An ϵ Arithmetic for Removing Degeneracies

D. Michelucci
Ecole Nationale Supérieure des Mines
Saint-Etienne, France 42023

Abstract

Symbolic perturbation by infinitely small values removes degeneracies in geometric algorithms and enables programmers to handle only generic cases: there are a few such cases, whereas there are an overwhelming number of degenerate cases. Current perturbation schemes have limitations, presented below. To overcome them, this paper proposes to use an ϵ-arithmetic, i.e. to represent in an explicit way infinitely small numbers, and to define arithmetic operations $(+, -, *, /, <, =)$ on them.

1 Introduction

Handling all degeneracies is a burden when implementing geometric algorithms: there are a few generic cases, but numerous degeneracies, for instance in *2D* alignments of more than two points, cocircularity of more than three points, intersection of more than two lines in a point, parallelism between lines or between a sweeping line and segments for some algorithms... Symbolic perturbation removes degeneracies, and so programmers, like theoreticians, can ideally focus on the treatment of a few generic cases. However, after reading previous papers on perturbation schemes [4] [5] [13] [14] implementing and using a perturbation library is not so easy a task.

First, whereas the principle of perturbation is very powerful and general, implementations are often restricted to some computations, typically some determinant computations. Next, programmers have to perform some kind of symbolic computations by hand, like expanding perturbed determinants in ϵ^α and they have to foresee what terms in ϵ will be needed at such-and-such moments at run time: a burden.

Yap's perturbation technique, probably the most powerful one, also has limitations. With Yap's formulation, the perturbation 'black-box', when deciding the perturbed sign of some polynomial $F(x_1, x_2 \ldots x_n)$ needs that values of $x_1, x_2 \ldots x_n$ be direct input parameters: they cannot be, say, coordinates of intersection points computed by some algorithm; in Yap's terminology, derivation degree is at most 1. Practically, the programmer cannot use the same function for testing the location of an initial vertex and the location of an intersection point, relatively to some line. Moreover, the programmer has to distinguish between lines: lines defined by two input vertices, by one input vertex and an intersection point, by two intersection points.... And intersection points are also of different kinds, depending on the intersecting lines. So the number of distinct generic cases the programmer has to handle grows very fast indeed, (it is a quite ironic situation, when remembering that the perturbation technique has been introduced to remove overwhelming numbers of degenerate cases) and Yap's formulation is unusable for on-line algorithms. The implementation scheme proposed in this paper solves these problems.

Actual implementations do not explicitly represent ϵ and polynomials in ϵ by data structures in the computer: the ϵ are just conceptual entities, guiding the programmer. I discuss here a straightforward and natural scheme for implementing a perturbation library. The main idea is to *explicitly represent*, in some way, polynomials in ϵ^α, and to suitably redefine the basic arithmetic operations: sum, subtraction, product, division, comparisons, so that *the arithmetic itself will remove degeneracies.* This approach removes limitations on derivation degree; it allows easy implementations and the comparison of several perturbations schemes. Moreover, it gives more control to the user on the perturbation: It becomes easy to specify special perturbations, for instance that a given point must stay on some given straight line, or semi straight line, after perturbation: In previous implementations, the user had very poor control on perturbation (typically and symptomatically described as a 'black box').

Remark: when using approximate arithmetic, degeneracies or near-degeneracies can lead to program crashes, or topological inconsistencies in results. However, taking into account all degeneracies on one hand, and avoiding side effects due to finite-precision arithmetic on the other hand are two distinct problems and must not be confused. Moreover, symbolic perturbation requieres an exact arithmetic. This paper assumes a rational arithmetic is used. It can be a lazy one, as defined in [1, 2, 3]; personally I have used the rational arithmetic library provided by Le-Lisp[1] 15.25. Note that a rational arithmetic is more often sufficient for geometric problems considered in Computational Geometry.

Section 2 gives definitions, notations, and summarizes more or less known properties on ϵ-extensions over a field, compatible orderings and so on. Section 3 proposes to use streams (potentially infinite lists) to

[1]Le-Lisp is a registered trademark by INRIA.

represent expansions in ϵ. Implementation of a naive perturbation scheme is given as a first example, then implementation of Yap's perturbation. Section 4 discusses first experiments.

2 Degeneracies and perturbation

2.1 Geometric degeneracies

In geometric algorithms a degeneracy occurs each time a numerical test on the sign of an arithmetic expression returns 0: generic answers are strictly positive, and strictly negative. This paper supposes, following Yap, that arithmetic expression tests are polynomials. Thus a degeneracy occurs when input parameters are not algebraically independent over $\mathbb{Q}$. As they are rational numbers (floating point numbers are rational numbers), they are never algebraically independent, obviously. The principle of the perturbation technique is to symbolically perturb input parameters by infinitely small deformations, in order to remove all algebraic dependencies between input parameters, or at least to remove some of them; those that can cause degeneracies in such-and-such algorithm. Computations are then performed on perturbed data, and algorithms have to handle only generic cases. A post processing step finally cleans output: typically it merges infinitely close vertices, removes infinitely short edges and so on. Several perturbation schemes have been proposed, in particular by Edelsbrunner and Mücke [4], by Emiris and Canny [5], by Yap [13, 14]. Due to lack of space, I will discuss only the latter; it is probably the most general one.

