Theoretical Informatics and Applications Will be set by the publisher
Informatique Théorique et Applications

FORMALLY CERTIFIED FLOATING-POINT FILTERS FOR HOMOGENEOUS GEOMETRIC PREDICATES

Guillaume Melquiond[1] and Sylvain Pion[2]

Abstract. Floating-point arithmetic provides a fast but inexact way of computing geometric predicates. In order for these predicates to be exact, it is important to rule out all the numerical situations where floating-point computations could lead to wrong results. Taking into account all the potential problems is a tedious work to do by hand. We study in this paper a floating-point implementation of a filter for the orientation-2 predicate, and how a formal and partially automatized verification of this algorithm avoided many pitfalls. The presented method is not limited to this particular predicate, it can easily be used to produce correct semi-static floating-point filters for other geometric predicates.

1991 Mathematics Subject Classification. 65G50,68Q60,65D18.

1. INTRODUCTION

Computational geometry algorithms, such as convex hull computations, Delaunay triangulations and arrangements computations, are notoriously sensitive to numerical instability. The most important characteristic of these algorithms is that they mix numerical computations, with combinatorial ones. Their input is for example a set of points in the plane or in space, given by their coordinates, and their output contains a combinatorial part such as a graph: a simple linear

Keywords and phrases: Geometric predicates, semi-static filters, formal proofs, floating-point

[1] Laboratoire de l'Informatique du Parallélisme
UMR 5668 CNRS, ENS Lyon, INRIA, UCBL
46 allée d'Italie, 69364 Lyon Cedex 07, France
E-mail: Guillaume.Melquiond@ens-lyon.fr

[2] INRIA Sophia Antipolis,
2004 route des Lucioles, BP 93, 06902 Sophia Antipolis, France
E-mail: Sylvain.Pion@sophia.inria.fr

sequence for the 2-dimensional convex hull, or more complex ones in the case of triangulations. Many such algorithms are gathered in the CGAL[1] library [3].

In order to derive a combinatorial, discrete, structure from a set of numerical inputs, a set of functions are used, called geometric predicates, which evaluate the relative positions of a few geometric objects, such as the orientation of three points in the plane. From the implementation point of view, using floating-point approximate arithmetic to evaluate these predicates has shown to be the source of many non-robustness problems, because the incorrect geometry of these approximate predicates violates basic geometric theorems on which the algorithms rely [8]. One of the most appreciated solutions to this problem, due to its generality, is to render these predicates exact, thus following the Exact Geometric Computation paradigm [14].

The use of exact multiprecision arithmetic instead of floating-point provides the necessary exactness, but it is not usable naively in practice, due to considerable efficiency loss. Therefore an additional step has been developed in what is called arithmetic filters, which are a way to filter out easy cases of the predicates using certified floating-point arithmetic, calling the slow exact arithmetic as a last resort, far less often. There are different kinds of filters, varying in efficiency and precision. Static filters have been first developed [6], and rely on static forward analysis of error propagation. They tend to be restricted in the input, but are fast. Many variants exist [2, 5, 12]. Dynamic filters are easier to use and more general, but slower [1].

Static filters variants use floating-point error propagation, which are risky and error prone when done by hand. Some more or less automatic tools have been developed to produce their code [2,7,9,11], but it remains an error-prone task due to the complex nature of floating-point arithmetic.

In this paper, we detail the formal proof of a particular kind of static filters, which is implemented in CGAL [3]. We focus on the 2-dimensional orientation of points as predicate, since it is one of the most critical. We then show that the techniques apply similarly to many more important other predicates. And we conclude with a comparison to other existing methods.

