Lecture 6
Efficiency Issues in EGC

Tarski's Geometry provides a universal framework for describing most common geometric tasks. We also have the basic computational tools for Tarski's Geometry: resultants, sturm sequences, algebraic number manipulation, cell decomposition. We might conclude from this that the geometric robustness problem within this framework is solved "in principle". These facts were known in the early 80's and began to be exploited (e.g., [15] addresses topologically correct decisions in CAD primitives). But serious practical obstacles remain, mostly to do with efficiency. Consider the simplest situation: geometric (CAD) modeling involving only linear objects. Yu [21] addresses this case. Here, only rational arithmetic is needed, and yet Yu concluded that exact computation in this setting is not practical in the foreseeable future. His reason was that the numbers in such computations become too large. But the approach of reducing all computation to exact rational arithmetic is only the first cut – at least it tells us where the difficulties might lie. We already know from the previous lecture that approximate arithmetic can be useful in exact geometric computation. In this lecture, we examine several computational techniques which speed up our computation, especially by exploiting approximate arithmetic.

§1. Expression Packages

An efficient Big Number package is clearly very important. One effort at creating an efficient Big Number core is [17, 18]; the experience there suggests that the use of low level assembly language coding can speedup a big number integer package by a factor of 3. We feel that there are many other important ideas that remains to be more fully explored at the arithmetic front: one is the idea of redundant arithmetic [10]. Another is the notion of incremental arithmetic [20]. Another potent idea is some infrastructure to allow users to freely substitute alternative Big Number packages into the system (see `LiDIA` [14]).

Here we want to focus on another key observation: *arithmetic in a geometric algorithm is invariable performed during the repeatedly evaluating a small number of primitives, each evaluation on different arguments.* This suggest that we focus on such complex aggregates of arithmetic operations (called "expressions"); our optimization efforts should be aimed at such expressions rather than at the level of individual arithmetic operations. The software embodiment of these idea is called **expression packages**, built upon Big Number packages. The simplest class of primitives are based on polynomial expressions $(+, -, \times)$. Next, we have rational expressions $(+, -, \times, \div)$, and radical expressions $(+, -, \times, \div, \sqrt{\cdot})$. We shall describe expression packages and their properties.

§1.1. The `LN` System.

One key technique here is the "compilation" of expressions. To see what can be optimized along this line, we describe a system called `LN` ("Little Numbers") from Fortune and Van Wyk [11, 10, 12]. It provides a simple language to define (1) numerical data types and (2) expressions. The numerical data types are basically fixed-size multiprecision numbers (only integers are available in the initial implementation). A preprocessor (also called `LN`) compiles such an `LN` program into `C++` code. This code is extremely efficient since

- Loops are unrolled into straightline arithmetic programs
- Function calls are eliminated by direct code substitution
- Multiprecision numbers are used for exact computation, but their storage sizes are known at compile time. This avoid dynamic allocation of storage.
- Floating point filters are generated to avoid multiprecision arithmetic when possible.

- Various classic compiler optimization techniques can be introduced: common subexpressions, constant folding, etc.

Here is an example of an `LN` program:

```
[[
tuple Point(x,y)
macro orient(a:Point, b:Point, c:Point) =
sign( ( ....);
class integer = int<31>;
Point;
proc orient;
]]
...
// C++ code
...
```

The portion inside `[[...]]` is compiled into `C++` classes and methods, while code outside `[[...]]` are copied out verbatim. If this code is stored in geom.e, then `LN` will output two files, geom.h, and geom.cc. These files can now be included or compiled in the usual way.

The key words "tuple" and "macro" define (respectively) `LN` data types and `LN` expressions. The key words "class" and "proc" produces (respectively) `C++` class (or typedefs) definitions and `C++` methods.

An interesting side effect of `LN` 's aggressive optimization of time efficiency is the code bloat – the generated `C++` code can be very large. Milenkovic and Chang [8] reported that the compiler ran out of space while compiling an `LN` code for Boolean operations on planar polygons. This suggests that a less aggressive version of `LN` may be useful.

§2. Program Filters

In the previous lecture, we exploited approximate computation in EGC through the use of BigFloat arithmetic. But another obvious avenue for exploiting approximate is machine arithmetic: machine arithmetic is fast and highly optimized! More generally, we examine the uses of programs that are "always fast but sometimes incorrect". Let us euphemistically call such programs **useful**. This seems antithetical to the whole idea of EGC. But it will actually hold the key to making EGC practical. Of course, we must deploy useful programs in our algorithms in a principled way.

