Lecture 10
NUMERICAL FILTERS

This theme of this chapter is the certification of approximate computation. This is best exemplified by a numerical program P that uses machine floating-point arithmetic. How do we certify that the output of P has certain properties? Conceptually, let us think of the certifier C as a program which, given the output of P, will always answer YES or MAYBE. We see that certification is a "partial predicate": it does not have to answer NO. So the MAYBE output could be produced even when the correct answer is YES. Certifiers are very important for the ultimate practical efficiency of EGC algorithms.

§1. Numerical Filters

An avenue to gain efficiency in EGC computation is to exploit machine floating-point arithmetic which is fast and highly optimized on current hardware. The idea is simply this: we must "check" or "certify" the output of machine evaluation of predicates, and only go for the slower exact methods when this fails.

In EGC, certifiers are usually **(numerical) filters**. These filters certify property of computed numerical values, typically its sign. This often amounts to computing some error bound, and comparing the computed value with this bound. When such filters aim to certifying machine floating-point arithmetic, we call them **floating-point filters**. We can also consider a cascade of certifiers of increasing effectivity for the problem. Such cascades can be quite effective [**?**, **?**, **?**]. There is an obvious connection between the notion of certifiers and the area of program checking [**?**, **?**]. It is also worth mentioning that similar techniques have later been used on large determinant sign evaluations based on distance to the nearest singularity [**?**].

There are two main classifications of numerical filters: static or dynamic. Static filters are those that can be computed at compile time for the most part, and they incur a low overhead at runtime. However, static error bounds may be overly pessimistic and thus less effective. Dynamic filters exhibit opposite characteristics: they have higher runtime cost but are much more effective (i.e., fewer false rejections). We can have semi-static filters which combine both features.

Certifiers can be used at different levels of granularity: from individual machine operations (e.g., arithmetic operations for dynamic filters), to subroutines (e.g., geometric predicates [**?**]), and to algorithms (e.g., [**?**]). See Funke et al. [**?**] for a general framework for filtering each "step" of an algorithm.

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. How can you compute an upper bound on $|z|$ where $z = xy$? We first compute

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

Here, $|\cdot|$ is done exactly by the IEEE arithmetic, but the multiplication $\odot$ is not exact. One aspect of IEEE arithmetic is that we can change the rounding modes [**?**]. Thus changing the rounding mode to round towards $+\infty$, we will have $\widetilde{z} \geq |z|$. 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. We will describe the way to use the rounding modes later, in the interval arithmetic section. But here, instead to avoid rounding modes, we further compute $\widetilde{w}$ as follows:

$$\widetilde{w} \leftarrow \widetilde{z} \odot (1 + 4\mathbf{u}). \tag{2}$$

It is assumed that overflow and underflow do not occur during the computation of $\widetilde{w}$. Note that $1+4\mathbf{u} = 1+2^{-51}$ is exactly representable. Therefore, we know that $\widetilde{w} = \widetilde{z}(1+4\mathbf{u})(1+\delta')$ 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 this result:

Lemma 1 *Let E be any rational numerical expression and let $\widetilde{E}$ be the approximation to E evaluated using IEEE double precision arithmetic. Assume the input numbers in E are IEEE doubles and E has $k \geq 1$ operations.*
(i) We can compute an IEEE double value $\texttt{MaxAbs}(E)$ *satisfying the inequality* $|E| \leq \texttt{MaxAbs}(E)$*, in $3k$ machine operations.*
(ii) If all the input values are positive, $2k$ machine operations suffice.
(iii) The value $\widetilde{E}$ is available as a side effect of computing $\texttt{MaxAbs}(E)$*, at the cost of storing the result.*

Proof. We simply replace each rational operation in E by at most 3 machine operations: we count 2 flops to compute $\widetilde{z}$ in equations (1), and 1 flop to compute $\widetilde{w}$ in (2). In case the input numbers are non-negative, $\widetilde{z}$ needs only 1 machine operation. Q.E.D.