2. ORIENTATION-2 PREDICATE

This predicate is one of the most often encountered geometric predicates. Given three points p, q, and r in the plane, it answers if they are collinear, or if they are clockwise or counter-clockwise oriented. If the Cartesian coordinates of the three points are known, the answer is given by the sign of a 3×3-determinant of the coordinates, or the sign of a 2×2-determinant of the vectors.

$$\mathrm{orient}_2(p,q,r) = \begin{vmatrix} p_x & q_x & r_x \\ p_y & q_y & r_y \\ 1 & 1 & 1 \end{vmatrix} = \begin{vmatrix} q_x - p_x & r_x - p_x \\ q_y - p_y & r_y - p_y \end{vmatrix}$$

[1] http://www.cgal.org/

2.1. Naive implementation

Algorithm 1 Floating-point orientation-2 naive implementation

```
double pqx = qx - px, pqy = qy - py;
double prx = rx - px, pry = ry - py;
double det = pqx * pry - pqy * prx;
if (det > 0) return POSITIVE;
if (det < 0) return NEGATIVE;
return ZERO;
```

Algorithm 1 shows a naive implementation of the orientation-2 predicate. It first computes floating-point approximations of the vectors pq and pr. Then it simply computes an approximation of the 2×2-determinant and returns its sign.

If the computations were done on real numbers, this implementation would lead to the correct result. Unfortunately floating-point numbers suffer from both a limited precision and a limited range. These limitations can lead to a `det` value with a different sign.

Let us consider the following example. When the result of a floating-point computation is outside the range of representable numbers, it is replaced by an infinite value. Let M be the biggest power-of-2 number that is still representable. Let p, q, and r be

$$(p, q, r) = \begin{pmatrix} -M & M & -7M/8 \\ 1 & 3 & 17/16 \end{pmatrix}$$

Computing `pqx` will produce an unrepresentable value $2M$. As a consequence the variable will contain $+\infty$. None of the other variables nor `pqy * prx` will contain an infinite value. Hence the infinite value contained in `pqx` will propagate till the final variable `det` and the predicate will answer `POSITIVE`. Unfortunately the real value of the determinant is a negative number: $-M/8$.

This example shows that the limited range of floating-point numbers can lead to the wrong sign. The limited precision can also lead to a wrong answer, especially when the points are almost aligned. A small rounding error will spoil each floating-point computations and the computed values may slowly drift away from the real values.

2.2. Floating-point considerations

The IEEE-754 standard [13] covers both the data formats and the precise behavior of the arithmetic operators. In particular, it describes how rounding and limited precision affect the operators.

A floating-point operation $\tilde{c} \leftarrow \tilde{a} \oplus \tilde{b}$ can theoretically be decomposed into two steps: the first one computes the exact real result $c = \tilde{a} + \tilde{b}$ and the second one rounds this exact result c to the nearest floating-point number $\tilde{c}$. There exist two constants ϵ_0 and η_0 such that the distance between the exact result and the computed value is bounded by $\epsilon_0 \cdot |c|$ and by $\epsilon_0 \cdot |\tilde{c}|$, if the result is in the range of

normal numbers, and η_0 otherwise. If $|c|$ is bounded by r, and if $\epsilon_0 r \geq \eta_0$, then the computational error is bounded by $\epsilon_0 r$.

Both ϵ_0 and η_0 depend on the floating-point format chosen to implement the algorithm. In case of the IEEE-754 double-precision floating-point arithmetic with rounding to nearest, these constants are $\epsilon_0 = 2^{-53}$ and $\eta_0 = 2^{-1075}$.

Unfortunately, the standard does not prevent a processor to use an excess precision when computing in its default mode of operations. It does not describe how programming languages and their compilers should comply with it either. As a consequence, a double rounding phenomenon can appear. A value will first be computed and rounded in the default extended precision of the processor. It will then be stored in memory with a reduced precision. This storage induces a second rounding.