The intuitive idea is easy: we must "check" or "verify" or "certify" the output of useful programs. In our discussions, we will try to make the following formal distinction between checking and filtering: both are procedures which, given an input-output pair, will answer "yes" or "no". Both are required to say "no" for an inappropriate input/output pair. However, a checker must also say "yes" for an appropriate input/output pair. See the formal definition of these concepts below.

The related area of program checking [2, 3] lends some perspective on how to do this. The approach of program checking is a great relief from a burden that theoreticians imposed on themselves in the 1970s: then, under the weight of increasingly complex programs, the battle cry was "program verification". This is the task of verifying that each purported algorithm P for a problem π is indeed an algorithm for π. Program checking says – I don't care how you got the answer (using P or not) – I just need to check that the output is correct. Checkers are easier to construct than verifiers. Our focus is on filters – these are even easier to construct than checkers. They are used for efficiency purposes, not for correctness purposes.

Actually, this lecture will focus on a special class of filters called **numerical filters**. These certify some property of a computed numerical value. This often amount to computing error bounds. When such filters aim to certifying machine floating-point arithmetic, we call them **floating point filters**.

There are two main classification of numerical filters: static or dynamic. Static filters are those that can be computed at compile time, and they incur a low overhead at runtime. However, static bounds may be overly pessimistic and thus less effective. Dynamic filters exhibit opposite characteristics: they have higher runtime cost but are much more effective. In fact, we can have have semi-static filters which combines both features.

Filters can be used at different levels of granularity: from individual machine operations (e.g., arithmetic operations), to subroutines (e.g., geometric primitives) to algorithms (e.g. convex hull or meshing algorithm) to entire systems.

Computing Upper Bounds in Machine Arithmetic. In the implementation of numerical filters, we need to compute sharp upper bounds on numerical expressions. To be specific, suppose you have IEEE double values x and y and you want to compute an upper bound on $|z|$ where $z = xy$. How can you do this? We can compute

$$\widetilde{z} = |x \odot y|. \tag{1}$$

Here, $|\cdot|$ is done exactly by the IEEE arithmetic, but the multiplication is not exact. One aspect of IEEE arithmetic is that you can change the rounding modes (see Lecture I). So if you have change the the rounding mode to round away from 0 or round towards $+\infty$, we will have $\widetilde{z} \geq xy$. Otherwise, we only know that $\widetilde{z} = z(1+\delta)$ where $|\delta| \leq \mathbf{u}$. Here $\mathbf{u} = 2^{-53}$ is the "unit of rounding" for the arithmetic. Unfortunately, turning the rounding mode one or off seems like an expensive proposition, not meant to be used in the middle of a program frequently. We will show that $\widetilde{w}$, computed as follows

$$\widetilde{w} := \widetilde{z} \oplus (\widetilde{z} \odot 4\mathbf{u}), \tag{2}$$

is an upper bound on $z = xy$. It is assumed that overflow and underflow does not occur in computing $\widetilde{w}$.

Note that $4\mathbf{u} = 2^{-51}$ is exactly representable. Furthermore, the product $\widetilde{z} \odot 4\mathbf{u}$ is exact as it involves shifting the exponent only. Therefore, we know that $\widetilde{w} = \widetilde{z}(1+2\mathbf{u})(1+\delta')$ for some δ satisfying $|\delta'| \leq \mathbf{u}$. Hence,

$$\begin{aligned} \widetilde{w} &= z(1+\delta)(1+\delta')(1+4\mathbf{u}) \\ &\geq z(1-2\mathbf{u}+\mathbf{u}^2)(1+4\mathbf{u}) \\ &= z(1+2\mathbf{u}-7\mathbf{u}^2+4\mathbf{u}^3) \\ &> z \end{aligned}$$

Note that if any of the operations $\oplus$, $\ominus$ or $\oslash$ is used in place of $\odot$ in (1), the same argument still shows that $\widetilde{w}$ is an upper bound on the actual value.

We summarize a general result for later application. Let E be any rational numerical expression and let $\widetilde{E}$ be IEEE double value approximation to E.

Lemma 1 *Assume the input numbers in E are IEEE doubles and E has k operations. Then we can compute a IEEE double* `MaxAbs`(E) *which satisfies $|E| \leq maxabs(E)$ in $4k$ machine operations. Moreover, the value $\widetilde{E}$ is available at no extra costs if desired.*

Proof. In proof, we simply replace each rational operation in E by 4 machine operations: we count 2 operations in each of equations (2) and (1) for computing $\widetilde{z}$ and $\widetilde{w}$. **Q.E.D.**

