundary Representations, cannot achieve a perfect robustness without exact computations. Exact computations do not mean all computations must be done exactly, it means all test decisions must be exact. This is the "Exact Computation Paradigm".

• For algebraic problems, the Exact Computation Paradigm is today an emerging field: it is very probably a fertile research area.

• For linear problems, *i.e.* when a rational arithmetic is sufficient, Exact Computation Paradigm has proved to be feasible. However:

• Exact (rational or algebraic) computations can only be used *temporarily*, typically to protect some routines against inaccuracy and inconsistency: rounding steps are sooner or later unavoidable, since they are the only way to stop the exponential growth in the algebraic complexity. Rounding geometric objects without modifying their topology is NP-complete, thus geometers are obliged to use rounding operations which may modify the topology and introduce self-intersections. Thus, even with the Exact Computation Paradigm, data structures and algorithms have to be modified to account for the need of rounding operations.

• With Exact Computation Paradigm, CG basically deals with discrete problems. Discreteness has been mainly exploited on the arithmetic level, but not really on the algorithmic one. Exploiting discreteness on the algorithmic level is an open research area for CG.

• Robustness is possible without exact computations. This is the "Approximate Computation Paradigm". It is more and more used in CADCAM and Computer Graphics. This approach rejects non reliable methods and data structures, not robust enough against inaccuracy, namely methods exploiting, relying and depending on geometric consistencies and thus requiring exact computations. This includes CG methods and Boundary Representations. It is a complete review since classical CADCAM modellers are based on Boundary Representations.

• The last approach dealing with robustness issue is Discrete Geometry. This paper has shown than CG problems are not so far from Discrete Geometry.

• Whatever the used paradigm, Exact or Approximate, the robustness issue has to be accounted for at the very first start when designing geometric methods, data structures and modellers: they are profoundly modified.

# References


[ABD+95] F. Avnaim, J-D. Boissonnat, O. Devillers, F.P. Preparata, and M. Yvinec. Evaluation of a new method to compute signs of determinants. In *Proceedings of the 11th Symposium on Compututational Geometry*, pages C16–C17. ACM Press, 1995.

[AR94] A. Agrawal and A.G. Requicha. A paradigm for the robust design of algorithms for geometric modeling. *Computer Graphics Forum (EUROGRAPHICS'94)*, 13(3):C–33–C–44, 1994.

[BCRO86] H.-J. Boehm, R. Cartwright, M. Riggle, and M.J. O'Donnell. Exact real arithmetic: a case study in higher order programming. In *Proc. ACM Conference on Lisp and Functional Programming*, pages 162–173, 1986.

[BJMM93] M.O. Benouamer, P. Jaillon, D. Michelucci, and J-M. Moreau. A lazy arithmetic library. In *Proceedings of the IEEE 11th Symposium on Computer Arithmetic*, pages 242–269, Windsor, Ontario, June 30-July 2, 1993.

[BJMM94] M.O Benouamer, P. Jaillon, D. Michelucci, and J.M. Moreau. Hashing lazy numbers. *Computing*, 53(3–4):205–217, 1994.

[BM97] M.O. Benouamer and D. Michelucci. Bridging the gap between csg and brep via a triple ray representation. In *Proceedings of the Symposium on Solid Modeling Foundations and CAD/CAM Applications*, May 1997.

[BMP94] M.O Benouamer, D. Michelucci, and B. Péroche. Error-free boundary evaluation based on a lazy rational arithmetic: a detailed implementation. *Computer-Aided Design*, 26(6):403–416, June 1994.

[Bow95] A. Bowyer. *SVLIS – Introduction and User Manual.* Information Geometers Ltd, 47 Stockers Avenue, Winchester, SO22 5LB, UK, second edition, 1995.

[BPR96] S. Basu, R. Pollack, and M-F. Roy. On the combinatorial and algebraic complexity of quantifier elimination. *Journal of the ACM*, 43(6):1002–1045, november 1996.

[Can88] J. Canny. *The complexity of robot motion planning.* M.I.T. Press, Cambridge, Mass., 1988.