The total rounding error of these two operations can then exceed the rounding error that would have happened if a single rounding had been done. For example, this phenomenon can happen in a program relying on double-precision floating-point computations compiled by GCC on Linux for the x86 processor family. Depending on whether the result of a floating-point operation is only stored in a processor register or goes through the memory, the value will be rounded one or two times. The floating-point registers use 64-bit wide mantissas while the `double` type causes the compiler to store the values in memory with 53-bit wide mantissas. In order to take the double rounding phenomenon into account, the constants have to be changed accordingly: $\epsilon_0 = 2^{-53}(1 + 2^{-11})$ and $\eta_0 = 2^{-1075}(1 + 2^{-12})$.

2.3. Robust implementation

Algorithm 2 Floating-point orientation-2 filter

```
double pqx = qx - px, pqy = qy - py;
double prx = rx - px, pry = ry - py;

double maxx = max(abs(pqx), abs(prx));
double maxy = max(abs(pqy), abs(pry));
double eps = 8.8872057372592758e-16 * maxx * maxy;
if (maxx > maxy) swap(maxx, maxy);

if (maxx < 1e-146) {         // underflows?
  if (maxx == 0) return ZERO;
} else if (maxy < 1e153) {  // no overflow?
  double det = pqx * pry - pqy * prx;
  if (det > eps)  return POSITIVE;
  if (det < -eps) return NEGATIVE;
}

// fall back to a more accurate, slower method
```

Algorithm 2 takes into account the properties of floating-point arithmetic in order to return the correct sign. In case the sign cannot be computed by using floating-point arithmetic, the algorithm falls back on a more accurate yet slower method in order to determine the exact result. The cost of this slower method is amortized due to the fact that the filter part is expected to be able to determine the exact result with a high probability. We are not going to describe this part in this paper.

Like the naive implementation, this algorithm will first compute the vectors pq and pr and then compute the determinant of their coordinates. It will however contain some logic to prevent returning a wrong sign. In particular it will compute a value ϵ bigger than the rounding error caused by the limited precision of floating-point computations.

Indeed the computed value `det` is not the exact result det of the determinant. If there exists a value ϵ that bounds the error $|det - \texttt{det}| \leq \epsilon$, then `det` and det have the same sign if $|\texttt{det}| > \epsilon$. Otherwise the slower method is used. As a consequence, certifying that the floating-point fast path of the predicate is robust means certifying that the computed `eps` value is bigger than this theoretical ϵ value.

3. CERTIFYING THE ALGORITHM

3.1. The case of the 2×2-determinant

When computing a 2×2-determinant by its simplest formula with a floating-point arithmetic, an error will appear. The computed value $\tilde{z}$ is not the exact value z, and their signs may be different if z is close enough to 0. Moreover, the inputs $\tilde{s}$, $\tilde{t}$, $\tilde{u}$, and $\tilde{v}$ may already be spoiled by rounding errors and differ from the real inputs s, t, u, and v.

$$\begin{aligned} z &= s \cdot v - t \cdot u \\ \tilde{z} &= (\tilde{s} \otimes \tilde{v}) \ominus (\tilde{t} \otimes \tilde{u}) \end{aligned}$$

If the inputs were bounded, a simple forward error analysis would suffice to bound the absolute error between z and $\tilde{z}$. The relative error is not usable since the results of the computations are potentially subnormal. So we artificially bound the inputs by involving the values $m = \max(|\tilde{s}|, |\tilde{t}|)$ and $n = \max(|\tilde{u}|, |\tilde{v}|)$.

If m or n is zero, the computed value $\tilde{z}$ will be exactly zero. Thanks to subnormal numbers, the result of a subtraction is zero only if the two floating-point operands are equal. Consequently, if m or n is zero, it means that the real determinant also has a column of zeros, and is then equal to zero. The algorithm can directly answer that the three points are aligned.