2.2 Yap's perturbation

Each input parameter x_i is perturbed into $P(x_i) = x_i + \epsilon_i$ where $(\epsilon_1, \epsilon_2, ...\epsilon_n)$ are infinitely small positive numbers. For performing comparisons between perturbed x_i, when unperturbed values are equal, we need an ordering between the ϵ_i, say, for instance:

$$1 \gg \epsilon_1 \gg \epsilon_2 \gg \ldots \gg \epsilon_n$$

where $\gg$ means 'infinitely greater': if $a \gg b$, then for all real values v, $a \gg bv$. Multiplying the last sorted sequence by ϵ_1 implies that:

$$\epsilon_1 \gg \epsilon_1^2 \gg \epsilon_1\epsilon_2 \gg \epsilon_1\epsilon_3 \ldots$$

However, even by multiplying by all possible monomials, some comparisons remain undeterminate: for instance we can freely choose the ordering between ϵ_1^2 and ϵ_2. But if $x_1 = x_2 = 0$, we will need such an ordering to compare the perturbed values of x_1^2 and x_2. So we need a consistent ordering (the good word is 'compatible' ordering) between all the monomials in ϵ_i: 'consistent' means that the ordering must not introduce any contradiction like $a < b < c < a$ where[2] a, b, c are some computed values. Assume for the moment that such an ordering is available.

[2]Of course: $a \ll b \Rightarrow a < b$. $<$ is sometimes used for $\ll$.

Let $X = (x_1, \ldots x_n)$ and $\epsilon = (\epsilon_1, \ldots \epsilon_n)$. Then each polynomial expression $F(X)$ becomes, after perturbation, a polynomial: $F(X+\epsilon)$. By Taylor's expansion:

$$F(X+\epsilon) = F(X) + \sum_{|\alpha|=1}^{deg(F)} \frac{1}{\alpha!}\frac{\partial^{|\alpha|}F(X)}{\partial X^\alpha}\epsilon^\alpha$$

where:

$\alpha = (\alpha_1, \ldots, \alpha_n)$ is a multi-index
$|\alpha| = \alpha_1 + \ldots + \alpha_n$
$\alpha! = \alpha_1! \ldots \alpha_n!$
$x^\alpha = x_1^{\alpha_1} \ldots x_n^{\alpha_n}$.

$F(X+\epsilon)$ is a polynomial in ϵ, and its terms can be sorted by decreasing size of the monomials ϵ^α, such that $\epsilon^{(\alpha)_1} \gg \epsilon^{(\alpha)_2} \gg \epsilon^{(\alpha)_3} \ldots$. The sign of $F(X+\epsilon)$ is thus the sign of the first non null term in the sequence:

$$(F(X), \frac{1}{(\alpha)_1!}\frac{\partial^{|(\alpha)_1|}F(X)}{X^{(\alpha)_1}} \ldots \frac{1}{(\alpha)_i!}\frac{\partial^{|(\alpha)_i|}F(X)}{X^{(\alpha)_i}} \ldots),$$

i.e. the sign of the first non null term in the sequence of the partial derivatives of F, noted for short:

$$(F(X), F_{(\alpha)_1}(X), F_{(\alpha)_2}(X), \ldots F_{(\alpha)_i}(X) \ldots),$$

the α being ordered like the ϵ^α. Of course, this sequence has finite length, and only the identically null polynomal has a sequence with all terms equal to 0.

This last formulation for the sign of $F(X)$ after perturbation does not make reference to any ϵ: it only uses derivatives and any compatible order on monomials x^α. It was first proposed by Yap in [12] and his proof of this perturbation scheme did not use any ϵ.

2.3 Classic compatible orderings

A total and consistent ordering between monomials ϵ^α is needed. 'Consistent' means compatible with multiplication of monomials, *i.e.* for all monomials m, n, k with $k > 0$ (as it is always the case), $m < n \Rightarrow mk < nk$. It is equivalent and simpler to discuss consistent sign of multi-indices, instead of consistent ordering of monomials. Equivalence follows from: $\epsilon^\alpha < \epsilon^\beta \Leftrightarrow \epsilon^{\alpha-\beta} < \epsilon^0 = 1 \Leftrightarrow sign(\alpha - \beta) = +1$. With this terminology, compatibility means that the sum of two positive multi-indices must be a positive multi-index.

The most frequently used compatible signs of multi-indices are the following:

The lexicographic sign (*lex* for short): the *lex* sign of α=$(\alpha_1, \alpha_2 \ldots \alpha_n)$ is the sign of the first non null α_i met in the sequence: $\alpha_1, \alpha_2 \ldots \alpha_n$. When all α_i are null, α has *lex* sign 0. For example, with (α_1, α_2) and total degree smaller than 2:

$$(0,0) < (0,1) < (0,2) < (1,0) < (1,1) < (2,0)$$
$$\Rightarrow 1 \gg \epsilon_2 \gg \epsilon_2^2 \gg \epsilon_1 \gg \epsilon_1\epsilon_2 \gg \epsilon_1^2.$$

The inverse lexicographic sign (*ilex* for short): the *ilex* sign of α=$(\alpha_1, \alpha_2 \ldots \alpha_n)$ is the *lex* sign of: $(\alpha_n, \alpha_{n-1} \ldots \alpha_1)$. For example:

$$(0,0) < (1,0) < (2,0) < (0,1) < (1,1) < (0,2)$$
$$\Rightarrow 1 \gg \epsilon_1 \gg \epsilon_1^2 \gg \epsilon_2 \gg \epsilon_1\epsilon_2 \gg \epsilon_2^2.$$