0.1 Static Filters

Fortune and Van Wyk [**?**] were the first to implement and quantify the efficacy of filters for exact geometric computation. Their filter was implemented via the `LN` preprocessor system. Let us 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 are easily maintained for a polynomial expression E with integer values at the leaves. Let $\widetilde{E}$ denote the IEEE double value obtained by direct evaluation of E using IEEE double operations. Fortune and Van Wyk compute a bound $\texttt{MaxErr}(E)$ on the absolute error,

$$|E - \widetilde{E}| \leq \texttt{MaxErr}(E). \tag{3}$$

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. For simplicity, assume this action is to immediately use an infallible method, namely computing exactly using a Big Number package.

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

Table 1: Parameters for the FvW filter

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**,

$$\texttt{MaxErr}(E), \quad \texttt{MaxLen}(E). \tag{4}$$

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 rules in Table 1.

Observe that the formulas in Table 1 assume exact arithmetic. In implementations, we 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. Checking for exceptions has an extra overhead. Since $\texttt{MaxLen}(E)$ is an integer, we can evaluate the corresponding formulas using IEEE arithmetic exactly. But the formulas for $\texttt{MaxErr}(E)$ will incur error, and we need to use some form of lemma 1.

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; this is standard procedure. This base performance is then compared to the performance of some standard (off-the-shelf) Big Integer packages. This serves as the **top line** for speed. The numbers cited in the paper 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 top line time yields 60 time increase over the base line. We will say

$$\sigma = 60 \tag{5}$$

in this case; the symbol σ reminds us that this is the "slowdown" factor. Clearly, $\sigma = \sigma(L)$ 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 [**?**, Figure 14]. For simplicity, we say $\sigma = 20$, for some mythical combination of platforms and CPUs. Thus the static filters improve the performance of exact arithmetic by the factor

$$\phi = 60/20 = 3. \tag{6}$$

In general, using unfiltered exact integer arithmetic as base line, the symbol ϕ (or $\phi(L)$) denotes the "filtered improvement". We use it as a measure of the efficacy of filtering.

The above experimental framework is clearly quite general, and estimates the efficacy of a filter by a number ϕ. 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 used random 31-bit integers). Another measure of efficacy is the fraction ρ of approximate values $\widetilde{E}$ which fail to pass the filter. In [**?**], a general technique for assessing the efficacy of an arithmetic filter is proposed based on an analysis which consists of evaluating both the threshold value and the probability of failure of the filter.

For a true complexity model, we need to introduce size parameters. In EGC, two size parameters are of interest: the combinatorial size n and the bit size L. Hence all these parameters ought to be written as $\sigma(n, L)$, $\phi(n, L)$, etc.

Realistic versus synthetic problems. Static filters have an efficacy factor $\phi = 3$ (see (6)) in evaluating the sign of randomly generated 4-dimensional matrices ($L = 31$). Such problems are called "synthetic benchmarks" in [**?**]. It would be interesting to see the performance of filters on **realistic benchmarks**, i.e., actual algorithms for natural problems that we want to solve. But even here, there are degrees of realism. Let us[1] equate realistic benchmarks with algorithms for problems such as convex hulls, triangulations, etc. The point of realistic benchmarks is that they will generally involve a significant amount of non-numeric computation. Hence the ϕ-factor in such settings ought to be different from (in fact, less than) the synthetic setting. To quantify this, suppose that a fraction

$$\beta \quad (0 \leq \beta \leq 1) \tag{7}$$

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 $1 + (\sigma - 1)\beta\phi^{-1}$. So "realistic" efficacy factor ϕ' for the algorithm is

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

It is easy to verify that $\phi' < \phi$ (since $\phi > 1$). 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 [**?**] noted that $\beta \in [0.2, 0.5]$. Burnikel, et al. [**?**] suggest a simple method 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.

