# A "Lazy" Solution to Imprecision in Computational Geometry

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: moreau@emse.fr

**Abstract**

*Making safe and consistent decisions is essential to geometric algorithms. Quite a few solutions to this problem have been suggested in the recent years, but they generally ask that drastic changes be made to algorithms. A simple and efficient paradigm is suggested here, which enables programmers to forget about precision issues altogether, whatever algorithm they intend to implement.*
*The paper describes a lazy exact arithmetic library (LEA) based on this paradigm and its operation in a typical situation from Computational Geometry.*

## 1 Introduction

Lazy languages (ML, Miranda) emerged a few years ago, introducing the strong paradigm of *laziness*: "Why should a quantity be evaluated if it is never going to be used later?" The same applies to precision: "Why should a quantity be computed *exactly* if it is never involved in *conflicting issues* later on?"

Of course, if brutally implemented, such an observation would be fatal in situations where some exact information – that was available but not needed at one stage – becomes indispensable but no longer accessible, later on!

Although there is no way to predict when they do arise, such situations are not infrequent in ordinary computations, and the simplest, safest prevention is to memorize – when *still* possible – as little as will prove sufficient for complete information retrieval, *should* this ever become necessary at a later stage.

Why should we ever be concerned with precision issues, let alone lazy paradigms? Section 2 briefly reviews precision problems in Computational Geometry and their treatment by the most widely used techniques. Section 3 presents the objectives and principles of a lazy library. In section 4, laziness is shown in operation on a famous problem from Computational Geometry and Section 5 concludes on future work.

## 2 Precision and Geometry

Computational Geometry deals with algorithms on real points, lines, planes, solids... as modelled on a finite-precision machine. Most of the time, processors truncate digits, round off results, and almost every decision made by any program is biased! Naive algorithms often process data independently. The consequence of an error is then most likely local. However, optimal algorithms generally depend on the coherence of the inputs to guarantee coherent outputs. For them, any error may start a chain reaction that will corrupt the global solution, or even cause system crashes, because the logic of the objects they model and that of the data they manipulate are no longer consistent.

We shall use the following problem to illustrate our discussion: Given $N$ distinct segments in the plane, detect and identify all $K \in O(N^2)$ intersections between them. There are basically three methods for solving this problem: a brutal and always quadratic algorithm (test the intersection of all possible pairs of segments), the more sophisticated $O((K + N) \log N)$ algorithm due to J.L. Bentley and T. Ottmann ([1]), and finally the optimal $O(K + N \log N)$ algorithm by B. Chazelle and H. Edelsbrunner ([3]).

The first, "brute-force" algorithm is straightforward and is known to be extremely robust, if not computationally efficient.

"*Bentley-Ottmann*'s" algorithm is so famous that we refer the reader who would be unaware of its principles to [1] (the original paper), [24] or [16] where it is further described, commented upon and analyzed. We shall just need to know that this algorithm uses a vertical sweep-line moving from $-\infty$ to $+\infty$, supposed to model, at the end of each iteration, the vertical order of all segments it intersects. It is also well-known that this algorithm is extremely sensitive to machine errors (cf. 2.1), and therefore is a very good candidate for testing the virtues of laziness.

We intend to implement *Chazelle-Edelsbrunner*'s algorithm in the future to measure its sensitivity to precision, which should be much less than *Bentley-Ottmann*'s. Interestingly enough, the authors of [3] use a "functional approach to data structures" ([2]), that shows similar concerns, to some extent, to the lazy paradigm described in this paper.

## 2.1 *Djinns*

Define $x$-order ($y$-order) as the natural order on abscissæ (ordinates) – noted $>_x$ ($>_y$) – and $xy$-order as the usual *lexicographical* order on points ($x$-order, then $y$-order for ties) – noted $>_{xy}$. Now suppose a facetious *djinn* exchanges the positions of two elements in the $xy$- or $y$-order while *Bentley-Ottmann*'s algorithm is running: There is no way to guarantee that the results will be valid, or even that the process will terminate.