The total then lexicographic sign (*tlex* for short): the *tlex* sign of $\alpha{=}(\alpha_1, \alpha_2 \ldots \alpha_n)$ is the *lex* sign of $(|\alpha|, \alpha_1, \alpha_2 \ldots \alpha_n)$, with $|\alpha| = \alpha_1 + \alpha_2 + \ldots \alpha_n$. It is also the *lex* sign of $(|\alpha|, \alpha_1, \alpha_2 \ldots \alpha_{n-1})$. For example:

$$(0,0) < (0,1) < (1,0) < (0,2) < (1,1) < (2,0)$$
$$\Rightarrow 1 \gg \epsilon_2 \gg \epsilon_1 \gg \epsilon_2^2 \gg \epsilon_1\epsilon_2 \gg \epsilon_1^2.$$

The total then inverse lexicographic sign (*tilex* for short): the *tilex* sign of $\alpha{=}(\alpha_1, \alpha_2 \ldots \alpha_n)$ is the *ilex* sign of $(\alpha_1, \alpha_2 \ldots \alpha_n, |\alpha|)$. It is also the *ilex* sign of $(\alpha_2 \ldots \alpha_n, |\alpha|)$. For example:

$$(0,0) < (1,0) < (0,1) < (2,0) < (1,1) < (0,2)$$
$$\Rightarrow 1 \gg \epsilon_1 \gg \epsilon_2 \gg \epsilon_1^2 \gg \epsilon_1\epsilon_2 \gg \epsilon_2^2.$$

2.4 Ordering representation

Let $(\Omega_0, \Omega_1, \ldots \Omega_n)$ be infinitely big numbers, ordered by some compatible order. We say we have a 'representation' of this order if we can express all Ω_i, and all their power products, with a single, unique infinite integer, say: ω. Having a single infinite number ω is interesting, because the ordering is obvious and intuitive:

$$\ldots \omega^2 \gg \omega^1 \gg 1 \gg \omega^{-1} \gg \omega^{-2} \ldots$$

There is no more the question of ordering in a consistent way Ω_0^2 and Ω_1 and so on. Then this representation can give a more intuitive insight into orders and infinitesimals. I discuss infinitely big numbers for convenience of notation but results are easily extended to infinitely small numbers by considering negative powers of ω and Ω_i. Suppose we set:

$$\Omega_0 = \omega,\ \Omega_1 = \Omega_0^\omega,\ \Omega_2 = \Omega_1^\omega\ \ldots,\ \Omega_k = \Omega_{k-1}^\omega$$

Here it becomes convenient to define a logarithm in basis ω:

$$L(x) = log(x)/log(\omega).$$

Then we have:

$$\begin{aligned} &L(\Omega_0) = L(\omega) = 1 \\ &L(\Omega_1) = L(\Omega_0^\omega) = \omega \\ &L(\Omega_2) = L(\Omega_1^\omega) = \omega^2 \\ &\vdots \\ &L(\Omega_k) = L(\Omega_{k-1}^\omega) = \omega^k \end{aligned}$$

To compare monomials in Ω_i, we now use Logarithms. For instance:

$$L(\Omega_3^5 \Omega_1^3 \Omega_0^2) = 5\omega^3 + 3\omega + 2 = [5 \quad 0 \quad 3 \quad 2]_\omega$$

We see that the Logarithm of a monomial is just a number expressed in basis ω, and it is also the multi-index of the monomial, but written in reverse order. So power products of these Ω_i are ordered in reverse lexicographic order.

I now give a representation for a total then inverse lexicographic order. Let (G_0, G_1, G_2) be 3 (the generalization is left to the reader) infinitely big numbers, ordered this way, that is:

$$G_0 \ll G_1 \ll G_2 \ll G_0^2 \ll G_0G_1$$
$$\ll G_1^2 \ll G_0G_2 \ll G_1G_2 \ll G_2^2 \ll G_0^3 \ldots$$

We have:

$$\begin{aligned} &G_0^{t_0} G_1^{t_1} G_2^{t_2} > 1 \\ \Leftrightarrow \quad &(t_0 \quad t_1 \quad t_2) >_{tilex} (0 \quad 0 \quad 0) \\ \Leftrightarrow \quad &(t_0 \quad t_1 \quad t_2 \quad t_0+t_1+t_2) >_{ilex} (0 \quad 0 \quad 0 \quad 0) \\ \Leftrightarrow \quad &(t_1 \quad t_2 \quad t_0+t_1+t_2) >_{ilex} (0 \quad 0 \quad 0) \end{aligned}$$

Here we can suppress t_0, because scanning the sequence: $t_0 + t_1 + t_2, t_2, t_1, t_0$, one must find a first term greater than 0, all the previous being null, by definition of this order. When in fact t_0 cannot be the first term greater than 0: otherwise $t_0 + t_1 + t_2$ would be non nul, a contradiction. Now:

$$\begin{aligned} &(t_1 \quad t_2 \quad t_0+t_1+t_2) >_{ilex} (0 \quad 0 \quad 0) \\ \Leftrightarrow \quad &\Omega_0^{t_1} \Omega_1^{t_2} \Omega_2^{t_0+t_1+t_2} > 1 \end{aligned}$$

