ZEA – A zero-free exact arithmetic

D. Michelucci, J-M Moreau

LISSE/ENSM.SE, Saint-Étienne – France

1 Introduction

This paper describes the foundations of a new and entirely different exact arithmetic for geometric needs, based on the observation that the main problem when dealing with numerical issues is the handling of zero. With this in mind, the new arithmetic takes its strength from the fact that it forbids zero, or, said in a less provocative fashion, it forbids degenerate cases. In the long range, we strongly believe that such strategies could allow Computational Geometry to address non-linear problems.

There are two classes of exact arithmetics:

1. Those that always provide comparisons between numbers, and the signs of numbers; *bigint* and rational arithmetics are typical examples in this class, along with various algebraic arithmetics that unfortunately prove very slow and are seldom used in Computational Geometry (*CG*).
2. Arithmetics in the second class can only compare numbers with different values: they take an infinite time to detect the equality of two numbers. Equivalently, although they do "recognize" positive and negative numbers, they cannot detect zero (in a finite number of steps). Such arithmetics are called *real exact* or *constructive real* arithmetics. They can be based on Cauchy sequences of nested intervals, continued fractions expansions, or on-line computations with redundant numeration systems. They can compute algebraic nonrational functions (*e.g.* square root, cubic root, *etc.*), but also transcendental functions, as quickly and easily—as seen from outside—as any rational operation. It is worth mentioning that, by principle, they can only compute continuous functions. Of course, they all use some kind of laziness, since they compute with (potentially) infinite objects. To quote a few, see [23, 2, 22, 15, 10] and the references therein.

As is well known, algorithms in Computational Geometry do not withstand the inaccuracy of floating-point arithmetics, which results in topological inconsistencies or infinite loops at run time. To prevent those, computational geometers typically round initial numerical data (*e.g.*, coordinates of vertices for Delaunay triangulations) to integers or rational values, and then compute exactly, using a Class 1 arithmetic, since algorithms in that domain usually require the detection of equality (and nullity)[1].

In practice, this approach precludes *CG* from addressing nonrational problems. But even in the rational case, it is not proven that using a Class 1 exact arithmetic is the only approach, let alone the best one. So the question arises: is it possible for *CG* to use only arithmetics from Class 2? We now examine two positive answers to that question, one already published, and one we suggest and think has never been used before.

2 Gap arithmetics

One way to use an arithmetic from Class 2 in a geometric environment is to convert it to Class 1 (see *e.g.*, [17, 5, 12]). What is mainly required to do so is a way to detect zero, and this may be achieved by using various so-called gap theorems or conjectures. Consider, for instance, Canny's theorem [7]: *Let* $(x_1, x_2 \dots x_n)$ *be one of the finitely many roots of an algebraic system in* n *unknowns,* n *equations, each with total degree not exceeding* d*, and integer coefficients with absolute value not greater than* M*. Then, there is a threshold value*

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

such that either $x_i = 0$*, or* $|x_i| > \varepsilon$*, for all* $i \in 1 \dots n$.

(See [7] for a proof. Hint: consider the u-resultant, and apply Hadamard's bound.)

Assuming all the numbers involved in any geometric computation are either initial integer numbers (after rounding), or roots of some algebraic system at hand, this kind of "gap theorem" gives a method to compute the sign of a number (or to compare two numbers, by considering their difference): just compute an interval with half-width less than ε. Unfortunately, the threshold value is virtually unpractical,

[1] Naturally, to speed up *CG* programs, exact arithmetics are not systematically used in practice: many exact, and slow, computations may be avoided using floating point or interval filters, or some kind of laziness; see for instance [8, 4, 6, 16, 9]. However, this paper will not deal with this question.

and there is not much hope to improve (*i.e.*, increase) ε significantly[2]. Intuitively—and this seems confirmed by experiments—one may hope that finding the sign of a non zero number is fast, on the average. On the other hand, establishing the nullity of a given quantity is hopelessly slow: an intrinsically exponential number of digits must be computed.

