Lecture 13
NUMERICAL ALGEBRAIC COMPUTATION

We introduce the **numerical algebraic** *mode of computation. This can be viewed as a third form of computing with algebraic numbers. The main features of this computation is that we avoid all explicit manipulation of the polynomials that underlie algebraic numbers. Instead, we only use a numerical approximation. To recover the algebraic properties of the exact numerical values, we only use the theory of zero bounds as developed in the previous chapter.*

In general, the numerical algebraic mode leads to more efficient and practical algorithms. The complexity of numerical algebraic computation is adaptive, making it useful for applications.

§1. What is Numerical Algebraic Computation?

We have described two standard views of algebraic computation: manipulating algebraic relations in $\mathbb{Q}(\alpha)$ and real algebraic computation via Sturm theory. We now describe the third approach, based on numerical approximations.

Although it has some similarities to isolating intervals, the key difference is that we never explicit compute defining polynomials. Instead, we maintain some suitable bounds on algebraic quantities related to our numbers, and remember the construction history (which is an expression). Using such bounds, we can carry out exact comparisons of our algebraic numbers.

What exactly do we need? Suppose we want to compare two numbers, e.g.,

$$\sqrt{2} + \sqrt{3} : \sqrt{5 + 2\sqrt{6}}.$$

These two values are the same, and so regardless of how accurately we approximate the quantities, we cannot decisively conclude that the numbers are equal!

Our premise for this mode of computation goes as follows. Like Kronecker[1], we assume that the original numerical inputs are $\mathbb{Z}$; all other numbers must be explicitly constructed by application of a suitable operator. This is formalized as follows.

Let Ω be a set of algebraic operators. We restrict ourselves to real operators, and so each $f \in \Omega$ is a partial function $f : \mathbb{R}^n \to \mathbb{R}$ for some arity $n = n(f)$. Let $\Omega^{(n)} \subseteq \Omega$ denote the set of operators of arity n. Thus $\Omega^{(0)}$ are the constant operators of Ω. representing the constant operators (i.e., f with $n(f) = 0$). Let $\Omega_0 = \mathbb{Z} \cup \{\pm, \times\}$. We assume that $\Omega_0 \subseteq \Omega$.

[1] We are misappropriating Leopold Kronecker's famous statement, "Die ganzen Zahlen hat der liebe Gott gemacht, alles andere ist Menschenwerk" [God made the whole numbers, all else are man-made]. This statement appeared in H. Weber's memorial article (1892) on Kronecker, and which Weber attributes to a 1866 speech by Kronecker.

Remark: The choice of $\Omega^{(0)}$ can have significant impact on the efficiency of our algorithms. In the simplest case, we have $\Omega^{(0)} = \mathbb{Z}$. But other natural choices are $\Omega^{(0)} = \mathbb{Q}$ or $\Omega^{(0)} = \mathbb{F} = \{n2^m : n, m \in \mathbb{Z}\}$ (floats).

The class of algorithms we study is intuitively easy to understand, for it resembles programs in modern high-level computer languages. We have basic data types like Booleans, integers and also real numbers. Here "real" numbers really do refer to elements of $\mathbb{R}$. However, the input numbers are restricted to $\mathbb{Z}$ (or $\mathbb{Q}$ or $\mathbb{F}$). In computer languagues, these input numbers are sometimes called **literals**. You can perform basic arithmetic operations using the operators in Ω, and you can compare any two real numbers. Basic data structures like arrays and control structures for looping and branching will be available. The loops and branches are controlled by comparisons of real numbers. It is possible to formalize this class of algorithms as the **algebraic computation model** [3].

Our algorithm therefore manipulates real numbers which can be represented by expressions over Ω. Let $Expr(\Omega)$ denote the set of such expressions. Note that each $e \in Expr(\Omega)$ is either undefined or denotes a real algebraic number. We shall write $\texttt{val}(e)$ for the value of e ($\texttt{val}(e) = \uparrow$ when undefined). When we write expressions, we often view them as rooted oriented dags (directed acyclic graphs), with each node labeled by an operator in Ω of the appropriate arity. For instance, all the leaves of the dag must be elements of $\Omega^{(0)}$.

[FIGURE of an EXPRESSION]

CONVENTION: It is convenient to write a plain "e" instead of "$\texttt{val}(e)$" when the context is clear.

§1.1. Relative and Absolute Approximations

Since we will be using approximations extensively, it is good to introduce some convenient notations. We are interested in both relative and absolute approximations. Indeed, the interaction between these two concepts will be an important part of our algorithms.

For real numbers x and p, let $\texttt{Rel}(x, p)$ denote the set of numbers $\widetilde{x}$ such that $|x - \widetilde{x}| \leq 2^{-p}|x|$. Any value y in $\texttt{Rel}(x, p)$ is called a **relative p-bit approximation** of x. Similar, let $\texttt{Abs}(x, p)$ denote the set of all $\widetilde{x}$ such that $|x - \widetilde{x}| \leq 2^{-p}$, i.e., the set of **absolute p-bit approximations** to x. In practice, p will be an integer.