So one representation amongst others for (G_0, G_1, G_2) can be obtained by identifying $\Omega_0^{t_1} \Omega_1^{t_2} \Omega_2^{t_0+t_1+t_2}$ and $G_0^{t_0} G_1^{t_1} G_2^{t_2}$. Thus:

$$\begin{aligned} &G_0 = \Omega_2 \text{ and } L(G_0) = \omega^2 \\ &G_1 = \Omega_0\Omega_2 \text{ and } L(G_1) = \omega^2 + 1 \\ &G_2 = \Omega_1\Omega_2 \text{ and } L(G_2) = \omega^2 + \omega \end{aligned}$$

The reader can verify that, as it must be:

$$L(G_0) < L(G_1) < L(G_2) < L(G_0^2) < L(G_0G_1)$$
$$< L(G_1^2) < L(G_0G_2) < L(G_1G_2) < L(G_2^2) < \ldots$$

This section has shown that:

$$\alpha >_{tilex} (0\ 0\ 0) \Leftrightarrow \alpha \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} >_{ilex} (0\ 0\ 0)$$

and a natural question arises: is any compatible order σ on multi-index reducible to *ilex* order by a linear transformation on multi-index, *i.e.* is there a matrix M_σ such that $\alpha >_\sigma (0 \ldots 0) \Leftrightarrow \alpha M_\sigma >_{ilex} (0 \ldots 0)$? The answer is obviously positive for *lex*, *ilex*, *tlex* and *tilex* order (for instance M_{lex} is the n by n matrix obtained by reversing columns of the identity matrix). In fact, it is a general property of all compatible orders, as next section shows.

2.5 Compatible signs properties

This section summarizes known properties of all compatible signs, already published in various forms; see for instance [10]. It is convenient to see multi-indices as points in $\mathbb{Z}^n$. The goal is to partition $\mathbb{Z}^n$ in P, $-P$ and 0_n: P is the set of positive multi-indices, having sign $+1$; $-P$ is the set of negative multi-indices, having sign -1; $0_n = (0, 0 \ldots)$ is the only multi-index with sign 0. Sometimes notation P_σ will be used for the set of positive points of order σ. Clearly, P and $-P$ are symmetrical relatively to 0_n. We have the following properties:

Property P_1. $\alpha \in P, \beta \in P \Rightarrow \alpha + \beta \in P$.

Proof: compatibility.

Property P_2. $\alpha \in P, \lambda \in I\!N^+ \Rightarrow \lambda\alpha \in P$.

Proof: $\lambda\alpha = \alpha + \alpha + \ldots, \lambda$ times; then P_1.

Property P_3. $\lambda\alpha \in P$, $\alpha \in \mathbb{Z}^n$ and $\lambda \in I\!N^+ \Rightarrow \alpha \in P$.

Proof: clearly $\alpha \neq 0_n$; suppose $\alpha \notin P \Rightarrow -\alpha \in P \Rightarrow \lambda(-\alpha) \in P \Rightarrow \lambda\alpha \notin P$: a contradiction.

Property P_4. P and $-P$ are convex.

Proof: $\alpha \in P$, $\beta \in P$, and suppose $\gamma = \frac{a}{b}\alpha + \frac{b-a}{b}\beta \in \mathbb{Z}^n$ with $0 < a < b$, a and $b \in I\!N^+$. Then $a\alpha \in P$ by P_2, and $(b-a)\beta \in P$ by P_2; thus $a\alpha + (b-a)\beta \in P$ by compatibility (P_1). Thus $\gamma \in P$ by P_3.

Now it is clear that P and $-P$ are, roughly speaking, two halfspaces of $\mathbb{Z}^n$. More precisely, there is a hyperplane H_{n-1} with dimension $n-1$, crossing 0_n, such that all points on one side are positive, all points on the other side are negative. Recursively, there is an hyperplane H_{n-2} in H_{n-1}, crossing 0_n, such that all points of H_{n-1} on one side of H_{n-2} are positive, all points on the other side are negative, and so on, by iteration on dimension, until hyperplane $H_0 = 0_n$ is reached.

Thus it is possible, for all compatible ordering and sign σ, to find a 'good' basis for σ: $(b_1, b_2 \ldots b_n)$ where multi-index x has coordinates $(x_1, x_2 \ldots x_n)$ and where σ sign of x is nothing else but, say, the lexicographic sign of $(x_1, x_2 \ldots x_n)$ –supposing *lex* is your favorite compatible sign. Such a good basis is built as follow: If $n = 1$, take for b_1 any positive point for σ. Otherwise let H_{n-1} be the hyperplane with only negative points on one side, and only positive points on the other side; let B_{n-1} be a good basis of H_{n-1} and P_n any positive point not lying in H_{n-1}. Then (P_n, B_{n-1}) is a good basis. Thus:

Property P_5. All compatible orders have a good basis.

In other words, all compatible orders on multi-indices are reduced to *lex* (or *ilex* or *tlex* or *tilex* or ...) order by a linear transformation. Thus all compatible orders have a representation.