Still, it is worth noting that Canny's theorem also applies in the linear rational case, *i.e.*, when $d = 1$: the εs are of course much more usable in practice. To our knowledge, no attempt has ever been made to use gap theorem-based rational arithmetics.

Several such "gap arithmetics" have been proposed and used in *CG*, especially to handle the square root operation [17, 5, 12]. In the transcendental case, D. Richardson [18, 14, 19] has proposed a gap conjecture, called the *Uniformity Conjecture*, for numbers defined by an expression (not by a system of equations) over the 19 symbols:

$$0 \dots 9 \quad (\quad) \quad + \quad - \quad \times \quad / \quad \exp \quad \log \quad \sqrt[k]{\ }$$

If the value of an expression has magnitude smaller than $1/19^l$, where l is the expression length, then it is zero[3]. In more intuitive words, the 19^l possible numbers are uniformly distributed modulo 1.

In conclusion, gap arithmetics lack gap theorems in the transcendental case, and generally yield unpractical methods: an exponential number of digits must be computed to prove nullity. Hence, exploring another direction seems justified.

3 Zero-free computations

We claim that a new approach to Class 2 arithmetic—the "zero-free exact arithmetic" (*zea*)—may be used with profit in *CG*. Its principle is as follows: instead of being rounded to integral or rational numbers, the initial numerical data are "rounded to" algebraically independent numbers[4]. To do so, the initial numbers are perturbed by a (potentially infinite) stream of random digits; for instance the initial value 0.47 will be perturbed into 0.470000845289... where the first zeros are present by respect for the initial value, and the other digits are random.

One may think that the numbers thus genereated are transcendental with high probability, but we do not claim this nor intend to use such a property: algebraic independence is sufficient for our needs!

CG programs branch according to the sign of polynomials (often expressed as determinants) with integer coefficients over initial numerical data[5]:

```
if F(p1,p2...pn) > 0
 then POSITIVE_CASE
else-if F(p1,p2...pn) < 0
 then NEGATIVE_CASE
 else ZERO_CASE
```

After the perturbation, the p_i's are algebraically independent, thus $F(p_1, p_2 ... p_n)$ cannot be zero: the ZERO_CASE may no longer occur (except if F is the identically null polynomial, which is supposed never to be used). Hence, Class 2 arithmetics may indeed be used in *CG*. Moreover degeneracies (3 aligned points, 4 cocyclic points, etc) may no longer occur: they are removed by the perturbation. This greatly simplifies the programming of *CG* methods. And finally it becomes possible for *CG* to consider non linear or non affine problems (*e.g.*, intersection between algebraic surfaces).

Such a scheme should be used with the typical constructions of *CG*, for instance convex hulls, intersections of generic (*e.g.*, non regular) polytopes, Delaunay triangulations or Voronoi diagrams, generic arrangements, and so forth. A typical domain where such a scheme *should not* be used is geometric theorem proving.

4 Discussion

Some remarks or questions are worth mentioning:

1. Dependencies between data are forbidden, to allow separate random perturbations. This means for instance that convex polytopes must be described either by the intersection of halfplanes, or by the convex hull of a set of points. Actually, there is nothing new here: it is already the case when *CG* methods round data to integer values.

2. What is the average number of digits required to determine the sign of a number?

3. The most frequently used random generators yield periodical sequences; in our case this would

[2]For example, in the following fairly simple system, ε must be smaller than $1/M^{2^n-1}$:

$$\begin{aligned} x_1(Mx_1 - 1) = 0 \quad &\rightarrow \quad x_1 = 0 \text{ or } 1/M \\ Mx_2 - x_1^2 = 0 \quad &\rightarrow \quad x_2 = 0 \text{ or } 1/M^3 \\ &\vdots \\ Mx_n - x_{n-1}^2 = 0 \quad &\rightarrow \quad x_n = 0 \text{ or } 1/M^{2^n-1}. \end{aligned}$$

[3]Actually, another condition is required: see the article for details.

