International Journal of Computational Geometry & Applications

EXACT GEOMETRIC COMPUTATION USING CASCADING *

CHRISTOPH BURNIKEL

and

STEFAN FUNKE

and

MICHAEL SEEL

Max-Planck-Institut für Informatik,
Im Stadtwald
66123 Saarbrücken
Germany

{burnikel,funke,seel}@mpi-sb.mpg.de

ABSTRACT

In this paper we talk about a new efficient numerical approach to deal with inaccuracy when implementing geometric algorithms. Using various floating-point filters together with arbitrary precision packages, we develop an easy-to-use expression compiler called EXPCOMP. EXPCOMP supports all common operations $+, -, \cdot, /, \sqrt{}$. Applying a new semi-static filter, EXPCOMP combines the speed of static filters with the power of dynamic filters. The filter stages deal with all kinds of floating-point exceptions, including underflow. The resulting programs show a very good runtime behaviour.

Keywords: floating-point filter, exact arithmetic, expression compiler

1. Introduction

When computer scientists design geometric algorithms, they usually assume the availability of exact arithmetic on real numbers. Since no computer directly provides exact arithmetic on real numbers, programmers implementing these algorithms must find some substitution. Quite commonly, they resort to floating-point arithmetic due to its support by hard- and software as well as its convenient use. The resulting programs may not behave as expected, though. There are several ways a geometric algorithm may behave when exact arithmetic is replaced by floating-point arithmetic.

Some algorithms produce usable results in spite of some incorrect decisions – these algorithms are called stable (e.g. [16])– but most do not, they either produce

*This research was partially supported by the ESPRIT IV LTR Project No. 28155 (GALIA).

completely inconsistent results, crash or loop. Hence for most problems it is crucial to ensure correctness of every predicate evaluation.

There are several exact arithmetic schemes designed specifically for computational geometry; most of them are methods for exactly evaluating the sign of a determinant using IEEE double precision floating-point arithmetic, and hence can be used to perform e.g. the orientation and incircle tests or even the insphere test [1]. Difficulties arise, if the tests to be performed involve previously computed geometric objects which require extended precision to be exactly represented.

A more general approach, which is not specific to determinants or even predicates, are multiprecision packages [2,11,15]. They allow arbitrary precision arithmetic on integers, fixed-point numbers or floating-point numbers. Of course, exact arithmetic with these packages has its cost, which is considerably higher than floating-point arithmetic. Depending on the input bit-length the arbitrary precision primitives are at least about 10-100 times slower than their floating-point counterparts.

If a predicate expressed as the sign of an arithmetic expression, is to be computed, an obvious technique to reduce this overhead is trying to decide the predicate using floating-point arithmetic first. Only if no guarantee for the correctness of the outcome can be given, one resorts to arbitrary precision arithmetic. This technique is called a *floating-point filter* (also see [6]).

The focus of this paper is the conception of such a floating-point filter to be applied to both, integer and algebraic arithmetic; important issues are efficacy and ease of use.

In section 2 we focus on the problem of deriving error bounds for floating-point arithmetic at a reasonable cost. The approach presented is not restricted to integer values and handles all possible situations such as underflow exceptions and errors in the computation of the error bounds themselves. In section 3 we discuss packaging the results of the first part in a way such that it can be easily used within C++ programs. We design and implement an expression compiler which gives transparent access to the floating-point filter techniques. Finally, section 5 presents some results of applying the expression compiler – both with synthetic benchmarks and existing implementations of geometric algorithms.

Prior results in the field of floating-point filter techniques are mostly restricted to integer computation without divisions and square roots or neglect the problems of underflow. More importantly, most of these floating-point filter techniques are still unavailable for everyday programming work as they have to be encoded manually. There are similar approaches of packaging the filter techniques such as the expression compiler LN by Fortune/vanWyk [7], which is restricted to integer computation, though. Furthermore LN depends on fixed bit-lengths of the input data which, though allowing efficient exact evaluation code, only leads to strong filters when the presumed bit-lengths are close to the bit-lengths of the actual values.

2. A floating-point filter

If the floating-point arithmetic on a machine complies to the IEEE standard, we

can guarantee certain bounds for the error occurring in *one particular* operation, see [10]. But as we evaluate complex expressions involving more than one operator, errors are propagated from the "earlier stages" of computation to the final result. Assuming we have evaluated a complex expression e with floating-point arithmetic to $\widetilde{e}$, what we then want is an upper bound err for the error of this value:

$$|e - \widetilde{e}| \leq err$$

2.1. A short review of floating-point arithmetic

The following review about double precision floating-point arithmetic according to the IEEE standard is taken from [5], pages 7/8, but modified in some respects. It restricts to the knowledge required for the further considerations of this section. For a more intuitive tutorial on floating-point arithmetic, [9] is highly recommended.

A **double** is specified by a sign $s \in \{0, 1\}$, an exponent $e \in [0 \dots 2047]$ and a binary fraction $f = f_1 f_2 \dots f_p$ with $f_i \in \{0, 1\}$ for all i, $1 \leq i \leq p$. p is called *mantissa length* and in case of a **double,** $p = 52$. We also use f to denote the number $\sum_i f_i \cdot 2^{-i}$. The number v represented by the triple (s, e, f) is defined as follows:

- if $e = 2047$ and $f \neq 0$ then no number is represented (v is a NaN)
- if $e = 2047$ and $f = 0$ then $v = (-1)^s \cdot \infty$
- if $0 < e < 2047$ then $v = (-1)^s \cdot (1 + f) \cdot 2^{e-1023}$
- if $e = 0$ and $f \neq 0$ then $v = (-1)^s \cdot f \cdot 2^{-1022}$and v is called a denormalized number
- if $e = 0$ and $f = 0$ then $v = (-1)^s \cdot 0$ (= signed zero)

Let $MinDbl = 2^{-1022}$ and $MaxDbl = (2 - 2^{-52}) \cdot 2^{1023}$. We call a real number approximable if $|a| \leq MaxDbl + 2^{-53} \cdot 2^{1023}$.

For any approximable number a let $round(a)$ be the floating-point number nearest to a. If there are two floating-point numbers equally near to a then $round(a)$ is the one where the least significant bit is zero. For any of the arithmetic operations $+, -, \cdot, /, \sqrt{}$ the IEEE standard guarantees the following. Let z be the *exact* result of an operation applied to floating point operands.