Aside: A Model for Checking and Filtering. We give a simple formal model for checking and filtering. Assume I and O are some sets called the **input and output spaces**. In numerical problems, it is often possible to identify I and O with $\mathbb{R}^n$ for some n. A **computational problem** is simply a subset $\pi \subseteq I \times O$. A **program** is simply partial function $P : I \to O$. So $P(x)$ may be undefined for some x, denoted $P(x)\uparrow$; if $P(x)$ is defined, then we write $P(x)\downarrow$. We say P is a **partial algorithm** for π if for all $x \in I$, if $P(x)\downarrow$ then $(x, P(x)) \in \pi$. An **algorithm** A for π is a partial algorithm that happens to be a total function. A **filter** for π is a total program $F : I \times O \to \{0, 1\}$ such that $F(x, y) = 1$ implies $(x, y) \in \pi$. A **checker** C for π is a filter for π such that if $C(x, y) = 0$ then $(x, y) \notin \pi$ Thus, a checker is a filter, but not necessarily vice-versa. Finally, a **filtered program** for π is a pair (P, F) such that P is a total program and F is a filter for π. We view (P, F) to be a new program P_F, such that on input x, if $F(x, P(x)) = 1$ then $P_F(x) = P(x)$; otherwise $P_F(x)\uparrow$. Thus P_F is a partial algorithm for π. We want to "anchor" a filtered program (P, F) for π with an algorithm A for π. The triple (P, F, A) would constitute a program $P_{F,A}$ for π: on any input x, we define $P_{F,A}(x) = P_F(x)$ if $P_F(x)\downarrow$, otherwise $P_{F,A}(x) = A(x)$.

We consider the filtered program (P, F) for π because P is presumably some useful program for π. We can consider filter cascades: let F be a sequence

$$F = (F_1, \ldots, F_n) \tag{3}$$

where each F_i is a filter for π. We call F a (n-level) **filter cascade** for π. We view F as a filter for π by defining $F(x, y) = \max_{i=1}^{n} F_i(x, y)$. Thus $F(x, y) = 1$ iff some $F_i(x, y) = 1$. In practice, we evaluate $F(x, y)$ by searching for the first i such that $F_i(x, y) = 1$; and otherwise output 0. Funke, et al [13, 7] are among the first to exploit multi-level filter cascades. A filter cascade (3) in practice would have the property that each F_i is more "effective" but less efficient than F_{i-1}.

Note that this simple model must be augmented by complexity considerations if it is to lead to useful results. Otherwise, the concept of "useful" programs, "efficient" solutions, etc, cannot be captured. These concepts are the motivations for filters after all!

Exercises

Exercise 2.1: (a) Write a `C/C++` program which performs a sequence of arithmetic operations, and reports on the types of IEEE exceptions turned on by each.
(b) Perform some timing to determine the cost of testing for these exceptions. □

Exercise 2.2: (a) Write a `C/C++` program which turns on the various rounding modes, and prints the results of rounding.
(b) Perform some timing to determine the cost of switing rounding modes. □

End Exercises

§3. Static Filters

Expression E	$\texttt{MaxLen}(E)$	$\texttt{MaxErr}(E)$
Variable x	$\texttt{MaxLen}(x)$ given	$\max\{0, 2^{\texttt{MaxLen}(E)-53}\}$
$E = F \pm G$	$1 + \max\{\texttt{MaxLen}(F), \texttt{MaxLen}(G)\}$	$\texttt{MaxErr}(F) + \texttt{MaxErr}(G) + 2^{\texttt{MaxLen}(F\pm G)-53}$
$E = FG$	$\texttt{MaxLen}(F) + \texttt{MaxLen}(G)$	$\texttt{MaxErr}(F)^{\texttt{MaxLen}(G)} + \texttt{MaxErr}(G)^{\texttt{MaxLen}(F)}$ $+2^{\texttt{MaxLen}(FG)-53}$

Figure 1: Table 1: Parameters for the FwW Filter

Fortune and Van Wyk [12] were one of the first to implement and quantify the the efficacy of filters for exact geometric computation. Their filter was implemented via the `LN` preprocessor system. We now look at the simple filter they implemented (which we dub the "FvW Filter"), and some of their experimental results.

The FvW Filter. Static error bounds is easily maintained for a polynomial expression E with integer values at the leaves. Let $\widetilde{E}$ denote the IEEE double value obtained by straightforward evaluation of E using IEEE double operations. Fortune and Van Wyk computes a bound $\texttt{MaxErr}(E)$ on the absolute error,

$$|E - \widetilde{E}| \leq \texttt{MaxErr}(E). \tag{4}$$

It is easy to use this bound as a filter to certify the sign of $\widetilde{E}$: if $|\widetilde{E}| > \texttt{MaxErr}(E)$ then $\texttt{sign}(\widetilde{E}) = \texttt{sign}(E)$. Otherwise, we must resort to some fall back action. Unless we specify otherwise, the default action is to immediately use an infallible method, namely computing exactly using a Big Number package.