This supernatural intervention is likely to perturb the execution of procedures manipulating $x$- and $xy$-order data structures since the invariants of such structures are no longer preserved. The program will invariably abort or wind up in theoretically impossible (topolo-)logical conflicts ([17]).

*Bentley-Ottmann*'s method assumes it is possible to sort endpoint or intersection coordinates reliably, and this may no longer be guaranteed when finite precision is used. Consider the following example (refer to Figure 1): Each square has unit length sides, so $A = (0,0), B = (1,3), C = (0,2), D = (1,2)$. Furthermore, the abscissa of both $E$ and $F$ is equal to the floating point number 0.666667. $AB \cap CD = \Omega = (\frac{2}{3}, 2)$. In the machine, and in the best case, $\Omega$ will be represented as a point $\Omega^\star$ such that, say, $\Omega^\star = (0.666667, 2)$. The vertical segment $EF$ has been chosen so that, according to the *real* $x$-order $\Omega <_{xy} E <_{xy} F$, whereas we have $E <_{xy} \Omega^\star <_{xy} F$ on the machine. $\Omega^\star$ will therefore be inserted after $E$ in the $x$-order, which is wrong: This is just how a minor numerical error becomes topological, and imprecision plays the part of a facetious *djinn*.

Figure 1: *Djinns* and *Bentley-Ottmann*'s algorithm.

A last remark: Imprecision only has serious consequences when it alters the order between numbers. This means that finding a good lazy remedy to precision problems is to cure them when they become hazardous for coherence, and to leave them alone otherwise!

## 2.2 A collection of published solutions

Quite a lot of research has been devoted to finding solutions to numerical imprecision. Extensive accounts on this topic may be found in [18] and the other sources below. For the purpose of this paper, we shall only give the outlines of a (non-exhaustive) classification.

1. Solutions based on pure floating point arithmetic
   (a) Numerical solutions
      i. Epsilons (popular folklore)
      ii. Finite exact precision ([4], [9], [18], [19])
      iii. Epsilon geometry ([8], [25])
   (b) Geometrical solutions
      i. Adaptive geometry ([7], [18], [20])
      ii. Robust geometry ([10], [13], [18])
      iii. Constructive geometry ([11])
      iv. Symbolic geometry ([18])
2. Solutions based on an exact library
3. Perturbation techniques ([5], [26])
4. Mixed solutions
   (a) Solutions based on one exact operator ([23])
   (b) Semi-exact solutions ([14], [17])
   (c) Reluctant algorithms ([22])

Solutions in class 1 compensate for numerical imprecision with floating point tools using either purely numerical or purely geometrical strategies. Most of these solutions may only be applied to specific problems, and generalizations are difficult.

Solutions in class 2 solve imprecision problems using an exact representation for data, and hence are extremely greedy in time and space.

Solutions in class 3 do not attempt to solve precision problems, but remove degeneracies from computations by appropriately perturbing the data, and require an exact module – all the same.

Solutions from class 4 are much more universal in essence, and use two types of representations for the data to achieve consistent decisions. The present work originates from solutions 4(b) and 4(c).

## 3 *LEA*

### Objectives

The lazy rational library is an independent module, which may be used by virtually any program in a *C/C++* environment. It has five major objectives:

1. It must be exact (yielding consistent results, and results consistent with the data). Moreover, the library must provide for the basic arithmetic operations on lazy numbers $(+, *, inv_+, inv_*)$ and comparisons. Extensions have in fact been added to allow more sophisticated operators.
2. It must be transparent to the programmer who will use `LazyNumbers` *instead* of (but in exactly the same way as) `floats`.
3. It must be fast, and use as little resources as possible (*i.e.* slightly more than floating point solutions, but much less than exact solutions).
4. It must depend on the built-in floating point arithmetic of the machine and an exact arithmetic module – possibly user-defined – without explicitly depending on whatever exact representation is chosen. Thus, it should be possible to replace rational numbers with algebraic numbers without disturbing the operation of the lazy library. This means that specific modules should be made available for working with algebraic numbers, etc.
5. Programs based on valid algorithms, running successfully with floating point arithmetic (but possibly crashing in degenerate situations) must terminate successfully with the lazy library (even in the presence of such degeneracies as accepted by the underlying algorithms).

