Hashing Lazy Numbers

M.O. Benouamer, P. Jaillon, D. Michelucci, J-M. Moreau
Département Infa, E.M.S.E., 158, Cours Fauriel
F 42023 Saint-Étienne, Cedex 02
e-mail: author@emse.fr

Abstract

This paper describes an extension of the "lazy" rational arithmetic (LEA) presented in [1]. A lazy arithmetic *is an optimized version of the usual exact arithmetics used in Symbolic Calculus, in Computational Geometry or in many other fields. We present a method originating from modular arithmetic to compute efficient hash values for lazy numbers. Hashing is frequently used in geometric algorithms for fast searching purposes.*

1 Introduction

Finite precision often introduces inconsistencies in the results of otherwise correct algorithms. At the other end of the spectrum, exact arithmetics guarantee error-free computations, but they require considerable amounts of time and memory resources, and hence have a rather restricted domain of application.

An optimized rational arithmetic library is presented in [1]; it is based on the so-called "lazy evaluation" paradigm, in that it always delays exact computations until they are inevitable.

A *lazy number* is a rational represented as a record with two fields: One for an approximation and another for a symbolic definition of the number (to be detailed later). In most cases, the approximations carry enough information to operate "consistently" on numbers. When this is no more true, the library decides to perform exact evaluations to either "refresh" approximations or to obtain the exact expression of the result or test that caused trouble.

The lazy arithmetic module is implemented as an independent library which may be used by all sorts of application programs. Because "laziness" is inherent to the library, it is not necessary to know – when a computation is called for – whether it will involve exact operations or whether finite precision will suffice.

To give but one example of application, solving a linear system involves giving a floating point value to each unknown. All known pivot-based solvers produce well-behaved answers in most cases, but are extremely sensitive to matrix conditioning.

Using finite-precision all along would be extremely hazardous (what with the risk of never finding out that the results are erroneous). Using exact arithmetic only might prove irrelevant 9 times out of 10. A solution based on lazy arithmetic will only perform the necessary operations, using either finite *and/or* infinite precision, according to each case.

The major goal of lazy exact arithmetics is to leave the decision of switching from finite-precision to exact calculus (and then possibly back to finite-precision) to the library itself. Lazy schemes, such as the one described here, statistically save a large amount of unnecessary evaluations, and thus guarantee consistent computations (which is not true of finite-precision methods) at a much lower cost than purely exact solutions, in most situations.

In this paper, we shall focus on hash-coding methods ([4]) for lazy numbers. There are basically two motivations for introducing such a notion:

1. Geometric algorithms often require to retrieve objects from their coordinates, a process that is made possible by appropriate hash code techniques.
2. The lazy library itself may use such a scheme to discriminate numbers with close approximations, in which case finite-precision cannot be of any help: Since different hash keys necessarily designate different numbers, a certain amount of evaluations – although not all of them – may be avoided.

We shall see how modular arithmetic allows to define efficient hash keys from the symbolic definition of lazy numbers. In Section 2, we briefly review how lazy numbers are represented and manipulated (see [3] for a detailed account). Sections 3 and 4 present two methods to compute hash keys for lazy numbers. Performance issues, future research and open problems are presented in Section 5.

2 Laziness fundamentals

Any rational quantity r with exact value ϱ may be represented as a *lazy number* by a two-field record:

1. an approximation – in our setting, a floating-point interval that contains ϱ – and
2. a symbolic definition for r, specifying a method to retrieve its exact value ϱ, when desired. More precisely, it is either a rational (evaluated) quantity, or a symbolic (unevaluated) expression representing the sum, the product, the opposite (additive inverse) or reciprocal (multiplicative inverse) of other lazy numbers.

Each time an elementary arithmetic operation $(+, *, inv_+, inv_*)$ is requested, the library computes a consistent interval for the result, and then creates a new symbolic definition record for it.

In most situations, intervals will convey sufficient information to allow consistent computations[1]. Let us illustrate this with a simple example. Suppose we wish to compare two lazy numbers a and b. If their bounding intervals I_a and I_b are disjoint, it suffices to compare the floating point bounds of I_a and I_b to infer the relative positions of a and b. Conversely, if $I_a \cap I_b \neq \emptyset$, the intervals for the two numbers have grown too large, and only an exact (rational) comparison will settle the matter. Let us now investigate the basic tools required by the lazy library.