[CC96] F. Chaitin-Chatelin. Is finite precision arithmetic useful for physics ? *Journal of Universal Computer Science*, 2(2):380–395, May 1996. *http://www.iicm.edu/jucs.*

[Cla92] K.L. Clarkson. Safe and effective determinant evaluation. *IEEE Foundations of Computer Science*, 33:387–395, 1992.

[CM93] J.D. Chang and V. Milenkovic. An experiment using ln for exact geometry computations. In *Proceedings of the 5th Canadian Conference on Computational Geometry*, pages 67–72, Waterloo, Canada, August 5-9, 1993.

[csg94] *CSG94: Set Theoretic Solid Modelling Techniques and Applications.* Information Geometers Ltd, 47 Stockers Avenue, Winchester, SO22 5LB, UK, 1994. Proceedings of the CSG 94 Conference, Winchester, UK, 13-15 april 1994.

[csg96] *CSG96: Set Theoretic Solid Modelling Techniques and Applications.* Information Geometers Ltd, 47 Stockers Avenue, Winchester, SO22 5LB, UK, 1996. Proceedings of the CSG 96 Conference, Winchester, UK, 17-19 april 1996.

[DD93] D. Duval and T. Gomez Diaz. A lazy method for triangularizing polynomial systems. In *SEA 93*, nov. 1993.

[dFS95] L.H. de Figueiredo and J. Stolfi. Adaptive enumeration of implicit surfaces with affine arithmetic. In *Proceedings Eurographics Workshop on Implicit Surfaces*, pages 161–170. INRIA, 1995.

[Duv89] D. Duval. Handling algebraic numbers in computer algebra. In *ISSAC'89*, 1989.

[DY95] T. Dubé and C.K. Yap. The exact computation paradigm. In 2nd Edition Du andHwang, World Scientific Press, editor, *Computing in Euclidean Geometry*, 1995.

[EC92] I. Emiris and J. Canny. An efficient approach to removing geometric degeneracies. In *Proc. 8th ACM Symp. on Comp. Geometry*, pages 74–82, Berlin, Germany, 1992.

[EM90] H. Edelsbrunner and E.P. Mücke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graph*, 9:66–104, 1990.

[Emi94] I.Z. Emiris. *Sparse Elimination and Applications in Kinematics.* PhD thesis, Computer Science Division, Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, December 1994.

[EY88] L.W. Ericson and C.K. Yap. The design of linetool a geometric editor. In *Symposium on Computational Geometry*, pages 83–92, 1988.

[For87] S. Fortune. A sweep-line algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[For95] S. Fortune. Polyhedral modelling with exact arithmetic. In *Proceedings of the Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 225–233, May 1995.

[FvDF+94] James D. Foley, A. van Dam, Steven K. Feiner, John F. Hughes, and Richard L. Phillips. *Introduction to Computer Graphics.* Addison-Wesley, Reading, Mass., 1994.

[FVW93] S. Fortune and C. Van Wyk. Efficient exact arithmetic for computational geometry. In *Proceedings of the 9th ACM Symposium on Computational Geometry*, pages 163–172, San Diego, May 1993.

[GD94] T. Gomez-Diaz. *Quelques applications de l'évaluation dynamique*. PhD thesis, Université de Limoges, janvier 1994.

[GHPT89] M. Gangnet, J.C. Hervé, T. Pudet, and J.M. Van Thong. Incremental computation of planar maps. *ACM Computer Graphics (SIGGRAPH 89)*, 23(3):345–354, July 1989.

[GM84] M. Gangnet and D. Michelucci. Un outil graphique interactif. In *Proceedings of MICAD 84*, pages 95–110. Hermès, Feb.-Mar 1984.

[Grü67] B. Grünbaum. *Convex polytopes*. London Interscience, 1967.

[GT91] M. Gangnet and J.M. Van Thong. Robust boolean operations on 2d paths. In *Proceedings of COMPUGRAPHICS91*, volume 2, pages 434–443, Sesimbra, Portugal, 1991.