Let us now see how to compute $\texttt{MaxErr}(E)$. It turns out that we also need the magnitude of E. The base-2 logarithm of the magnitude is bounded by $\texttt{MaxLen}(E)$. Thus, we say that the FvW Filter has two **filter parameters**,

$$\texttt{MaxErr}(E), \quad \texttt{MaxLen}(E). \tag{5}$$

We assume that each input variable x is assigned an upper bound $\texttt{MaxLen}(x)$ on its bit length. Inductively, if F and G are polynomial expressions, then $\texttt{MaxLen}(E)$ and $\texttt{MaxErr}(E)$ are defined using the following rules.

Observe that the formulas in Table 1 assumes exact arithmetic. In implementations, we have to be careful to compute upper bounds on these formulas. We assume that the filter has failed in case of an overflow; it is easy to see that no underflow occurs when evaluating these formulas.

Since $\texttt{MaxLen}(E)$ is integer, we can evaluate the corresponding formulas using IEEE arithmetic exactly. But the formulas for $\texttt{MaxErr}(E)$ will incur error, and have to use the techniques noted about for computing upper bounds.

Framework for Measuring Filter Efficacy. We want to quantify the efficacy of the FvW Filter. Consider the primitive of determining the sign of a 4×4 integer determinant. First look at the unfiltered performance of this primitive. We use the IEEE machine double arithmetic evaluation of this determinant (with possibly incorrect sign) as the base line for speed. It is, of course, the standard procedure. This base performance is then compared to the performance of some (off-the-shelf) Big Integer packages. The numbers cited here are for the Big Integer package in `LEDA` (circa 1995), but the general conclusion for other packages are apparently not much different. For random 31-bit integers, the time increases by a factor of 60 over the base line. We will say

$$\sigma = 60 \tag{6}$$

in this case; the symbol σ reminds us that this is the "slowdown" factor. Clearly, σ is a function of the bit length L as well. For instance, with random 53-bit signed integers, the factor σ becomes 100. Next, still with $L = 31$, but using static filters implemented in `LN` , the factor σ ranges from 13.7 to 21.8, for various platforms and CPU speeds [12, see Figure 14]. For simplicity, we simply say $\sigma = 20$, for some mythical combination of these platforms and CPUs. Thus the static filters improve the performance of exact arithmetic by the factor

$$\phi = 60/20 = 3. \tag{7}$$

In general, using unfiltered exact integer arithmetic as base line, the symbol ϕ denotes the "filtered improvement". It gives us a measure of the efficacy of filtering.

In summary, the above experimental framework is clearly quite general, and aims to reduce the efficacy of a particular filter by estimating a number ϕ. In general, the framework requires the following choices: (1) a "test algorithm" (we picked one for 4×4 determinants), (2) the "base line" (the standard is IEEE double arithmetic), (3) the "top line" (we picked `LEDA`'s Big Integer), (4) the input data (we consider random 31-bit integers). Its simplicity means that we can extract this ϕ factor from almost any published experimental papers. Another measure of efficacy is the fraction ρ of correct approximate values $\widetilde{E}$ which fails to pass the filter. Unfortunately, this data is less readily available in the papers we cite.

Realistic versus Synthesis Problems. Static filters has an efficacy factor $\phi = 3$ (see (7)) in evaluating the sign of randomly generated 4-dimensional matrices ($L = 31$). BFS [7] call such problems are "synthetic benchmarks". So it would be interesting to see the performance of filters using "realistic benchmarks", meaning actual algorithms for natural problems that we want to solve. But even here, there are degrees of realism. Holding on tightly to our theoretician's hat, we will equate realistic benchmarks with algorithms for problems such as convex hulls, triangulations, etc. We will look at one of these benchmarks from Sugihara below.

But before we do this, we might note that realistic benchmarks will involve a significant amount of non-numeric computation. Hence the ϕ-factor in such settings may not look as good as our synthetic ones. To quantify this, suppose that a fraction

$$\beta \quad (0 \le \beta \le 1) \tag{8}$$

of the running time of the algorithm is attributable to numerical computation. After replacing the machine arithmetic with exact integer arithmetic, the overall time becomes $(1-\beta)+\beta\sigma = 1+(\sigma-1)\beta$. With filtered arithmetic, the time becomes $1 + (\sigma - 1)\beta\phi^{-1}$. Thus efficacy factor ϕ' for the algorithm is really

$$\phi' := \frac{(1 + (\sigma - 1)\beta}{1 + (\sigma - 1)\beta/\phi}.$$

It is easy to verify that $\phi' < \phi$ (assuming $\phi > 1$).