2.1 Intervals

Interval techniques ([4], [5], [7]) make it possible to derive the interval for the result of an arithmetic operation directly from those of the operand(s). For instance, the interval for the sum of two lazy numbers x and y with bounding intervals $I_x = [a_x, b_x]$ and $I_y = [a_y, b_y]$ is, in general (cf. [3]), given by:

$$[\bigtriangledown(a_x \oplus a_y), \bigtriangleup(b_x \oplus b_y)]$$

where $\bigtriangledown(\chi)$ (resp. $\bigtriangleup(\chi)$) is the *machine* number immediately below (resp. above) number χ, and $\oplus$ denotes the addition on machine numbers.

2.2 Dags

On the other hand, the resulting symbolic definition for the sum of x and y is obtained by creating a symbolic node (to hold the $+$ operator), with two pointers to the operands.

[1]To clarify this, suppose ρ and σ are the machine representations of two *real* numbers, r and s. These numbers may result from any sequence of rational computations. ρ and σ are said to be *consistent* results *iff* they lie in the same order as r and s. Clearly, no consistent statement may be made when $\mid \rho - \sigma \mid$ is smaller than machine precision!

In general, the underlying data structure is a tree-like representation in which internal nodes are "unevaluated" numbers – i.e. unary or binary operators[2] – and terminal nodes ("leaves") are evaluated numbers – i.e. with known rational values. This structure is a directed acyclic graph (*dag*, for short) rather than a tree, since any lazy number may be referred to by more than one node.

Thus, the *generation* (interval, definition) of any *elementary* lazy arithmetic operation takes *constant* time and space, and requires *no* evaluation.

2.3 Evaluation

Indeed, the exact rational value of a given lazy number may even *never* be computed, depending on the subsequent operations it will be subjected to. The only occasions when a lazy number should be evaluated are:

1. Each time the sign of a number whose interval contains zero is requested, or more generally, each time two numbers with overlapping intervals are to be compared.
2. Each time the reciprocal of a number whose interval contains zero is required.
3. Each time an "ancestor" of (i.e. containing in its own definition dag) the lazy number in question must itself be evaluated.

Evaluation is based on a simple recursive mechanism. Different strategies and heuristics are liable to reduce the computational cost of this operation. See [3] for a discussion.

2.4 Putting it all together

The lazy exact arithmetic library (*LEA*) is written in *C++*, mainly because this language allows classes and operator overloading. Consider any *C* program using `Number` as a synonym for `float`, for instance "`typedef float Number;`". Schematically, it now suffices to replace this definition with "`typedef LazyNumber Number;`" to compile the program and link it with the library. As a consequence, all standard floating point operators and syntax constructs are directly available, and the mechanisms of the lazy library are completely transparent to the user.

To illustrate this in a typical situation from Computational Geometry, suppose $A_{(x,y)}$, $B_{(x,y)}$, and $C_{(x,y)}$

[2](inv_+, inv_*) and $(+, *)$, respectively. There is no restriction on the *arity* of the operators: It is quite legitimate to want to provide for built-in, specialized functions to compute such things as the determinant of a 3×3 matrix, and so forth... The larger the number of basic operations available, the more versatile the library.

are three points in the euclidian plane. The triple (A, B, C) is said to form a right turn (left turn) if the measure of the angular sector $(\vec{BA}, \vec{BC})$ around B is smaller (greater) than π. Of course, A, B, C are aligned *iff* the angle is null.

Now consider the following function to discriminate all three cases:

```
int LeftRight? (LazyNumber Ax, Ay, Bx, By, Cx, Cy)
{ LazyNumber Δ;
  Δ = (Bx − Ax) * (Cy − By) − (Cx − Bx) * (By − Ay);
  if (Δ > 0.0)
        return LeftTurn;
  if (Δ < 0.0)
        return RightTurn;
  return Aligned;
}
```

Note that (cf. [8], p. 43):

$$\Delta \equiv \begin{vmatrix} 1 & 1 & 1 \\ A_x & B_x & C_x \\ A_y & B_y & C_y \end{vmatrix}$$

represents twice the signed area of triangle $\triangle_{ABC}$, and thus gives the sign of the cross product $\vec{AB} \times \vec{BC}$.

