A Lazy Exact Arithmetic

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

Abstract

Systems based on exact arithmetic — as they are known in Symbolic Calculus or Computational Geometry — are very slow. In practical situations, very few computations need be performed exactly as approximating the results is very often sufficient. Unfortunately, it is impossible to know in advance (i.e. at the time when the computation is called for) whether an exact evaluation will be necessary or not. The arithmetic library presented here achieves laziness *by postponing any exact computation until it is* proven *to be indispensable. This yields very substantial gains in performance while allowing exact decisions.*

The paper presents lazy arithmetic techniques in the context of rational computations and uses the field of Computational Geometry as background.

1 Introduction

Exact (arithmetic) libraries are used, among other fields, in Symbolic Calculus or in Computational Geometry. It is well known that they require a large amount of memory space and computational resources. Lazy implementations offer interesting optimisation schemes, and, as such, allow to broaden the until now rather restricted application field of exact arithmetics. The lazy arithmetic library presented here operates on floating point numbers and rational numbers represented by unbounded integers.

This section presents the problem and research related to it. The basics and objectives of the lazy arithmetic are listed in 2. Section 3 deals with the manipulation of lazy numbers. Section 4 addresses implementation issues; section 5 presents some extensions and/or future work, and section 6 gives information about performance issues.

The term "lazy" was chosen with the lazy evaluation scheme of functional languages (Miranda, LazyML, etc.) in mind. However, unlike these languages, lazy arithmetics exploit multiple representations for data: one representation approximates each piece of data, and another stores a symbolic definition for it, which may, in turn, yield the exact value associated with the data, if necessary.

1.1 Previous work

In Computational Geometry, one has to make topological decisions based on the outcome of arithmetic tests, such as the sign of a determinant. Errors related to finite precision induce topological inconsistencies, and hence a vast amount of research has been devoted in the last few years to finding efficient solutions to this problem. A first class of such methods attempt to solve precision problems by either allowing imprecise computations, or compensating finite precision by various floating point schemes:

Epsilons: All computations are performed to within a given *tolerance* (basically the unit roundoff error). Roughly speaking, any two quantities differing by less than the tolerance are declared equal. This popular heuristic proves very poor in practical situations as it makes it impossible to discriminate events or even objects conflicting within the tolerance.

Epsilon Geometry [5], [15]: Computations are performed within a certain degree of *confidence*. Predicates help determine whether any result is within its associated confidence interval or not, but no comprehensive solution may be given in practice as soon as decisions happen to be made *outside* the associated confidence intervals.

Symbolic Geometry [12], [6]: Things are organized so that any new decision be consistent with all the ones made before. This technique is extremely powerful but calls for a total and clever redesign of existing algorithms (minute bookkeeping of past history).

Adaptive Geometry [4]: Objects are 'bent' in order to adapt their geometry to a discrete set of positions (integral grid). For instance, two intersecting segments are transformed so that their intersection occurs at the nearest appropriate point on the grid. As this induces possible perturbations on the true topology of the initial data, the consistent results produced are not necessarily coherent with those of the original problem.

Obviously, another solution is to only perform exact computations, with the help of an exact library. Such a technique has major drawbacks:

- Exact computations are extremely greedy both in time and memory resources.
- In most situations, exact arithmetic is not used to get exact results but to reach coherent decisions and to ensure program consistency. So there is a vast amount of computational effort wasted in systematic exact evaluations.
- Unlike their floating point counterparts, exact arithmetic libraries are not available in standard high-level language environments, with the exception of Lisp and SmallTalk (which provide for built-in rational operations). Furthermore, implementing exact arithmetic libraries in such languages as *C*, *FORTRAN* or even *C++* yields severe memory management problems. Next, from a syntactic point of view, exact numbers may never be treated as machine numbers in *C* nor *FORTRAN*, but require extensions, external to the language (as opposed to libraries), such as *PASCAL-XSC*, *C-XSC*, *ACRITH-XSC*. Solutions of this type do not offer the flexibility of the library presented here.
- Until now, the only exact modules used in Computational Geometry (but not in symbolic calculus) work on unbounded integers or rationals. This forbids the manipulation of algebraic (and *a fortiori* transcendental) curves or surfaces. Using other schemes (such as the one presented in this paper) might shed another light on these problems and perhaps help solve them in the future.