- if z is approximable then $round(z)$ is delivered
- if $|z| > MaxDbl + 2^{-53} \cdot 2^{1023}$ then an overflow exception is signaled and $\pm\infty$ is delivered where the sign is determined by z
- if any argument of an operation is infinity or not a number, then the result is guaranteed to be infinity or not a number

We will try to establish a relation between an approximable number a and its floating-point approximation $round(a)$.

Let us first consider the case $a \notin [0, MinDbl)$. Let e be such that $2^e \leq |a| \leq 2^{e+1}$. Then $round(a) = a + \rho(a)$ where $|\rho(a)| \leq 2^{-53} \cdot 2^e$ and $2^e \leq |round(a)| \leq 2^{e+1}$. Thus

$$\frac{|a - round(a)|}{|a|} \leq \frac{2^{-53} \cdot 2^e}{2^e} = 2^{-53}$$

$$\frac{|a - round(a)|}{|round(a)|} \leq \frac{2^{-53} \cdot 2^e}{2^e} = 2^{-53}$$

By the two bounds derived above we have

$$round(a) \in \left[(1 - 2^{-p-1}) \cdot a, (1 + 2^{-p-1}) \cdot a\right]$$

$$a \in \left[(1 - 2^{-p-1}) \cdot round(a), (1 + 2^{-p-1}) \cdot round(a)\right]$$

On the other hand, if $a \in [0, MinDbl)$, $round(a)$ is $MinDbl$, a denormalized number or zero. Obviously, for $round(a) = MinDbl$ the same error bound as above applies but for the other cases it may not be true any more, e.g. if $a = 1.5 \cdot 2^{-1074}$ then $round(a) = 2^{-1073}$ and $\frac{|a-round(a)|}{|a|} > \frac{|a-round(a)|}{|round(a)|} = \frac{1}{4}$. If $round(a) = 0$, it is even impossible to give any error bound of this form. The *absolute* error is bounded by 2^{-1075} though. The case, in which the relative error bound is no more valid — i.e. $0 \leq round(a) < MinDbl$ — , is called *underflow*.

2.2. Error Analysis

There exist several approaches for computing *err*. There are dynamic error estimation schemes, where the error bound is computed completely at runtime, making use of the actual input values to get the best error estimation possible. This may lead to a considerable overhead, though. See [3] and [11] for example.

On the other hand, there are static error estimation schemes, see [7] for example, which compute an upper bound for the error before runtime, hence reducing the overhead at runtime to a minimum. This requires some knowledge about the actual input values at runtime, in particular the bit-length. The problem with this scheme is that if the presumed bit-length is choosen too long, the resulting error bounds are pessimistic and the filter is likely to fail. On the other hand, if the bit-length is chosen too short, using the program for larger input values requires recompiling.

For our purposes, we will divide the computation of the error bound in a static part and a dynamic part. The static part can be precomputed before runtime without any knowledge of the actual values of the expressions, whereas the dynamic part is computed during runtime.

For every expression e we not only compute the floating-point approximation $\widetilde{e}$ but also an upper bound $\widetilde{e_{sup}}$ for $|\widetilde{e}|$, called the supremum of $\widetilde{e}$, and an integer

expr. e	approx. $\widetilde{e}$	supremum $\widetilde{e_{sup}}$	index ind_e
$x+y$	$\widetilde{x} \oplus \widetilde{y}$	$\widetilde{x_{sup}} \oplus \widetilde{y_{sup}}$	$1 + MAX(ind_x, ind_y)$
$x-y$	$\widetilde{x} \ominus \widetilde{y}$	$\widetilde{x_{sup}} \oplus \widetilde{y_{sup}}$	$1 + MAX(ind_x, ind_y)$
$x \cdot y$	$\widetilde{x} \odot \widetilde{y}$	$\widetilde{x_{sup}} \odot \widetilde{y_{sup}}$	$1 + ind_x + ind_y$
x/y	$\widetilde{x} \oslash \widetilde{y}$	$\frac{\widetilde{(\widetilde{\lvert x \rvert} \oslash \widetilde{\lvert y \rvert})} \oplus \widetilde{(x_{sup} \oslash y_{sup})}}{\widetilde{(\widetilde{\lvert y \rvert} \oslash y_{sup})} \ominus (ind_y + 1) \cdot 2^{-p}}$	$1 + MAX(ind_x, ind_y + 1)$
$x^{\frac{1}{2}}$	$\sqrt{\widetilde{x}}$	$\begin{cases} (\widetilde{x_{sup}} \oslash \widetilde{x}) \odot \sqrt{\widetilde{x}} & if\ \widetilde{x} > 0 \\ \sqrt{\widetilde{x_{sup}}} \odot 2^{\frac{p}{2}} & if\ \widetilde{x} = 0 \end{cases}$	$1 + ind_x$

Table 1: Rules for computing approximations, suprema and indices.

ind_e – the index of e – , such that the following bound for the absolute error of the floating-point approximation is true at all times (assuming that the index ind_e never exceeds $2^{\frac{p-2}{2}}$, which is not a critical constraint in practice, though):

$$|\widetilde{e} - e| \leq \widetilde{e_{sup}} \cdot ind_e \cdot 2^{-p} \tag{1}$$

where p is the mantissa length, which is 52 for the standard C/C++ datatype **double** . An input value x exactly representable by a **double** has the floating-point approximation $\widetilde{x} = x$, the supremum $\widetilde{x_{sup}} = |\widetilde{x}|$ and the index 0. An input value not exactly representable by a **double** has the floating-point approximation $\widetilde{x} = round(x)$, the supremum $\widetilde{x_{sup}} = |\widetilde{x}| = |round(x)|$ and the index 1.

The index ind_e may be computed statically whereas $\widetilde{e}$ and $\widetilde{e_{sup}}$ must be computed at runtime using the inductively given rules in table 1. In the following $+, -, \cdot, /, .^{\frac{1}{2}}$ denote exact addition, subtraction, multiplication, division and square root, whereas $\oplus, \ominus, \odot, \oslash, \sqrt{}$ denote their floating-point counterparts.

Note that the computation of the supremum is quite similar to the computation of the floating-point approximation itself and therefore the implied overhead is reasonably small. In case of $+, -, \cdot$, only twice as many operations are needed, three times as many in case of $\sqrt{}$. The division has a considerably higher overhead but its use often can be avoided in practice.