Note that our derivation assumes the original time is unit! This normalization is valid in our derivation because all the factors σ, ϕ that we use are ratios and are not affected by the normalization.

The factor β is empirical, of course. But even so, how can we estimate this? For instance, for 2- and 3-dimensional Delaunay triangulations, Fortune and van Wyk [12] noted that $\beta \in [0.2, 0.5]$. Burnikel, Funke and Schirra [7] suggests a very simple idea for obtaining β: simply execute the test program in which each arithmetic operation is repeated $c > 1$ times. This gives us a new timing for the test program,

$$T(c) = (1 - \beta) + c\beta.$$

Now, by plotting the running time $T(c)$ against c, we obtain β as the slope.

Static Filters for 4-Dimensional Convex Hulls. To see static filters in action in the context of an actual algorithm, we follow a study of Sugihara [19]. He uses the well-known beneath/beyond algorithm for computing the convex hull of a set of 4-dimensional points, augmented with symbolic perturbation techniques. By using static filters, Sugihara reported a 100-fold speedup for nondegenerate inputs, and 3-fold speedup for degenerate inputs (see table below).

The main primitive concerns 5 points A, B, C, D, E in $\mathbb{R}^4$. The sign of the following determinant

$$D = \begin{bmatrix} 1 & A_x & A_y & A_z & A_u \\ 1 & B_x & B_y & B_z & B_u \\ 1 & C_x & C_y & C_z & C_u \\ 1 & D_x & D_y & D_z & D_u \\ 1 & E_x & E_y & E_z & E_u \end{bmatrix}$$

determines whether the tetrahedron (A, B, C, D) is visible from E. Of course, this is the primitive for all standard convex hull algorithms, not just the beneath/beyond method. If the coordinates are L bit integers, Hadamard bound says that the absolute value of the determinant is at most $25\sqrt{5}2^{4L}$, and thus a $6 + 4L$ bit integer. If $L = 28$, it suffices to use 4 computer words. Using IEEE 64-bit doubles, the mantissa is 53 bits, so δ the relative error in a number, is at most $2^{-53} < 4 \times 10^{-15}$.

The determinant, evaluated as a 4×4 determinant has at most $4! = 24$ terms, and each term is at most d. Thus, if the error in the determinant is at most

$$\Delta := \frac{24}{10^{15}} d.$$

So if $|D'|$ (the computed value) is at least Δ, the sign is correct.

[Display table of values from Sugihara here]

[COMPUTE the PHI factor]

Exercises

Exercise 3.1: Determine the value of β for any realistic algorithm of your choice. Does this β depend on the size of the input? □

End Exercises

§4. Dynamic Filters

To improve the quality of the static filters, we can use runtime information about the actual values of the variables, and dynamically compute the error bounds. We can again use `MaxErr`(E) and `MaxLen`(E) as found in Table 1 for static error. The only difference lies in the base case: for each variable x, the `MaxErr`(x) and `MaxLen`(x) can be directly computed from the value of x.

Dynamic Version of the FvW Filter. Looking at the formulas in table 1, we see that each arithmetic operation may be accompanied by up to 6 additional operations plus a comparison. This operation count assumes each $+$, taking powers, etc, in Table 1 takes one operation. This implies a factor of at least 7 overhead.

But in a machine implementation, the overhead is much more because the Table 1 assumes exact arithmetic. This is not a problem when computing $\mathtt{MaxLen}(E)$ because they involve integer arithmetic. But the computation of $\mathtt{MaxErr}(E)$ is a floating point computing. In lemma 1, we show that each exact arithmetic operation can be replaced by at most 4 flops. The worst case in the formula for $E = FG$:

$$\mathtt{MaxErr}(E) = \mathtt{MaxErr}(F)^{\mathtt{MaxLen}(G)} + \mathtt{MaxErr}(G)^{\mathtt{MaxLen}(F)} + 2^{\mathtt{MaxLen}(FG)-53}.$$

We assume $2^{\mathtt{MaxLen}(FG)-53}$ can be computed exactly in 2 flops. There remain 4 other exact operations, which requires $4 \times 4 = 16$ flops. Hence the bound here is 18 flops. A similar analysis for $E = F + G$ yields 10 flops. When the flops for computing $\mathtt{MaxLen}(E)$ and the single comparison are added, we get 14 flops for $E = F + G$ and 20 flops for $E = FG$.

Fortune and Van Wyk gave somewhat similar numbers: let f_e count the extra runtime operations, including comparisons; let f_r count the runtime operations for storing intermediate results. In the LN implementation, $12 \le f_e \le 14$ and $24 \le f_r \le 33$. The $36 \le f_e + f_r \le 47$ overhead is needed even when the filter successfully certifies the approximate result $\widetilde{E}$; otherwise, we may have to add the cost of exact computation, etc. Hence $f_e + f_r$ is a lower bound on the σ-factor when using filtered arithmetic in LN. Roughly the same bound of $\sigma = 48$ was measured for LEDA's system.