To overcome the previous drawbacks, some research has been devoted to finding better solutions based on exact libraries, or a mixture of exact and floating point arithmetics:

Semi-exact techniques [8]: The authors use integer intervals to accelerate computations in conjunction with a *C++* implementation of rational operations. Results (determinants for instance) are first computed in terms of intervals with integer bounds. If the intervals do not allow for safe decisions (e.g. finding the sign of a determinant whose value interval contains 0), the intervals are refined and the process is repeated. Although it yields exact decisions, the main disadvantage of this method is that its computational cost is hard to evaluate. See also [11] and [3] for other possible exact arithmetic schemes.

Reluctant algorithms [14]: Each basic decision may be expressed as a functional whose associated diffidence interval (which depends on various parameters such as number size and the functional itself) may be calculated at compile time (although possibly not so easily). When a decision must be made, it suffices to check whether the float test value lies outside its associated diffidence interval ("safe" decision) or not (the decision must be "backed up" by exact computations). Such a technique saves a lot of computational effort but requires severe rewriting of programs and forces the maintenance of large data structures to store intermediate results or objects that may prove totally useless if the associated values have not been involved in any other computation.

2 Laziness fundamentals

In view of the preceding presentation, laziness stands out as a new and important paradigm. It is important to emphasize the fact that laziness is exploited at the arithmetic level and not at the program level: any program using a lazy library automatically benefits from the lazy exact arithmetic without the programmer having even to notice: from his or her point of view, only (abstractions of) "real" numbers, called *lazy numbers*, are used and manipulated with the help of standard *C++* operators. This is a major improvement over any method requiring the rewriting of all programs for the sake of taking precision idiosyncracies into account. This section briefly outlines the fundamentals of laziness.

2.1 Objectives

The lazy library:

1. Must yield consistent decisions, with the help of floating point operations or of an infinite precision

arithmetic. It must also be fast, using as little resources as possible (although more than standard floating-point computations, by definition).

2. Must only perform the exact computations that can be proven to be indispensable in the context of floating point numbers.

3. Must be totally transparent to the user who will want to manipulate lazy numbers as traditional floating point numbers or integers.

4. Must provide for the four basic arithmetic operations ($+, *, inv_+, inv_*$), and the elementary comparisons ($<, \leq, =, \neq, \geq, >$). In the restrictive rational setting we are imposing here, it will be impossible to use square roots or more complex operators. Multiple representation of data will be the clue to ensuring exact decisions and fast computations. Transparency and basic arithmetic on lazy numbers will be achieved through sophisticated language constructs (for instance *operator overloading* in *C++*).

2.2 Lazy numbers and multiple representation

Any real number is represented as a lazy number via an *interval* and a *definition field*. The interval is bounded by two floating point numbers and must contain the exact value. Most of the time, the interval will be sufficient for numerical tests. When this is no more the case, the definition field will enable exact computations. This field initially contains an unevaluated symbolic definition of the number in the form of an *expression dag* (refer to 3.2) with nodes for operators and references to other lazy quantities. When evaluation is in order, the symbolic definition dag of the lazy number to be evaluated is replaced with a single node containing the exact (rational) value.

2.3 Elementary operations

To perform an elementary lazy operation ($+, *, inv_+, inv_*$):

1. allocate a new dag node for the result,

2. compute the floating-point interval with the help of interval arithmetic operations as decribed below, and

3. fill in the field of the unevaluated definition with an operator node and pointers to the operands.

No exact computation is performed. Furthermore, in the case where the exact value was not needed after all, it will never have been computed!

2.4 Lazy evaluation

The only reasons why a lazy number should ever be evaluated are when

- its sign or its reciprocal must be determined and its associated interval includes 0,

- it is to be compared to another lazy number and their associated intervals overlap (see 3.3),

- another lazy number whose definition refers to it needs being evaluated.

3 Creating and manipulating lazy numbers

3.1 Interval arithmetic for lazy needs

This section presents the basics of interval arithmetics, according to our (lazy) needs. It is a rather lazy account on an anotherwise well-studied field. The courageous reader may wish to refer to more authoritative sources on the topic, such as [9], [10], [13]. Let us state the minimum definitions that will be needed. To restrict the discussion of over- and underflows to the minimum, let us just suppose that all lazy numbers belong to the subset $\Sigma =]-M, -\epsilon[\cup \{0\} \cup]+\epsilon, +M[$ of the real line, where ϵ and M can be understood, in first approximation, to be the smallest and largest positive floats on the machine.

A thorough presentation of this topic and the following will be available in [7].