[Hof89] C. M. Hoffmann. *Geometric and Solid Modeling: An Introduction*. Morgan Kaufmann, 1989.

[Hof90] C.M. Hoffmann. A dimensionality paradigm for surface interrogations. *IEEE Computer Aided Geometric Design*, 7:517–532, 1990.

[Hon86] J.W. Hong. Proving by example and gap theorem. In IEEE Computer Society Press, editor, *27th symposium on Foundations of computer science*, pages 107–116, Toronto, Ontario, 1986.

[HPY96] C.-Y. Hu, N. Patrikalakis, and X. Ye. Robust interval solid modelling. part 1: Representations. part 2: Boundary evaluation. *CAD*, 28(10):807–817, 819–830, 1996.

[IS89] M. Iri and K. Sugihara. Construction of the Voronoi diagram for one million generators in single-precision arithmetic. In *Proceedings of the 1st Canadian Conference on Computational Geometry*, Montréal, 1989.

[Jac95] D. Jackson. Boundary representation modelling with local tolerances. In *Proceedings of the Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 247–253, 1995.

[Jus92] N.P. Juster. Modelling and representation of dimensions and tolerances: a survey. *CAD*, 24(1):3–17, jan 1992.

[Kea96] R.B. Kearfott. *Rigorous Global Search: Continuous Problems*. Kluwer, Dordrecht, Netherlands, 1996.

[KKM97] J. Keyser, S. Krishnan, and D. Manocha. Efficient brep generation of low degree sculptured solids using exact arithmetic. In *Proceedings of the Symposium on Solid Modeling Foundations and CAD/CAM Applications*, May 1997.

[KLN91] M. Karasick, D. Lieber, and L.R. Nackmann. Efficient delaunay triangulation using rational arithmetic. *ACM Transactions on Graphics*, 10:71–91, Jan. 1991.

[KM96] S. Krishnan and D. Manocha. Efficient representations and techniques for computing b-reps of csg models with nurbs primitives. In Information Geometers Ltd, editor, *Proceedings of CSG96*, pages 101–122, Winchester, UK, April 1996.

[Knu81] D.E. Knuth. *Seminumerical Algorithms*, volume 2. Addison-Wesley, Reading, Mass., 1981.

[Knu92] D.E. Knuth. *Axioms and hulls*. Lecture Notes in Computer Science (606), Springer-Verlag, 1992.

[Kon92] K. Kondo. Algebraic method for manipulation of dimensional relationships in geometric models. *Computer Aided Design*, 24(3):141–147, mars 1992.

[Loo83] R. Loos. Computing in algebraic extensions. In *Computer algebra, symbolic and algebraic computation*. Springer-Verlag, 1983.

[MG94] M.B. Monagan and G.H. Gonnet. Signature functions for algebraic numbers. In *Proceedings ISSAC*, pages 291–296. ACM Press, 1994.

[Mic87] D. Michelucci. *Les représentations par les frontières : quelques constructions; difficultés rencontrées (in french)*. PhD thesis, École Nationale Supérieure des Mines de Saint-Étienne, 1987.

[Mic95] D. Michelucci. An epsilon-arithmetic for removing degeneracies. In *Proceedings of the IEEE 12th Symposium on Computer Arithmetic*, pages 230–237, Windsor, Ontario, July 1995.

[Mic96] D. Michelucci. Arithmetic issues in geometric computations. In J-C. Bajard, editor, *Proceedings of the 2nd real Numbers and Computers*, April 1996.

[Mil88] V.J. Milenkovic. *Verifiable Implementations of Geometric Algorithms Using Finite Precision Arithmetic*. PhD thesis, Carnegie-Mellon, 1988.

[Mis93] B. Mishra. *Algorithmic Algebra*. Springer-Verlag, New York, 1993.

[MM] D. Michelucci and J-M. Moreau. Lazy arithmetic. *To be published in IEEE Transactions on Computers*. Available at: <ftp://ftp.emse.fr/pub/papers/LAZY/lazy.ps.gz>.

[MM94] V. Ménissier-Morain. *Arithmétique exacte*. PhD thesis, Université Paris VII, 1994.

[MMZ94] J. Menon, R.J. Marisa, and J. Zagajac. More powerful solid modeling through ray representations. *IEEE Computer Graphics and Applications*, 14(3):22–35, May 1994.

[MN90] V.J. Milenkovic and L.R. Nackmann. Finding compact coordinate representations for polygons and polyhedra. *IBM Journal of Research and Development*, 34(5):753–769, Sept. 1990.

[MV95] J. Menon and H. Voelcker. On the completeness and conversion of ray representations of arbitrary solids. In Chris Hoffman and Jarek Rossignac, editors, *Solid Modeling '95*, pages 175–186, May 1995.

[NSTY93] J. Nakagawa, H. Sato, K. Toshimitsu, and F. Yamagushi. An adaptive error-free computation based on the 4x4 determinant. *The Visual Computer*, 9:173–181, 1993.

[OTC87] G. Ottmann, G. Thiemt, and Ullrich C. Numerical stability of geometric algorithms. In *Proceedings of the 3rd ACM Symposium on Computational Geometry*, pages 119–125, 1987.

[PA94] R.M. Persiano and A. Apolinário. Boundary evaluation of csg models by adaptative triangulation. In *CSG 94 : Set Theoretic Solid Modelling Techniques and Applications*, Information Geometers Ltd, april 1994.

[Pri96] M.G. Prisant. Application of the ray-representation tp problems of protein structure and function. In Information Geometers Ltd, editor, *Proceedings of CSG96*, pages 33–47, Winchester, UK, April 1996.

[SBE95] P. Stucki, J. Bresenham, and R. Earnshaw. Computer graphics in rapid prototyping technology. *IEEE Computer Graphics and Applications (special issue on Rapid Prototyping)*, 15(6):17–19, Nov. 1995.

[Sch89] J. Schwarz. Implementing infinite precision arithmetic. In *Proceedings of the IEEE 9th Symposium on Computer Arithmetic*, pages 10–17. IEEE Computer Society, 1989.

[Seg90] M. Segal. Using tolerances to guarantee valid polyhedral modeling results. *Computer Graphics (SIGGRAPH '90 Proceedings)*, 24(4):105–114, August 1990.

[Sny92] J.M. Snyder. Interval analysis for computer graphics. *Computer Graphics*, 26(2):121–130, july 1992.

[Tau93] G. Taubin. An accurate algorithm for rasterizing algebraic curves. In *Second Symposium on Solid Modeling and Applications, ACM/IEEE*, pages 221–230, May 1993.

[TGP96] R.F. Tobler, T.M. Galla, and W. Purgatofer. Acsgm–an adaptative csg meshing algorithm. In Information Geometers Ltd, editor, *Proceedings of CSG96*, pages 17–31, Winchester, UK, April 1996.

[Vui90] J.E. Vuillemin. Exact real computer arithmetic with continued fractions. *IEEE Trans Computers*, 39(8):1087–1105, 1990.

[WB96] K.D. Wise and A. Bowyer. Using csg models to map where things can and cannot go. In Information Geometers Ltd, editor, *Proceedings of CSG96*, pages 359–376, Winchester, UK, April 1996.

[WG95] Chionh Eng Wee and Ronald N. Goldman. Elimination and resultants, part 1: Elimination and bivariate resultants. *IEEE Comp. graphics and applications*, pages 69–77, jan. 1995.

[Wie80] E. Wiedmer. Computing with infinite objects. *Theoretical Computer Science*, 10:133–155, 1980.

[Yam87] F. Yamagushi. Theoretical foundations for the 4x4 determinant approach in computer graphics and geometrical modeling. *The Visual Computer*, 3:88–97, 1987.

[Yap88] C.K. Yap. A geometric consistency theorem for a symbolic perturbation scheme. In *Proceedings of the 4th ACM Symposium on Computational Geometry*, pages 134–142, 1988.

[Yap96] C.K. Yap. *Fundamental Problems in Algorithmic Algebra*. Princeton University Press, 1996.