With this error bound, it is now easy to decide if a floating-point evaluation $\widetilde{e}$ of an expression e has the correct sign. This is obviously the case if

$$|\widetilde{e}| > \widetilde{e_{sup}} \cdot ind_e \cdot 2^{-p}$$

Another assumption even allows a statement about the numerical exactness of $\widetilde{e}$. If the input values are of integer type and $+, -, \cdot$ the only operations used, we know that $\widetilde{e} = e$ if

$$\widetilde{e_{sup}} \cdot ind_e \cdot 2^{-p} < 1$$

Nothing has been said yet about what happens if an underflow occurs during the computation of $\widetilde{e}$ or $\widetilde{e_{sup}}$. Normally this is a pathological point of many error estimation schemes. For our scheme, we have proven that in case of addition, subtraction and square root underflow can be neglected, and in case of the division and multiplication it can be compensated by adding a small constant ($MinDbl$) to $\widetilde{e_{sup}}$. If the "original" input values are of integer type and no divisions occur –

which is often the case in practice – underflow can totally be neglected. Apart from that, our scheme also deals with second order errors, i.e. errors which occur during the computation of the error bounds themselves.

Looking at the rules for computing the suprema it turns out that if only $+, -, \cdot$ occur, and upper bounds for $\lceil |x| \rceil$ of all input values x exist, an upper bound for $\widetilde{e_{sup}}$ can be statically computed, which leads us to the following evaluation strategy:

1. evaluate the expression using floating-point arithmetic to get the approximation $\widetilde{e}$
2. if only $+, -, \cdot$ occur and upper bounds for the input values are available, check $\widetilde{e}$ with the statically determined error bound $e_{sup_{static}} \cdot ind_e \cdot 2^{-p} \Rightarrow$ *fully-static filter*
3. if 2. fails, compute dynamically the supremum $\widetilde{e_{sup}}$ and check $\widetilde{e}$ with the error bound $\widetilde{e_{sup}} \cdot ind_e \cdot 2^{-p} \Rightarrow$ *semi-static filter.*
4. if 2. and 3. fail, evaluate the expression using arbitrary precision arithmetic

2.2.1. Proof of correctness

In the following we will proof that our invariant is maintained when computing $\widetilde{e}$, $\widetilde{e_{sup}}$ and ind_e according to the rules given in table 1 for the case of addition. The other proofs are similar and can be found in detail in [8]. A similar error bound is also shown in [12].

$$|(\widetilde{a} \oplus \widetilde{b}) - (a + b)| =$$

$$= |(\widetilde{a} \oplus \widetilde{b}) - (\widetilde{a} + \widetilde{b}) + (\widetilde{a} - a) + (\widetilde{b} - b)|$$

$$\leq \underbrace{|(\widetilde{a} \oplus \widetilde{b}) - (\widetilde{a} + \widetilde{b})|}_{operation-induced} + \underbrace{|\widetilde{a} - a| + |\widetilde{b} - b|}_{operand-induced}$$

Assuming $|\widetilde{a} \oplus \widetilde{b}| \geq MinDbl$ we get for the operation-induced error according to section 2.1:

$$|(\widetilde{a} \oplus \widetilde{b}) - (\widetilde{a} + \widetilde{b})| \leq |\widetilde{a} \oplus \widetilde{b}| \cdot 2^{-p-1}$$

$$\leq \widetilde{a_{sup}} \oplus \widetilde{b_{sup}} \cdot 2^{-p-1} = \widetilde{e_{sup}} \cdot 2^{-p-1}$$

On the other hand, if we have an underflow $|\widetilde{a} \oplus \widetilde{b}| < MinDbl$, then two cases can be distinguished:

1. $|\widetilde{a}| \oplus |\widetilde{b}| < MinDbl$, i.e. there may also be an underflow in the supremum. But then we know that $|\widetilde{a}|, |\widetilde{b}| < MinDbl$ and hence $\widetilde{a} + \widetilde{b} = \widetilde{a} \oplus \widetilde{b}$. In other words, the operation-induced error is zero.

2. $|\widetilde{a}| \oplus |\widetilde{b}| \geq MinDbl$, i.e. the supremum does certainly not underflow. But since the operation-induced error for $\widetilde{e}$ is bound by 2^{-1075}, we get

$$|(\widetilde{a} \oplus \widetilde{b}) - (\widetilde{a} + \widetilde{b})| \leq 2^{-1075} = MinDbl \cdot 2^{-p-1}$$

So in all cases – even if an underflow has occurred either during the computation of $\widetilde{e}$ or $\widetilde{e_{sup}}$, $\widetilde{e_{sup}} \cdot 2^{-p-1}$ is an upper bound for the operation-induced error.

For the operand-induced error we get:

$$|\widetilde{a} - a| + |\widetilde{b} - b| \leq \widetilde{a_{sup}} \cdot ind_a \cdot 2^{-p} + \widetilde{b_{sup}} \cdot ind_b \cdot 2^{-p}$$

$$\leq (\widetilde{a_{sup}} + \widetilde{b_{sup}}) \cdot MAX(ind_a, ind_b) \cdot 2^{-p}$$

As above we know that if $\widetilde{a_{sup}} \oplus \widetilde{b_{sup}}$ underflows, then $\widetilde{a_{sup}} \oplus \widetilde{b_{sup}} = \widetilde{a_{sup}} + \widetilde{b_{sup}}$. Hence we may continue using $ind_e \leq 2^{\frac{p-2}{2}}$ with

$$\leq (\widetilde{a_{sup}} \oplus \widetilde{b_{sup}}) \cdot (1 + 2^{-p-1}) \cdot MAX(ind_a, ind_b) \cdot 2^{-p}$$

$$\leq \widetilde{e_{sup}} \cdot 2^{-p} \cdot (MAX(ind_a, ind_b) + 2^{-1})$$

which finally leads us to:

$$|(\widetilde{a} \oplus \widetilde{b}) - (a + b)| \leq \widetilde{e_{sup}} \cdot 2^{-p} \cdot (1 + MAX(ind_{a,} ind_b))$$

Obviously, $\widetilde{e_{sup}}$ remains an upper bound for $\widetilde{e}$ and an underflow occurring during the computation of $\widetilde{e}$ or $\widetilde{e_{sup}}$ can be neglected.