3.1.1 Primitives and intervals

If χ is any floating point number in Σ, $\nabla(\chi)$ (resp. $\Delta(\chi)$) is the floating point number immediately below (resp. above) χ. Hence the "natural" interval bounding any real number X in Σ is $[\nabla(\chi), \Delta(\chi)]$, where χ is the machine approximation nearest to X.

3.1.2 Operators and intervals

Let $\perp$ be any arithmetic operator amongst $\{+, *, -, /\}$ in the real domain. Its machine equivalent will be denoted by $\boxed{\perp}$. Suppose a and b are two floating point numbers in Σ. Due, among other things, to truncature and alignment considerations, the result of $a \perp b$ has no reason to be a representable floating point number. But one can be assured ([9]) that

$$\nabla(a \boxed{\perp} b) < a \perp b, a \boxed{\perp} b < \Delta(a \boxed{\perp} b),$$

whence a bounding interval for $a \perp b$.

3.1.3 Arithmetic operations on intervals

Let x and y be any two lazy numbers. Denote by $I_x = [a_x, b_x]$ and $I_y = [a_y, b_y]$ their bounding intervals. We wish to determine the intervals bounding their sum and product, or the opposite and the reciprocal of x. Using the above definitions and excluding exceptions for the sake of conciseness, we may write:

$I_{x+y} = [\bigtriangledown(a_x \boxed{+} a_y), \triangle(b_x \boxed{+} b_y)]$.

$I_{x*y} = [\bigtriangledown(m), \triangle(M)]$, where m (M) is the minimum (maximum) of $\{a_x \boxed{*} a_y, a_x \boxed{*} b_y, b_x \boxed{*} a_y, b_x \boxed{*} b_y\}$.

$I_{-x} = [-b_x, -a_x]$.

$I_{1/x} = [\bigtriangledown(1.0 \boxed{/} b_x), \triangle(1.0 \boxed{/} a_x)]$, except when not applicable (see 3.3).

Typically, the amplitudes of bounding intervals grow with the number of lazy operations. Evaluation may be seen as the ultimate tool to stop this growth, albeit the most expensive and most unwanted one.

In the implementation of the above definitions, the only difficulty is to avoid the errors related to over- and underflows during floating point computations. One is often forced to simulate IEEE standards for floating point numbers [2], which are designed to signal and account for all such events – and which all systems should abide by but rarely do! A neat way out of this is to use floats (32 bits single precision floating numbers) for the interval bounds and to make computations with doubles (64 bits double precision floating numbers) while checking at every step that no over- nor underflow occurs. Special flags $(-\infty, 0^-, 0^+, +\infty)$ may be used to label each situation and to help discriminate out-of-range lazy numbers from standard ones (for which a valid bounding interval may be expected). Interval arithmetics must take these "exceptions" into account, which calls for complex *Pythagorean tables* for all operators. Refer to [7] for more details.

3.1.4 Data alignment

As the lazy library may happen to handle exact quantities at various stages, some convention must be used to "align" the initial data on proper exact numbers, and to associate consistent bounding intervals with them. Note that such alignment is inherent to any exact arithmetic library and that we shall not consider the case where data are inputted as decimal numbers and read as strings, which may be easily treated much the same as described below. Three major techniques may be considered:

Exact alignment: floating point numbers are converted into exact (rational) values by a straightforward algorithm without loss of precision! But this may be a little akward as the resulting rational numbers will most likely be cumbersome, and as such "original" numbers may themselves be the results of imprecise computations.

Regular alignment: inputs are aligned on the nearest multiple of a given unit. For instance, if the unit were 1, floating point numbers would be aligned on the nearest integer (cf. [16]).

Adaptive alignment: the inputs are aligned on the nearest rational number for a given precision. Several techniques may be used for this purpose, among which continued fractions expansion (*CFE*) [9]. This method yields the fastest algorithms and the most concise rational numbers for a given precision.

Note that the definition of lazy numbers can be augmented with a new class for each alignment method. For instance, instead of storing rational 2/3 in a definition, one may store something like "the number obtained through continued-fractions-alignment with original data .6666667 and with relative precision 10^{-6}". This technique allows to delay alignments until they become inevitable.

3.2 Dags

Trees are the natural structures for storing expressions. Any arithmetic expression may be translated into a tree whose internal (binary or unary) nodes contain operators $(+, *, inv_+, inv_*)$ and whose leaves are lazy operands. (Note that this may be generalized to trees with arbitrary *arity* to allow composite operators, such as determinants or universal functionals). The definition field of a lazy number typically points to a fixed number of nodes (two for binary operators, etc.), but the tree at the root of which it resides may have an arbitrary depth. Moreover, any lazy number may be referred to by more than one "father". Hence the proper structure for storing a lazy definition is not a tree but a directed acyclic graph (*dag*).