### Principles

**Multiple representation:** A lazy number is represented as an *interval* and a formal *definition.* The interval is bounded by two `floats` and must contain the underlying number (the one modelled by the lazy number). The definition is a field that conveys, at any moment, sufficient information to produce the exact representation of the underlying number.

**Intervals:** Any data inputted in a program must be assigned an interval (by the library itself). Methods for this, justifications, treatment of over- and underflow exceptions will be covered in depth in [12].

**Definition fields:** When a lazy number is first encountered, its definition field is *unevaluated* and is basically made up of a node for an operator and pointers to operands. Another way to look at this is to say that the formal expression for any lazy number is a tree[1]. For instance, if $a, b, c$ are three lazy numbers, the definition field for

$$z = a + \frac{1}{b+c}$$

has a node containing the binary operator "+", itself pointing to two "subtrees" (one for "$a$", another for subexpression "$inv_*(b+c)$").

The important thing is that no evaluation is done when the definition field of a lazy number is constructed. In other words, the construction of the *dag* for any $\underline{\text{elementary}}$ arithmetic operation is a *constant* time and space process, as opposed to evaluation.

**Evaluation:** Intervals tend to grow with the number of arithmetic operations. If this becomes a problem (see comparing lazy numbers, or computing their reciprocal below), the only way out is evaluation, a process involving the definition fields. When a lazy number actually needs being evaluated, its definition field is simply filled with the exact representation of the evaluated underlying number. The definition field is then said to be *evaluated.*

Incidentally, if a node in the definition *dag* of an evaluated number is not referred to by any other expression, it may disposed of. Hence, some sort of garbage collection must be arranged.

There are only three reasons why a lazy number *should* ever be evaluated:

1. When it is compared with another lazy number whose interval intersects its own,
2. When its reciprocal is required and its interval lies on both sides of 0, and
3. When the evaluation of a lazy number referring to it is called for.

**Refreshing intervals:** The most natural evaluation strategy consists in a recursive procedure that evaluates all the children of a given node, and then evaluates it with the help of the operator it contains. During this process, tighter interval bounds from lower levels bubble up to the current node, allowing to update and "refresh" the node interval itself.

**Comparisons:** If the numbers have non-overlapping intervals, the relative positions of these intervals (and hence of the two numbers) may be found in $O(1)$. Else, the intervals have grown too large and must be "refreshed". If intervals still overlap after evaluation, compare the numbers using exact arithmetic.

[1]Actually, it is a *dag* (for directed acyclic graph), since the node for a lazy number may be pointed to by as many other lazy numbers as needed in the application programs

**Arithmetic operations:** Interval arithmetic ([15], [21]) allows a straightforward definition of the intervals resulting from the sum, difference and product of two lazy numbers (represented by their own intervals). As for the reciprocal of a lazy number, the situation is more complex. Let $z$ be any lazy number, and $I_z = [\alpha, \beta]$ its associated interval. Clearly, if $0 \notin I_z$, the image of $I_z$ under $z \rightarrow \frac{1}{z}$ is another interval $[\frac{1}{\beta}, \frac{1}{\alpha}]$ with only one connected component, and the reciprocal of $z$ is well defined. Else, the image has two disjoint connected components and exact evaluation is required. Note that evaluation will yield a new and tighter interval for both $z$ and its reciprocal.