Any ordering has a lot of good bases. Suppose that a multi-index has coordinates $(x_1, x_2 \ldots x_n)$ in a good basis, and $(y_1, y_2 \ldots y_n)$ in another good basis. Let M be the n by n matrix such that $y = xM$. Clearly M can not be any invertible matrix. More precisely:

M must 'work' for all x, *i.e.* for all x, $y = xM$ must have the same *lex* sign as x. Either $x_1 \neq 0$, and since

$y_1 = \sum_{i=1}^{n} x_i M_{i,1}$

must have same sign as x_1, the only possibility is $M_{1,1}$ to be strictly positive, and $M_{i>1,1}$ to be 0. Either $x_1 = 0$, and so $y_1 = 0$; in this case, the sign of x is the *lex* sign of $(x_2, x_3 \ldots x_n)$, and $(y_2, y_3 \ldots y_n)$ must have the same *lex* sign.

$$y_2 = \sum_{i=1}^{n} x_i M_{i,2} = x_1 M_{1,2} + \sum_{i=2}^{n} x_i M_{i,2}$$
$$= \sum_{i=2}^{n} x_i M_{i,2}.$$

So $M_{1,2}$ does not matter, $M_{2,2}$ must be positive, and $M_{i>2,2}$ must be null. And so on. Thus:

Property P_6. Multi index $y = xM$ has same *lex* sign as x iff M is a n by n upper triangular matrix, with all diagonal elements strictly positive.

3 A new approach: an ϵ-arithmetic

To explicitly represent ϵ expansions, a natural idea is to use 'streams' *i.e.* (potentially) infinite lists. Some kinds of streams and continuations have already been used in an explicit way to represent continued fraction expansions [11], and in an implicit way in on-line arithmetic [6].

3.1 Lazyness and streams

Streams are already available in functional languages [8], such as LazyML, Scheme, Haskell, Miranda to cite a few. These languages allow delaying computations while their results are not needed, and so they allow us to define and use streams, lists of elements which are created only when needed. With such languages, implementing a perturbation library is very easy. However, in the real world, these languages are still considered much too exotic. So I now describe how streams can be partially simulated in usual programming languages such as Fortran, Pascal, C, C++, Lisp, Smalltalk, Modula ...These streams cannot be as powerful as the inspiring ones; but they will be sufficient for the needs of the perturbation technique.

Using object-oriented terminology, a stream s of items is an object answering the following questions:

Available?(s): Is the first item of s available ?

Head(s), *Tail*(s): If available, s returns a pointer on its first item, and a pointer on its tail: another stream.

Expert(s): Otherwise, s returns a pointer on its 'expert': e. *Expert* e is an object, may be with some private fields the description of which depends on its exact class. These fields will enable e to answer to the following messages:

ComputeHead(e): e computes the first item of s.

ComputeTail(e): e computes the tail of s.

Finally, a stream s replies to two other messages:

ForceHead(s), *ForceTail*(s): s returns its head, or its tail, be they available or not; if not, s forces evaluation by using its expert e.

3.2 A naive perturbation

As an example, I first present a naive perturbation scheme, where each input parameter x is perturbed into a sequence with *infinite length*:

$$x + x_1\epsilon^1 + x_2\epsilon^2 + \ldots,$$

where x_i are by default random integers or rational numbers. There is only one ϵ, the same for all x: so there is no more ordering problems. $x + x_1\epsilon + x_2\epsilon^2 \ldots$ is represented by a stream $(x, x_1, x_2 \ldots)$, beginning by the non perturbed value x, and whose expert is able to generate random integers (or rational) x_i when asked for. Addition and multiplication must be redefined to cope with such ϵ streams. Division can be defined but is useless (see below). For conciseness, I detail only multiplication:

$$(x_0 + x_1\epsilon + x_2\epsilon^2 + \ldots)(y_0 + y_1\epsilon + y_2\epsilon^2 + \ldots)$$
$$= x_0y_0 + (x_0y_1 + x_1y_0)\epsilon + (x_0y_2 + x_1y_1 + x_2y_0)\epsilon^2 + \ldots$$

or with another notation, more recursive:

$$(x_0 + \epsilon X_1)(y_0 + \epsilon Y_1) = x_0y_0 + \epsilon(x_0Y_1 + y_0X_1 + \epsilon X_1Y_1)$$

The product of two streams is a new allocated stream, whose expert has class (say) *ExpertProduct*; this expert has two private fields X_0 and Y_0 referencing the two operands. In *ExpertProduct* class, methods $ComputeHead(e)$ and $ComputeTail(e)$ first recover streams X_0 and Y_0 to be multiplied, then ask for first items of X_0 and Y_0: $x_0 = ForceHead(X_0)$ and $y_0 = ForceHead(Y_0)$. Read access to x_0 and y_0 can force some evaluations but recursion automatically takes care of that. $ComputeHead(e)$ then returns x_0y_0. $ComputeTail(e)$ asks for tails of X_0 and Y_0: $X_1 = ForceTail(X_0)$ and $Y_1 = ForceTail(Y_0)$, and then returns the new stream: $a_0Y_1 + b_0X_1 + stream(0, X_1Y_1)$; $stream(f, s)$ built a stream whose first item is available element f, and whose tail is stream s. Note method $ComputeTail()$ recursively calls addition of streams and multiplication between two streams or between a scalar and a stream.