3. Packaging the floating-point filter

Manually encoding the filter mechanisms shown in the last chapter is much too cumbersome. Not only has a programmer to implement a given predicate evaluation several times – at least once for the floating-point filter stage and once for the exact evaluation – she also has to compute the indices manually. Even small changes in a predicate then require a considerable effort. There are basically two ways of packaging the floating-point filter:

1. a C++ class wrapper
2. an expression compiler

In the following we will discuss both approaches and show their advantages and drawbacks.

3.1. Class Wrapper

As an example of a class wrapper we implement a class **FpFilt** replicating the functionality of the C/C++ number types. Internally though, an error bound is

computed for each expression using the methods derived in section 2. If the sign of such a value is asked for using the **::sign()** member function, it may return -1, 0, +1 or NO_IDEA if no guarantee for the exactness of the sign can be given.

We will illustrate the use of this new class with a simple example occuring in many geometric algorithms – the 2d orientation test.

> Given three points $p, q, r \in \mathbf{Z^2}$ with $p = (p_x, p_y)$, $q = (q_x, q_y)$, $r = (r_x, r_y)$; to which side of the oriented line $\overrightarrow{pq}$ is r located ?

The following pseudo-code evaluates this predicate using the LEDA datatype for arbitrary precision integers:

integer $DX \leftarrow p_x - q_x$
integer $DY \leftarrow p_y - q_y$
integer $TL \leftarrow q_x \cdot DY - q_y \cdot DX$
integer $RES \leftarrow DX \cdot r_y - DY \cdot r_x + TL$
$ressign \leftarrow sign(RES)$

The resulting sign in *ressign* indicates, where with respect to $\overrightarrow{pq}$ the point r is located: left of ($ressign \equiv -1$), right of ($ressign \equiv 1$) or on the line ($ressign \equiv 0$). The same predicate, when enhanced with a floating-point filter using the new C++ class `FpFilt`:

FpFilt $DX \leftarrow$ **FpFilt**$(p_x) -$ **FpFilt**(q_x)
FpFilt $DY \leftarrow$ **FpFilt**$(p_y) -$ **FpFilt**(q_y)
FpFilt $TL \leftarrow$ **FpFilt**$(q_x) \cdot DY -$ **FpFilt**$(q_y) \cdot DX$
FpFilt $RES \leftarrow DX \cdot$ **FpFilt**$(r_y) - DY \cdot$ **FpFilt**$(r_x) + TL$
$ressign \leftarrow sign(RES)$
if ($ressign \equiv NOIDEA$)
{**integer** $DX \leftarrow p_x - q_x$
integer $DY \leftarrow p_y - q_y$
integer $TL \leftarrow q_x \cdot DY - q_y \cdot DX$
integer $RES \leftarrow DX \cdot r_y - DY \cdot r_x + TL$
$ressign \leftarrow sign(RES)$
}

We see that the "filter" code looks just like a reimplementation of the predicate in the filter datatype – something which could be automated. Ideally the floating-point filter should be invisible for the programmer. Nevertheless, with this approach, the code evaluating the predicate can be interleaved or nested arbitrarily within other language constructs such as if-statements or loops. This allows a maximum of flexibility in the use of the floating-point filter. Apart from that, no preprocessing of the source file is necessary.

But there are disadvantages as well. Wrapping functionality in a C++ class always implies some overhead as we will see in our experimental results. This agrees with the observations made in [7]. Apart from that, this approach obviously does not allow static precomputations before runtime, i.e. in our case both the computation of the suprema and the indices have to take place at runtime.

3.2. Expression compiler

An expression compiler remedies most of the drawbacks mentioned above. It parses marked code segments within a source file which represent the evaluation of a predicate and replaces them with the equivalent two-level evaluation scheme.

The source code for our example when using an expression compiler may look like that:

BEGIN_PREDICATE
{
integer $DX \leftarrow p_x - q_x$
integer $DY \leftarrow p_y - q_y$
integer $TL \leftarrow q_x \cdot DY - q_y \cdot DX$
integer $RES \leftarrow DX \cdot r_y - DY \cdot r_x + TL$
$ressign \leftarrow sign(RES)$
}
END_PREDICATE

Note that the use of a floating-point filter is almost invisible to the programmer. The only difference are the **BEGIN_PREDICATE** ... **END_PREDICATE** directives around the original code, which are required by the expression compiler to recognize the predicate evaluation. The output of the expression compiler may look like that:

`<floating-point filter evaluation>`
`<with result in ressign>`
if $(ressign \equiv NOIDEA)$
{**integer** $DX \leftarrow p_x - q_x$
integer $DY \leftarrow p_y - q_y$
integer $TL \leftarrow q_x \cdot DY - q_y \cdot DX$
integer $RES \leftarrow DX \cdot r_y - DY \cdot r_x + TL$
$ressign \leftarrow sign(RES)$
}

Before generating code, the expression compiler can perform static precomputations, such that less work is done at runtime. In particular, if we use the semi-static error analysis, the expression compiler could precompute the indices of all (sub)expressions.

There are disadvantages as well, though. The programmer has to restrict to a reasonably small subset of C/C++ language constructs, to keep parsing and code generation manageable. Loops for example would complicate static analysis considerably.

In an actual implementation of the expression compiler, the programmer must provide some information about the types of the identifiers (such as p_x, p_y, q_x ...) referenced within the marked code sequence.

The advantages and drawbacks of both approaches suggest the use of an expression compiler for which we will describe our implementation in the following.

4. EXPCOMP – an expression compiler

In this section we sketch our implementation of an expression compiler as suggested in the last section. We designed it in such a way that it can be easily used with the datatypes for arbitrary precision provided by the LEDA software library (Library of Efficient Datatypes and Algorithms) (see [11]). In LEDA the programmer has access to three C++ classes for integer, rational and algebraic computation (**integer** , **rational** and **real**). They can be used very similarly to the standard C/C++ number datatypes due to operator overloading, but also have additional functions for conversion, sign determination etc. Adapting EXPCOMP to other arbitrary precision datatypes is not difficult, though.

The expression compiler EXPCOMP is basically a preprocessing tool which is invoked with a source and a target filename. The source file contains C++ code mixed with special statements which allow the expression compiler to identify the code sequences to be replaced using the floating-point filter techniques introduced in section 2.