**Implementation:** We have chosen *C++* to implement the lazy library, as this language allows operator overloading (writing "`a < b`" in a program although `a` and `b` are not `floats` but "`LazyNumbers`"). Garbage collection is not provided for in this language, but has been made possible through the use of reference counters on rationals and arbitrary length integers. The library consists in an interval module (arithmetic on floating-point intervals), a rational module (arithmetic on arbitrary length integers in large base, and rationals), and a lazy module (operations on lazy numbers).

## 4 Using the lazy library

From a general standpoint, using a library such as the one described above, is straightforward. Data structures and objects refer to lazy numbers explicitly, instead of floating point numbers. Such a test as "**if** point $p$ is to the left of the directed line through $(q, r)$" is coded using determinants ([24]) as usual. Whether the outcome of this test is computed with floating point arithmetic only or with the extra help of exact arithmetic is not for the programmer to know, but for the lazy library to decide.

### 4.1 Putting it all together

Writing *Bentley-Ottmann*'s algorithm in program form is no easy process, and even less so when testing a new library. Specifying the library as was done obscures many details that only become apparent when things don't go the way they should.

Although it is only possible to scratch the surface of things in this paper, here is a brief list of the problems one is faced with when implementing a lazy exact library. Remember that the underlying goal is to solve these problems ahead of time, and at the lowest level, to relieve programmers from precision issues.

- Preventing useless evaluations when comparing lazy numbers. For an interesting subproblem, consider a test for comparing segment slopes, such as "**if** `(Slope(s) == Slope(t))`". The only way to prevent evaluation when the two segments are indeed equal is to teach the library how to detect that program-defined functions (such as `Slope`) may yield *clones*, *i.e.* different versions of *dags* with identical structures and equal "leaves".
- Allowing or forbidding the creation of identical dags, whichever is best. This may mean using "union-find" algorithms to take advantage of past experience in future computations (for instance: From $a = b$, and $b = c$, *infer* that $a \equiv c$).
- Computing hash keys from unevaluated dags. Hash tables may then be used to retrieve geometric information from co-ordinates, as often required in Computational Geometry.

### 4.2 Comparing performances

We have implemented the 'brute-force' and *Bentley-Ottmann*'s algorithms, both as a unique program for 3 versions. Linking either program with the appropriate library module generates its floating-point, exact, or lazy version. This means that programs running with floating-point arithmetic may almost instantly be converted into lazy applications. One of the parameters used to compare the lazy and exact versions is the relative precision (ranging from $10^{-1}$ to $10^{-12}$) with which inputs are encoded.

#### Brute-force algorithm

Note that although this version never crashes from imprecision, it is not guaranteed to yield consistent outputs (*i.e.* the resulting segment graph is not necessarily planar!).

On random data (segments with endpoints drawn randomly in $[0, 1] \times [0, 1]$), the lazy version is 4 to 10 times slower than its floating point counterpart, and always yields valid results, of course. Moreover, it performs no exact computation and is considerably faster than the exact version: for a relative precision of $10^{-6}$, it is 40 times faster; for a relative precision of $10^{-9}$, it is 75 times faster; for a relative precision of $10^{-12}$, it is more than 100 times faster.

On more realistic data, only the indispensable computations are performed by the lazy version, whence a substantial gain. Data for which all computations should be done in exact form are extremely rare and artificial (for instance all segments on the same line of support and partially overlapping). But even in this case, the lazy version outperforms the exact version, but is not ridiculed by the floating point version.

Figure 2: Charts for *Bentley-Ottmann*'s algorithm on 50 (a) and 100 (b) segments, respectively. The charts show the floating point version (bottom curves), the lazy version (middle curves), and the exact version (top curves). Horizontal axes show precision, and vertical axes indicate times (in sixtieths of a second).

### ***Bentley-Ottmann*'s algorithm**

The results for this algorithm are extremely encouraging for our paradigm. The floating point version crashes, as expected, when running on special cases (vertical segments, intersection of more than two segments at a same point, etc.). The overall performance of the lazy version is slightly slower than the floating point version on random data, owing to the extra overhead for dag maintenance, as one would suspect. The charts on Fig. 2 were obtained after running the three versions of *Bentley-Ottmann*'s algorithm on 50 and 100 segments, respectively, with a relative precision ranging from $10^{-1}$ up to $10^{-9}$. Increasing precision does not affect floating point computations (bottom curves) nor lazy computations (middle curves), but does affect rational computations (top curves): the more accurate the computations, the more expensive the exact solution! The lazy-to-exact ratio ranges from 4 to 75 when precision varies from $10^{-1}$ to $10^{-9}$. Roughly speaking, the floating point and lazy curves follow asymptotic complexity for random cases, whereas the exact curves show the overhead of unbounded precision. When the segments are randomly chosen, there is virtually no exact computation, and the overhead for laziness is quite moderate. 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 *exact/lazy* ratio may be anywhere between 1 (deadly pathological cases) and 150.

## 5 Conclusion

The paradigm presented in this paper relegates imprecision handling at the lowest level. It has been demonstrated on a classical problem from Computational Geometry, but is intended to be used for any algorithm (of rational essence) in the literature, sensitive as it may be to imprecisions.

Laziness yields fast and efficient solutions which output consistent results, even in the presence of the worst pathological cases. Using a lazy library is straightforward, frees programmers from precision issues, and from strenuous rewriting of algorithms.

There are a few references to laziness in the literature. [14] has pioneered the use of a double representation (floats and arbitrary-length-integer intervals), but this solution differs significantly from ours, in that (*i*) it is dedicated to a specific problem, and involves non trivial algebraic manipulations related to the problem at hand, and (*ii*) it relies on successive refinements of intervals with initially loose bounds, until they can unambiguously be declared not to include 0. Such a process may be iterated an arbitrary number of times, and hence its computational cost is hard to evaluate (refer to [18] for a discussion).

A reference to lazy evaluation is made in [6]. (The authors wish to thank the referees for pointing out, and O. Devillers, INRIA, for providing this very recent and otherwise unobtainable publication.) Fortune and van Wyk seem to have a rather negative opinion of such a scheme. The reason may well be that the main stream of their research led them away from investigating the true potentials of laziness. Tuning the library, and tailoring it to the strong constraints we had opted for, required long hours and patient (un-lazy?) work.

There are many questions left, however, among which the detection of algebraic identity (*e.g.* $a*(b+c) \equiv b*a+a*c$), and such topics should act as a strong incentive for future research. One of the major tasks that remains to be done is to adapt the concepts in this paper to more general settings, such as algebraic arithmetic.

## 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] B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.*, 17(3), June 1988.

[3] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. Technical Report CS-TR 148-88, Dept of Computer Science, Princeton University, 1988.

[4] O. Devillers. Robust and efficient implementation of the Delaunay tree. Technical Report 1619, INRIA, 1992.

[5] H. Edelsbrunner and E.P. Mücke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. In *Proceedings of the 4th ACM Symposium on Computational Geometry*, pages 118–133, 1988.

[6] 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.

[7] 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.

[8] 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.

[9] K. Heinrichs, J. Nievergelt, and Schorn P. A sweep algorithm for the all-nearest-neighbours problem. In *Lecture notes in Computer Science 333, H. Noltmeieir (ed.), CG' 88 International Workshop on CG*, Wilzburg, FRG, March 1988.

[10] C. Hoffman, J. Hopcroft, and M. Karasick. Towards implementing robust geometric computations. In *Proceedings of the 4th ACM Symposium on Computational Geometry*, pages 106–118, 1988.

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

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

[13] M. Karasick. *On the Representation and Manipulation of Rigid Solids.* PhD thesis, McGill University, Montréal, 1988.

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

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

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

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

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

[19] V.J. Milenkovic. Double precision geometry: A general technique for calculating line and segment intersections using rounded arithmetic. In *Proceedings of the 30th Annual Symposium on the Foundations of Computer Science*, 1989.

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

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

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

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

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

[25] 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.

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