3.3 Computing sign

The sign of a stream is the sign of its first non null item: this method is trivial to implement and generally works fine. However, the perturbation technique does not remove all nullities, because $x_k - x_k$ is always null, and $h(x_k) - h(x_k)$ too, where $h()$ is any function. This kind of thing can happen, for instance, when testing the location of a vertex P relatively to some segment s, incident to P, by naively computing the sign of $A_sX_P + B_sY_P + C_s$. Note that it cannot happen with other segments, non incident to P, because of the perturbation. When performing such a pathological test, the user of a naive perturbation will face a problem: the comparison will never stop. There are two possible viewpoints:

1. Pathological tests are mistakes of the programmer. In the previous example, the informed programmer must first verify if segment s is not incident to the vertex by using topological information stored in his data structures. Possibly the library provides a (possibly slow) debug mode to detect pathological tests.

2. The library must detect all equalities or nullities in finite time, and quickly: the programmer is allowed to use pathological comparisons.

Detecting nullity can be done with few symbolic computations: each stream x_k must be considered as a symbolic variable, and operations +, -, *, / only create multivariate polynomials $p(x_1, x_2, \ldots x_n)$ or rational polynomial functions $p(x_1, x_2, \ldots x_n)/q(x_1, x_2, \ldots x_n)$. By hypothesis (the perturbation), all these variables are not only different, but also algebraically independent; that means they never verify some algebraic equation, like, say $x_1^2 - x_2 = 0$. So the only way for a polynomial $p(x_1, x_2, \ldots x_n)$ (or a rational function) to be null is to be identically null. Symbolic computations of sum, product, division on rational polynomial functions are straightforward, once the symbolic definition of the streams at hand is available -as it is the case here. Though simple, these symbolic computations are slow and so the second approach, allowing pathological tests, seems not to be practical, unless we use a fast but *probabilistic* test: if $p(x_1, x_2, \ldots x_n) = 0$ modulo a big enough prime integer for random x_i, then p is likely identically null [2, 7, 9]. This approach remains to be investigated.

3.4 Implementing Yap's perturbation

Let us now study implementing Yap's perturbation with ϵ-streams. First these streams have always *finite length*. So there is no termination problem when testing the sign of a stream, or comparing two equal streams. *However, to ensure efficiency, the programmer always has to avoid pathological tests.*

Then stream items are terms $a_\alpha\epsilon^\alpha$, where a_α is a rational number, and ϵ^α is a monomial ($\epsilon_1^{\alpha_1}$, $\epsilon_2^{\alpha_2}$, $\epsilon_3^{\alpha_3} \ldots \epsilon_n^{\alpha_n}$). Each term is thus represented by a rational number: a_α and by a multi-index: α. A multi-index α is represented by, say, a list (or a dynamically allocated array) of couples $(i, \alpha_i \neq 0)$ representing $\epsilon_i^{\alpha_i}$; i and α_i are integers. When using *lex* (*ilex*) ordering, it is convenient to sort couples by increasing (decreasing) i. When using *tlex* or *tilex* ordering, total degree can be inserted first in this list. This multi-index representation is of course more compact than a vector of n degrees: $(\alpha_1, \alpha_2 \ldots \alpha_n)$ (n the number of input parameters), because most often used monomials are very sparse; moreover, in on-line applications, n evolves at run time.

Multiplication is now detailed. Expert e for computing the first term and the tail of the product $C = AB$ contains in its private fields pointers on the two operands A and B. Let $A = a\epsilon^\alpha + A'$ and $B = b\epsilon^\beta + B'$ where $a\epsilon^\alpha$ is obtained by calling $ForceHead(A)$, A' by calling $ForceTail(A)$, and symmetrically for $b\epsilon^\beta$ and B'. To compute the first term of AB, $ComputeHead(e)$ returns $ab\epsilon^{\alpha+\beta}$. To compute the tail of AB, $ComputeTail(e)$ returns the new stream: $a\epsilon^{\alpha}B' + b\epsilon^{\beta}A' + A'B'$, recursively using additions and multiplications.

The expert for adding previous streams A and B behaves as follows: if $\epsilon^{\alpha} \gg \epsilon^{\beta}$, *ComputeHead* returns $a\epsilon^{\alpha}$, otherwise if $\epsilon^{\alpha} \ll \epsilon^{\beta}$, it returns $b\epsilon^{\beta}$, otherwise $(\alpha = \beta)$ it returns[3] $(a+b)\epsilon^{\alpha}$. Following these cases, *ComputeTail* returns: $A' + B$, or $A + B'$ or $A' + B'$.

In the worst case, Yap's perturbation is inefficient, as all terms of the stream have to be computed; the stream has $O(n^d)$ terms, with n the number of occuring parameters and d the degree of the polynomial represented by the stream. However, in practice, the worst case seldom happens, or, more precisely, it happens only with a naive programming style, *i.e.* when performing pathological tests. A suitable programming style (called 'informed' style in the sequel) easily avoids such bugs.

3.5 Another implementation

Another way to implement Yap's perturbation uses towers of infinitesimal extensions:

$$K_0 = \mathbb{Q},\ K_1 = K_0[\epsilon_1],\ K_2 = K_1[\epsilon_2]\ \ldots.$$

Suppose an arithmetic $(+, -, *, <, =)$ exists to compute in an ordered ring K; initially, $K = K_0 = \mathbb{Q}$. An infinitesimal extension of K by a symbolic variable ϵ, noted $K[\epsilon]$, is a new ordered ring, whose elements are polynomials $p(\epsilon) = \sum a_i \epsilon^i$, with $a_i \in K$. Definition of sum and product is straightforward (mathematically speaking, $K[\epsilon]$ is a vector space over K, with infinite dimension). Moreover, we decide that ϵ is positive and smaller than all positive values in K. So the sign of $p(\epsilon) = \sum a_i \epsilon^i$ is the sign of the first non vanishing term a_i, with a_i sorted by increasing i. Thus all operations $(+, -, *, <, =)$ in $K[\epsilon]$ have been reduced to operations in K.