There are two ways in which the expression compiler can be used:

1. Predicate evaluation
2. Value computation

For simple programs where no extensive numeric precomputations take place, i.e. the numeric part of computation mainly consists of evaluating predicates which do not reference precomputed values, it is sufficient to apply the expression compiler to the predicate evaluations. In programs though, where large parts of the numeric computation consist of computing values for the later use in predicates, the expression compiler can also be applied. In both cases, only relatively small modifications of the original code are necessary.

4.1. Predicate evaluation

When used to speed up predicate evaluation, the following modifications have to be made:

- marking the code sequences which represent the predicate evaluations using **BEGIN_PREDICATE** { ... } **END_PREDICATE**
- declaring the types of all identifiers/functions referenced within the code sequence
- optionally giving the expression compiler more information about the input values, such as bit-length or sign

In the previous section we have come across the orientation test occurring in many geometric algorithms. The following code fragment evaluates this predicate using LEDA **integer** arithmetic, provided the points P, Q, R are stored as LEDA **rat_point**s. An instance of a **rat_point** represents a point by homogenous coordinates of type **integer**, which can be accessed by member functions **rat_point::X()**, **rat_point::Y()** and **rat_point::W()**. For sake of simplicity, we assume that the third homogenous coordinate (**::W()**) is 1 for our input points (meaning that all points have integer coordinates).

```
integer DX=P.X()-Q.X();
integer DY=P.Y()-Q.Y();
integer TL=Q.X()*DY-Q.Y()*DX;
integer RES=DX*R.Y()-DY*R.X()+TL;
ressign=sign(RES);
```

To make use of the floating-point filter, this code sequence is mixed with special expression compiler statements. In our example we tell EXPCOMP that all input values of the predicate are nonnegative LEDA **integer**s which have at most bit-length 16:

```
BEGIN_PREDICATE
{
DECLARE_ATTRIBUTES integer_type pos_val FOR
    P.X() P.Y() Q.X() Q.Y() R.X() R.Y();
DECLARE_BITLENGTH 16 FOR
    P.X() P.Y() Q.X() Q.Y() R.X() R.Y();
integer DX=P.X()-Q.X();
integer DY=P.Y()-Q.Y();
integer TL=Q.X()*DY-Q.Y()*DX;
integer RES=DX*R.Y()-DY*R.X()+TL;
ressign=sign(RES);
}
END_PREDICATE
```

EXPCOMP then replaces this code sequence by the code implementing the following evaluation strategy:

1. `RES` is evaluated using floating-point arithmetic to $\widetilde{RES}$
2. $\widetilde{RES}$ is checked with the error bound that EXPCOMP has determined before runtime using the bit-length of the input.
3. if no decision could be made in step (2), the supremum of `RES` is computed and used to check whether the sign of $\widetilde{RES}$ is reliable
4. if still no decision could be made, `RES` is evaluated using LEDA **integer** arithmetic

If all input values of a predicate are of integer type, but square roots or divisions occur in its computation (this implies that the exact implementation is using the LEDA types **real** or **rational**), EXPCOMP can generate code in which an intermediate filter stage is introduced. Then the evaluation scheme is first to try making a decision based on floating-point arithmetic; if this fails, the predicate is tested using LEDA **integer** arithmetic. Note that this requires checking after every square root and division if the result is really an integer value. Only if this second stage also fails, LEDA **real** arithmetic is used. In practice this has proven to be very efficient for detecting degenerate cases.

4.2. Value precomputation

So far, we have assumed that the input values for a predicate are either input data or exactly precomputed using one of the LEDA datatypes for exact arithmetic. In more complex applications, though, these precomputations dominate the runtime, and hence applying the floating-point filter mechanism only to the predicate evaluation but still precomputing the input values for the predicate exactly, is not very promising with respect to the performance gain.

So the idea is to approximate the precomputations as well. Of course this requires more effort than just marking some code sequences. In the following we will illustrate the idea using a simple example.

Again, we will consider the orientation test from above with the only difference being the point r which is to be checked against the line $\overrightarrow{pq}$ results from an earlier computation, namely r is the intersection of two lines l_1, l_2, each of which is given by two points, namely $l_1 = \overrightarrow{s_1, t_1}$, $l_2 = \overrightarrow{s_1, t_1}$.

A natural approach would be to compute the intersection point using exact LEDA **rational** arithmetic and then store the result (without loss of precision) as a LEDA **rat_point** as above. Note though, in that case, even if the predicate can be decided using floating-point arithmetic, the intersection point is *always* computed exactly. If the exact intersection point is not required as an output of the algorithm and all orientation tests involving this point can be evaluated reliably using floating-point arithmetic, we better only "approximate" the computation of the intersection point. This can be done by writing a new class **intersection**:

```
class intersection {
public:
 integer X() {
  if (!exactly_computed){ComputeExact(); exactly_computed=true;}
  return _X;}
 integer Y() {
  if (!exactly_computed){ComputeExact(); exactly_computed=true;}
  return _Y;}
 integer W() {
  if (!exactly_computed){ComputeExact(); exactly_computed=true;}
  return _W;}
```

```
 interval XAPX(){return _XAPX;}
 interval YAPX(){return _YAPX;}
 interval WAPX(){return _WAPX;}

 intersection(rat_point s1, rat_point t1, rat_point s2, rat_point t2)
  { _s1=s1;_t1=t1;_s2=s2;_t2=t2;
   exactly_computed=false; ComputeApprox();}

private:
 rat_point _s1,_t1,_s2,_t2;
 integer _X,_Y,_W;
 interval _XAPX, _YAPX, _WAPX;
 bool exactly_computed;

 void ComputeExact();
 void ComputeApprox();
}
```

The purpose of this class is to provide a kind of "lazy evaluation" scheme for the computation of an intersection point. When an intersection point is to be constructed, one creates a new object of type **intersection**, giving the four defining **rat_point**s in the constructor. On instantiation, only the floating-point approximations (with error bounds) of the coordinates are computed and stored in **_XAPX**,...

Remark: the class **interval** is a simple interface class, which stores two **double** values representing an interval in which the exact value of a computation is contained.

We will now show, how intersection points represented by objects of this class are used within the orientation predicate:

```
BEGIN_PREDICATE
{
DECLARE_ATTRIBUTES integer_type FOR P.X() P.Y() Q.X() Q.Y();
DECLARE_ATTRIBUTES integer_APX_type FOR R.X() R.Y() R.W();
integer DX=P.X()-Q.X();
integer DY=P.Y()-Q.Y();
integer RES=DX*(P.Y()*R.W()-R.Y()) -
            DY*(P.X()*R.W()-R.X());
ressign=sign(RES);
}
END_PREDICATE
```

Remember that we assume integer coordinates for the input data (here: points P and Q), but of course not for the intersection point R.