Each time the function is called, an interval I_Δ is computed for the local lazy variable, and a definition is constructed for it, as explained above. Depending on the "real values" of the lazy parameters, 0.0 will sometimes lie outside I_Δ – in which case finite precision is sufficient to determine Δ's sign –, and will sometimes lie inside this interval – in which case exact evaluation is in order. However, all these situations are potentially encompassed in the unique *standard C* expression of `LeftRight?` sketched above!

All initial (raw) data are assigned an interval containing them. It may happen that two initial values are so close that their bounding intervals overlap; in a way, it is as if finite-precision was not sufficient from the start, but this happens extremely rarely, and makes no difference for the library.

Interval amplitudes grow with operations. All is fine until the library hits, say, a comparison test between two lazy quantities, the intervals of which overlap: Obviously, all computations made prior to the test were consistent, but now the wind is turning. The library's first action is to try and contract intervals in the hope of disconnecting them. This may be done with the help of partial evaluations in the dags (from the "leaves" upwards). If the (refreshed) intervals are disjoint after this, the "interval" comparison may be safely carried out; otherwise, the rational expressions of both quantities must be compared using straightforward rational arithmetic methods.

3 Computing hash keys

Frequently, geometric algorithms use hash-tables to speed up searching over elements ([6]) such as points, lines, or planes ([2]). Each element is assigned a hash key derived from the numerical data (i.e. coordinates, line or plane coefficients, etc.) that define it. In this context, the user of a lazy arithmetic library will be confronted with the problem of computing hash keys for lazy numbers, the exact values of which are not necessarily available. The interested reader is referred to [4] for a thorough treatment of hash methods and their performance, the distribution of hash keys, and the techniques for solving collision problems, as these topics are not relevant here. Our goal simply is to exhibit an efficient way to produce keys in the presence of lazy numbers, on the basis of well-known results in hash-coding theory.

Notations and preliminaries

Let $\mathbb{Z}$ denote the set of all integers and $p \in \mathbb{Z}_+^*$ be a (large) positive *prime*. In this section, we shall see how each lazy number z may be assigned a hash key $\Psi(z) \in [0, p[$ such that $\Psi(z) \neq \Psi(z') \Rightarrow z \neq z'$, for any two lazy numbers z and z'.

The binary relation defined in $\mathbb{Z}$ by $x \equiv y\,[p]$ (i.e. $\exists\, k \in \mathbb{Z}$ such that $x - y = k * p$) induces the *field* $\mathbb{Z}_p = \mathbb{Z}/p\mathbb{Z}$. From now on, we shall denote by $x \,\%\, p$ the unique integer in $[0, p[$ defined as the remainder in the *Euclidian division* of x by p. *GCD* will stand for *Greatest Common Divisor*.

3.1 Hash keys for evaluated lazy numbers

By definition, each integer $x \in \mathbb{Z}$ is assigned the hash key $\Psi(x) = x \,\%\, p$. Rational numbers will be represented in the canonical form $\frac{x}{y}$ where $x \in \mathbb{Z}, y \in \mathbb{Z}_+^*$, and $GCD(x, y) = 1$. The hash key $\Psi(\frac{x}{y})$ is defined as follows:

1. If $GCD(y, p) = 1$, $\exists!\, y^{-1} \in]0, p[$ (the reciprocal of y in $\mathbb{Z}_p$) such that $(y * y^{-1}) \,\%\, p = 1$.

   In this general case, we define

   $$\Psi(\frac{x}{y}) = (x * y^{-1}) \,\%\, p \equiv [(x \,\%\, p) * (y \,\%\, p)^{-1}] \,\%\, p.$$

   Note that the second form is used in practice, for the sake of efficiency, and that this definition also applies to a fraction with reducible numerator and denominator, provided their *GCD* is not a multiple of p. The computation of reciprocals in $\mathbb{Z}_p$ is detailed in 3.3.

2. If $GCD(y,p) \neq 1$ (i.e. if $y = k*p$ for some $k \in \mathbb{Z}$) then y admits no reciprocal in $\mathbb{Z}_p$. However, we may define $0^{-1} = \Omega$ and $\Omega^{-1} = 0$, where Ω is any number outside $[0,p[$. (For convenience, we shall choose $\Omega = p$.)

   Whenever $GCD(y,p) \neq 1$, let $\Psi(\frac{x}{y}) = \Omega$. Note that the larger the value of p, the smaller the probability ($\frac{1}{p}$) of such an event to occur.