Some detailed experiments on 3D Delaunay triangulations have been made by Devillers and Pion [**?**], comparing different filtering strategies; they conclude that cascading predicates is the best scheme in practice. Other experiments on interval arithmetic have been done by Seshia, Blelloch and Harper [**?**].

0.2 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 $\texttt{MaxErr}(E)$ and $\texttt{MaxLen}(E)$ as found in Table 1 for static error. The only difference lies in the base case: for each variable x, the $\texttt{MaxErr}(x)$ and $\texttt{MaxLen}(x)$ can be directly computed from the value of x. It is

[1] While holding on tightly to our theoretician's hat!

possible to make a dynamic version of the FvW filter, but we will not detail it here due to lack of space.

The BFS filter. This 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

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

The first is easy to understand: $\texttt{MaxAbs}(E)$ is an upper bound on $|E|$. The second, called the **index** of E, is a natural number whose rough interpretation is that its base 2 logarithm is the number of bits of precision which are lost (i.e. which the filter cannot guarantee) in the evaluation of the expression. Together, they satisfy the following invariant:

$$|E - \widetilde{E}| \leq \texttt{MaxAbs}(E) \cdot \texttt{Ind}(E) \cdot 2^{-53} \tag{8}$$

The value 2^{-53} may be replaced by the unit roundoff error $\mathbf{u}$ in general. Table 2 gives the recursive rules for maintaining $\texttt{MaxAbs}(E)$ and $\texttt{Ind}(E)$. 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 an IEEE double. In any case, x is assumed not to lie in the overflow range, so that the following holds

$$|\texttt{round}(x) - x| \leq |x| 2^{-53}.$$

The bounds are computed using IEEE machine arithmetic, denoted

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

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

Table 2: Parameters of the BFS filter

Assuming (8), we have the following criteria for certifying the sign of $\widetilde{E}$:

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

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

$$1 > \texttt{MaxAbs}(E) \cdot \texttt{Ind}(E) \cdot 2^{-52}. \tag{10}$$

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`. The last column adds compilation to the filtered case. It is based on an expression compiler, EXPCOMP, somewhat in the spirit of `LN` (see Section 0.3). 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 BFS filter is more efficacious than the FvW 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

Table 3: Random 3×3 determinants

While the above results look good, it is possible to create situations where 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 indicate that filters have very little effect. Indeed, we might have expected it to slow down the computation, since the filtering efforts are strictly extra overhead. In contrast, for random data, the filter is almost always effective in avoiding exact computation.

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

Table 4: Degenerate 3×3 determinants

The original paper [**?**] 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.

Dynamic filter using interval arithmetic. As mentioned, a simpler and more traditional way to control the error made by floating-point computations is to use interval arithmetic [**?**, **?**]. Some work has been done by Pion et al. [**?**, **?**, **?**] in this direction.

Interval arithmetic represents the error bound on an expression E at runtime by an interval $[E_m; E_p]$ where E_m, E_p are floating-point values and $E_m \leq E \leq E_p$. Interval operations such as $+, -, \times, \div, \sqrt{}$ can be implemented using IEEE arithmetic, exploiting its rounding modes. Changing the rounding mode has a certain cost (mostly due to flushing the pipeline of the FPU), but the remark has been made that it can usually be done only twice per predicate: at the beginning by setting the rounding mode towards $+\infty$, and at the end to reset it back to the default mode. This can be achieved by observing that computing $a+b$ rounded towards $-\infty$ can be emulated by computing $-((-a)-b)$ rounded towards $+\infty$. A similar remark can be done for $-, \times, \div$. Therefore it is possible to eliminate most rounding mode changes, which makes the approach much more efficient.

Most experimental studies (e.g. [**?**, **?**]) show that using interval arithmetic implemented this way usually induces a slowdown factor of 3 to 4 on algorithms, compared to floating-point. It is also noted that interval arithmetic is the most efficacious dynamic filter, failing rarely. This technique is available in the CGAL library, covering all predicates of the geometry kernel.