[4]algebraically independent: n real numbers $v_1, ..., v_n$ are said to be *algebraically dependent* if and only if there exists a non-identically zero polynomial $f(x_1, .., x_n)$ with integral coefficients such that $f(v_1, ..., v_n) = 0$. There are algebraically independent if no such polynomial may be found.

[5]This assertion is not entirely true, but let us admit it for the moment.

theoretically result in generating algebraically dependent—because rational—numbers. However, in practice such generators could be used and the method could work with high probability, provided the period of the random generator is long enough.

Another track could be to use smarter random generators, producing truly aperiodic sequences: although we do not know whether this is possible, we do know that there exist deterministic automata that produce aperiodic sequences (*e.g.*, Thue-Morse sequences, [1]). If everything else failed, it would still be possible to generate, once and for all, tables of algebraically independent numbers, in much the same way that random numbers tables were edited before the advent of modern computers.

4. Another idea is to note that in most situations in *CG*, since the depth of the computations is finite (*i.e.*, the algorithms are not re-entrant), it is possible to have a "general upper bound" $[d^+]$ on the degree of the test polynomials involved in a given instance of algorithm. This means that algebraically independence is not necessary, and may be advantageoulsy replaced with the following weaker constraint: define the dependency-degree of n numbers to be the smallest degree $[d_-]$ of the non-identically null polynomial with coefficients in $\mathbb{Z}$ that vanishes at $(v_1, ..., v_n)$. It suffices that the dependency-degree of the generated numbers be bigger than the general upper bound, *i.e.*, $d_- > d^+$.

   What is basically required here is that the dependency-polynomial be *more complex* (in a certain sense) than the test polynomials. As opposed to the first "boolean" and qualitative complexity evaluation (algebraic dependency *versus* algebraic independency), using the degree in the fashion just described is a more quantitative approach. However, there exist other measures for this notion (*e.g.*, Mahler's measure, magnitude of the coefficients, polynomial height, etc.), some of which might prove applicable and useful in this case.

5. Until now, we have assumed that the numbers involved in any test were initial data, not computed values (note that other researchers have made the same assumption [24, 8]). However, it turns out that such a restriction is unconvenient for programmers, who had rather use the same predicate (*i.e.*, the same function call) to compute, for instance, the location of an initial or intersection point relatively to some given line or plane, be it defined by initial points or not: `location(pt, line)` returning `below` or `above`.

6. Allowing initial and computed values in the test polynomials may lead to problems: for instance, a program computes the intersection point I between two (perturbed) lines AB and CD. However, whatever the perturbation on A, B, C, D, points A, B, I are aligned, by definition, and the call to `location(I, AB)` or `location(I, CD)` will never return: indeed, after substitution, the polynomial for `location` is identically null.

   Of course, this is an old problem to programmers in *CG*, who are accustomed to using book-keeping information (I aligned with A and B, and with C and D) rather than letting the exact library recompute it.

7. Re-entrant or on-line methods raise more difficult issues (next section.)

5 Re-entrance

Re-entrant or online methods raise difficult issues, but such methods are rare in *CG*. Imagine, for instance, a method to compute arrangements of lines in 2D allowing the dynamic insertion of new lines through the intersections of previous lines. Let P_1, P_3, P_5 be three intersection points on a first line, and P_2, P_4, P_6 three others, on a second line. Then, by Pappus's theorem, $A = P_1P_2 \cap P_4P_5$, $B = P_2P_3 \cap P_5P_6$ and $C = P_3P_4 \cap P_6P_1$ are aligned (whatever the perturbation of lines $P_1P_3P_5$ and $P_2P_4P_6$). Thus the call to `location(A, BC)` will not return: the underlying polynomial behind this call to `location` is the zero polynomial. Pappus's theorem being not trivial, the corresponding zero is more difficult to predict for the programmer.