3.2 Hash keys for unevaluated lazy numbers

Recall that a lazy number z may be either "evaluated" (i.e. the rational value of z is available), or "unevaluated". If the former case, $\Psi(z)$ may be computed from the rational value, as shown in 3.1. In the latter case, z is represented by a symbolic expression like "$a+b$", "$a*b$", "$inv_+(a)$", or "$inv_*(a)$", where a and b are references to lazy numbers.

3.2.1 General rules

In the general case, $\Psi(z)$ is computed by recursively applying the following well-known properties of $\mathbb{Z}_p$ (see exceptions in 3.2.2):

$$\begin{aligned}
\Psi(z+z') &= (\Psi(z)+\Psi(z'))\,\%\,p \\
\Psi(z*z') &= (\Psi(z)*\Psi(z'))\,\%\,p \\
\Psi(inv_+(z)) &= (-\Psi(z))\,\%\,p \equiv p-\Psi(z) \\
\Psi(inv_*(z)) &= [\Psi(z)]^{-1}\,\%\,p, \Psi(inv_*(0)) = \Omega.
\end{aligned}$$

This scheme yields hash keys for unevaluated lazy numbers without computing their exact rational values. Moreover, it always returns identical hash keys for lazy numbers the evaluation of which would result in the same rational quantity (e.g. $\frac{4}{8}+\frac{11}{3}$ and $\frac{5}{3}*\frac{5}{2}$).

3.2.2 Special cases

However, it was implicitly assumed that neither $\Psi(z)$ nor $\Psi(z')$ is equal to Ω. If this is no longer true, we may still give consistent and easily justified rules for computing hash keys using the key(s) of the operand(s) in the following cases:

$$\begin{array}{ll}
\Omega*\Omega = \Omega & \\
\Psi*\Omega = \Omega*\Psi = \Omega, & \forall\,\Psi \in [1,p[\\
\Psi+\Omega = \Omega+\Psi = \Omega, & \forall\,\Psi \in [0,p[\\
-\Omega = \Omega & \\
0^{-1} = \Omega \text{ and } \Omega^{-1} = 0. &
\end{array}$$

This only leaves two cases which lead to indeterminations and cannot be decided upon, as illustrated below:

1. $0*\Omega = \Omega*0 = ?$
   Consider the rational number $a*b$ where $a = \frac{p}{1}$, and $b = \frac{k}{p}$ for $k \in [1,p[$. Clearly,

   $$\Psi(a) = 0, \Psi(b) = \Omega, \;\; \Psi(a*b) \equiv \Psi(k) = k.$$

   Therefore, $\Psi(a*b)$ may take any value k in $[1,p[$. Moreover, choosing $a = p$ and $b = \frac{1}{p^2}$ yields $\Psi(a) = 0$, $\Psi(b) = \Omega$ and $\Psi(a*b) = \Psi(\frac{1}{p}) = \Omega$.

2. $\Omega+\Omega = ?$
   Consider the rational number $a+b$ where $a = \frac{1}{p}$, and $b = (k - \frac{1}{p})$ for $k \in [0,p[$. Clearly,

   $$\Psi(a) = \Psi(b) = \Omega, \;\; \Psi(a+b) \equiv \Psi(k) = k.$$

   Therefore, $\Psi(a+b)$ may take any value k in $[0,p[$. Moreover, choosing $a = \frac{1}{p}$ and $b = \frac{1}{p}$ yields $\Psi(a) = \Omega, \Psi(b) = \Omega$ and $\Psi(a+b) = \Psi(\frac{2}{p}) = \Omega$.

In both cases, a simple way to compute the key is to evaluate the whole definition tree for z, and to deduce $\Psi(z)$ from the resulting rational value as shown in 3.1. As these indeterminate cases are not very frequent, they have little influence on the overall performance of the library. But more about this in Section 4.

3.3 Elementary arithmetic in $\mathbb{Z}_p$

Quite naturally, the sum, the product and the opposite may be found in constant time in $\mathbb{Z}_p$, using straightforward properties of modular arithmetic.

Let us now detail the computation of u^{-1}, the reciprocal of any u such that $GCD(u,p) = 1$. This may be done by applying:

Fermat's theorem Given a prime number p, if u is any integer such that $GCD(p,u) = 1$ (i.e. not a multiple of p), then $u^{p-1}\,\%\,p = 1$.