0.3 Tools for automatic generation of code for the filters

Given the algebraic formula for a predicate, it is tedious and error-prone to derive the filtered version of this predicate manually. Therefore tools have been developed to generating such codes.

We have already mentioned the first one, LN, which targets the FvW filter [**?**]. This tool does not address the needs of more complex predicates, which may contain divisions, square roots, branches or loops. Another attempt has been made by Funke et al. [**?**] with a tool called EXPCOMP (standing for expression compiler), which parses slightly modified `C++` code of the original predicate, and produces static and semi-static BFS filters for them.

The `CGAL` library implements filtering using interval arithmetic for all the predicates in its geometry kernel. The filtered versions of these predicates used to be generated by a Perl script [**?**, **?**], but the current approach uses template mechanisms to achieve this goal entirely within C++. The advantage of dynamic filters is that the code generator need not analyze the internal structure of the predicate.

Most recently, Nanevski, Blelloch and Harper [**?**] have proposed a tool that produces filters using Shewchuk's method [**?**], for the SML language, from an SML code of the predicate.Seeing these past and ongoing works, it seems important to have general software tools to generate such numerical code. Such work is connected to compiler technology and static code analysis.

§2. EXTRA NOTES for Filters

Model for certifying and checking. We give a simple formal model for certifying (filtering) and checking. 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 suitable subsets of $\mathbb{R}^n$ or $\cup_{n\geq 1}\mathbb{R}^n$. A **computational problem** is simply a subset $\pi \subseteq I \times O$. For simplicity, we assume that for all $x \in I$, there exists y such that $(x, y) \in \pi$ (for most problems, this condition can be artificially enforced without changing the overall complexity). A **program** is a pair (P, C_P) where $P : I \to O$ is a partial function and $C_P : I \to \mathbb{R}_{\geq 0}$ is its complexity function. Thus P on input $x \in I$ takes time $C_P(x)$. Note that even if $P(x) \uparrow$, the time taken is $C_P(x)$. We usually refer to P as "the program" and leave its complexity function implicit. As a partial function, $P(x)$ may be undefined at x, written $P(x) \uparrow$; otherwise 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. Informally, we say a partial program P is "efficacious" if for "most" $x \in X$, $P(x) \downarrow$.

A pair (P, A) is an **anchored algorithm** for π if P is partial algorithm for π and A is an algorithm for π. This pair (P, A) represents a new algorithm for π in which, for each input x, we first compute $P(x)$; if $P(x) \downarrow$ then output $P(x)$ and otherwise we compute and output $A(x)$. Logically, the algorithm (P, A) could be identical to A. It is the complexity considerations that motivates (P, A). We have

$$C_{(P,A)}(x) = C_P(x) + \delta(x)C_A(x)$$

where[2]

$$\delta(x) = \begin{cases} 0 & \text{if } P(x) \downarrow, \\ 1 & \text{else.} \end{cases}$$

Thus the algorithm (P, A) could be more efficient than A when P is efficacious and more efficient than A.

How do we obtain partial algorithms for π? Let P be an arbitrary total program. A **certifier** for P (relative to π) is a program that computes a total function $F : I \times O \to \{0, 1\}$ such that for all $x \in X$, if $F(x, P(x)) = 1$ then $(x, P(x)) \in \pi$. A **checker** C for P (relative to π) is a certifier for P (relative to π) such that if $C(x, P(x)) = 0$ then $(x, P(x)) \notin \pi$. Thus a checker is[3] a certifier, but not necessarily vice-versa. We call the above pair (P, F) a **certified program** for π. Thus (P, F) is seen as a new program P_F, such that on input x, we first compute $P(x)$ and certify that $F(x, P(x)) = 1$; if certified, we output $P(x)$ and otherwise $P_F(x) \uparrow$.