When a lazy number is first met, its definition field is filled with a pointer to the formal dag representing the rational expression defining the number. For instance, if number x is defined by $x = a/(b-c)$ where a, b and c are other lazy numbers, it may be easily rewritten in terms of the four basic operators as $x = ((a) * (inv_*((b) + (inv_+(c)))))$. Its definition field will contain a tree with operator $*$ at the root, a and operator-node inv_* as left and right children. The latter will itself have a "unique" child (operator $+$), and so forth. This implicitly means that subexpressions are stored as lazy numbers, and may, of course, themselves be lazy numbers or subexpressions referring to constants (such as "1", "22/7", etc.).

Therefore, filling the definition field of a lazy number (such as resulting from the computation of a determinant) takes space and time proportional to its length in the worst case, as each operator uses up $O(1)$ space and is constructed in constant time.

3.3 Elementary operations on lazy numbers

We have just seen how to fill in the definition fields for all arithmetic operations considered here. Let us now consider interval management. Adding, multiplying two lazy numbers or taking the opposite of another simply amounts to computing a new bounding interval using the rules in 2.2.

Suppose we now want to compute the reciprocal of a lazy number x. If its bounding interval I_x does not include 0, the image by $x \rightarrow 1/x$ of this interval only has one connected component and the simple rule stated in 2.2 may be applied to build it. If $0 \in I_x$, then the interval image is clearly splitted into two connected components: an exact evaluation is called for, and is decided upon by the lazy library itself. The main advantage of evaluation in this case is to refine the interval for x (which has obviously grown beyond control) and to provide one for its reciprocal. Evaluation is followed by replacement of the definition for x by the node with the exact value.

3.4 Comparing two lazy numbers

Suppose the bounding intervals of the two numbers to be compared do not overlap. Then the numbers are necessarily different and it takes $O(1)$ to determine the relative positions of their bounding intervals, and hence of the numbers themselves.

If the intervals overlap, then the idea is that they have grown too large and the numbers must be evaluated: in other words, machine precision is insufficient to discriminate the two numbers! Evaluation will result in new and tighter bounds for the intervals, which may then be disjoint (the numbers may then be compared with interval arithmetic). If they are not, the last resort is to compare the rational values, using the exact library.

Now, the most difficult lazy task is to actually prove that two given numbers are equal. In fact, the previous algorithm says that the only way out is to evaluate both numbers! Needless to say, the very first thing to do to improve on this is to check, before anything else, whether the two lazy numbers are not indeed references to the same memory location. This obvious test saves a substantial amount of evaluations in practice. As we shall see later, this is by no means sufficient nor satisfactory to settle all cases.

3.5 Evaluation

The most natural and simple evaluation strategy consists in evaluating the definition trees from the leaves upward while propagating new bounds from each level to the next. This yields complete evaluations of trees and new, tighter bounds for intervals.

Each lazy number evaluated in the process is replaced with its exact value. Incidentally, each evaluated lazy number that is not referred to by any other element may be disposed of. This implies that the definition field must also allow some sort of reference for bookkeeping and garbage collection purposes.

Other evaluation strategies have been tried, but all are equivalent to within 10 percent. See [7] for details.

4 Implementation

We chose *C++* to implement the lazy library. This language provides for the overloading of arithmetic operators. Thus, the lazy library allows one to manipulate lazy numbers as standard numbers in a straightforward manner. The lazy rational arithmetic library includes three modules: the interval arithmetic module, the rational arithmetic module and the dag management module. See [7] for more details. The major classes are:

Interval: Instances include two floats. Main methods: Arithmetic operations, `IncludesZero?`, `Disjoint?`.

Arbitrary length integer: Instances have the following fields: number of digits, pointer over dynamically allocated digit array. Main methods: Arithmetic operations, comparisons, greater common divisor, next or previous float.

Rational: Instances have two pointers on arbitrary length integers (numerator and denominator). Main methods: Reduction to the common divisor, arithmetic operations, comparisons, next or previous float.

Lazy number: Instances include a bounding interval and a definition field. Main methods: Arithmetic operations, evaluate.

Definition: Virtual class. Subclasses: rational and expression. Essential method for this class and all inheriting classes: Evaluate.

Expression: Virtual class. Subclasses: sum, product, opposite, reciprocal.