Other ideas can be brought into play: we need not immediately invoke the dynamic filter. We still maintain a static filter, and do the more expensive runtime filter only when the static filter fails. And when the dynamic filter fails, we may still resort to other less expensive computation instead of jumping immediately to some failsafe but expensive big Integer/Rational computation. The layering of these stages of computations is called **cascaded filtering** by Burnikel, Funke and Schirra (BFS) [6]. This technique seems to pay off especially in degenerate situations. We next describe the BFS filter.

The BFS Filter. The is a dynamic filter, but it can also be described as "semi-static" (or "semi-dynamic") because one of its two computed parameters is statically determined. Let E be a radical expression, *i.e.*, involving $+, -, \times, \div, \sqrt{\cdot}$. Again, let $\widetilde{E}$ be the machine IEEE double value computed from E in the straightforward manner (this time, with division and square-roots). In contrast to the FvW Filter, the filter parameters are now

$$\mathtt{MaxAbs}(E), \quad \mathtt{Ind}(E).$$

The first is easy to understand: $\mathtt{MaxAbs}(E)$ is simply an upper bound on $|E|$. The second, called the **index** of E, is a natural number whose interpretation is harder to characterize. Together, they satisfy the following invariant:

$$|E - \widetilde{E}| \le \mathtt{MaxAbs}(E) \cdot \mathtt{Ind}(E) \cdot 2^{-53} \tag{9}$$

Of course, the 53 may be replaced by other suitable parameter p if the IEEE arithmetic had other precision.

Table 2 gives the recursive rules for maintaining $\mathtt{MaxAbs}(E)$ and $\mathtt{Ind}(E)$. The base case (E is a variable) is covered by the first two rows: notice that they distinguish between exact and rounded input variables. A variable x is **exact** if its value is representable without error by a IEEE double. In any case, x is assumed not to lie in the overflow range, so that the following holds

$$|\mathtt{round}(x) - x| \le |x| 2^{-53}.$$

Also note that the bounds are computed using IEEE machine arithmetic, denoted

$$\oplus, \quad \ominus, \quad \odot, \quad \oslash, \quad \widetilde{\sqrt{\cdot}}.$$

The question arises: what happens when the operations lead to over- or underflow in computing the bound parameters? It can be shown that underflows for $\oplus$, $\ominus$ and $\widetilde{\sqrt{\cdot}}$ can be ignored, and in the case of $\odot$ and $\oslash$, we just have to add a small constant $MinDbl = 10^{-1022}$ to $\texttt{MaxAbs}(E)$.

Expression E	$\texttt{MaxAbs}(E)$	$\texttt{Ind}(E)$
Exact variable x	x	0
Approx. variable x	$\texttt{round}(x)$	1
$E = F \pm G$	$\texttt{MaxAbs}(F) \oplus \texttt{MaxAbs}(G)$	$1 + \max\{\texttt{Ind}(F), \texttt{Ind}(G)\}$
$E = FG$	$\texttt{MaxAbs}(F) \odot \texttt{MaxAbs}(G)$	$1 + \texttt{Ind}(F) + \texttt{Ind}(G)$
$E = F/G$	$\frac{\lvert\widetilde{E}\rvert \oplus (\texttt{MaxAbs}(F) \oslash \texttt{MaxAbs}(G))}{(\lvert\widetilde{G}\rvert \oslash \texttt{MaxAbs}(G)) \ominus (\texttt{Ind}(G)+1)2^{-53}}$	$1 + \max\{\texttt{Ind}(F), \texttt{Ind}(G)+1\}$
$E = \sqrt{F}$	$\begin{cases} (\texttt{MaxAbs}(F) \oslash \widetilde{F}) \odot \widetilde{E} & \text{if } \widetilde{F} > 0 \\ \widetilde{\sqrt{\texttt{MaxAbs}(F)}} \odot 2^{26} & \text{if } \widetilde{F} = 0 \end{cases} =$	$1 + \texttt{Ind}(F)$

Figure 2: Table 2: Parameters of the BFS Filter

Under the assumption (9), we can use the following criteria for certifying the sign of $\widetilde{E}$:

$$|\widetilde{E}| > \texttt{MaxAbs}(E) \cdot \texttt{Ind}(E) \cdot 2^{-53} \tag{10}$$

Of course, this criteria should be implemented using machine arithmetic (see (2) and notes there). One can even certify the exactness of $\widetilde{E}$ under certain conditions. If E is a polynomial expression (*i.e.*, involving $+, -, \times$ only), then $E = \widetilde{E}$ provided

$$1 > \texttt{MaxAbs}(E) \cdot \texttt{Ind}(E) \cdot 2^{-52}. \tag{11}$$

Correctness of the BFS Filter. Let us just verify the rules just for the case of addition/subtraction.