We can rule out this case and now consider both m and n to be positive. All the inputs are bounded by m and n: $|\tilde{s}| \leq m$, $|\tilde{t}| \leq m$, $|\tilde{u}| \leq n$, and $|\tilde{v}| \leq n$. If these inputs were normalized ($m = 1$ and $n = 1$), then we could use forward error analysis methods in order to bound the absolute error caused by rounding:

$|\tilde{z} - z| \leq \delta$. We could then try to adapt this property when the inputs are not normalized:

$$|\tilde{z} - z| \leq m \cdot n \cdot \delta.$$

The value δ optimal for the normalized problem cannot be used directly: it would not catch all the wrong answers from the fast path. However, δ can be slightly modified so that it can be always used robustly. And this can be done by considering the normalized problem only.

3.2. Involving δ

The objective is to inductively prove a property on $\tilde{z} - z$ of the form $x \in f(m,n) \cdot I$. I is a closed and bounded interval of the real numbers $\mathbb{R}$, such that it could be obtained by considering the normalized problem. And f is a positive function, easily computable. In the case of $\tilde{z} - z$, this function is simply the product $m \cdot n$, and we expect to find an interval close to $[-\delta, \delta]$.

We will enclose expressions dealing with real numbers. These expressions are built of $\mathbb{R}$ ring operators (addition and multiplication) and a rounding operator $\circ : \mathbb{R} \to \mathbb{F}$. This unary function expresses the following property: in normal situation, a floating-point operator shall behave as if the computation was first done with an infinite precision and the result was then rounded to the working precision. As a consequence, an expression like $\tilde{s} \otimes \tilde{v}$ can be rewritten as $\circ(\tilde{s} \cdot \tilde{v})$.

The induction will be done on the structure of these expressions.

3.2.1. *First rounding*

Let us initialize the induction. The values of both expressions $\tilde{s}$ and $\tilde{t}$ are in the interval $m \cdot [-1, 1]$, and both $\tilde{u}$ and $\tilde{v}$ are in $n \cdot [-1, 1]$.

We will also need to bound the first rounding errors. For example, as explained in Section 2.2, the error $\tilde{s} - s$ will be bounded by $(m \cdot [-\epsilon_0, \epsilon_0]) \cup [-\eta_0, \eta_0]$. So if $m \cdot \epsilon_0 \geq \eta_0$, then

$$\tilde{s} - s \in m \cdot [-\epsilon_0, \epsilon_0]$$

In both cases, the intervals can be obtained by studying the normalized problem only. We can now start the structural induction.

3.2.2. *Addition and multiplication*

Let a and b be two real expressions respectively bounded by $f(m,n) \cdot A$ and $g(m,n) \cdot B$.

$$\begin{aligned} a + b &\in (f(m,n) \cdot A) + (g(m,n) \cdot B) \\ &\in \max(f(m,n), g(m,n)) \cdot (A + B) \\ &\in f(m,n) \cdot (A + B) \quad \text{when} \quad f = g \\ a \cdot b &\in (f(m,n) \cdot g(m,n)) \cdot (A \cdot B) \end{aligned}$$

The intervals $A + B$ and $A \cdot B$ must verify the inclusion property of interval arithmetic [10]. $[\underline{a} + \underline{b}, \overline{a} + \overline{b}]$ is such an interval for $A + B$ since it contains all the

possible results $x + y$ for $x \in A$ and $y \in B$. A similar formula involving only the bounds of the intervals is available for the product $A \cdot B$ of two intervals.

Once again, the intervals can be computed by simply considering the problem restricted to normalized inputs. Indeed, be it for the addition / subtraction (restricted to $f = g$) or the multiplication, we were able to bound the composed expression by the product of an expression involving m and n and interval that does not depend on the values of m or n.

3.2.3. *Rounding error*

The last expression that needs to be bounded is the rounding error $\circ(a) - a$. Thanks to the property described in Section 2.2, if $|A|$ is the magnitude $\max(|\underline{a}|, |\overline{a}|)$ of the interval $A = [\underline{a}, \overline{a}]$, and if $f(m, n) \cdot |A| \cdot \epsilon_0 \geq \eta_0$, then

$$\circ(a) - a \in f(m, n) \cdot [-|A| \cdot \epsilon_0, |A| \cdot \epsilon_0]$$

If we simply consider the normalized problem $a' \in A$, then $[-|A| \cdot \epsilon_0, |A| \cdot \epsilon_0]$ is an interval containing the rounding error $\circ(a') - a'$, but this is generally not the sharpest. This widened interval is necessary in order for it to be usable in the general case. As a consequence, when computing δ on the normalized problem, we cannot search for the best interval. We have to restrict ourselves to the theorems shown here. This is the reason why we will have to define non-standard rounding operators in Section 3.3.

In both rounding error computations, a condition $r \cdot \epsilon_0 \geq \eta_0$ has to be verified so that the inclusion properties are valid. This condition is directly included in Algorithm 2. By testing if `maxx` (the smallest of m and n) is big enough, we verify that the global error `eps` is not underestimated. If this test fails, we discard the floating-point determinant and switch to another method for evaluating its sign.

3.2.4. *Rewriting formulas*

Since none of the previous rules applies to $\circ(a)$, we would never find a bound on $\tilde{z} - z$. Indeed, this formula can only be split between $\tilde{z}$ and z; and although the right hand side z can be bounded thanks to the rules we described, the left hand side cannot.

To avoid this problem, we rewrite $\tilde{z} - z$ such that we have an expression that can be bounded by induction. In particular, since the addition in these formulas is associative (it is the addition of $\mathbb{R}$, not a floating-point addition), we can replace $\tilde{z} - z$ with $(\tilde{z} - z') + (z' - z)$, and $z' = \tilde{s} \otimes \tilde{v} - \tilde{t} \otimes \tilde{u}$.

The left hand side of the addition is now the rounding error $(\tilde{z} = \circ(z'))$ of the last floating-point operation. The right hand side is the error caused by the approximated inputs and the previous rounding errors. It cannot be directly bounded though, so it will be rewritten too, and so on until we reach expressions directly involving the inputs.

3.3. Computing and proving δ

We use the tool Gappa[2] [4] to generate both the value of δ and a formal proof of the correctness of this value. Given a logical property involving real expressions containing rounding operators, Gappa tries to certify it. More precisely, we will ask Gappa to bound the expression $\tilde{z} - z$ knowing that the four floating-point elements of the determinant are between -1 and 1. This is our normalized problem and we will use Gappa to compute the corresponding δ.

Algorithm 3 Gappa's algorithm description.

```
# some notations:
pqx = homogen80x_init(qx - px);
pqy = homogen80x_init(qy - py);
prx = homogen80x_init(rx - px);
pry = homogen80x_init(ry - py);
det homogen80x= pqx * pry - pqy * prx;
exact = (qx-px)*(ry-py) - (qy-py)*(rx-px);
# the property Gappa has to find and verify:
{ pqx in [-1,1] /\ pqy in [-1,1] /\
  prx in [-1,1] /\ pry in [-1,1] ->
  det - exact in ? }
```

Mathematical version of the property:

$$|\texttt{pqx}| \leq 1 \ \wedge \ |\texttt{pqy}| \leq 1 \ \wedge \ |\texttt{rqx}| \leq 1 \ \wedge \ |\texttt{rqy}| \leq 1 \ \rightarrow$$
$$|\texttt{det} - ((\texttt{qx} - \texttt{px}) \cdot (\texttt{ry} - \texttt{py}) - (\texttt{qy} - \texttt{py}) \cdot (\texttt{rx} - \texttt{px}))| \leq \texttt{?}$$

Algorithm 3 shows the code fed to Gappa. The question mark in the property means Gappa does not have to try to validate a given bound; it just has to compute an absolute error bound it can formally prove.

The first equalities are just notations that Gappa will expand in the property it has to analyze. The `homogen80x_init` (noted $\circ_i$ in the following) and `homogen80x` are both rounding operators. The first one corresponds to the initialization of the induction. The equality `det homogen80x= pqx * pry - pqy * prx` is just syntactic sugar to express that `det` denotes $\circ(\circ(\texttt{pqx} \cdot \texttt{pry}) - \circ(\texttt{pqy} \cdot \texttt{prx}))$.

We are not using the standard Gappa rounding operators like `float<ieee_64,ne>`, although they provide some powerful theorems on floating-point arithmetic. Indeed, Gappa is a generic tool for bounding expressions involving rounding operators, so it does not know about the induction we are performing. As explained in Section 3.2.3, we are restricted to a subset of floating-point arithmetic theorems. As a consequence, we define two new rounding operators in order to force Gappa to create a suitable proof. For each of them, only one Gappa theorem is defined,

[2] http://lipforge.ens-lyon.fr/www/gappa/

and hence there is only one way Gappa can deal with these operators.

$$\begin{array}{rcll} \circ_i(a) - a & \in & A \cdot [-\epsilon_0, \epsilon_0] & \text{when} \quad \circ_i(a) \in A \\ \circ(a) - a & \in & A \cdot [-\epsilon_0, \epsilon_0] & \text{when} \quad a \in A \end{array}$$

As described in Section 3.2.4, the shape of the expressions are not such that they can be bounded directly. They first have to be rewritten, so that the sub-terms are in a suitable form. Fortunately, Gappa automatically does this rewriting, the user does not have to do it beforehand. This will be especially important for the predicates of Section 4.1.

3.4. Other parts of the algorithm

Gappa gives us the value δ described in Section 3.2. This is a numerical value and it can be multiplied by m and n in order to obtain ϵ. Unfortunately, these multiplications will be computed by floating-point arithmetic. Consequently they may be slightly inaccurate and the computed value `eps` could then be smaller than the real ϵ value. The determinant would then be wrongly assumed to be bigger than the error threshold.

Once again, we will have to use a slightly pessimistic value of δ. This value is found by analyzing the errors that appear during the computation of `eps`. This computation only involves multiplications, so we can look at the relative error. This is especially important since we do not want to bound m and n, and consequently the absolute error cannot be used.

The relative error however requires us to avoid the range of subnormal results in order for it to stay bounded. To this end, Algorithm 2 puts some constraints on `maxx` and `maxy`. They guarantee none of the multiplications will underflow when computing `eps`.

Algorithm 4 Gappa's script for checking there is no overflow.

```
# some notations:
t1 = pqx * pry;
t2 = pqy * prx;
det = float80x(t1) - float80x(t2);
# the property Gappa has to verify:
{ pqx in [-1e153,1e153] /\ pqy in [-1e153,1e153] /\
  prx in [-1e153,1e153] /\ pry in [-1e153,1e153] ->
  t1 in [-1b1023,1b1023] /\ t2 in [-1b1023,1b1023] /\
  det in [-1b1023,1b1023] }
```

These constraints also prevent the computations from overflowing. Script 4 contains the Gappa description of this property of the algorithm. The computation of the determinant only happens when all the values are bounded by 10^{153}, this is expressed by the hypotheses on the expressions `pqx`, `pqy`, etc. The three goals

express that none of the theoretical non-rounded values is outside the range of floating-point numbers. Hence no overflow will happen during the computation of their rounded counterparts.

We use the notation `-1b1023` to write dyadic numbers, it means -1×2^{1023}: this is the smallest power-of-two number still representable by a floating-point double-precision number. It could be written in decimal notation `-8.98846567431e+307`, but it would be a lot less readable.

Another rounding mode, `float80x`, was used in order to describe this property. A standard rounding mode like `float<ieee_64,ne>` could not have been used, since it would not have taken into account the possibility of a double rounding phenomenon. The `float80x` rounding, however, provides a theorem that is still true when there is first a rounding to extended precision (80-bit format). This theorem expresses the range of a rounded expression when the range of the non-rounded expression is known.

This is ensured by rounding the lower and upper bounds of the range of the non-rounded expression respectively toward $-\infty$ and $+\infty$ in floating-point double precision. Although the reason is different, this is the same method interval arithmetic libraries apply to round their intervals, so that they contain any possible real results.

4. GENERALIZATION TO OTHER PREDICATES

4.1. Homogeneous predicates

The orientation-2 predicate is the sign of a 2×2-determinant. As a consequence, the numerical value the filter has to compute is multilinear with respect to the rows (or columns) of this determinant. Hence this value is related to the product $m \cdot n$. But more importantly, the absolute error on this value is also related to $m \cdot n$. Hence the bound $\delta \cdot m \cdot n$ we were trying to reach is not especially pessimistic.

More generally, this method applies to any predicate such that its numerical value is computed by an homogeneous polynomial. Indeed, by correctly choosing the initial bounding functions, the rules of Section 3.2 will lead to tight bounds on the absolute error, with simple bounding functions.

Fortunately, orientation-2 is not the only homogeneous predicate: a good number of common predicates follow the same shape. Indeed, due to their geometric nature, they mainly deal with distances. Hence they are homogeneous expressions, and the arguments that we have detailed for the 2D orientation predicate apply to the higher dimensional versions.

We have applied our techniques to the 2D and 3D orientation predicates, as well as the 2D and 3D predicates testing if a point is inside the circumscribing circle (resp. sphere) of 3 (resp. 4) other points. They are called `incircle` and `insphere` respectively. These predicates are the most useful in building triangulations and meshes.

4.2. Generating Gappa code

As shown by Algorithm 3, the file describing the `orient2` predicate for Gappa can be written by hand, it is only a few lines long. But such a work will become tedious for higher order predicates. The `insphere` is for example a 5×5-determinant; as such, its expanded expressions, both exact and computed, are not that easy to write.

To circumvent this difficulty, we have written a C++ class. This class interfaces with CGAL generic geometric predicates in order to automatically generate the input file for Gappa. We soon expect to be able to directly generate the C++ robust floating-point filters, in addition to Gappa files. We also intend to generalize the method for some other geometric predicates, still homogeneous, but containing more complex control structures.

4.3. Comparisons to other existing methods

As written in the introduction, there are various kinds of filters which have been proposed in the literature. Dynamic filters based on interval arithmetic are immune to problems due to boundary conditions, since these problems are supposed to be dealt with by the interval arithmetic which is used. However, they tend to be slower than methods requiring static analysis such as the one presented here.

The original paper on static filters by Fortune did not address the underflow issue, nor did Shewchuk's work (as explicitly stated in his paper). Indeed, it is not hard to find a data set which produces an underflow leading to a wrong answer. Taking a large degree predicate such as the `insphere` predicate, the following set of points appears to be cospherical while it is obviously not:

p = (0, 0, 0)
q = (1e-67, 0, 0)
r = (0, 1e-67, 0)
s = (0, 0, 1e-67)
t = (1e-67, 1e-67, 2e-67)

4.4. Benchmarks

We have conducted some benchmarks in order to evaluate the overhead of all necessary checks at run time. The benchmark consists of computing the 3D Delaunay triangulation of 10^5 random points in the unit cube, using CGAL [3]. The algorithm makes extensive use of the `orientation` and `insphere` predicates.

The experiments have been performed on a Pentium 4 PC at 1.7 GHz, we have used the GNU G++ compiler version 4.0 with the command line options `-O3 -DNDEBUG`. Table 1 reports the timings in seconds, for the average of 3 consecutive runs. Note that the choice of a random data set hardly triggers a robustness failure.

The various methods that we have compared are:

TABLE 1. Benchmarks of a 3D Delaunay triangulation

Method	Random
uncertified floating-point	3.29
our filter + interval + exact	4.33
interval + exact	12.5
exact	296
Shewchuk's predicates	4.39

- `uncertified floating-point`: the pure floating-point evaluation using doubles, which is uncertified.
- `our filter + interval + exact`: the filter we have described in this paper, which is proved, and in case of uncertainty calls a more accurate evaluation based on interval arithmetic, followed by an exact computation.
- `interval + exact`: uses interval arithmetic, followed by an exact computation if the intervals are not accurate enough.
- `exact`: uses exact multiprecision computation.
- `Shewchuk's predicates`: uses the predicates provided by Shewchuk [12].

As shown by Table 1, our filter is the first component of a geometric predicate a bit faster than Shewchuk's on average on the test machine. But its main quality does not lie in its speed, it lies in its robustness: floating-point underflow and overflow are taken into account, and the code is still valid on hardware architectures that suffer from double rounding.

5. CONCLUSION

Once some pitfalls of floating-point arithmetic like underflows, overflows, or double rounding are set aside, it becomes quite easy to design an algorithm which is not impacted by floating-point rounding errors. However such an algorithm can not be called robust.

We have formalized in this paper a method that allows us to design simple and fast yet robust floating-point filters. This method only applies to homogeneous algorithms; but since geometric predicates mainly deal with distances, they generally are homogeneous.

This homogeneity is used to compute a bound on the general error by studying the behavior of the filter when fed with normalized entries. This error bound does not require the intermediate results not to underflow: as long as a few constraints are satisfied, the filter will correctly handle any loss of precision.

The homogeneity also makes it so that the error bound is tight enough for the floating point filter to actually be useful. Indeed, as shown by Table 1, our predicates are almost as fast as the naive uncertified floating-point predicates: the slow path (interval arithmetic and then exact arithmetic) is seldom taken.

References

[1] H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. *Discrete Applied Mathematics*, 109:25–47, 2001.

[2] C. Burnikel, S. Funke, and M. Seel. Exact geometric computation using cascading. *Internat. J. Comput. Geom. Appl.*, 11:245–266, 2001.

[3] *The CGAL Manual*, 2004. Release 3.1.

[4] M. Daumas and G. Melquiond. Generating formally certified bounds on values and round-off errors. In *6th Conference on Real Numbers and Computers*, Dagstuhl, Germany, 2004.

[5] O. Devillers and S. Pion. Efficient exact geometric predicates for Delaunay triangulations. In *Proc. 5th Workshop Algorithm Eng. Exper.*, pages 37–44, 2003.

[6] S. Fortune and C. J. Van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Trans. Graph.*, 15(3):223–248, July 1996.

[7] S. Fortune and C. V. Wyk. *LN User Manual*. AT&T Bell Laboratories, 1993.

[8] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap. Classroom examples of robustness problems in geometric computations. In *Proc. 12th European Symposium on Algorithms*, volume 3221 of *Lecture Notes Comput. Sci.*, pages 702–713. Springer-Verlag, 2004.

[9] A. Nanevski, G. Blelloch, and R. Harper. Automatic generation of staged geometric predicates. In *International Conference on Functional Programming*, Florence, Italy, 2001. Also Carnegie Mellon CS Tech Report CMU-CS-01-141.

[10] A. Neumaier. *Interval methods for systems of equations*. Cambridge University Press, 1990.

[11] S. Pion. *De la géométrie algorithmique au calcul géométrique*. Thèse de doctorat en sciences, Université de Nice-Sophia Antipolis, France, 1999. TU-0619.

[12] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete Comput. Geom.*, 18(3):305–363, 1997.

[13] D. Stevenson et al. An American national standard: IEEE standard for binary floating point arithmetic. *ACM SIGPLAN Notices*, 22(2):9–25, 1987.

[14] C. K. Yap and T. Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, volume 4 of *Lecture Notes Series on Computing*, pages 452–492. World Scientific, Singapore, 2nd edition, 1995.

Communicated by (The editor will be set by the publisher).