The only difference from the orientation test we have seen before is that the type of the identifiers **R.X()**,**R.Y()**, **R.W()** has changed from **integer_type** to **integer_APX_type**. When this special type is used, EXPCOMP assumes that

the coordinate is not only available as exact **integer**, but also as a floating-point approximation together with an error bound.

So in the first filter stage of this computation, only the floating-point approximations of the intersection point is referenced. Therefore, if the predicate can be provably decided by floating-point arithmetic, the exact coordinates of the intersection point are not computed.

It remains to give implementations for the **::ComputeExact()** and **::ComputeApprox()** member functions of our new **intersection** class. This is where EXPCOMP comes into play again:

```
DEFINE_FUNCTION intersection::Compute();
{
 BEGIN_COMPUTATION
 {
 //computation of the intersection point
 //using the ''constructors''
 //in _s1, _t1, _s2, _t2
 //storing the result in _X,_Y,_W
 //expressed using exact LEDA integer arithmetic
 }
 END_COMPUTATION
}
END_FUNCTION
```

This code sequence is replaced by EXPCOMP with the two member functions **::ComputeExact()**, **::ComputeApprox()**.

Note that by convention of EXPCOMP, assignments to variables non-local to the member function itself — in our example the assignments of the homogenous coordinates of the intersection point — require that not only variables for the exact values exist in the class (here: **_X**, **_Y**, **_W** of type **integer**), but also for the approximated values (**_XAPX**, **_YAPX**, **_WAPX** of type **interval**) (see above in the class definition).

Furthermore, to keep parsing, analysis and code generation simple, EXPCOMP restricts the C++ language constructs within a **BEGIN_COMPUTATION** ... **END_COMPUTATION** block. Only assignments are allowed within such blocks. In our implementations, it was was always possible to rearrange the computation code in such a way. Loops and recursive calls, as they may occur in some computation, are rejected by EXPCOMP.

5. Experimental Results

5.1. Synthetic benchmarks

In this section we will determine experimentally to which extent the use of the expression compiler affects the performance of synthetic benchmarks as well as actual implementations of geometric algorithms. As test environment we used a 200

Bit-length	int.	real	FpFilt	int. filter	real filter	stat. filter
k=8	42.8	24.1	2.9	1.9	2.2	2.6
k=12	46.6	24.6	2.9	1.8	2.3	2.7
k=16	46.2	56.1	2.8	1.9	2.3	2.7
k=24	61.5	62.6	2.8	1.7	2.3	2.6
k=32	65.7	64.9	2.9	1.9	2.2	2.5
k=40	123.3	64.6	2.9	1.8	2.3	1.5
k=48	125.1	60.3	2.9	1.8	2.3	1.5

Table 2: Det3x3, random data

MHz UltraSparc running Solaris 2.5. All programs are compiled using g++ 2.7.2 with the –O option set.

5.1.1. Signs of 3x3 determinants

In this experiment we measure the runtimes of several number types and the code generated by the expression compiler when computing 100000 times the sign of 3x3 determinants with integer entries.

The following implementations are tested and compared to **double** arithmetic evaluation (i.e. the time overhead $\frac{runtime\,impl}{runtime\,double}$ is measured):

- LEDA **integer**
- LEDA **real**
- LEDA **integer** with **FpFilt** class (as described in section 3)
- LEDA **integer** tuned with EXPCOMP
- LEDA **real** tuned with EXPCOMP — implies underflow "countermeasures"
- LEDA **integer** tuned with EXPCOMP — full- and semi-static filter for a presumed bit-length of 48

All implementations ensure that the delivered sign result is exact, which is not the case for the **double** evaluation — so the comparison is somewhat unfair.

We will test both, matrices with random entries and non-regular matrices where the determinant is zero. The determinants are evaluated according to Sarrus' rule. The test results are listed in tables 2 and 3.

As we can see, all tests with random data can be decided by the filter-stages. The fully static filter only comes into effect when the actual bit-length is near the presumed one, otherwise the second stage – the semi-static filter – succeeds. The differences between the integer filter and the static filter for bit-lengths < 40, can be explained by the fact that in case of the integer filter, the instructions for computing $\widetilde{e}$ and $\widetilde{e_{sup}}$ are interleaved, otherwise they aren't. Due to the compiler's optimizer and the architecture of the FPU this may lead to differences in runtime.

Bit-length	int.	real	FpFilt	int. filter	real filter	stat. filter
k=8	37.9	24.0	2.4	1.4	26.3	2.0
k=12	44.5	24.1	2.7	1.4	26.3	2.3
k=16	45.3	1378	2.4	1.4	1378	2.0
k=24	54.3	1719	56.3	56.5	1721	54.7
k=32*	56.3	2633	56.5	58.4	2636	59.3
k=40*	117.4	4353	119.4	117.5	2644	117.3
k=48*	135.2	5151	136.5	135.1	5157	136.9

Table 3: Det3x3, degenerate data

The degenerate cases could only be decided by the integer filter for short bit-lengths, so most of the time the arbitrary precision calculation dominates the run-time. It is remarkable how badly LEDA **real**s perform in degenerate cases.

5.1.2. Incircle test

In this experiment we use a predicate occurring in the construction of the Voronoi diagram of line segments and measure the different implementations. The code is taken from [4], page 59.

> Let $v = (v_x, v_y, v_z),\ v_z > 0$ be a Voronoi vertex defined by three sites p, l_1, l_2, where $p = (0, 0, 0)$ is the origin and l_1, l_2 are lines. Given a line l_3, does l_3 touch, intersect or miss the clearance circle of v ?

Assuming that the input variables are all of integer type, we will test three implementations (which all guarantee the exact sign result) of this incircle test and compare them with the (possibly faulty) **double** arithmetic implementation:

- LEDA **real**
- EXPCOMP 3-level (**double** – **integer** – **real**, as suggested in 4.1)
- EXPCOMP 2-level (**double** – **real**)

We will measure both, incircle tests with randomly chosen as well as degenerate input. For the degenerate cases we construct three lines that touch a common circle passing through $p = (0, 0)$, as described in [4] page 59. The test results are listed in tables 4 and 5.