Finally, we look at some experimental results. Table 3 shows the σ-factor (recall that this is a slowdown factor compared to IEEE machine arithmetic) for the unfiltered and filtered cases. In both cases, the underlying Big Integer package is from `LEDA`'s. The last column adds compilation to the filtered case. It is based on an expression compiler, EXPCOMP, somewhat in the spirit of `LN`. At $L = 32$, the ϕ-factor (recall this is the speedup due to filtering) is $65.7/2.9 = 22.7$. When compilation is used, it improves to $\phi = 65.7/1.9 = 34.6$. [Note: the reader might be tempted to deduce from these numbers that the the BFS filter is efficacious than the FwW Filter. But the use of different Big Integer packages, platforms and compilers, etc, does not justify this conclusion.]

BitLength L	Unfiltered σ	BFS Filter σ	BFS Compiled σ
8	42.8	2.9	1.9
16	46.2	2.9	1.9
32	65.7	2.9	1.9
40	123.3	2.9	1.8
48	125.1	2.9	1.8

Figure 3: Table 3: Random 3×3 Determinant

While the above results look good, it is possible to create situations that filters are ineffective. Instead of using matrices with randomly generated integer entries, we can use degenerate determinants as input. The results recorded in Table 4 indicates that filters have very little effect. Indeed, we might have expected it

BitLength L	Unfiltered σ	BFS Filter σ	BFS Compiled σ
8	37.9	2.4	1.4
16	45.3	2.4	1.4
32	56.3	56.5	58.4
40	117.4	119.4	117.5
48	135.2	136.5	135.1

Figure 4: Table 4: Degenerate 3×3 Determinants

to slow down the computation, since the filtering effort are strictly extra overhead. In contrast, for random data, the filter is almost always effective in avoiding exact computation.

The original paper [6] describes more experimental results, including the performance of the BFS filter in the context of algorithms for computing Voronoi diagrams and triangulation of simple polygons.

Exercises

Exercise 4.1: Implement and perform a direct comparison of the FvW Filter and the BFS Filter to determine their efficacy (*i.e.*, their ϕ-factors) for sign of small determinants. □

End Exercises

§5. Sign of Determinants

We know that the single most important class of expressions in geometric computation is the determinant. We examine some algorithmic techniques for solving this problem.

Clarkson [9] gave a general method for $n \times n$ matrices with L-bit integer entries, assuming limited precision exact integer arithmetic. The method can be very efficient when combined with machine floating point arithmetic for n and L that are moderate. When $n = 2$ or 3, Avnaim et al [1] gave efficient solutions when the inputs are machine representable. In [5], Brönnimann and Yvinec revisit Clarkson's algorithm and generalizes the method of Avnaim et al.

Filters for Determinants. If we use a standard Gaussian elimination algorithm for computing the determinant, then a simple filter is just to compare the absolute value of the computed determinant with the well-known error bounds found in Numerical Analysis books. Pan and Yanquiang [16] pointed out that this filter (which is static) is not very efficacious. They propose instead to use the the distance from the nearest singular instance of a matrix. This requires only a few computation, and leads to much better filtering ratios.

§6. Convex Hulls and Partial Compilation of Expressions

Sometimes, part of the arguments of an expression are fixed while others are vary. We can exploit this by constructing "partially compiled expressions". To see an example of this phenomenon, we look at the Beneath-Beyond Algorithm for convex hulls.

See [Page 147ff in Edelsbrunner].

Let S be a set of points in d dimensions. FOR SIMPLICITY, ASSUME GENERAL POSITION. We sort the points by their first coordinate, and add them in this order. The method (called Beneath-Beyond) begins with a $d+1$ simplex. At each step, we add a new point p to the current hull H:

§7. Root Bounds

We now derive some root bounds on algebraic expressions

§8. Sign Determination in Modular Arithmetic

Modular arithmetic (i.e., residue number systems) is an attractive method of implementing arithmetic because it allows each digit to be computed independently of the others. That is, is avoids the problem of carries, and thus is amenable to parallel execution in machine precision (assuming each digit fits into a machine word). Unfortunately, determining the sign of a number in modular representation seems to be non-trivial. We describe a solution provided by Brönnimann et al [4].

§9. Partial Compilation of Expressions

Sometimes, part of the arguments of an expression are fixed while others are vary. We can exploit this by constructing "partially compiled expressions". To see an example of this phenomenon, we look at the Beneath-Beyond Algorithm for convex hulls.