In this last implementation, an element of $K_i = K_{i-1}[\epsilon_i]$ is thus represented by a stream of elements of K_{i-1}. For instance, the polynomial represented in the previous implementation by stream (with, say, *lex* order):

$$a_{0,0} + a_{0,1}\epsilon_2 + a_{0,2}\epsilon_2^2 + a_{1,0}\epsilon_1 + a_{1,1}\epsilon_1\epsilon_2 + a_{2,0}\epsilon_1^2$$

will now be represented by:

$$[a_{0,0} + a_{1,0}\epsilon_1 + a_{2,0}\epsilon_1^2] + [a_{0,1} + a_{1,1}\epsilon_1]\epsilon_2 + [a_{0,2}]\epsilon_2^2.$$

This last formulation implicitly uses *ilex* order.

3.6 User's control on perturbation

Perturbation schemes were previously considered to be 'black boxes', thus users had no control on perturbation: for instance it was impossible to control the perturbation sign for each parameter, or to perturb a point in such a way so that it stays on a given straight line or semi straight line. With our ϵ-arithmetic, it is very easy for the user to specify the sign of a perturbation. He just writes (up to Lisp syntax): $x + \epsilon$, or $x - \epsilon$ as he needs.

Suppose now p and q are two given points, P and Q are corresponding perturbed values of p and q; a point M is needed, such that M is very close to P and belongs to perturbed edge PQ. First create ϵ_M (or reuse a previous ϵ); then define $M = P + \epsilon_M(Q - P)$, that is (with non homogeneous coordinates, say): define $M_x = P_x + \epsilon_M(Q_x - P_x)$ and $M_y = P_y + \epsilon_M(Q_y - P_y)$. Of course, it works also with the naive perturbation, and with homogeneous coordinates. Thus the control by the user is straightforward.

3.7 Handling division

There are two simple ways to handle division. The first is to use homogeneous coordinates. The second is to represent a perturbed number not by one ϵ-stream, but by two ϵ-streams: a numerator and a denominator; the rest (implementing $+, -, *, /, <, =$) is obvious. Note, however, that it is not possible to reduce a fraction $\frac{a}{b}$ with a and b two streams: the computation of $gcd(a, b)$ will force evaluations.

3.8 Do we really need infinite ?

For convenience and simplicity, ϵ, ω and Ω used in this paper have been first defined as infinitely small or infinitely big numbers. However, we do not need so much: it is sufficient that all real values met during computation are greater (smaller) than ϵ (ω).

Suppose for instance the naive perturbation is used. Roughly speaking, it is like computing with numbers expressed in basis $\omega = \epsilon^{-1}$ big enough to assure no carry can occur during computation. Some digits can be negative, as in redundant notation used in on-line arithmetic [6]. Let m be such that all digits met at run time have absolute value lower than m. Once the computation is done, m can be known; thus it is possible to assign an actual numeric value to ω and ϵ To preserve order between ϵ expansions (consider for instance $\omega - m > m$), it suffices to choose a basis ω greater than $2m$. Then it is possible to visualize perturbed data as ordinary ones (say, for debuging purposes). The same holds for Yap's arithmetic.

4 Experiments

Yap's and naive ϵ-arithmetic have been tested with a simple *2D* geometric algorithm: it detects generic intersection points between n segments with $\frac{1}{2}n(n-1)$ tests, and uses homogeneous coordinates (x, y, h) with h always positive. Coordinates of the $2n$ non perturbed vertices are set with a random rational value amongst v distinct possible ones: with $v = 1$, all vertices are confused and the situation is highly degenerate; at the other end of the spectrum, when v is much greater than $\sqrt{n}$, there is next to no degeneracy.

[3]It is better to remove null terms: $0\epsilon^{\alpha}$.

v	1	2	3	5	10	15	20
Yap/Rat	2.	14.	10.	6.	5.	4.	4.
Naive/Rat	125.	14.	8.	5.	5.	4.	4.

Table 1: Time ratios, $n = 50$ segments.

v	1	2	3	5	10	15	20
i_g	0	28	146	242	287	304	224
i_ϵ	0	202	242	338	321	318	238
x_t	0	835	511	547	429	350	273
x_m	0	4.13	2.11	1.61	1.33	1.10	1.15
$x_{[\,]}$	0, 0	0, 10	0, 10	0, 10	0, 7	0, 4	0, 4
h_t	0	798	502	546	429	350	273
h_m	0	3.90	2.07	1.61	1.33	1.10	1.15
$h_{[\,]}$	0, 0	0, 9	0, 9	0, 9	0, 7	0, 4	0, 4

Table 2: Yap's ϵ-arithmetic, $n = 50$ segments.

Table 1 shows that, with $n = 50$ segments, versions with ϵ-arithmetic are roughly speaking 15 times slower[4] than the pure rational one with very highly degenerate data, and 4 or 5 times slower with a 'normal rate of degeneracies' or next to no degeneracy. Using greater values for n, or another algorithm, does not change orders of these ratios a lot.