Sum, Product: Instances include two pointers on lazy operands.

Opposite, Reciprocal: Instances include a pointer on lazy operand.

Memory management is ensured through reference counters associated with lazy numbers, rationals and arbitrary length integers. Garbage collection of temporary variables is automatic (note that lazy numbers are represented by acyclic graphs). Of course, it is still the responsibility of the programmer to dispose of any data structure he or she may allocate in the program.

5 Extensions

Simple ideas may be suggested to improve the lazy library. We shall only mention a few problems we have been confronted with.

A straightforward implementation of the previous concepts results in disappointing performances for applications with realistic, non-random data. Fortunately, this may be easily seen to.

5.1 Detecting clones

One of the main reasons for such misbehaviors is that a lot of useless evaluations are done, owing to the fact that the most straightforward and "natural" algorithms tend to check that equal things are equal!

Let us take an example. Suppose we store geometric items – for instance segments – in a binary tree, ans we wish to delete one segment in this structure. We must compare this segment against all those along the search path until it is actually compared to itself. At this point, we have seen in 3.4 that a sane strategy is to compare addresses to detect equality.

Let us now suppose that slopes are used as keys. The problem is that, for such simple tests as `if (slope(x) == slope(y))`, there is no way to "teach" the lazy library to remember where the quantities on both sides of the `==` sign come from (without asking the programmer to cooperate: An easy way out would be to ask such tests to be written out as: `if (x == y || slope(x) == slope(y))`!)

Luckily, there is an elegant and easy solution. Tree a is said to be the clone of tree b if both have the same structure, nodes and leaves. It is clear that although `slope` might cast a shadow on the origin of the previous quantities to be compared, the images of x and y by the function yield perfect clones when x and y are pointers to the same location.

Checking whether two trees are actual clones is a classical problem that can be solved using any standard tree traversing procedure combined with the appropriate tests, as in:

```
function Clone? (a, b: LazyNumber): boolean;
{
  if (address(a) == address(b)) return true;
  if (a.interval ∩ b.interval == ∅) return false;
  if (class(a) ≠ class(b)) return false;
  if (class of a is UnaryOperator)
    return Clone?(a.onlyson, b.onlyson);
  if (class of a is BinaryOperator)
    if (Clone?(a.lson, b.lson) and Clone?(a.rson, b.rson))
      return true;
    else return false;
  return EqualLeaves?(a, b);
}
```

Comments: To optimize the process, recursion is stopped as soon as the intervals of the two subtrees being tested are disjoint or the nodes are references to the same memory location. Testing intervals is especially useful for the comparison of trees with identical structures differing only at the leaves (e.g. two determinants). `EqualLeaves` is a simple function that returns true iff its arguments are bitwise equal.

5.2 Comparing two lazy numbers (final version)

The comparison of two lazy numbers a and b may now benefically be rewritten as:

```
function LazyCompare (a, b: LazyNumber): -1..1;
{Comment: returns the sign of (a-b)
  if (address(a) == address(b)) return 0;
  if (a.interval ∩ b.interval == ∅)
    if (b.interval.lbound > a.interval.rbound) return +1;
    else return -1;
  if (Clone?(a, b)) return 0;
  return ExactCompare(a, b);
}
```

`ExactCompare` is, of course, the comparison function from the exact library, returning values consistent with those returned by `LazyCompare`.

5.3 Hashing and other related topics

Lazy arithmetic provide for the basic arithmetic and comparison operators. Such operators are not sufficient for geometric algorithms, for instance those in which numbers are treated as identifiers. Typically, associating a hash-code to each number allows geometric algorithms to retrieve vertices from coordinates. To this effect, it is possible to choose a hash-function such that, in the general case, if a and b are two lazy numbers and $\perp$ any operator in $\{+, -, *, /\}$,

$$hash(a \perp b) \equiv hash(a) \perp hash(b).$$

Some care must be taken to treat special cases, but this is beyond the scope of this paper. The interested reader will find ample information in [7].

Hashing lazy numbers helps cutting down on evaluation: if two lazy numbers with overlapping intervals have different hash-keys, they are necessarily different! Hence hashing schemes are used by the library itself, and may be used by the programmer, if needed.

5.4 Prospects

Quite a number of extensions to the lazy arithmetic library described here have been designed and implemented by the authors. Others are in the making, still.

The basic goal we have set (designing a tool to relieve the programmer from finite-precision concerns) has proven, so far, extremely powerful, and brings all kinds of interesting problems, new or old.