As for the tests with signs of determinants, it turns out that LEDA **real**s perform badly in degenerate cases. In the non-degenerate cases, the overhead of LEDA **real** compared to **double** is by far not as large as for the determinants' tests. This is due to the expensive square root operation which dominates the runtime cost.

5.2. Real-world benchmarks

When measuring the runtime of real-world applications, care has to be taken, since not only time spent in the numeric part is measured but also the time spent

Bit-length	real	3-level filter	2-level filter
k=8	20.2	2.4	2.5
k=12	20.3	2.4	2.5
k=16	21.1	2.4	2.5
k=24	21.7	2.5	2.6
k=32	29.2	2.5	2.6
k=40	28.8	2.5	2.6
k=48	28.3	2.4	2.5

Table 4: Incircle, random data

Bit-length	real	3-level filter	2-level filter
k=8	14.7	31.7	16.7
k=12	37.5	53.5	37.4
k=16	115.5	43.7	115.8
k=24	367.5	70.3	366.3
k=32	673.3	76.9	674.3
k=40	888.3	106.4	889.1
k=48	1169	127.9	1169

Table 5: Incircle, degenerate data

in the symbolic part. Therefore the total speedup when using floating-point filter techniques is usually much smaller than the results of the previous section may suggest. Of course, the user is only interested in the total speed-up of the program, but for us, a closer examination may be interesting.

The runtime T of a program consists of a symbolic part S and a numeric part N:

$$T = T_{(1)} = S + N$$

If we modify the program such that each numeric computation is performed $c > 1$ times, e.g. $c = 2$, the runtime of this modified program should be

$$T_{(c)} = S + c \cdot N$$

With these two measurements it is easy to get approximate values for S and N.

5.2.1. Triangulation of simple polygons

We implemented an algorithm for the incremental randomized construction of triangulations of simple polygons according to Seidel ([14]). The algorithm first computes the trapezoidal decomposition induced by a set S of n line segments in the plane. If S is a simple polygonal chain the expected runtime of the algorithm is O(n log* n). Given the trapezoidation of a simple polygon it is easy to get its triangulation in O(n) time. Hence the triangulation of a simple polygon can be computed in O(n log*n) time with this algorithm.

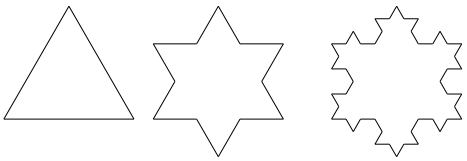

Figure 1: Koch-isles of recursion depth 0, 1 and 2

bit-length	double	integer	filter
16	4.25	20.91	5.09
24	4.36	20.86	5.19
32	4.31	21.38	5.71
40	4.25	24.93	8.20
48	-	26.69	8.81

Table 6: PolyTriang: time (in seconds)

We are interested in the geometric predicates used in the algorithm. The algorithm is quite simple and hence there is only one non-trivial predicate — the orientation test which we have already come across in the previous sections.

We assume that the input data (the coordinates) is of integer type and has **double** precision, i.e. it can be exactly represented by **double**s.

Runtime experiments were made with Koch-isles (see figure 1) of several recursion depths where the corner coordinates were rounded to the next integers.

The following tables show the average runtime for different implementations and different input data bit-lengths. We measured three implementations of the predicate:

- **double** arithmetic
- **integer** arithmetic
- **integer** arithmetic with EXPCOMP

and several input bit-lengths between 16 and 48 bit for a "snowflake" of recursion depth 7, i.e. a polygon with 49152 nodes. The results are shown in table 6.

Note though that the implementations do not only differ in the way the predicate is evaluated. Each implementation is optimized with respect to its predicate evaluation. In particular, each implementation stores the coordinates of the input data in the most suitable type, which is **double** for the filter implementation.

Up to an input data bit-length of 24, all predicates can be decided by the floating-point filter. With increasing bit-length more and more decisions require exact (**integer**) arithmetic, so the overhead increases. For bit-length 48 the **double** implementation crashed due to predicates being evaluated incorrectly.

For a more precise analysis we examine the number of "difficult" tests that require **integer** arithmetic in the filter implementation. Table 7 shows the results.

bit-length	# of tests	diff.	diff. "0"
16	373610	0	0
24	374558	0	0
32	385855	14213	13954
40	384232	18339	12443
48	387932	21425	13386

Table 7: PolyTriang: #of tests

	double	integer	filter
total time	4.43	22.94	5.90
symbolic	4.18	12.45	4.84
numeric	0.25	10.49	1.06

Table 8: PolyTriang, bit-length 32: time (in seconds)

In our example, even with bit-length 48 less than six percent of all tests require resorting to **integer** arithmetic. Most of these were degenerate cases, which obviously are more difficult to decide.

Finally we measure how much time is actually spent in the symbolic and in the numeric part of the computation using the method explained at the beginning of this section. We take the example with 49152 nodes and input bit-length 32. Table 8 shows the results. One does not expect differences in the symbolic runtimes; but as each implementation is optimized with respect to its predicate evaluations, i.e. for example that the integer implementation uses LEDA **integer**s to store the original data, whereas the other implementations use **double**s, their symbolic runtimes may vary.

5.2.2. Voronoi diagram of points and line segments

In this last experiment, the expression compiler was used to tune an implementation of a randomized incremental algorithm for computing the Voronoi diagram of points and line segments (see[13]).

As to be expected there are much more predicates than in the previous algorithm. What is also different from the previous algorithm is the fact that the input values for some of the predicates are values that have been precomputed. Moreover, these values may be non-rational, for the coordinates of Voronoi nodes are usually non-rational if at least one defining site is a segment.

As a very first step, we only concentrated on the predicates mentioned above, such as the incircle test. By applying the expression compiler, we replaced the exact evaluations by two-level evaluations, where we first try to evaluate the predicates using **double** arithmetic and only if this fails, exact arithmetic is used. The result was quite disappointing, performance improved by less than 15 % only.