The main examples we have in mind is when P is the machine-arithmetic implementation of a mathematically valid algorithm for π, so the output of P is not always correct. In many situations, we can construct a certifier F that knows about the operations of P and can certify most outputs of P. An example is where $\pi = \pi_{asd}$ is the problem of "approximate signed determinant". For any input matrix x, and for any real number y, let $(x, y) \in \pi_{asd}$ iff $\texttt{sign}\det(x) = \texttt{sign}y$. Note that we do not care whether y is even close to the true determinant of x, as long as y has the right sign. Let P be some standard

[2]Unlike in recursive function theory, our model assumes that we can detect in a finite (namely, $C_P(x)$) amount of time whether $P(x) \uparrow$ In this example, after detecting $P(x) \uparrow$, we simply call $A(x)$.

[3]To be certified in our sense is similar to passing a driver's test. As in real life, having a driver's certificate meant that one can drive, but not having one does not mean one cannot drive.

algorithm to compute determinants, implemented in machine-arithmetic. In this case, a certifier $F(x, y)$ for P (relative to π_{asd}) can compute the standard error bound $b(x) > 0$ on the output $P(x)$, and output 1 iff $b(x) < |P(x)|$. The pair (P, F) is thus a certified program for π. It turns out that more efficacious certifiers F' based on distance to the nearest singularity can be constructed (Pan and Yu [**?**]). Note that we could insist that F be a checker, in order to increase the efficacy of (P, F). Unfortunately, this may be too strong a requirement. In the case of π_{asd}, we know of no non-trivial checker F that does amount to actually computing the determinant exactly. This would defeat the whole purpose of certifying.

The intuitive idea is easy: we must "verify", "check" or "certify" the output of useful programs. We will make a formal distinction among these three concepts. They are ordered in a hierarchy of decreasing difficulty, with verification having the most stringent requirement. The verification procedure takes as input a program P and some input-output specification S. The goal is to determine whether or not P has the behavior S. A checking procedure (say, specialized to a behavior S) takes an input-output pair (I, O) and will determine whether or not $(I, O) \in S$. This checker does not care how this (I, O) pair is produced (in practice it probably comes from running some program P). Finally, a certifying procedure (again, specialized to S) takes the same input pair (I, O) and again attempts to decide if $(I, O) \in S$. If it says "yes" then surely $(I, O) \in S$. If it says "no", it is allowed to be wrong (we call this **false rejections**). Of course, a trivial certifier is to say "no" to all its inputs. But useful certifiers should certainly do more than this. The main tradeoff for designing certifiers is between **efficacy** (the percentages of false rejections) and efficiency of the procedure.

The area of program checking [**?**, **?**] lends some perspective on how to do this. The related area of program verification requires all code to be proved correct. This requirement is onerous as it is, and is the reason why verification was abandoned. But imagine verifying numerical code. Presumably we must take into account the numerical accuracy issues. But if so, practically all numerical code will fail the program verification process! Program checking from the 1990s takes a more realistic approach: it does not verify programs, it basically checks specific input/output pairs. This ought to be called "problem checking". In program checking, one wants to do more than problem checking, by actually calling the program as a blackbox. This requires additional properties such as self-reducibility.

But our main emphasis in EGC is certifiers which turns out to be very useful in practice. The term **filters** here is used as an alternative terminology for certifiers. A filter not have to recognize that any particular input/output pair is correct – to be useful, it only needs to recognize this for a substantial number of such pairs. Thus, filters are even easier to construct than checkers. Ultimately, filters are used for efficiency purposes, not for correctness purposes.

§2.1. 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 [**?**]. 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 non-degenerate inputs, and 3-fold speedup for degenerate inputs (see table below).

Let us briefly review the method: let the input point set be S and suppose they are sorted by the x-coordinates: $P_1, \ldots, P_n$. Let S_i be the set of first i

points from this sorted list, and let $CH(S_i)$ denote their convex hull. In general, the facets of the convex hull are 3-dimensional tetrahedra. The complexity of $CH(S)$ can be $\Theta(n^2)$. Since this beneath/beyond algorithm is $O(n^2)$, it is optimal in the worst case. We begin by constructing $CH(S_5)$. In general, we show how to transform $CH(S_{i-1})$ to $CH(S_i)$. This amounts to deleting the boundary tetrahedra that are visible from P_i and addition of new tetrahedra with apexes at P_i.