The simplest solution is to forbid re-entrant methods: thus points A, B, C of the previous example cannot be used twice, but only some perturbations of them. In this way, no geometric construction can be made! A less authoritarian approach would allow geometric constructions, but then would have to detect resulting zeros, at run time.

The existence of (fast?) detection methods is an open question. It is clear that these zeros are not accidental, but occur whatever the perturbation, *i.e.*, for all data or at least for all data in some open set. Since they are theorems, one may consider detecting them with the help of symbolic computations or gap theorems, but also by means of stochastic computations—in the spirit of the Schwartz's test [20] or of Hong's prover [11]. See also [3, 21, 13] for a more recent work in this wake. Let us give more insight on the previous points:

This problem has several difficulty levels.

1. In the simplest case, the perturbed numbers generated are $v = (v_1, ..., v_k)$ and the computed values

are

$$
\begin{aligned}
x_1 &= f_1(v) \\
x_2 &= f_2(v, x_1) \\
&\vdots \\
x_n &= f_n(v, x_1, ..., x_{n-1})
\end{aligned}
$$

where the f_i's are polynomials with integral coefficients, and the tests involve polynomials $T(v, x_1, \ldots, x_n)$.

2. In a more complex version, the x_i's are implicitly described by:

$$
\begin{aligned}
g_1(v, x_1) &= 0 \\
g_2(v, x_1, x_2) &= 0 \\
&\vdots \\
g_n(v, x_1, ..., x_{n-1}) &= 0
\end{aligned}
$$

where the g_i's are polynomials with integral coeffcients, assumed to be known, and the values of the x_i's are computed by a series, or an algorithm. For instance, $x_1 = \sqrt{v}, (g_1(v, x_1) = x_1^2 - v)$ may be computed using the Egyptian algorithm $(x_1^0 = v, x_1^{k+1} = \frac{x_1^k + v/x_1^k}{2})$.

Fortunately, there are methods to detect unexpected zeros in such settings. For instance, the operations leading to the irruption of a zero as a direct consequence of Pascal's Theorem[6] may be expressed in the previous framework. Let us detail a "stochastic proof" of this theorem:

(a) In a finite field $\mathbb{Z}/p\mathbb{Z}, p$ prime, define a random conic Γ.

(b) Pick 6 random lines, and use their intersections with the conic (one out of two points, at random) to define six points on Γ. Computing the intersecton of a line and the conic requires computing a surd, an operation which proves (im)possible in the finite field in 50% of the trials, on average (draw another line when not lucky).

(c) Check that the three points are aligned.

p-adic variants may also be considered (*i.e.*, computation modulo $p, p^2, p^3, ...$) There exist theorems that allow transforming this heuristic proof into a formal one (said rapidly: use Hong's methods with modular arithmetic, and then apply the Chinese Remainder Theorem). Note that the present argument also applies to the previous, simpler case.

3. The algebraic system is not triangular. Finding a solution in a finite field is harder.

4. In the most general and of course much harder case, the system is no longer algebraic, but uses transcendental (*e.g.*, trigonometric, logarithms, etc.) functions.

A last related question is: is it possible to detect such very special kinds of zeros at compile time, assuming some functional language is used, at least in the simplest cases described above?

6 Conclusion

This paper has hinted at a new solution to imprecision in *CG*. We have discussed how a zero-free arithmetic could be defined, and considered several options for its implementation. We have addressed the problem of detecting undesirable zeros in the case of re-entrant algorithms, and stressed out the fact that, although some methods for detecting such problems could be designed, they should not be confused with theorem-proving strategies, but considered as a means to debug programs. We are considering implementing ZEA in CAML and testing its capabilities in the near future.

Acknowledgements

ZEA is the starting point for a common research action between the authors and two INRIA Projects (PRISME, Sophia-Antipolis (J-D. Boissonnat) and ARENAIRE, Lyons (J-M. Muller)).