This can be explained by the fact, that the "input values" of these predicates – the precomputed coordinates of the Voronoi nodes – are still computed using exact (LEDA **real**) arithmetic. As shown in [4], the computation of the Voronoi node

Bit-length	orig.	r	dr	dir	d
12	189.2	218.8	85.5	82.3	-
16	204.1	242.8	91.5	86.7	-
24	240.0	279.8	101.4	93.8	80.4
32	336.4	387.4	183.9	136.5	-
40	459.0	511.7	282.7	200.3	-
48	494.2	548.6	288.0	187.7	-

Table 9: VoroDiag: time (in seconds)

	r	dir	dr	d
total time	278.9	96.8	99.3	80.9
symbolic	76.8	82.1	81.9	71.0
numeric	202.1	14.7	17.4	9.9

Table 10: VoroDiag, bit-length 24: time(in seconds)

coordinates is far more complex than the evaluation of the predicates.

Hence we incorporated the computation of the Voronoi node coordinates into the floating-point filter mechanism using the expression compiler as sketched section 4.2. Moreover we introduced a third intermediate filter stage where we try to decide a predicate using **integer** arithmetic if the floating-point filter fails, as suggested at the end of section 4.1. We have measured five different implementations:

orig	original impl. using LEDA **real**
r	lazy-eval, no filter stages
dr	lazy-eval, with fp-filter
dir	lazy-eval, with 3-level filter
d	lazy eval, only fp filter (not always exact)

For each set of input data we not only measure the runtimes of the different implementations but also analyse, how much time is spent in the symbolic/numeric part of computation as we did for the triangulation program. See tables 9 and 10 for the results. Unfortunately, due to the internal representation of LEDA **real**, a second computation of a predicate does not take the same amount of time. So in these cases (especially for the implementations **r** and **orig**), the runtime of the numeric part of computation is *underestimated*.

Furthermore we will analyse all occurring instances of the incircle test — how many can be decided at which stage and whether they are degenerate. See table 11.

As for the triangulation algorithm we use the "Koch isle" for benchmarking the Voronoi program. Due to the considerably more complex computations of the RIC-AVD algorithm, we restrict to a recursion depth of 4 (768 nodes).

5.3. Summary

	double	integer	real	$\sum$
non deg.	497166 (99.22%)	3042 (0.61%)	66 (0.01%)	500274 (99.84%)
deg.	0 (0%)	718 (0.14%)	62 (0.01%)	780 (0.16%)
$\sum$	497166 (99.22%)	3760 (0.75%)	128 (0.03%)	501054 (100%)

Table 11: VoroDiag, bit-length 24: #of tests

The synthetic (low-level) benchmarks showed that the floating-point filter itself only introduces an overhead of about factor 2-3 compared to **double** arithmetic, whereas the LEDA datatypes for arbitrary precision introduce an overhead of at least factor 20. Of course, when applying the expression compiler to actual applications, the speed-up is not as high as the low-level benchmarks may suggest, since the algorithms also spend time in symbolic computations, which are not or less affected by the use of a floating-point filter. Nevertheless, the two examples where the expression compiler was used, showed a considerable speed-up. After a closer examination, the speed-up of the numerical part of computation showed to be near the one suggested by the low-level benchmarks. It also turned out that LEDA **real** s, which perform quite badly in degenerate cases, can be efficiently tuned using an intermediate LEDA **integer** stage.

6. Conclusion

The expression compiler we have implemented has proven to be very useful for tuning implementations of geometric algorithms. In contrast to existing solutions it supports all operations $(+, -, \cdot, /, \sqrt{})$, storing intermediate results for later use in predicates and various filter stages. Existing code can be reused and converted in a very transparent manner. The performance of the resulting programs is very good. Nevertheless, correctness of the optimized code is always ensured, e.g. possible underflow conditions are not neglected.

Our implementation can easily be adapted to other non-LEDA datatypes for arbitrary precision numbers, in the future it may also be possible to let the expression compiler generate directly the arbitrary precision code and not use an external datatype (like for example Fortung/vanWyk's LN).

The original work ([8]) and the source code can be found at

`http://www.mpi-sb.mpg.de/~funke`

References

1. F. Avnaim et al.: *Evaluating signs of determinants using single-precision arithmetic*, (INRIA, 1995)
2. D. H. Bailey: *A Portable High Performance Multiprecision Package*, (Technical Report RNR-90-022, NASA Ames Research Center, May 1993)
3. H. Brönnimann, C. Burnikel, S. Pion: *Interval Analysis Yields Efficient Arithmetic*

Filters for Computational Geometry, (Proceedings of the 14th Symposium on CG 1998, ACM Press)

4. C. Burnikel: *Exact Computation of Voronoi Diagrams and Line Segment Intersections*, (PhD thesis, Max-Planck-Institut für Informatik, 1996)
5. Christoph Burnikel, Kurt Mehlhorn, Stefan Schirra: *The LEDA class real number*, (Research report MPI-I-96-1-001, Max-Planck-Institut für Informatik, 1996)
6. O. Devillers, P. Preparata: *A Probabilistic Analysis of the Power of Arithmetic Filters*, (Technical Report CS 96-27, Center for Geometric Computing, Dept. Computer Science, Brown Univ.)
7. Steven Fortune, Christopher J. Van Wyk: *Efficient Exact Arithmetic for Computational Geometry*, (Proceedings of the 9th Annual Symposium on CG 1993, ACM Press)
8. S. Funke: *Exact Arithmetic using Cascaded Computation*, (Master-Thesis, Max-Planck-Institut für Informatik, 1997)
9. David Goldberg: *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, (ACM Computing Surveys 23(1):5-48, March1991)
10. *IEEE standard 754-1985 for binary floating-point arithmetic*, reprinted in SIGPLAN 22, 2:9-25, 1987
11. K. Mehlhorn, S. Näher, C. Uhrig: *The LEDA User Manual (Version R 3.4)* (Max-Planck-Institut für Informatik, 1996)
12. K. Mehlhorn, S. Näher: *The Implementation of Geometric Algorithms*, Proc. 13th World Computer Congress IFIP94, vol.1, 223-231
13. M. Seel: *Eine Implementierung abstrakter Voronoidiagramme*, (Master Thesis, Universität des Saarlandes 1994)
14. R. Seidel: *A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons*, (Computational Geometry: Theory and Applications 1 (1991) 51-64)
15. B. Serpette, J. Vuillemin, J.C. Herve: *BigNum: a portable and efficient package for arbitrary-precision arithmetic*, (INRIA)
16. K. Sugihara, M. Iri: *A robust Topology-Oriented Incremental Algorithm for Voronoi diagrams*, (Internat. J. Comput. Geom. Appl. 4 (1994) 179-228)