The main primitive concerns 5 points A, B, C, D, E in $\mathbb{R}^4$. The sign of the following 5×5 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 can be evaluated as a 4×4 determinant $A = (a_{ij})$ with $4! = 24$ terms. Let us define

$$d := \max_{i,j,k,\ell} |a_{1i}a_{2j}a_{3k}a_{4\ell}|$$

where i, j, k, ℓ range over all permutations of $\{1, 2, 3, 4\}$. The computed determinant $\det(A')$ is only an approximation to the true determinant $\det(A)$, where A' is some perturbed version of A. It can be shown that if we define

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

then

$$|\det(A') - \det(A)| \leq \Delta.$$

Thus, if the absolute value of $\det(A')$ is greater than Δ, we have the correct sign.

[Display table of values from Sugihara here]

| ... |

[COMPUTE the PHI factor]

Exercise 2.1: Determine the value of β for any realistic algorithm of your choice. Does this β depend on the size of the input? ♢

0.4 Dynamic Filters

The following idea is important in practice: many numbers, at run time, have much smaller value than the worst case bounds. It is important to exploit this but it can only be done at run time.

Suppose we have a predicate $P(x_1, \ldots, x_n)$. Typically, we use a single parameter function $B(L)$ such that if each x_i has at most L bits then $|P(x_1, \ldots, x_n)| > B(L)$ when non-zero. If we have different estimates (or exact

bounds as in dynamic filters) for each of the x_i's then we would like some easy-to-compute function $B(L_1, \ldots, L_n)$ such that $|P(x_1, \ldots, x_n)| > B(L_1, \ldots, L_n)$ if non-zero. We have seen this in our static analysis of Fortune's Voronoi diagram algorithm.

Dynamic version of the FvW filter. It is easy enough to convert the FvW filter into a dynamic one: looking at Table 1, we see that the only modification is that in the base case, we can directly compute $\texttt{MaxLen}(x)$ and $\texttt{MaxErr}(x)$ for a variable x. Let us estimate the cost of this dynamic filter. We already note that $\texttt{MaxLen}(E)$ can be computed directly using the formula in Table 1 since they involve integer arithmetic. This takes 1 or 2 operations. But for $\texttt{MaxErr}(E)$, we need to appeal to lemma 1. It is easy to see that since the values are all non-negative, we at most double the operation counts in the formulas of Table 1. The worst case is the formula for $E = FG$:

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

The value $2^{\texttt{MaxLen}(FG)-53}$ can be computed exactly in 2 flops. There remain 4 other exact operations, which require $2 \times 4 = 8$ flops. Hence the bound here is 10 flops. Added to the single operation to compute $\texttt{MaxLen}(F) + \texttt{MaxLen}(G)$, we obtain 11 flops. A similar analysis for $E = F + G$ yields 8 flops.

After computing these filter parameters, we need to check if the filter predicate is satisfied:

$$|\widetilde{E}| > \texttt{MaxErr}(E).$$

Assuming $\widetilde{E}$ is available (this may incur storage costs not counted above), this check requires up to 2 operations: to compute the absolute value of $\widetilde{E}$ and to perform the comparison. Alternatively, if $\widetilde{E}$ is not available, we can replace $\widetilde{E}$ by $2^{\texttt{MaxLen}(E)}$. In general, we also need to check for floating-point exceptions at the end of computing the filter parameters (the filter is assumed to have failed when an exception occurred). We may be able to avoid the exception handling, e.g., static analysis may tell us that no exceptions can occur.

Fortune and Van Wyk gave somewhat similar numbers, which we quote: 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) [**?**]. This technique seems to pay off especially in nearly degenerate situations. We next describe the BFS filter.

Exercise 2.2: 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. ♢

0.5 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. [**?**].

§2.2. 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: