Lazy arithmetic

D. Michelucci - J-M. Moreau
LISSE-ENSM.SE[1]

March 1995

Technical report **RR 95.4**

[1]Address: 158, cours Fauriel, F 42 023 Saint-Étienne. Email: `moreau,micheluc@emse.fr`

Abstract

Finite-precision leads to many problems in geometric methods from CAD or Computational Geometry. Until now, using exact rational arithmetic was a simple, yet much too slow solution to be of any practical use in real-scale applications. A recent optimization – the lazy rational arithmetic ([4]) – seems promising: It defers exact computations until they become either unnecessary (in most cases) or unavoidable; in such a context, only indispensable computations are performed exactly, that is: Those without which any given decision cannot be reached safely using only floating-point arithmetic. This rapport takes stock of the lazy arithmetic paradigm: Principles, functionalities and limits, speed, possible variants and extensions, difficulties, problems solved or left unresolved.

Keywords: Computational Geometry, Exact Rational Arithmetic, Hash Coding, Inconsistencies, Interval Arithmetic, Lazy Arithmetic, Robustness.

Résumé

La précision finie des machines conduit à un grand nombre de problèmes dans les méthodes géométriques de la CA) ou de la Géométrie Algorithmique. Jusqu'ici, le recours à une arithmétique exacte rationelle était considéré comme une solution simple, mais bien trop lente pour pouvoir s'appliquer à des applications de grandeur réelle. Une optimization récente – l'arithmétique rationalle paresseuse ([4]) – seble prometteuse: elle repousse les calculs exacts jusqu'à ce qu'ils deviennent soit inutiles (le plus souvent) ou inévitable ; dans un tel contexte, seuls les calculs indispensables sont effectués en exact, c'est-à-dire : ceux sans lesquels une décision donnée ne peut être prise de manière fiable à l'aide de la seule arithmétique flottante. Ce rapport fait le bilan du paradigme paresseux : principes, fonctionnalités et limites, rapidité, variables éventuelles et extensions, difficultés, problèmes résolus ou toujours sans réponse.

Contents

1

Framework

A large number of Computer Science communities are concerned with finite-precision. Among them, we find designers of electronic circuits or arithmetic libraries; specialists of Numerical Analysis, Analytical or Symbolic Calculus; researchers or practictioners in Computational Geometry (CG), Computer Aided Design (CAD, solid modelling, robotics, ...), Image Synthesis, Signal Processing, Cryptography, and programming language experts. None of these communities has a full understanding of the techniques used by the other "tribes", nor of the way their experts handle their own problems.

In some such communities, it is often thought that finite-precision problems only have serious consequences in marginal applications, such as Symbolic Calculus or Cryptography. However, the geometrical algorithms used in CAD or CG are also greatly perturbed by numerical imprecision, and this even for such mathematically trivial problems as the detection of segment intersections in the plane.

Initially, the CAD and CG communities have also underestimated the problems raised by imprecision, and their consequences, *viz.* inconsistencies in the results and unreliability of the applications. As the research in these fields developed, it became apparent that such problems were inherent to the implementation of geometric methods on finite-precision machines, and some renowned authors (among which Ottmann, Thiemt, Ullrich, or Edelsbrunner, Karasick, Milenkovic and Fortune) suggested more and more elaborate solutions. For example, the *XSC* language family was developed to try and solve precision problems in a systematic way, although not entirely transparently for programmers. Some aspects of this research are summarized in the paper.

Today, correctly handling the side-effects of imprecision with reasonable losses in execution speed, memory consumption, or programming comfort has become one of the major goals in Computational Geometry (from both theoretical and practical points of views) for the coming years. The difficulties are such that some specialists from Solid Modelling (among which Requicha and Rossignac), are willing to replace the fundamental *Boundary Representation* data structure, on which most industrial software applications are built, with the *Constructive Solid Geometry* model. The 1994 CSG Symposium was intended to stimulate research in this direction:

> *Constructive Solid Geometry representations of objects are based on set theory and are unambiguous and robust; but commercial CAD systems are mostly built on boundary modellers, even though there are difficult numerical problems in maintaining a boundary representation reliably. Is this a sound technical choice, or an accident of history? Research aimed at extending the scope and flexibility of CSG modelling systems will answer this question*[1]*...*

Others, like Françon, or Réveillès, have opted for Discrete Geometry because it is a fairly new and interesting branch of computer mathematics where practical applications have virtually

[1] *in* Call for papers, "Set-theoretic Solid Modelling : Techniques and Applications", Winchester, UK, pages 13-15, April 1994.

no chance of being impaired by imprecision. The first two sessions of the major symposium in this field (called "Géométrie Discrète en Imagerie : Fondements et Applications") were held in Grenoble, France, 1992, and Strasbourg, France, 1993.

A third approach is to resort to exact arithmetic. This constitutes the most obvious and general solution to precision problems, and has been used for quite some time. Unfortunately, its prohibitive cost makes it unapplicable to Computational Geometry. One has to remember that typical applications in this field manipulate very large amounts of data (*e.g.* for modelling terrains or real-scale scenes), and that representing exact quantities may become next to untractable when dealing *on-line* with thousands of points whose co-ordinates involve several digits in a very large base!

However, the authors have recently suggested a new solution ([4], [3]) for efficiently handling imprecision in Computational Geometry. A specialized library (*LEA*, short for *Lazy Exact Arithmetic*), has the responsibility of switching from finite-precision to exact computations (and then back...) whenever approximations are not sufficient to allow safe decisions. Such switchings are made by the library itself, without explicit code for this in the programs. The expected result is to defer exact computations until they are either not needed or inevitable: This agrees strongly with the intuition that most decisions may be made using approximations, while only a few are too "tight", and cannot be made without the help of an exact arithmetic.

To paraphrase David Goldberg ([12]), this paper could be entitled "What Every Computer Scientist Should Know about *Laziness* in Computer Arithmetic". Section 2 presents the consequences of numerical imprecision on geometric methods. These problems are hardly acknowledged at all outside the fields of Computational Geometry and CAD. Section 3 lists classical solutions for solving such problems, and analyzes their advantages and drawbacks. Section 4 presents the lazy paradigm. Section 5 presents two important problems that had to be solved for implementing our lazy library, Section 6 extensions for the lazy library, and Section 7 some of its limitations. Finally, Section 8 concludes on future directions.

2

Imprecision in geometry

2.1 Stating the problem

From our point of view (theory and application in Computational Geometry), there are two types of numerical computations with machines:

- Those for which an approximation of numbers *is sufficient*, provided of course the committed error is less than a given tolerance.
- Those for which an approximation of numbers *is not sufficient*, whatever its quality. This is typically the case when the sign of a quantity is requested. The approximation x_ε of a number x to within ε is insufficient to determine the sign of x as soon as $|x_\varepsilon| < \varepsilon$, whatever the magnitude of ε. Furthermore, with such arithmetics, it is fundamentally impossible to detect the *equality* of two numbers.

Symmetrically, there are also two types of arithmetic:

- *Approximate arithmetics* such as floating-point arithmetic, interval arithmetic ([32], [25] [26]), or stochastic arithmetic ([41]) cannot guarantee the nullity of a number, or, equivalently, the equality of two numbers.
- *Exact arithmetics* which represent numbers in an exact way[1], and allow to compute the sign of a number, or to compare two numbers, with no possible error.

It is well known that Symbolic Calculus or Cryptography cannot be content with approximate arithmetics. However, the geometric algorithms from CAD or Computational Geometry are themselves greatly perturbed by numerical imprecision for the following reason:

The many decisions made by a geometric algorithm are not independent.

A few simple examples of this are given in 2.2. Numerical imprecision may then force geometric algorithms to make contradicting decisions, the implications of which are detailed in 2.3.

2.2 Examples of imprecision

Let us consider a few bi-dimensional examples to illustrate the imprecision problems that the finite-precision implementation of a typical geometric algorithm may run into. We shall use the usual following notations:

1. Infinite lines with equation $\alpha x + \beta y + \gamma = 0$ are represented by a triple of real numbers (α, β, γ).

[1] Provided the problems to be solved may effectively be modelled in such conditions! This naturally calls for squaring distances if one is using a rational arithmetic package, and so forth...

2. The triple for the infinite 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 Ω with co-ordinates

$$x_\Omega = \frac{\beta\gamma' - \beta'\gamma}{\alpha\beta' - \alpha'\beta} \quad y_\Omega = \frac{\alpha'\gamma - \alpha\gamma'}{\alpha\beta' - \alpha'\beta}$$

4. Geometric algorithms often use the *lexicographical order* on co-ordinates, defined by:

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

Example 1

From a theoretical point of view, the *power* of point $\Omega(x_\Omega, y_\Omega)$ with respect to D and its power *w.r.t.* D', defined respectively as $P_D(\Omega) = (\alpha x_\Omega + \beta y_\Omega + \gamma)$ and $P_{D'}(\Omega) = (\alpha' x_\Omega + \beta' y_\Omega + \gamma')$, should both be null. In practice, this is not true, due to floating-point imprecision, as this is shown in the following example:

$$D : (\alpha, \beta, \gamma) = (3, 13, 6), D' : (\alpha', \beta', \gamma') = (5, -17, 11)$$

$$P_D(\Omega) = -8.88 \cdot 10^{-16}, \quad P_{D'}(\Omega) = 0.$$

Note that, by modifying the value of any coefficient in either triple, it is possible to simulate situations in which both powers are 0, different from 0, or only one is! As a consequence, checking the position of Ω against either line at a later stage of the algorithm may yield contradicting results. Worse still, this example shows that, in transforming any given graph into a planar one, finite-precision alters the true geometry of objects: One initial segment is transformed into two subsegments that are not collinear with it!

Example 2

Let us note that all numbers in the preceding example were integers, and that the problem is even worse as soon as decimal numbers are entered. Consider for instance solving such a simple system as $Ax = b$, where x is the unknown vector:

$$A = \begin{pmatrix} 0.2 & 0.3 \\ 0.3 & 0.45 \end{pmatrix} \quad \text{and} \quad b = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

Obviously, the determinant of A *is* null, because the lower row is equal to the upper one multiplied by 1.5. However, since converting decimal numbers such as 0.2 and 0.3 or 0.45 in base 2 does not terminate in a finite number of steps, the algorithms using these figures as inputs may fail to detect nullity, and come out with incongruous answers. This example[2] is typical of any geometric algorithm for finding, say, the inverse image of a point through a given linear transformation.

But machines can be even more mischievous than this! Consider the following determinant (quoted from [18]), the exact evaluation of which is -1:

$$\begin{vmatrix} 72450100 & 732698713 \\ 212345677 & 2147483637 \end{vmatrix}$$

The following pseudo-*C* program segment implements two methods for computing this determinant using `doubles`:

[2]Communicated by G. Bohlender, Universität Karlsruhe, during the September 1993 SCAN Symposium in Vienna.

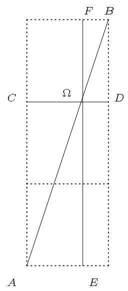

Figure 2.1: *A numerical and topological error.*

```
double a =  72450100.0, b =  732698713.0;
double c = 212345677.0, d = 2147483637.0;
double det1, ad, bc, det2;

  det1 = a*d - b*c;

  ad = a*d;
  bc = b*c;
  det2 = ad - bc;
```

In the first case `det1` yields the correct result (-1), while `det2` contains 0. The reason for this is that the larger number of bits used in the registers of the floating-point co-processor in the first case allows to get the correct answer, while storing intermediate results in double precision floating-point numbers induces a loss in precision and a wrong result in the second. Thus,

	holds	*instead of*
`ad`	155585404249013690.0	155585404249013700.0
`bc`	155585404249013690.0	155585404249013701.0

Such discrepancies between exact and computed results are enough to lose one's faith in machines

Example 3

Now consider Figure 2.1, where dotted squares have unit-length sides:

Points E and F have been chosen so as to have the same abscissa 0.6666667. The intersection between $[A, B]$ and $[C, D]$ should be $\Omega = (\frac{2}{3} \equiv 0.\overline{6}, 2)$, and thus, the exact lexicographical order should be $\Omega <_L E <_L F$. Unfortunately, Ω is computed as $\Omega^\star$, whose abscissa (0.6666667) is such that, the lexicographical order in the machine will be $E^\star <_L \Omega^\star <_L F^\star$. As a consequence, all subsequent decisions regarding the relative positions of these three points will be irrelevant! This is how a numerical error, small though it may be, becomes a topological error, an extremely frequent situation in Computational Geometry.

In general, intersecting a vertical segment $[A, B]$ and another, non-vertical, segment yields a point Ω with the same abscissa as A or B. However, finite-precision transforms Ω into a "machine" point $\Omega^\star$ such that either $x_A < x_{\Omega^\star}$ or $x_A > x_{\Omega^\star}$, which means that relying on a (real space) property such as:

$$\Omega \in [A, B] \Leftrightarrow A <_L \Omega <_L B$$

always fails in finite-precision contexts.

The most efficient geometric algorithms exploit order properties, in particular transitivity: If it is known for instance that $A <_L B$ and $B <_L C$, it will be inferred that $A <_L C$, without further checking. This shows how initial errors do not remain isolated, but are propagated and amplified all along the execution of the algorithm.

Example 4

If β and β' are both different from zero, the three following definitions for y_Ω are algebraically equivalent:

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

When evaluated in floating-point precision, they generally yield different results, due to truncature. 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, in different routines from the same program.

Example 5

In the projective plane, if the 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 $[P_1, P_2] \cap [P_4, P_5]$, $[P_2, P_3] \cap [P_5, P_6]$, and $[P_3, P_4] \cap [P_6, P_1]$ by Pappus 's theorem. Numerical imprecision generally prevents the detection of this property.

Another example of the same class is the following: If six distinct points $P_1, P_2, P_3, P_4, P_5, P_6$ belong to the same conic, then the three points $[P_1, P_2] \cap [P_4, P_5]$, $[P_2, P_3] \cap [P_5, P_6]$, and $[P_3, P_4] \cap [P_6, P_1]$ are aligned, by Pascal's theorem. Here again, numerical imprecision will rarely allow to check this property.

One last example of the same kind is given by the following problem: Prove that $N(> 4)$ given points are on the same circle. It is always possible to state the problem of cocircularity for four points in terms of a determinant ([14]), by observing that four points on a same circle in the plane are transformed into four coplanar points on the paraboloid of revolution

$$\Gamma : \{Q(x, y, z) \in \mathbb{R}^3 \quad | \quad z = x^2 + y^2\}.$$

Repeating this process on more than four points is doomed to failure, due to imprecision. Note that the other solution, which consists of comparing the center of the circumscribed circles of all triples of points, is even more unstable.

Example 6

To detect whether three given points A, B, C in the plane are aligned, form a convex angle (greater than π) or a reflex one, one may use the sign of the following determinant, expressing twice the signed area of triangle ABC:

$$\begin{vmatrix} 1 & 1 & 1 \\ x_A & x_B & x_C \\ y_A & y_B & y_C \end{vmatrix}$$

There are many ways to fool a program into finding a wrong answer to the original question, using this scheme. In general, it is sufficient to choose A and B very close, and B and C very far apart, or to choose triples such that the angle they form is very close to (but provably different from) π. It is thus possible to pick many (about $100,000$) random points on the unit circle, and to construct inconsistent convex hulls on them, through numerical imprecision.

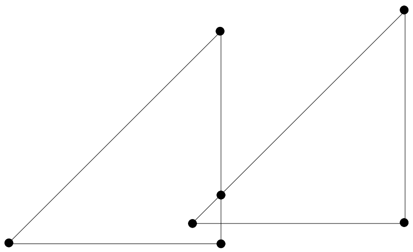

Figure 2.2: *An example of inconsistent topology: a forgotten vertex.*

2.3 Consequences

Inconsistent decisions have two types of consequences on a given algorithm:

1. The algorithm may terminate normally. This is what generally happens for "brute-force" algorithms (which process one item at a time, without relying on properties of all the already processed ones). However, such algorithms may (and will) produce results which are not only approximate but incoherent, *i.e.* the topology of which relates to no possibly valid object: For instance, a brute-force algorithm to detect all intersections among a set of segments in the plane will most likely output non-planar graphs (refer to Figure 2.2). Any such graph fed to another program that will assume its planarity, will invariably cause errors or infinite loops at execution time.

2. The algorithm may abort. This is what generally happens with the more sophisticated algorithms, which exploit geometrical consistency (*e.g.* transitivity of orders between geometric elements, or some of the properties listed in the examples), and whose execution is guided by intermediate results. Such algorithms err in theoretically impossible situations: Trying to delete an element which has never been inserted from a data structure, or ending up in an infinite loop while scanning a supposedly closed boundary that never returns to its starting point. Misadventures of this kind are detailed in [28], [33], or [6].

3

A catalogue of solutions

As we have just seen, any valid geometric property may be "proved wrong" due to numerical imprecision. Hence, even the simplest problem may be badly handled by finite-precision, however accurate the approximations used.

The examples above should have clearly illustrated that:

1. In some cases, there is a solid quantum leap between the conclusions of a finite-precision test, and its conclusions using exact computations.

2. It is impossible to identify all such cases in advance!

Let us emphasize that the problems here have nothing to do with the ill-conditioning considerations of Numerical Analysis: The wrong ordering of vertices in Example 3 cannot be imparted to ill-condition!

Somehow, the only way-out is to rely on an impartial *Oracle* to let the programs know whether they are right or wrong to make the decision they are currently making... Solutions to simulate this Oracle differ, and may be classified as follows.

3.1 Approximate solutions

In this class of solutions, all decisions are made on the basis of finite-precision only. Once the idea that the results may be erroneous is accepted, there are different ways to cope with it:

3.1.1 Warned programming

To our knowledge, forewarned programming (as in "forewarned is forearmed") has never been addressed in any Computer Science paper, so far. This term encompasses the whole set of common good-sense tricks used by practitioners in order to avoid the simplest and most common inconsistencies. A more detailed account of these may be found in [2] and [18].

Warned programming is about preventing certain kinds of inconsistencies by judiciously tailoring (or sometimes curbing) algorithms and data structures, so they "behave": For instance, in Example 1, a warned programmer will tag intersection points with where they come from, and use this information when checking points against lines. He or she will also be quite cautious not to split the original segments into new, fundamentally different ones, but will rather keep track of the fact that the newly created vertices belong to original segments, without altering the structures of the latter! In Example 2, a warned programmer might design special input functions to allow exact representation of decimals, or control the accuracy of data representations. To prevent errors in lexicographical order (Example 3), warned programmers will handle the intersection of vertical segments as a special case (imposing the abscissa of intersection points to be *equal* to that of the endpoints of the vertical segment). As for Example 4, no warned programmer will ever expect computing the same value using different (but equivalent) algebraic expressions!

Warned programming, as opposed to naive programming, manages to prevent in this fashion the "simplest" cases of inconsistencies (Examples 1-4). But preventing more complex inconsistencies, as those in the last two examples, looks much more like an *I.M.F.* task!... It is this sort of impossible mission that Symbolic Programming (presented below) is addressing, with more or less success.

3.1.2 Epsilon heuristics

Using "epsilons" is another popular heuristic. It has been widely used among practitioners ([16], [29]) because it is simple and straightforward: As soon as two values are closer than a given ε, consider them to be equal. Such an ε may depend on the values to be tested (angle, length, area...), on the application, or on the dynamic range of the data. While this solution may, in practice, improve the reliability of algorithms, the fine tuning of epsilons for a given application and its various typical data sets remains empirical and rather involved. Moreover, there is no way to guarantee that such a technique will effectively succeed: If one is relying on lexicographical order to decide on the next action, losing track of the true topology of objects inside an ε-wide strip has the same effect as shuffling objects inside that portion of space, and invariably results in making hazardous decisions.

3.1.3 Symbolic programming

This technique, suggested by well-known theoricians in Computational Geometry ([15], [29], [27], [17]), proscribes the use of exact arithmetics, on the (reasonable) pretense that their cost is prohibitive, and compensates by designing algorithms that exploit strong topological or geometrical properties of the objects being manipulated. For instance, V. Milenkovic and Z. Li ([27]) explain how to build ε-strong convex hulls, that is, objects that remain convex when any of their vertices is moved in any direction by a distance less than ε. Another technique, suggested by K. Sugihara and M. Iri ([17]) is to ensure certain topological properties during the incremental updation of a specific data structure, whatever actions the floating-point results would otherwise drive the program to take. For instance, if it is known that inserting a new point in a dynamic structure yields a well-identified graph invariant, then this invariant is ensured from the topological point of view, independently of what the available arithmetic would drive the algorithm to do.

3.1.4 Historical book-keeping

(first suggested by V. Milenkovic ([29])) consists in recording all decisions made at any time by a program, and then in consulting these "minutes" to make consistent decisions: For instance, if, at one point of execution, two objects have been found to lie in a certain order, any subsequent decision involving these two objects must be made in agreement with this order. Although this technique may induce inconsistencies between the computed and ideal results, it allows self-consistent results. On the other hand, it is extremely cumbersome and asks for vast amounts of memory space.

The straightforward advantages of the last two techniques are their robustness, and the fact that they allow consistent results. Their main drawback is that a specific solution must be developed for each and every problem to be solved, with a heavy implication for programmers. Until now, only a few algorithms have undergone such a dramatic treatment.

3.1.5 Confidence intervals

"ε-Geometry" ([13]) formalizes the empirical heuristic of epsilons. It allows computing "confidence intervals" for specialized predicates, within which floating-point results are reliable. Unfortunately, ε-Geometry does not help much in deciding what to do outside such intervals. A typical situation is given by Example 6, where one wishes to build a routine to decide whether the angle formed by

three distinct points is reflex, null, or convex. The associated predicate `TypeOfAngle` computes a determinant as mentioned in the same example, and returns a certified answer ("convex", "reflex") when this determinant is reliably found to be different from zero (using absolute error analysis) and a "fuzzy" answer ("don't know") otherwise. Of course, this last situation is exactly the one where something more "precise" would be expected from any predicate designed to help solve precision issues!

3.2 Classical exact solutions

3.2.1 Definitions

Suppose an algorithm is implemented on a computer as a program using a certain arithmetic. Let us use the term "ideal" to refer to the real mathematical constructs (tests, arithmetic operations, and so forth) in the abstract real space. Then, a computed value is said to be *exact* if it is equal to the ideal corresponding value, and two computed values are said to be *consistent* if and only if their mutual ordering is the same as their corresponding ideal ordering. An *exact* decision in a computed test is one that is guaranteed to be consistent with the decision implied by the ideal underlying test, for any set of operands. More prosaically, exact decisions are outcomes of tests which can never be biased by the implementation (finite-precision, overflow, underflow, ...).

As most problems from Computational Geometry may be modelled with an algebraic *decision tree*[1], what is actually needed is a method that yields consistent (if not exact) results, by means of exact decisions. Since methods that would only compute approximate values cannot be guaranteed to yield consistent results – as the previous examples have shown – one has to resort to using exact values. There are two "classical" ways to do this, as we shall now see.

3.2.2 Entirely exact methods

Using an exact arithmetic obviously rids programs of all precision issues and of their consequences, *i.e.* inconsistencies in the results. Another advantage is that since such solutions do not require the existing algorithms to be redesigned, all classical geometric theories (compatible with the exact library at hand) become readily available.

However, exact arithmetics (either rational or algebraic) have the disadvantage to yield irrelevantly precise results, *i.e.* results with a much greater associated precision than that of the initial data – which are only, at their best, reasonable approximations. Obviously, such extremely precise solutions are unnecessary, if not absurd: Consistency is the only thing that matters.

Furthermore, exact arithmetics provide such minute information at a prohibitive price: They are extremely costly both in execution time and memory consumption. It must be understood that, in general, exactly represented numbers are unbounded integers (built up of machine integers), or rationals (built up of unbounded integers), and that any intermediate result may grow as large as the problem at hand has a need for. The number of digits required to compute intermediate values in arbitrary precision may indeed be astronomical. To quote M. Karasick ([19]) on this:

> *Ten random points with double-precision co-ordinates were triangulated using floating-point arithmetic in .1 seconds; ten random points with rational co-ordinates (2-digit numerator, 3-digit denominator, in base 2^{16}) were triangulated using rational arithmetic in 1 200 seconds, generating intermediate values with as many as 81 digits.*

For all these reasons, exact arithmetics are very seldom used in Computational Geometry or CAD. When one is used, it is most likely a rational arithmetic, since pure algebraic arithmetics (*i.e.* non-rational ones) still seem impractical today. Their use is limited to specific applications of Symbolic Calculus.

[1] Refer to the seminal book by M.I. Shamos and D.F. Preparata ([37]) for a discussion.

3.2.3 Mixed methods

Such techniques have been used at different levels by T. Ottmann *et al* ([36]), M. Karasick *et al* ([19]), Fortune and van Wyk ([10]), F. Yamagushi ([43]), M. Gangnet and J-M. van Thong ([11]), J. Nakagawa *et al* ([35]) and one of the authors ([33]). The main idea is to use a mixed representation for the data, one of which is approximate (*e.g.* floating-point), and the other exact (*e.g.* rational, or unbounded integers). Every program test is rewritten – by the programmer, or by a pre-compiler – in such a way as to be performed using approximations first, and if finite-precision is not sufficient, in exact form.

T. Ottmann, G. Thiemt, and C. Ullrich use the *XSC* language family in the following way ([20], [36]): They first perform numerical tests with approximate *XSC* long reals; the properties of these languages (exact truncature, etc.) enable the authors to implicitly attach a confidence interval to each approximation. When these intervals are insufficient to safely make a numerical decision, the exact corresponding test is performed by determining the sign of a scalar product involving standard floating-point numbers (considered as exact numbers) from original data. Partial multiplications and additions in scalar products are computed exactly, thanks to the long enough mantissa register emulated in the *XSC* library. Of course, it is often necessary to retrieve original data, and this is done through some kind of *book-keeping* by the programmer. In the example of the detection segment intersections, it is necessary to store, together with the usual standard information (co-ordinates, attributes, etc.), the way a point was generated (if original data: Co-ordinates, if intersection of two segments: references to the endpoints of these segments, etc.).

M. Karasick or F. Yamagushi reduce all tests to computing the sign of the 4×4-determinant with "long integer" coefficients. From an informal point of view, their methods are equivalent: They both consist of computing with more and more "digits" (starting from most significant digits) until the sign is safely determined. This may lead to complete exact evaluations in certain cases, but in most situations, only a few operations are needed.