As a consequence,

$$u^{-1} = u^{p-2}\,\%\,p,$$

which gives a first algorithm for computing reciprocals. Another solution is given by:

Bezout's theorem For any relatively prime numbers $u, v \in \mathbb{Z}, \exists\, x, y \in \mathbb{Z}$ such that $u*x+v*y = 1$.

Applying *Euclid's extended algorithm* to compute $GCD(u,v)$ with $v = p$ yields x and y such that $u*x+p*y = 1$. Therefore $(u*x)\,\%\,p = 1$, and

$$u^{-1} = x\,\%\,p$$

which suggests the second algorithm.

Both methods take $O(\log(p))$ time. In practice, it is more convenient to use a pre-computed table to store all multiplicative inverses in $\mathbb{Z}_p$. Next, since

$$\forall\, i \in\,]0, p[, (p-i)^{-1} \equiv (-i)^{-1} \equiv -(i^{-1}),$$

it is sufficient to store only $q = \frac{p-1}{2}$ reciprocals: $1^{-1}, 2^{-1}, \ldots, q^{-1}$.

However, since p is a fixed prime, this table may be computed independently once and for all – as a $O(p\log(p))$-time preprocessing – and included as data at compile time into the library. Thus, all elementary arithmetic operations in $\mathbb{Z}_p$ may be performed in constant time, provided $O(p)$ space for the table of reciprocals.

Choosing p The value of p is an important factor in the performance of this technique, for obvious reasons. On the one hand, p should be as large as possible, to reduce the frequency of indeterminations. But on the other hand, p should be small enough to prevent cumbersome overflow handling in $\mathbb{Z}_p$, and, *most important*, to limit the amount of memory required to store the table of reciprocals.

A reasonable choice, on 32-bit machines, is to set p to the largest prime less than 2^{16} (i.e. 65,521). This "only" consumes about 64Kb of memory resources.

4 A finer method

The main disadvantage of the former method is that the space occupied by the table is proportional to p. In order to rule this table out of the library altogether, let us now introduce a new hashing technique. The idea is to replace the keys with more manageable information, thereby eliminating the need for their evaluation.

Let z be any lazy number, $\Psi(z)$ be its hash-key – as defined earlier – and define couple $(\eta, \delta) \in \mathbb{Z}_p^2$ so that

$$\eta = (\Psi(z) * \delta) \,\%\, p, \text{ whenever } \delta \neq 0.$$

Thus, all couples $(k*\eta, k*\delta), k \neq 0$ represent the same key $\Psi(z)$ *modulo* p, and all couples $(k, 0), k \neq 0$ represent the "infinite key" Ω. Finally, define $(0,0)$ as the special couple for indeterminations.

4.1 General rules

A natural way to define the couple associated with an irreducible rational number $\frac{a}{b}$ is to choose $(a\,\%\,p, b\,\%\,p)$. Note that this definition also applies to a fraction with reducible numerator and denominator, provided their *GCD* is not a multiple of p.

If the couples assigned to lazy numbers z and z' are (η, δ) and (η', δ'), we may define those associated with $+$, $*$, inv_+, and inv_* as follows:

$(\eta, \delta) + (\eta', \delta') = ((\eta * \delta' + \eta' * \delta) \,\%\, p, (\delta * \delta') \,\%\, p).$
$(\eta, \delta) * (\eta', \delta') = ((\eta * \eta') \,\%\, p, (\delta * \delta') \,\%\, p).$
$inv_+(\eta, \delta) = (-\eta \,\%\, p, \delta) \equiv (p - \eta, \delta).$
$inv_*(\eta, \delta) = (\delta, \eta).$

All these operations are carried out in constant time and space, and division no more requires the computation of reciprocals, since it now consists in the simple inversion of couple components.

4.2 Indeterminations

Of course, indeterminate cases have not disappeared completely. They may indeed be seen to appear in just two cases, exactly as before:

$(\eta, 0) + (\eta', 0) = (0, 0).$
$(\eta, 0) * (0, \delta') = (0, 0).$

A simple way to solve indeterminations is yet again to evaluate the underlying rational number.

4.3 Discussion

The lazy library may use keys – or rather couples –, for its own proper needs. Suppose lazy numbers z and z' are assigned couples (η, δ) and (η', δ'), respectively, and we wish to compare keys $\Psi(z)$ and $\Psi(z')$, as they were defined in Section 3. There is no need to compute their actual values for this purpose, since

$$\Psi(z) = \Psi(z') \Leftrightarrow \eta * \delta' = \eta' * \delta.$$

In the general case, the key corresponding to a couple $(\eta, \delta), \delta \neq 0$, is $\Psi = (\eta * \delta^{-1}) \,\%\, p$, and the computation of δ^{-1} is required. As such requests are not likely to be frequent, it is possible to use one of the above mentioned $O(\log(p))$-time algorithms.

Of course, on such occasions, couple (η, δ) is replaced with the equivalent couple $(\eta * \delta^{-1}, 1)$. The only special case is the computation of the key from a couple $(\eta, 0), \eta \neq 0$, where the key is Ω.

This new solution is much simpler than the first one. In particular, there is no special treatment for the infinite key Ω. The computation of reciprocals in $\mathbb{Z}_p$ is no more needed, except when the key is explicitly requested by the user. Even in this case, the library may do without of table for reciprocals.

Choosing p Larger values for prime p may be used here, since overflow is the only concern! It is possible to use, for instance on 32-bit machines, the *Mersenne* prime

$$\mathcal{M} = 2^{31} - 1 = 2,147,483,647$$

as addition and multiplication overflows in $\mathbb{Z}_{\mathcal{M}}$ may be dealt with by standard techniques (cf. [4], p. 272 ff.). As a consequence, the probability of indeterminations ($\frac{1}{\mathcal{M}}$) becomes negligible.

5 Conclusion

Performance Roughly speaking, the "lazy-and-hash" version of an algorithm runs between 4 an 7 times slower than its floating point version (due to interval and dag updates), and may run up to 150 times faster than its exact version, depending on data pathology. It is worth noting that lazy versions only make about twice the floating point operations than finite-precision versions, but never one superfluous exact operation, which means none when a finite-precision version would yield consistent results!

Use Whenever precision and consistency are crucial issues, *laziness* is a strong and powerful paradigm, for which hashing techniques provide efficient complementary tools. As an example, the complete geometric algorithm presented in [2] makes intensive use of hash keys – as we have described them – for fast searching in lazy context. It is clear that either method presented in this paper might be appropriate in different situations. Favoring one or the other will depend, in practice, on whether the actual values of keys must be made available to the application programs or not.

Yet, application programs are not the only ones to benefit from hashing: LEA, the lazy arithmetic library itself, uses this technique (see [3] for a full description). As already pointed out, hashing is exploited to speed up discriminations between lazy numbers with overlapping intervals (obviously, $\Psi(z) \neq \Psi(z') \Rightarrow z \neq z'$). In the current version of the library, hash keys are computed each time a new lazy number is created. However, there is no particular difficulty in supplying a function Ψ to be called at request (i.e. for the user's own needs).

Open problems From a practical point of view, hashing lazy numbers widen the functionalities of the lazy arithmetic library, in that it allows to solve a certain number of problems related to our basic goal: Handling precision issues without interfering with algorithms.

Although lazy algebraic libraries are beyond the scope of this paper, let us just point out that the principle of hashing lazy numbers described above may be extended to the algebraic case.

For instance, suppose we work in a quadratic extension $Q(\sqrt{a})$. To hash such numbers, it suffices to use a finite field F_p where a is a quadratic residue *modulo* p, or else to use the quadratic extension of F_p.

Finally, one may ask if more efficient hashing methods could be found. One may even wonder if more powerful methods exist that would allow to detect equality between lazy numbers as they are created – be they defined by isomorphic expressions or not – via some specialized mechanism (possibly union-find techniques). To say the truth, detecting equality without evaluating remains a crucial problem.

Acknowledgements The authors wish to thank Jean-Michel Muller and all his colleagues from E.N.S. and I.N.P.G. for all their help.

References

[1] 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.

[2] 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.

[3] P. Jaillon. 'LEA', a lazy exact arithmetic: Implementation and related problems. Technical Report in preparation, École Nationale Supérieure des Mines de Saint-Étienne, 1993.

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

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

[6] K. Mehlhorn. *Data structures and Algorithms 3: Multidimensional Searching and Computational Geometry.* Springer-Verlag, Berlin, 1984.

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

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