For instance, laziness – at its best – would be to also define new compiler-level boolean tests, that should impose priorities on evaluation according to lazy paradigms. For instance, in a logical *and* test, it would be useful to check whether one condition or the other is not false by interval arithmetic standards, before *evaluating either*.

One may also wish to define new lazy operations, with larger scope, such as the maximum of a finite set of values, absolute values, etc. In a nutshell, detecting the need for laziness at language level would be the best one could hope for: this could only be dealt with by designing a truly lazy-oriented language.

The major future goal is to generalize the concepts in this paper to an algebraic arithmetic. This will allow, we hope, to address certain classes of problems in Computational Geometry that can hardly be tackled so far.

Some of the topics hinted at in this section, including hashing methods applied to lazy numbers, are presented in another paper (submitted to SCAN-93, Vienna).

6 Results and performance

We have implemented the lazy library presented in this paper and experimented it on a typical algorithm from geometry, due to Bentley and Ottmann [1], which computes the $K \in O(N^2)$ intersections between N segments in the plane in time $O((N+K)logN)$. It is well known to be extremely sensitive to precision (see [11]).

Two series of charts and corresponding tables are shown after the references. They describe the differences in running time of the algorithm using floating-point (solid curves), lazy-&-clone-detection (dashed curves) and exact (dotted, topmost curves) arithmetics, with increasing CFE precision (from $1e^{-1}$ to $1e^{-9}$) respectively for 50 segments (table and chart 1) and 100 segments (table and chart 2). Segments being randomly chosen, the worst-case running-time ratio is about $\frac{100^2 log 100}{50^2 log 50} \approx 4.7$, which is basically what comparing both series shows. Increasing precision does not affect floating point computations (hopefully) nor lazy computations (which is good news), but they do affect rational computations: the more accurate the computations, the more expensive the solution. The lazy/exact ratio ranges from 4 to 75 when precision varies from $1e^{-1}$ to $1e^{-9}$.

Roughly speaking, the floating point and lazy curves follow asymptotic complexity for random cases, whereas the exact curve shows the overhead of unbounded precision. When the segments are randomly positioned, there is next to no exact computation, and the overhead for laziness is obviously very moderate. On the other hand, it is important to point out that if the topology of the scene presents many special cases (vertical segments or very close ties), the "float" version invariably crashes contrary to the lazy and exact versions which behave much the same: the exact computations that the exact version would perform are done by the lazy version in addition to the interval computations it is naturally supposed to do and the construction of the dags.

Thus the overall price to pay for laziness is by far more reasonable than that for exact computations: in the vast majority of cases (segments from real scenes), the overhead is only equal to the cost of creating and updating dags, and the rat/lazy ratio may be anywhere between 50 and 150 (with $1e^{-12}$ precision).

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

References

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

[2] J.T. Coonen. An implementation guide for a proposed standard for floating point arithmetic. *IEEE Computer*, 13(1), 01 1980.

[3] H Edelsbrunner and E.P. Mücke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. In *Proceedings of the Fourth Symposium on Computational Geometry*, pages 118–133. ACM, 1988.

[4] D.H. Greene and F.F. Yao. Finite-resolution computational geometry. In *Proceedings of the 27th Annual Symposium on the Foundations of Computer Science*, pages 143–152, 1986.

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

[6] C. Hoffman, J. Hopcroft, and M. Karasick. Towards implementing robust geometric computations. In *Proceedings of the Fourth Symposium on Computational Geometry*, pages 106–118. ACM, 1988.

[7] P. Jaillon. Lea, a lazy exact arithmetic: Implementation and related problems. Technical report, Ecole Nationale Supérieure des Mines de Saint-Etienne, 1993.

[8] M. Karasick, D. Lieber, and L. Nackman. Efficient delaunay triangulation using rational arithmetic. Technical Report RC 14455, IBM, 1989.

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

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

[11] D. Michelucci. *Les représentations par les frontières : quelques constructions; difficultés rencontrées.* PhD thesis, Ecole Nationale Supérieure des Mines de Saint-Etienne, 1987.

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

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

[14] J-M. Moreau. *Facétisation et hiérarchisation de la représentation par segments d'un graphe planaire dans le cadre d'une arithmétique mixte.* PhD thesis, Ecole Nationale Supérieure des Mines de Saint-Etienne, 1990.

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

[16] K. Sugihara and M. Iri. *A Solid Modelling System Free from Topological Inconsistencies.* Research Memorandum, RM 89-03, University of Tokyo, 1989.