[6]Pascal's Theorem: The three intersection points of the opposite sides of an hexagon inscribed in a conic are aligned. (When the conic degenerates into 2 lines, Pascal's theorem reduces to Pappus's theorem).

References

[1] J-P. Allouche and J. Shallit. The ubiquitous Prouhet-Thue-Morse sequence. In C. Ding, T. Helleseth, and H. Niederreiter, editors, *Sequences and Their Applications: Proc. of SETA'98*, pages 1–16. Springer-Verlag, 1999. Available at `http://www.math.uwaterloo.ca/~shallit/Papers/ubiq.ps`.

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

[3] D. Bouhineau. *Construction automatique de figures géometriques et programmation logique avec contraintes.* PhD thesis, LSR-IMAG, Grenoble, France, 1997.

[4] H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. In *Proc. 14th Annu. Symp. on Comput. Geom.*, pages 165–174, 1998.

[5] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Shirra. Efficient exact geometric computation made easy. In *Proc. 15th Annu. Symp. on Comput. Geom.*, pages 341–350, 1999.

[6] C. Burnikel, S. Funke, and M. Seel. Exact geometric predicates using cascaded computations. In *Proc. 14th Annu. ACM Symp. on Comput. Geom.*, pages 175–183, 1998.

[7] J.F. Canny. *The complexity of robot motion planning.* PhD thesis, Massachussetts Institute of Technology, Cambridge, Massachussetts, 1988.

[8] S. Fortune and C. Van Wyk. Efficient exact arithmetic for computational geometry. In *Proc. 9th Annu. ACM Symp. on Comput. Geom.*, pages 163–172, San Diego, May 1993.

[9] S. Funke and K. Mehlhorn. LOOK – a lazy object-oriented kernel for geometric computations. In *Proc. of the 16th Annu. ACM Symp. on Comput. Geom.*, pages 156–165, Hong-Kong, 2000.

[10] P. Gowland and D. Lester. The correctness of an implantation and test of a library for exact arithmetic. In *Proc. 4th Real Numbers and Computers*, pages 125–140, Schloss Dagstuhl, Saarland, Germany, 2000.

[11] J.W. Hong. Proving by example and gap theorem. In IEEE Computer Society Press, editor, *Proc. 27th Annu. IEEE Symp. on FOCS*, pages 107–116, Toronto, Ontario, 1986.

[12] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A core library for robust numeric and geometric computation. In *Proc. 15th Annu. ACM Symp. on Comput. Geom.*, pages 351–359, 1999.

[13] U. Kortenkamp. *Foundations of dynamic geometry.* PhD thesis, Swiss Federal Institute of Technology, Zurich, 1999.

[14] S. Langley and D. Richardson. Exact computations with real algebraic numbers. In *Proc. 3rd Real Numbers and Computers*, pages 167–176, Paris, France, 1998.

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

[16] D. Michelucci and J-M. Moreau. Lazy arithmetic. *IEEE Transactions on Computers*, 46(9):961–975, 1997.

[17] K. Ouchi. Real/expr: implementation of exact computation. Master's thesis, Dept. of Computer Science, New York University, 1997.

[18] D. Richardson. How to recognise zero. *J. Symbolic Computation*, 1996.

[19] Dan Richardson. Multiplicative independence and the uniformity conjecture. Technical report, Bath University, Math. Dpt, 1999. `http://www.bath.ac.uk/~masdr/`.

[20] J.T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, 1980.

[21] D. Tulone, C. Yap, and C. Li. Randomized zero testing of radical expressions and elementary geometry theorem proving. Technical report, Dpt of Computer Science, New York University, december 1999.

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

[23] K. Weihraus. Representations of the real numbers and of the open subsets of the sets of the real numbers. *Annals of pure and applied logic*, 35:247–260, 1985.

[24] C.K. Yap. Symbolic treatment of geometric degenaracies. In *Proc. 13th IFIP Conf. on Sys. Modeling and Optimization*, pages 348–358, Tokyo, 1987.