In tables 2 and 3, i_g is the number of generic intersection points, detected by the algorithm with pure rational arithmetic; i_ϵ the number of intersection points when using ϵ-arithmetic and the same algorithm; x_t the total length of streams for x coordinates of all intersection points; $x_m = x_t/i_\epsilon$ the average length; $x_{[\,]}$ the min and max length. Idem for h_t, h_m, $h_{[\,]}$ with h coordinates. When using naive ϵ-arithmetic, values h_t, h_m and $h_{[\,]}$ are always equal to x_t, x_m and $x_{[\,]}$, so they are omitted in Table 3.

Full length (*i.e.* length with eager evaluation) for x, y streams is 33, and 17 for h streams, when using Yap's perturbation of course: with naive perturbation, full length is infinite and eager evaluation never stops.

In tables 1 and 2, *lex* ordering was used with Yap's ϵ-arithmetic: Experimentally, stream length is the more often a little shorter with *lex* than with *tlex* ordering. Due to lack of space, these facts and others experimentations (sorting intersection points by x coordinate, using other algorithms, and so on) are not detailed.

It's funny to see that y coordinates of intersection points are never computed, before display or other use: to verify that the intersection point between two lines belongs to a segment, the algorithm uses first abscissa, and then ordinates in case of a vertical line; but due to the perturbation, there are no more such lines...

[4]When ignoring the irrelevant case $v = 1$.

v	1	2	3	5	10	15	20
i_g	0	28	146	242	287	304	224
i_ϵ	347	328	328	315	328	315	240
x_t	1399	879	572	403	389	326	264
x_m	4.03	2.68	1.74	1.28	1.19	1.03	1.10
$x_{[\,]}$	0, 6	0, 4	0, 4	0, 3	0, 4	0, 2	0, 3

Table 3: Naive ϵ-arithmetic, $n = 50$ segments.

I have prefered simplicity to efficiency when implementing because my goal was mainly to validate the idea of an ϵ-arithmetic: a clever implementation would probably be faster. Anyway, following these first experiments, and when an informed programming style is used to avoid pathological tests, space and time overhead due to the ϵ-arithmetic is roughly constant relatively to pure rational arithmetic versions. Of course, pure rational arithmetic is itself much slower than native floating point arithmetic, but this is another problem, already treated in [1].

5 Conclusion

This paper has promoted the use of an ϵ-arithmetic to remove geometric degeneracies at the lowest level. This scheme makes easy and obvious the perturbation control by the user, and it removes the limitation on the derivation degree. This paper has pointed out that an informed programming style is needed to avoid pathological tests. Last but not least, this scheme is compatible with the use of a lazy rational arithmetic[5]. Thus ϵ-arithmetic appears to be an interesting tool for Computational Geometry. However this first implementation is still too slow for practical use, and further work is needed to obtain faster implementations.

A lot of packages and programming environments already provides arithmetics on big integers or big floats. I think they will soon provide exotic arithmetics, such as lazy exact arithmetic or ϵ-arithmetic.

Acknowledgements

I don't know how to thank the anonymous referees for correcting about one hundred English glitches.

[5]Empirical tests have not been performed.

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, P. Jaillon, D. Michelucci, and J-M. Moreau. Hashing Lazy Numbers. *Computing*, 53(3–4):205–217, 1994.

[3] M.O Benouamer, D. Michelucci, and B. Péroche. Error-Free Boundary Evaluation based on a Lazy Rational Arithmetic: a Detailed Implementation. *Computer-Aided Design*, 26(6):403–416, June 1994.

[4] H. Edelsbrunner and E.P. Mücke. Simulation of Simplicity: a Technique to Cope with Degenerate Cases in Geometric Algorithms. *ACM Trans. Graph*, 9:66–104, 1990.

[5] E. Emiris and J. Canny. An Efficient Approach to Removing Geometric Degeneracies. In *Proc. 8th ACM Symp. on Comp. Geometry*, pages 74–82, Berlin, Germany, 1992.

[6] Ercegovac M. On-Line Arithmetic : an Overview. *SPIE Real Time Signal Processing*, 495(VII):86–93, 1984.

[7] W.A. Martin. Determining the Equivalence of Algebraic Expressions by Hash Coding. *J. ACM*, 18(4), 549–558, 1971.

[8] C. Reade. Elements of Functional Programming. *Addison Wesley*, 1989.

[9] J.T. Schwartz. Fast Probabilistic Algorithms for Verification of Polynomial Identities. *J. ACM*, 27, 701–717, 1980.

[10] F. Schwarz. Monomial Orderings and Grobner Bases. *SIGSAM*, 25(1):10–23, 1991.

[11] J.E. Vuillemin. Exact Real Computer Arithmetic with Continued Fractions. *IEEE Trans Computers*, 39(8):1087–1105, 1990.

[12] C.K. Yap. Symbolic Treatment of Geometric Degenaracies. In *Proc. 13th IFIP Conf. on Sys. Modeling and Optimization*, pages 348–358, Tokyo, 1987.

[13] C.K. Yap. A Geometric Consistency Theorem for a Symbolic Perturbation Scheme. *J. Comput. Syst. Sci.*, 40:2–18, 1990.

[14] C.K. Yap. Symbolic Treatment of Geometric Degenaracies. *J. Symbolic Comput*, 10:349–370, 1990.