See [Page 147ff in Edelsbrunner].

Let S be a set of points in d dimensions. FOR SIMPLICITY, ASSUME GENERAL POSITION. We sort the points by their first coordinate, and add them in this order. The method (called Beneath-Beyond) begins with a $d+1$ simplex. At each step, we add a new point p to the current hull H:

§10. Other Topics

1) Precision Driven Computation: e.g., CORE Library.

2) A New Computational Paradigm: – robustness as a computational resource – speed/robustness tradeoff curve – How to Admit Non-robustness but useful code: Checking as an antidote to program verification.

3) Root Bounds for Radical Expressions.

Exercises

Exercise 10.1: Program the partial evaluation idea for the convex hull algorithm. □

End Exercises

References

[1] F. Avnaim, J.-D. Boissonnat, O. Devillers, F. Preparata, and M. Yvinec. Evaluating signs of determinants using single-precision arithmetic. *Algorithmica*, 17(2):111–132, 1997.

[2] M. Blum and S. Kannan. Designing programs that check their work. *J. of the ACM*, 42(1):269–291, Jan. 1995.

[3] M. Blum, M. Luby, and R. Rubinfeld. Self-testing and self-correcting programs, with applications to numerical programs. *J. of Computer and System Sciences*, 47:549–595, 1993.

[4] H. Brönnimann, I. Z. Emiris, V. Y. Pan, and S. Pion. Sign determination in residure number systems. *Theor. Computer Science*, 210:173–197, 1999.

[5] H. Brönnimann and M. Yvinec. Efficient exact evaluation of signs of determinants. *Algorithmica*, 27:21–56, 2000.

[6] C. Burnikel, S. Funke, and M. Seel. Exact arithmetic using cascaded computation. *ACM Symp. on Computational Geometry*, 14:175–183, 1998.

[7] S. Burnikel, Funke. Exact geometric computation using cascading. *to appear in special issue of IJCGA*, 2000.

[8] J. D. Chang and V. Milenkovic. An experiment using LN for exact geometric computations. *Proceed. 5th Canadian Conference on Computational Geometry*, pages 67–72, 1993. University of Waterloo.

[9] K. L. Clarkson. Safe and effective determinant evaluation. *IEEE Foundations of Computer Science*, 33:387–395, 1992.

[10] S. J. Fortune and C. J. van Wyk. Efficient exact arithmetic for computational geometry. In *Proc. 9th ACM Symp. on Computational Geom.*, pages 163–172, 1993.

[11] S. J. Fortune and C. J. van Wyk. LN User Manual, 1993. AT&T Bell Laboratories.

[12] S. J. Fortune and C. J. van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Transactions on Graphics*, 15(3):223–248, 1996.

[13] S. Funke. Exact arithmetic using cascaded computation. Master's thesis, Max Planck Institute for Computer Science, Saarbrücken, Germany, 1997.

[14] LiDIA Homepage, 1998. LiDIA: an efficient multiprecision number package for computational number theory. URL `http://www.informatik.th-darmstadt.de/TI/LiDIA/`.

[15] S. Ocken, J. T. Schwartz, and M. Sharir. Precise implementation of CAD primitives using rational parameterization of standard surfaces. In J. Hopcroft, J. Schwartz, and M. Sharir, editors, *Planning, Geometry, and Complexity of Robot Motion*, pages 245–266. Ablex Pub. Corp., Norwood, NJ, 1987.

[16] V. Y. Pan and Y. Yu. Certified computation of the sign of a matrix determinant. *Proc. 10th ACM-SIAM Symp. on Discrete Algorithms (SODA99)*, pages 715–724, 1999. Accepted for Algorithmica.

[17] B. Serpette, J. Vuillemin, and J. Hervé. BigNum: a portable and efficient package for arbitrary-precision arithmetic. Research Report 2, Digital Paris Research Laboratory, May, 1989.

[18] M. Shand, P. Bertin, and J. Vuillemin. Hardware speedups in long integer multiplication. *2nd ACM Symposium on Parallel Algorithms and Architectures*, 1990. Crete.

[19] K. Sugihara. Exact computation of 4-D convex hulls with perturbation and acceleration. *Proc. 7th Pacific Conference on Computer Graphics and Applications (Pacific Graphics '99)*, 1999. October 5-7, 1999, Seoul, Korea.

[20] C. K. Yap and T. Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, pages 452–486. World Scientific Press, Singapore, 1995. 2nd edition.

[21] J. Yu. *Exact arithmetic solid modeling.* Ph.D. dissertation, Department of Computer Science, Purdue University, West Lafayette, IN 47907, June 1992. Technical Report No. CSD-TR-92-037.