S. Fortune and C. van Wyk proceed in two stages: 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 floating-point arithmetic, using references to original data only;
2. to test, in the context of comparison against 0, if the absolute floating-point value is greater than the maximum possible error for the expression at hand, and thus to determine its sign reliably;
3. finally, to call the exact, "long integer", library to evaluate the expression.

The program is then compiled and linked with the exact library.

Note that in the three previous techniques, every exact test must be made with reference to original data. These techniques are fast, and yield consistent results. But they have several drawbacks: First, they systematically bound data range and computation depth, in order to use "long integers" (or long reals) of maximum constant size. Thus, they forbid fully on-line computations and reentrant algorithms: Suppose one wants to dynamically insert line segments in an initially empty set, and the endpoints of the newly inserted segments are allowed to be already detected intersection points. A simple back-of-the-enveloppe estimation shows that the quantity of information needed to allow exact computations is multiplied by four from one "generation" of points to the next. Secondly, they force programmers to provide explicit book-keeping mechanisms, which is most demanding. For instance, a given vertex will belong to one among different classes, depending on its history (original point, intersection point between original edges, intersection point between edges from different generations, and so on). Each time a procedure has to manipulate N vertices belonging to C possible classes, the programmer will have to write the code for the C^N different cases!...V. Milenkovic ([8]) gives a fairly good account of problems of this

kind in the case of Fortune-van Wyk's *LN* library ([10]). Remark. Very often, implementations do not use fully rational arithmetics, but only long integer or long real arithmetics, so they have to forbid or limit the use of divisions, since division introduces numbers with an infinite mantissa ($\frac{1}{3} = 0.333\ldots$). In Computational Geometry, this is not too much of a problem, because division may be easily avoided by using homogeneous co-ordinates.

The originality of the lazy paradigm presented below is to allow the book-keeping of any object to be made implicitly and dynamically, without *any* extra work from the programmer, as long as the essence of the problems to be treated remains consistent with that of the exact library being used. Moreover, lazy arithmetics allow fully dynamic (on-line) applications, and re-entrant algorithms.

4

Lazy arithmetic

4.1 Principles

The lazy rational arithmetic exploits the two following remarks ([2], [4], [18]):

1. A floating-point interval bracketing the rational value is very often sufficient to represent this value.
2. It is impossible to know beforehand (*i.e.* when a computation is called for) whether intervas will be sufficient or not.

A lazy arithmetic package performs computations in approximate form first. If it encounters a precision problem, it uses exact arithmetic to solve it, and then goes back to finite-precision. All this is done without any extra work from programmers: The library is totally transparent in that sense.

Hence, the basic idea is to defer exact evaluations until they become unnecessary (most of the time) or inevitable. Thus, no exact computation that is not strictly necessary will ever be performed.

4.2 Data structures for laziness

A (rational) lazy number is a cell containing:

- An interval with two floating-point bounds, guaranteed to bracket the rational number, be it known (exactly evaluated) or not.
- A symbolic definition, to allow retrieving the exact value of the underlying rational number.

Intervals are governed by interval arithmetic ([26], [32]). The usual rational numbers have minimum width intervals associated with them, and the interval associated with any cell has bounds determined by simple rules, such as: If z, z' are two lazy numbers with associated intervals $[\alpha_z, \beta_z]$ and $[\alpha_{z'}, \beta_{z'}]$, respectively, then the interval for the sum $z + z'$ is $[\bigtriangledown(\alpha_z + \alpha_{z'}), \bigtriangleup(\beta_z + \beta_{z'})]$, where $\bigtriangledown$ ($\bigtriangleup$) denotes the function returning the nearest machine number below (above) its argument. Some care must be taken to avoid overflow and underflow problems.

A symbolic definition is either:

- A standard (evaluated) rational number, represented, for instance, by a numerator and a denominator, *i.e.* two lists of digits in a given (large) base,
- A 'sum' or 'product' cell, referencing two other lazy numbers,
- An 'opposite' or 'reciprocal' cell, referencing another lazy number.

Thus, a lazy number is the root of a "tree" of lazy numbers, the internal nodes of which are binary (sum or product) or unary (opposite, reciprocal) operators, and the leaves of which are standard rational numbers. Actually, since any node or leaf may be shared, lazy numbers induce *directed acyclic graphs* (*dag* for short), rather than trees. Lazy numbers may access their children (the operand(s)) but not their ancestors.

4.3 Lazy elementary arithmetic operations

Performing an elementary lazy operation consists in allocating a cell (sum, product, opposite, reciprocal), in computing the bounds of the interval associated with the cell, and in assigning the references of the definition. This is done in constant time, except in the special case described in point 3 below. The exact arithmetic operation is not performed, and never will be if the interval proves sufficient for all subsequent computations.

The intervals associated with two lazy numbers are sufficient to compare them when they do not overlap. In the opposite case, the library must perform an exact evaluation of their symbolic definitions, in order to give a safe answer. The only situations where the exact rational evaluation of lazy number z is required are:

1. When the library needs to compare z to another lazy number with an overlapping interval.
2. When the library needs to determine the sign of z, and its interval contains 0.
3. When the library needs to compute the interval for the reciprocal of number z whose interval contains 0.
4. When the library is evaluating another lazy number whose dag references z.

As mentioned earlier, point 3 corresponds to the only situation where the simple initialization of the interval for a lazy number actually involves an evaluation, and is not, therefore, a constant time process: Since the interval for z includes 0, its image under $x \rightarrow x^{-1}$ is *not* a finite, closed interval: Evaluation is the most natural way to assign z^{-1} one valid interval in that case.

4.4 Evaluation strategies

The simplest form of evaluation is a recursive function which evaluates the children of the lazy number at hand, then performs the operation carried in its associated cell, using rational arithmetic. The unevaluated definition field is then physically replaced by the evaluated definition (*e.g.* the actual irreducible rational fraction). In this process, all lazy numbers that are no longer referenced become obsolete, and are recycled through a garbage collection mechanism.

The library also refreshes the intervals of the evaluated numbers during evaluation. Note that the more operations involved in a definition, the slacker its root interval. Complete evaluation can thus be regarded as the ultimate tool for refining the intervals of definitions corresponding to numbers which have become too imprecise due to the complexity of their definition.

In that regard, the multiplicative inverse may certainly be regarded as the worst possible operation: Suppose the interval for lazy number z is $[1\cdot 10^{-6}, 2\cdot 10^{-6}]$ (quite a reasonable assumption for standard input data), then the interval for its reciprocal will be something like $[\bigtriangledown(5\cdot 10^5), \bigtriangleup(10^6)]$, an interval with a very large amplitude indeed! Obviously, only the rational evaluation of z will allow the library to refine this interval down to near the *ulp*...

One may imagine different and more elaborate evaluation strategies. For instance, if some descendant d of a given lazy number z has been evaluated recently, the interval for z may be refined, but has no reason to have been refined during the evaluation of d. One so-called "refreshing" strategy first re-evaluates the interval for the considered number (z): Thus, it may happen that the usual bottom-up process makes the interval of z small enough to allow the operation (which primarily failed) to succeed without the library having to perform any evaluation.

Another ("up and down", or "yoyo") strategy consists in only evaluating the lowest internal nodes (those whose children are 'leaves'), then in bubbling interval information up to the root, as

Table 4.1: Running time ratios of the lazy, floating point, and exact (rational) versions of the same program. These figures are quoted from P. Jaillon's PhD Thesis ([18]).

Lazy/Float	4	8	12	16	20
10^{-3}	10.4	10	9.2	8.8	8.3
10^{-6}	10.4	9.8	9.2	8.8	8.3
10^{-9}	10.5	9.9	9.2	8.8	8.3
10^{-12}	10.5	9.9	9.3	8.8	8.3

Rats/Lazy	4	8	12	16	20
10^{-3}	57.8	80.9	91.1	97.9	101.6
10^{-6}	172.2	249.9	278.6	297.7	307.7
10^{-9}	348.9	485.7	531.7	568.6	588.6
10^{-12}	573.2	836.7	921.9	987.9	1038.2

in the previous strategy. This process is iterated as long as necessary (until all nodes referenced by the lazy number considered are evaluated, in the worst case).

A final 'asymmetrical' strategy first evaluates the children whose intervals are largest, then bubbles up interval information... All these strategies have been tested, and have proven to have about the same behaviour, within 10%, in our applications.

4.5 Usage, performance, applications

A lazy exact rational arithmetic library (*LEA*) has been developed in *C++* at EMSE, where it has also been tested. It is possible – if need be – to obtain the standard floating-point version, the exact version, and the lazy version of the same algorithm by simply using a nickname type, say `MyNumbers`, that will be a synonym of `double` in the floating-point version, of `Rat` in the exact version, and of `LazyNumber` in the lazy version. To obtain either version, one only has to make the appropriate changes in the type declaration, and to link the program with the appropriate library.

Empirical tests have shown that the lazy implementation of Bentley and Ottmann's segments intersection algorithm ([7]), can be more than 150 times faster than its purely rational counterpart. The lazy arithmetic version is only 4 to 10 times slower than the resident floating-point version (that is when the latter manages to terminate successfully, obviously...).

This library has also been tested on a program for computing the intersections of several polyhedra by members from the "lazy group research team" at EMSE ([6]). Table 4.1 shows the lazy-to-float and rational-to-lazy ratios for the running times of this algorithm on scenes with 4, 8, 12, 16, and 20 cubes and initial data precision ranging from 10^{-3} to 10^{-12}. The cubes have unit sides, and are randomly rotated around the origin. The lazy-to-float ratio is ten or less when the number of cubes increases, and does not depend on the precision of the initial data. On the other hand, the rational-to-lazy ratio increases with the precision of initial data.

LEA – which is to be shortly used in large-scale applications, such as the Delaunay triangulation of terrains, or the development of a geometric library – has also been used with some success to triangulate linear systems, and to find the sign of determinants using Gauss's elimination method. It is true that the now classical modular arithmetic ([21]) also allows to solve linear systems, without the hussle of numerical imprecision, nor the problems pertaining to ill conditioning. However, modular arithmetic neither allows to efficiently compute the sign of numbers, nor to order them. Thus, lazy arithmetic solutions are very promising alternatives.

Another potential usage for lazy arithmetic is Numerical Analysis, which often uses iterative and convergent algorithms. Numerical imprecision sometimes lead to problems, *e.g.* wrong convergence or non-convergence. Let us take an example from J-M. Muller's authoritative book on Computer Arithmetic ([34]): Define

$$a_{n+1} = f(a_{n-1}, a_n) = 111 - \frac{1130}{a_n} + \frac{3000}{a_n a_{n-1}}.$$

$x = f(x, x)$ has three solutions (5, 6 and 100). If one fixes $a_0 = \frac{11}{2}$ and $a_1 = \frac{61}{11}$:

1. The sequence *ideally* converges to 6.
2. Using floating-point arithmetic, it converges to 100: The successive values of a_n using double-precision 80-bit numbers (extended doubles) exhibit the following general pattern:

$$\begin{array}{c} a_1{=}5.54545454545454 \\ \dots \\ a_{15}{=}5.98143989459209 \\ a_{16}{=}6.65729723987226 \\ a_{17}{=}16.6000501930843 \\ a_{18}{=}70.0744205165950 \\ a_{19}{=}97.4532928295079 \\ \dots \\ a_{30}{=}99.9999999999999 \\ \dots \end{array}$$

3. Using *LEA*, it successfully converges to 6: The width of the lazy intervals for the same sequence grows from a_2 : $[5.59016\dots\ 5.59016\dots]$ to a_{10} : $[-40.9888\dots\ 47.7436\dots]$. Computing the next term implies division by an interval containing 0. This causes an exact evaluation which "shrinks" the intervals of all already computed terms (due to the structure of the computations), and, for instance, the interval for a_{11} is $[5.89915\dots\ 5.89915\dots]$. From then on, successive terms are computed using interval arithmetic only, until another degenerate interval is encountered by the library.

In conclusion, lazy arithmetic packages automatically detect when intermediate values should be computed with more precision. This property seems very interesting for Numerical Analysis, but we have been too lazy to study the question in detail.

5

Two lazy questions

Two important problems had to be specifically solved to allow an efficient implementation of *LEA*, as described in this section.

5.1 Labelling lazy numbers

Some geometric algorithms require that labels (*hash keys*) be attached to numbers. For instance, such labels allow to retrieve vertices from their co-ordinates in a *hash table*. The lazy paradigm does not readily allow this: Since the values of lazy numbers are not necessarily available, how is it possible to assign labels without evaluation?

One basic requirement is that the same label should be assigned to two equal numbers, with possibly different definitions, such as those from Example 4:

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

The solution presented in ([5]) exploits the morphism between $\mathbb{Q}$ and the finite field $\mathbb{Z}_p = \mathbb{Z}/p\mathbb{Z}$: Let p be a (large) prime, typically chosen so that p, and even p^2, may be represented by one machine integer. Labels are integers from $[0, p[\cup\{\Omega\}$, where Ω may be chosen equal to p.

The label assigned to an irreducible rational fraction $q = \frac{x}{y}$ is, by definition, $\Psi(\frac{x}{y}) = x(y^{-1})\%p$ (where % stands for the *modulo* operator), whenever y has an inverse in Z_p, and $\Psi(\frac{x}{y}) = \Omega$ otherwise. In general, the label for $q + q', q * q', -q, q^{-1}$ may be efficiently computed using the labels for q and q', as shown in table 5.1.

It is readily seen that there are only three indeterminate cases for which the label may not be uniquely defined (refer to [5] for more detail): this is mainly because, fundamentally, $\mathbb{Q}$ and $\mathbb{Z}_p$ are not isomorphic. In such cases, labels must be computed, for instance directly after the lazy numbers have themselves been evaluated.

One may notice that only one among the p^2 cells of the first table, and two in the second, yield indeterminate answers. Interestingly enough, empirical tests have shown that, for random data, the frequency of indeterminations was close to $\frac{1}{p}$, while it was under 1 in a million for "real-world" data (p being chosen equal to 65,521 in both cases).

The elementary arithmetic operations in $\mathbb{Z}_p$ may be performed in constant time, except, possibly, for the computation of reciprocals: If the library provides an $O(p)$-space table of reciprocals for numbers in $[1, \frac{p-1}{2}]$, this operation requires $O(1)$ time. Otherwise, the library must resort to using either Euclid's extended algorithm, or exponentiation ($q^{-1} = q^{p-2}\,\%p$), both of which methods take $O(\log p)$ time.

Another representation for hash keys virtually rids the library of the need to compute reciprocals in $\mathbb{Z}_p$: a key Ψ (as defined earlier) is represented by a couple $(\eta, \delta) \in \mathbb{Z}_p^2$, such that $\eta = \Psi \times \delta\,\%p$. All couples $(\lambda\eta\,\%\,p, \lambda\delta\,\%\,p)$ with $\lambda \neq 0$ are equivalent representations of key Ψ. Couples $(\eta \neq 0, 0)$ represent Ω. Couple $(0, 0)$ represents the indeterminate key.

Table 5.1: Computation rules for $\Psi(q+q')$, for $\Psi(q*q')$, and for $\Psi(-q)$ and $\Psi(q^{-1})$. (λ, λ' are any two values in $]0,p[$, and "?" stands for indetermination).

	$\Psi(q)=\Omega$	$\Psi(q)=0$	$\Psi(q)=\lambda$
$\Psi(q')=\Omega$	?	Ω	Ω
$\Psi(q')=0$	Ω	0	λ
$\Psi(q')=\lambda'$	Ω	λ'	$[\lambda+\lambda']\,\%\,p$

	$\Psi(q)=\Omega$	$\Psi(q)=0$	$\Psi(q)=\lambda$
$\Psi(q')=\Omega$	Ω	?	Ω
$\Psi(q')=0$	?	0	0
$\Psi(q')=\lambda'$	Ω	0	$[\lambda*\lambda']\,\%\,p$

$\Psi(q)$	Ω	0	λ
$\Psi(-q)$	Ω	0	$[p-\lambda]$
$\Psi(q^{-1})$	0	Ω	$[\lambda^{-1}]\,\%\,p$

Computation rules over couples may then be expressed as:

$$\begin{aligned}(\eta,\delta)+(\eta',\delta') &= ((\eta\delta'+\eta'\delta)\,\%\,p, \delta\delta'\,\%\,p)\\ (\eta,\delta)*(\eta',\delta') &= (\eta\eta'\,\%\,p, \delta\delta'\,\%\,p)\\ -(\eta,\delta) &= (-\eta\,\%\,p,\delta)\\ (\eta,\delta)^{-1} &= (\delta,\eta)\end{aligned}$$

When the result is $(0,0)$, the key is indeterminate, and the lazy number may be evaluated by the library. This only happens in the same cases as previously, the only difference being in the notation:

$$\begin{aligned}(\eta,0)+(\eta',0) &= (0,0)\\ (\eta,0)*(0,\delta') &= (0,0)\end{aligned}$$

With this representation, computing reciprocals in $\mathbb{Z}_p$ is only necessary when the hash key is effectively required (by user programs!), and no longer each time the reciprocal of a number is found in a lazy expression the key for which is being computed: This specific operation may now be done through a simple swap in the couple elements!

Labels may also be used for the needs of the lazy library proper: for instance, if two lazy numbers have different labels, then they must be different (although they still cannot be ordered). It is thus possible to detect the non-equality of two numbers when their intervals overlap, provided they are wise enough to have different labels. Note that, with this new representation, testing the equality between two couples (η,δ) and (η',δ') does not require computing an inverse in $\mathbb{Z}_p$: In effect,

$$(\eta,\delta)=(\eta',\delta') \Leftrightarrow \eta\delta^{-1}=\eta'(\delta')^{-1}\,\%\,p \Leftrightarrow \eta\delta'=\eta'\delta\,\%\,p.$$

5.2 Difficulties related to identity

Implementing a lazy library such as described in the preceding sections involves solving various problems, all of which there is not enough space to detail here. Let us only mention those related to dealing with identical definitions, which actually greatly hampered the very first implementation of *LEA*, before the cause of the problem could actually be identified and gotten rid of in a generic way.

Suppose a program uses a function `Slope` whose result is a lazy number representing the slope of the segment it is given as argument. If the program has tests like `if (Slope(s) = Slope(s'))` and if `s` and `s'` are two pointers on the same entity, a naive implementation of laziness will have to

evaluate the expressions for `Slope(s)` and `Slope(s')` before proving they are numerically equal. However, these expressions are necessarily equal in this specific case!

To prevent such useless evaluations, two solutions may be considered: the user program test should rather be written as `if (s = s' || Slope(s) = Slope(s'))`, or the library should detect such equalities without letting programmers worry about them. The latter solution has been chosen in *LEA*, simply because one global objective of such a library is to be easy to use, and to keep computer arithmetic issues at the lowest possible level.

In theory, identical expressions may be detected at two specific stages:

During comparisons: In its current version, the lazy library only detects identical expressions at this stage. The method used is a straightforward recursive "clone" checking procedure, which stops as soon as the subexpressions it is testing have the same addresses, disjoint intervals, or different labels. These tests greatly speed up the detection of "distinctness" when the library tries to compare two expressions the dags of which only differ at the leaves (*e.g.* two determinants). In practice, testing whether two expressions are identical only takes time proportional to their actual length when the expressions are indeed "clones" and do not share any common subexpression (think of comparing two $n\times n$ determinants with the same numerical coefficients, each represented by two different lazy numbers).

During creation: Identical expressions could also be detected whenever a copy of an already existing expression[1] is being created by a program. In this approach, any lazy expression can only be represented once in memory, and two identical expressions necessarily occupy the same memory location. To do this, the library needs to silently maintain a hash table containing all lazy numbers that have been created. The main drawback of this method is that it is not really *lazy* in its principle, since it may never be useful for a program to know that one particular couple of lazy numbers are clones, if they happen never to be compared in the sequel!

The final solution? A third, and better solution would be to wait for comparisons to detect equality, and then to use some sort of equivalence class scheme for equality, for instance the so-called *union-find* technique ([40]). One advantage is that it would be possible to deduce $a=c$ from knowing $a=b$ and $b=c$. It is also important to note that this scheme would be equally interesting whatever the method used to detect equality (clone detection, algebraic identity, etc.)

One final problem is that once a lazy number has been evaluated, it becomes impossible to compare it to one of its clones, because the two numbers now have different definitions: The "clone" still has the original symbolic definition which the other has lost in favor of a new, compact, irreducible fraction form. It would then suffice to use the last solution and to add an extra (initially void) field to store evaluated definitions, once the associated numbers have actually been evaluated. This would readily allow to keep the original symbolic definition untouched, and to make use of the clone detection procedure if needed.

[1] More explicitly, of a cell-expression, itself source of a lazy expression dag.

6

A few possible extensions

The wide collection of algorithms in the literature of Computational Geometry gives rise to certain classes of problems that may be dealt with using the following list of extensions to the lazy paradigm as it has been described earlier. It is our intention to implement some of these extensions in the later development of *LEA*.

6.1 Lazy exponentiation

The lazy arithmetic product is enough to compute z^n when z is a lazy number, and n an integer, but a specialized operator is more convenient. One reason for this is that it is possible to associate with such an operator a specific interval computation procedure that is finer and faster than the straightforward multiplication of n intervals. For instance, while $[-3,2][-3,2] = [-6,9]$, using the general definition of interval product, one may prefer getting $[0,9]$, using basic properties of the squaring function. The evaluation in rational form of z^n may also benefit from the classical "russian algorithm" using $O(\log n)$ multiplications or rational squaring operations. This new operator also saves memory space.

6.2 Lazy determinant

Lazy arithmetic must be paid for by allocating one memory cell for each single operation: Thus, the symbolic definition of the determinant of an $n \times n$ matrix, as computed using Gauss's algorithm, requires $O(n^3)$ such cells, since this method performs $O(n^3)$ operations. It is possible to consider a lazy determinant that would only store a matrix (or the reference of a matrix) and evaluate the determinant interval without explicitly developing the rather cumbersome symbolic definition.

6.3 Lazy booleans

When confronted with a test such as

```
if ((a < b) || (c < d)),
```

LEA – in its present state – may have to perform a rational evaluation of the first condition before even testing whether the second is not true using interval arithmetic only. Hence the need for *lazy boolean operators*, that could be defined in the following way (with the convention that $a_{[]}, !a_{[]}, a_{\mathbb{Q}}$ respectively mean "true according to intervals", "false according to intervals", and "true according to exact arithmetic"):

Lazy a **OR** b **::**
: if ($a_{[]}$ or $b_{[]}$) return true;

: if ($!a_{[]}$ and $!b_{[]}$) return false;
: if ($!a_{[]}$) return $b_{\mathbb{Q}}$;
: if ($!b_{[]}$) return $a_{\mathbb{Q}}$;
: if ($a_{\mathbb{Q}}$) return true;
: return $b_{\mathbb{Q}}$;

Note that "true according to intervals" means that the corresponding condition is tested using the interval arithmetic only, and that a negative answer to such a question is: The condition is either false according to intervals, or nothing can be said.

Unfortunately, rare are the languages that allow such redefinitions[1]. *C++*, the language used to implement *LEA*, does not allow the overloading of boolean operators. We may have to either switch to other programming environments, or else design specific functions, but this is quite contrary to the general philosophy of transparence we have opted for in the implementation of the library.

6.4 Lazy extrema

Whenever the lazy numbers z and z' have overlapping intervals $[\lambda_z, \mu_z]$ and $[\lambda_{z'}, \mu_{z'}]$, the lazy *maximum* operator should defer the evaluation of the two numbers by returning a symbolic definition of a new type, together with the interval $[max(\lambda_z, \lambda_{z'}), max(\mu_z, \mu_{z'})]$. The lazy *minimum operator* may be defined in a similar fashion.

These operators are unfortunately not compatible with the immediate computation of labels. However, using them would be rather interesting: Since one interval is known for the result *a priori*, it is possible to stop the evaluation of candidate z or z' as soon as its value is outside the interval. This optimization is similar to *$\alpha\beta$-pruning* in the *Minimax method* ([42]). (Refer also to [18] for more detail.)

6.5 Lazy absolute value

Whenever a lazy number z has an associated interval $[\lambda_z, \mu_z]$ containing 0, the lazy *absolute value* operator should defer the evaluation of z by returning a symbolic definition of a special kind, together with the interval $[0, max(|\lambda_z|, |\mu_z|)]$. The evaluation method is straightforward. Unfortunately, such an operator is not compatible with the immediate computation of the label for $|z|$.

6.6 Hardware implementation

Classical arithmetics over unbounded integers have already been successfully hardware implemented. Whether such a thing is possible (let alone interesting) for lazy arithmetics is still not known, although there is no apparent reason why it should not be. It remains to find out if using a rational arithmetic core is not necessarily too cumbersome, and if adapting the paradigm to a library driven by some inherently on-line scheme (for instance continued fractions ([22]), [23], [24]) would not be more beneficial.

6.7 Intervals as input data

Most of the time, data are realistic approximations of reality, with a known bound on errors. Data are often given as unique mean values in such intervals, but some applications may output actual

[1] *XSC* being one exception.

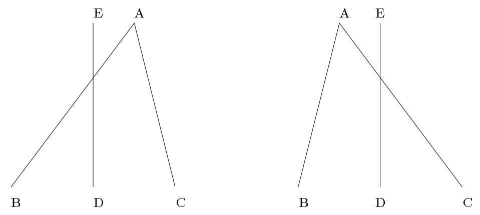

Figure 6.1: *Fuzzy topology and intervals. "Fuzzy" point A may be located at the left or at the right of the vertical line segment $[D, E]$ (that we have supposed fixed for simplicity), which induces fundamentally different topologies.*

intervals for their results. Is it possible to consider using lazy arithmetic packages when data are represented by intervals only?

In Computational Geometry, a few similar situations have been studied by authors like M. Segal and C.H. Séquin ([39]), Z. Li and V. Milenkovic ([27]). More recently, C. Barber ([1]) studied this problem from a very theoretical point of view in the case of the construction of 3-D convex hulls for sets of "fuzzy" points. He suggests very involved methods for defining and constructing such objects consistently, noting himself that the applications are necessarily slowed down by such constraints.

The main difficulty in dealing with interval inputs is that no simple topology may be defined, as illustrated in the simple example on Figure 6.1. Consequently, only complex topologies may be used, and only a few problems have beeen treated with that in mind, so far. Whether the lazy paradigm is of any interest in such approaches is an open question, still.

In other fields, where combinatorial properties are not exploited, the lazy paradigm may be extremely beneficial: The example about the badly convergent sequence, should have suggested the capability of lazy arithmetics to detect when intervals should be refined. This again is an open research area.

7

Limitations of the lazy solution

7.1 Programming environments

Because they are intended to be "user-friendly", lazy arithmetic packages require programming environments that allow to consider lazy numbers as standard numbers, *i.e.* floating-point numbers, or machine integers. The only environments with the necessary functionalities are *Lisp*, functional or object-oriented languages. *C++* is a limit case: While it allows to redefine arithmetic operators (but not boolean operators, as already mentioned), it does not allow garbage collection, nor does it allow to consider the basic numeric types (float or integer) as forming a class – a serious handicap.

According to us, such drawbacks are more related to the limitations of programming environments than to an inherent misconception in the lazy arithmetic principle.

7.2 Growth of numbers

During the tests that were performed at EMSE ([4], [18]), the initial size of the data had next to no influence on the running times of the applications running with *LEA*, as shown in Table 4.1. However, these applications were nowhere near geometric editors or modellers, for which it is known that incremental modifications on data (affine transformations, intersections, creation of new geometric objects from old ones) induce a considerable increase in the size of the numbers involved ([44]). This question calls for more experimentations.

7.3 Interface with the floating-point world

Currently, the constant use of an exact arithmetic is not considered to be realistic. Exact arithmetics are only used temporarily, during the execution of algorithms that do not withstand numerical imprecision: Most of the time, floating-point data are used. This approach has the merit to keep the size of numbers reasonably low (7.2), but raises the question of communicating with the "outside world", where only approximations are known.

It is of course necessary to provide the conversion of floating-point inputs into rationals, and *vice versa* for the output. Converting rationals into floating-point numbers can only be done approximately, in most cases. Conversely, converting floating-points into rationals may be done exactly. However, such a precision is useless, since floating point numbers are never more than reasonable approximations. One possible solution is to use continued fractions to perform conversions with a specified precision (refer to [2]).

Numerical conversions are not all there is to worry about: One must also preserve information consistency. One first minor problem is that of data redundancy. Consider a set of segments; the (x, y) co-ordinates of their endpoints cannot be converted independently from the coefficients (α, β, γ) for the lines supporting them without introducing inconsistencies. A trivial solution is to convert the endpoint co-ordinates, and then to deduce the $(\alpha', \beta', \gamma')$ triples associated with the lines through these converted endpoints, without trying to relate them to the triples associated with the unconverted segments. More generally, one must prevent any sort of such redundancy in the data.

The second problem is more difficult, and is related to preserving the topological consistency of the geometric objects. For instance, one should be careful not to introduce unwanted self-intersections when converting a "rational polyhedron" into its floating-point version ([2]). Unfortunately, this problem has been proved to be NP-hard, even in the two-dimensional case ([31]). However, S. Fortune has recently given a new and simple framework to avoid such problems ([9]).

7.4 The need for warned programming

We have already seen that floating-point arithmetic calls for some sort of warned programming to avoid the simplest and most frequent inconsistencies. Unfortulately, the lazy exact paradigm does not eradicate this need completely: When compiled with a lazy library, valid programs never fail, but may become extremely slow.

One reason for this is that a naive programming style will too often use non identical but algebraically equivalent expressions, such as $ab + ac$ and $a(b + c)$, or to mention more realistic examples, expressions such as those from Examples 1 or 4. The lazy library then has no solution other than to evaluate all these definitions in exact form before finding out that they are equal.

Three approaches to solve the problems related to algebraically equivalence may be considered: One is to use warned programming; we have already seen in 5.2 that another approach is to "teach" the lazy library how to handle equality between numbers; however, this is not really a solution for the problem at hand. The last approach, discussed in the next subsection, is to detect them in the compilation phase or even in a pre-processing phase.

To apply the first solution (warned programming style), it is best to first develop algorithms using floating-point numbers only: When the programs under development are robust enough, the simplest forms of inconsistencies have surely been seen to, and only the more sophisticated forms remain to be tackled. It is then possible to use the lazy rational arithmetic to solve these complex problems – and only those – with the help of exact computations. It is important to emphasize that in the lazy context, algorithmic issues are the ones that matter most: programmers are not asked to outsmart the idiosyncrasies of limited precision using wizards' tricks, but, on the contrary, to *not* worry about precision issues and to concentrate on the understanding of their algorithms, in terms of efficiency in a lazy programming environment. Whenever one has to write a test, it should be of the utmost importance to guess whether it is likely or not to induce unduly evaluations, even in the most trivial cases.

There is in fact a very simple rule of thumb to help programmers design efficient algorithms in a lazy setting: It is always possible to imagine very simple sets of data for a given geometric problem for which *absolutely* no evaluation should be performed in the lazy version. One good example of this is given by choosing a few random segments (preferably non vertical) to feed the algorithm supposed to detect their intersections. If the floating-point version works on such data sets, then the lazy version should be run (*i.e.* the same program should be linked with the lazy library, see 4.5), and no evaluation should be made – which may be easily checked using some specific information available from the lazy library itself. In the case where unduly evaluations are found, then something should be said about the way the program is written: Most likely, something is being done which is close to comparing equivalent but algebraically distinct expressions!

Let us now reconsider the problem described in Example 3. Obviously, it makes much more sense to impose that all the abscissæ of the intersections between a single vertical line ($x = x_0$) and a collection of non vertical segments be explicitly expressed as x_0, and not as $\frac{\Delta x_i}{\Delta_i}$. To be sure, the first version will never induce any rational evaluation, but nothing of that sort may be guaranteed for the second one!

One last word concerning warned programming: Consider a dynamic data structure for storing, say, random points in some structured order (*e.g.* a linked list, a binary tree, a priority queue,

an *AVL*, etc.). Insertion is usually straightforward, as random points will rarely come so close as to force the lazy library to evaluation. But deletion is a very much different matter: It requires searching the data structure until the element with the *same* key as the one given as parameter is identified. A naive programming of such an operation will invariably end up doing much more work than necessary (just imagine that the random points are now the results of complex operations). There may not be any evaluation performed, thanks to the clone detection procedure described earlier, but still, the library will have quite a hard time identifying each and every item to be deleted, from its key. It is a good idea to design a scheme to prevent the library from doing any extra work at all, besides searching the structure for the correct location. In general, it is most beneficial for programs to have their data structures slightly augmented with pointer information. Such modifications will never slow down the algorithms noticeably, while they will greatly speed up the performance of the lazy versions, for almost no extra programming work at all.

On the other hand, the major open question is: Is it possible to augment the data structures in the lazy library to prevent most of the evaluations that would be induced by naive programming? Of course, the limitation of such an entreprise is the ratio of extra work to put into it to the decrease of work performed by the library. In other words, how much work is one prepared to do, in order to be as lazy as possible!...

7.5 Automatic treatment of algebraic equivalences

Unfortunately, warned programming is slightly artificial, sometimes under-estimated, and not to be found in the books. The next question is, therefore: Is automatic detection of algebraic equivalences possible?

The current implementation of *LEA* only detects the most trivial algebraic equivalences:

$$-(-x) \equiv x, \frac{1}{\frac{1}{x}} \equiv x, x \cdot \frac{1}{x} \equiv 1, x + (-x) \equiv 0$$

and so forth. One may consider ordering operands for the sum and the product, so that the operands with larger label always lie to the left of all binary operator nodes, for instance. Thus $ab - ba$ would be found to be identically null without any rational evaluation. However, this scheme is not sufficient to detect the equivalence between such simple expressions as $(a + b)c$ and $ac + bc$.

Detecting algebraic equivalences like the ones above constitutes the *ABC* of symbolic calculus, but in the general case, the known methods are much more prohibitive than the simple identity test between two expressions we have described earlier: Symbolic computations may themselves involve computations on large integers or rationals (expansion of $(x + y)^n$, for instance). In short, a fully *deterministic* detection of algebraic identities during execution seems unrealistic. On the other hand, another approach is possible, by means of a fast, yet *probabilistic*, algorithm to detect algebraic identities ([38]), whose spirit is very close to the principle of hash coding. The authors did not test either approach.

Is detection during compilation or during a pre-processing phase possible? This remains to be seen. Let us simply point out that the whole problem of algebraic equivalences is not raised by the lazy paradigm only; it is already present in the conversion of algorithms to floating-point programs, and we have seen the number of tricks warned programmers have developed in order to prevent inconsistencies.

Even in the now classical framework of floating-point arithmetic, no general (*i.e.* independent of the programs) solution is known to solve the problems related to algebraic equivalences. To our knowledge, Computer Science has never fully addressed this phenomenon. It is our hope that this paper will have stated the question in a proper fashion, and that it will be recognized as worth considering and studying, in the future.

7.6 Discontinuous operators

Standard arithmetics over unbounded integers provide operators such as *modulo*, *gcd*, or *lcm*. These operators are fundamentally discrete and discontinuous, and are thus incompatible with interval arithmetic. The lazy rational arithmetic, as we have defined it, cannot therefore be applied to problems that require such operators – namely cryptography, calculus in finite fields, primality tests, or factoring large integers. Finding out whether other types of lazy arithmetic may allow to break this limit is an open problem.

7.7 Algebraic arithmetic

Finally, the lazy arithmetic package presented in this paper is of rational essence, while many problems that arise in Computational Geometry or Computer Aided Design are of algebraic nature. For instance, rotations with angle $k\pi, k \in \mathbb{Q}$, and intersections between algebraic curves or surfaces naturally involve algebraic numbers. The possibility to use a lazy algebraic arithmetic still is an open problem.

Remark. In general, when rotations are needed in applications, the coefficients of their matrices are approximated by rational values (see for example the paper by V. Milenkovic [30]). Another method is to perform the rotations usign finite-precision transcendental functions, and then to convert the finite-precision co-ordinates into rational ones, but this is likely to introduce topological inconsistencies in the objects. However, S. Fortune recently suggested a solution ([9]) to skirt this problem.

8

Conclusion

Numerical imprecision seriously impairs the implementation of geometric methods, forcing more and more authors to use mixed rational arithmetics in Computational Geometry ([44]).

The lazy rational arithmetic is a recent (1993) optimization of such arithmetics, which only performs necessary exact computations by deferring them until they either become unnecessary or unavoidable. Some variants and extensions of this arithmetic have been presented, which seem both promising and interesting.

The lazy paradigm leaves several problems open:

- The current implementation of the lazy library still calls for a careful, warned programming style, just as the standard floating-point arithmetic does. Such a problem is induced by algebraic equivalences. This paper has described the problem, shown its stakes, but fails to give a complete solution, so far.
- What about the use of the lazy paradigm with interval inputs in fields such as Numerical Analysis?
- Last, but not least: Is it possible to develop an algebraic arithmetic that would be practical in Computational Geometry and Computer Aided Design, as well as in Symbolic Calculus?

Acknowledgements

The authors wish to thank M.O. Benouamer and P. Jaillon from ENSM.SE for a first implementation of *LEA*, J-M. Muller (ENS Lyons (France)) and all the colleagues in his workgroup on Computers Arithmetic for their help and encouragements, and the anonymous referees for their comments and for pointing out various improvements to bring to the first draft of this paper.

Bibliography

[1] C. Barber. *Computational Geometry with Imprecise data and arithmetic.* PhD thesis, Princeton University, October 1992.

[2] M.O. Benouamer. *Operations booléennes sur les polyèdres représentés par leurs frontières et imprécisions numériques.* PhD thesis, École des Mines de Saint Étienne, 1993.

[3] M.O. Benouamer, P. Jaillon, D. Michelucci, and J-M. Moreau. A lazy solution to imprecision in computational geometry. In *Proceedings of the 5th Canadian Conference on Computational Geometry*, pages 73–78, Waterloo, Canada, August 5-9, 1993.

[4] 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*, Windsor, Ontario, June 30-July 2, 1993.

[5] M.O. Benouamer, P. Jaillon, D. Michelucci, and J-M. Moreau. Hashing lazy numbers. In *Proceedings of SCAN-93*, Vienna, Austria, September 1993.

[6] M.O. Benouamer, D. Michelucci, and B. Péroche. Boundary evaluation using a lazy rational arithmetic. In *Proceedings of the 2nd ACM/IEEE Symposium on Solid Modeling and Applications*, pages 115–126, Montréal, Canada, 1993.

[7] J.L. Bentley and T. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comp.*, C-28(9):643–647, 09 1979.

[8] J. D. Chang and V. Milenkovic. An experiment using LN for exact geometric computations. In *Proceedings of the 4th ACM Canadian Conference on Computational Geometry*, pages 67–72, August 1993.

[9] S. Fortune. Polyhedral modelling with exact arithmetic. In C. Hoffmann and J. Rossignac, editors, *Proceedings of the Third Symposium on Solid Modeling and Applications*, pages 225–233. ACM Press, May 1995.

[10] 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.

[11] 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.

[12] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Survey*, 23(1):5–48, March 1991.

[13] L. Guibas, D. Salesin, and J. Stolfi. Epsilon geometry: Building robust algorithms from imprecise computations. In *Proceedings of the 5th ACM Symposium on Computational Geometry*, pages 208–217, 1989.

[14] L.J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphic*, 4:74–123, 1985.

[15] C. Hoffman, J. Hopcroft, and M. Karasick. Robust set operations on polyhedral solids. *IEEE Comp. Graphics and Appli.*, 9(6):50–59, Nov. 1989.

[16] J.F. Hughes, D.H. Laidlaw, and J.F. Trumbore. Constructive solid geometry for polyhedral objects. In *Proceedings of SIGGRAPH86*, volume 20, no 4, pages 161–180, ACM Computer Graphics, Aug. 1986.

[17] M. Iri and K. Sugihara. Construction of the Voronoi diagram for one million generators in single-precision arithmetic. In *Abstracts of the 1st Canadian Conference on Computational Geometry*, Montréal, 1989. Also: *Proc. IEEE*, 80(9):1471–1484, 1992.

[18] P. Jaillon. *Proposition d'une arithmétique rationnelle paresseuse et d'un outil d'aide à la saisie d'objets en synthèse d'images.* PhD thesis, ENSM.SE, September 1993.

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

[20] R. Klatte, U. Kulisch, C. Lawo, M. Rausch, and A. Wiethoff. *C-XSC, A C++ Class Library for Extended Scientific Computing.* Springer-Verlag, Berlin/Heidelberg/New York, 1993.

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

[22] P. Kornerup and D.W. Matula. Foundations of finite precision rational arithmetic. *Computing Suppl.*, 2:85–111, 1980.

[23] P. Kornerup and D.W. Matula. Finite precision rational arithmetic: An arithmetic unit. *IEEE Transactions on Computers*, C-32(4):378–388, April 1983.

[24] P. Kornerup and D.W. Matula. An algorithm for redundant binary bit-pipelined rational arithmetic. *IEEE Transactions on Computers*, 39(8):1106–1115, August 1990.

[25] U.W. Kulisch and W.L. Miranker. *Computer Arithmetic in Theory and Practice.* Academic Press, New York, 1981.

[26] U.W. Kulisch and W.L. Miranker. *A New Approach to Scientific Computation.* Academic Press, New York, 1982.

[27] D. Michelucci. *Les représentations par les frontières : quelques constructions; difficultés rencontrées.* PhD thesis, ENSM.SE, 1987.

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

[29] V.J. Milenkovic and Z. Li. Constructing strongly convex hulls using exact or rounded arithmetic. In *ACM-SIAM Symposium on Discrete Algorithms*, 1989.

[30] V.J. Milenkovic and V. Milenkovic. Rational orthogonal approximations to orthogonal matrices. In *Proceedings of the 5th Canadian Conference on Computational Geometry*, pages 485–490, Waterloo, Ontario, 1993.

[31] 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.

[32] R.E. Moore. *Interval Analysis.* Prentice Hall, Englewood Cliffs, N.J., 1966.

[33] J-M. Moreau. *Facétisation et hiérarchisation de la représentation par segments d'un graphe planaire dans le cadre d'une arithmétique mixte.* PhD thesis, ENSM.SE, 1990.

[34] J-M.. Muller. *Arithmétique des ordinateurs, opérateurs et fonctions élémentaires.* Masson, Paris, Milan, Barcelone, Mexico, 1989.

[35] 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.

[36] G. Ottmann, G. Thiemt, and Ullrich C. *Numerical Stability of Simple Geometric Algorithms in the Plane.* in Borger, E. (Ed.): Computation Theory and Logic, Springer-Verlag, Berlin, 1987.

[37] F.P. Preparata and M.I. Shamos. *Computational Geometry – An Introduction.* Springer-Verlag, New York, N.Y., 1985.

[38] J.T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27:701–717, 1980.

[39] M. Segal and C.H. Séquin. Consistent calculations for solids modeling. In *Proceedings of the 1st ACM Symposium on Computational Geometry*, pages 29–38, 1985.

[40] R.E. Tarjan. *Data Structures and Network Algorithms.* SIAM, Philadelphia, Pa., 1983.

[41] J. Vignes. Estimation de la précision des résultats de logiciels numériques. *La Vie des Sciences*, Tome 7(2):93–115, 1990.

[42] P.H. Winston. *Artificial Intelligence.* Addison-Wesley, Reading, Mass., 1984.

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

[44] C.K. Yap. Towards exact geometric computation. In *Proceedings of the 5th Canadian Conference on Computational Geometry*, pages 405–419, Watreloo, Ontario, Aug. 5-9, 1993.