Example. Consider a binary floating point number, $\widetilde{x} = m2^n \in \mathbb{F}$. If $m \neq 0$ is a p-bit integer, then $\widetilde{x} = \pm(b_p b_{p-1}, \ldots, b_1)_2$ where $b_i \in \{0, 1\}$ and $b_p = 1$. Assume $\widetilde{x}$ is an approximation to some value x in the interval $[(m-1)2^n, (m+1)2^n]$. So $|x| \geq (m-1)2^n \geq (2^{p-1} - 1)2^n$. Assume $p \geq 2$. Then $|x| \geq 2^{p-2+n}$, or

$$|x - \widetilde{x}| \leq 2^n \leq |x|2^{p-2}.$$

Hence $\widetilde{x}$ has $p - 2$ relative bits of precision.

Instead of writing "$\widetilde{x} \in \texttt{Rel}(x, p)$", we also use the alternative notation,

$$\widetilde{x} \sim x \ (\texttt{Rel}[p]). \tag{1}$$

Similarly, we write $\widetilde{x} \sim x \ (\texttt{Abs}[p])$ instead of "$\widetilde{x} \in \texttt{Abs}(x, p)$". Note that $y \sim x \ (\texttt{Rel}[p])$ is not equivalent to $x \sim y \ (\texttt{Rel}[p])$, but $y \sim x \ (\texttt{Abs}[p])$ is equivalent to $x \sim y \ (\texttt{Abs}[p])$. Generally, we define concepts using relative approximations, leaving to the reader the task of extend the definition to the absolute case.

We note the following basic relationship:

Lemma 1 *If $\widetilde{x} \sim x \ (\texttt{Rel}[1])$ then $\texttt{sign}(\widetilde{x}) = \texttt{sign}(x)$. In particular, $x = 0$ iff $\widetilde{x} = 0$.*

The proof is left as an exercise.

We are interested in approximating numerical functions. If $f : \mathbb{R}^n \to \mathbb{R}$ is a numerical function, we call a function $\widetilde{f} : \mathbb{R}^{n+1} \to \mathbb{R}$ a **relative approximation** of f if for all $x_1, \ldots, x_n, p \in \mathbb{R}$, we have that

$$\widetilde{f}(x_1, \ldots, x_n, p) \in \mathtt{Rel}(f(x_1, \ldots, x_n), p).$$

Moreover, $\widetilde{f}(x_1, \ldots, x_n, p)$ is undefined iff $f(x_1, \ldots, x_n)$ is undefined. In general, we write $\mathcal{R}(f)$ or $\mathcal{R}f$ to denote any function that relative approximation of f. Similarly, we can define the notion of absolute approximation function, and use the notation $\mathcal{A}(f)$ or $\mathcal{A}f$ for any function that absolute approximation of f.

§2. Approximate Expression Evaluation

The central computational problem of numerical algebraic computation can be posed as follows: given an expression $e \in Expr(\Omega)$ and a p, to compute an approximate value $\widetilde{e} \sim e$ ($\mathtt{Rel}[p]$).

The basic method is to propagate the precision p to all the nodes in E using suitable inductive rules. In the simplest case, this propagation may proceed only in one direction from the root down to the leaves. Then, we evaluate the approximate values at the leaves and recursively apply the operations at each node from the bottom up. In implementations, the approximate values at each node is a float. Algorithms to propagate composite precision bounds are described in [2, 4, 1, 3].

Complications arise when the propagation of precision requires bounds on the magnitude of the values at some nodes. This is where the zero bounds become essential. This topic is taken up in the next lecture but for now, we make a definition:

A total function $B : Expr(\Omega) \to \mathbb{R}_{\geq 0}$ is called a **zero bound function** for expressions over Ω if for all $e \in Expr(\Omega)$, if $\mathtt{val}(e)$ is defined and non-zero, then

$$|\mathtt{val}(e)| > B(e).$$

For example if $B(e) = 0$ for all e, then we call $B(e)$ a trivial bound. Another equally useless zero bound function is $B(e) = |\mathtt{val}(e)|$ whenever $\mathtt{val}(e)$ is defined (and otherwise 0). It is useless because such a function cannot be effectively computed. In the following, we will assume the existence of some easy to compute zero bound function.

A well-known technique in the literature called "lazy evaluation" has similarities to precision-driven computation. The lazy evaluation typically "pumps" increasingly precise values from the leaves to the root, and simply tracks the forward error. If this error is larger than the desired precision at the root, the process is iterated. In other words, the lazy approach typically has only the upward phase of precision-driven computation. Such iterations alone is insufficient to guarantee the sign of an expression, which is another essential ingredient of the precision-driven approach.

Most Significant Bit (msb). Zero bounds aim to bound $|\mathtt{val}(e)|$ away from 0. We will also need to bound $|\mathtt{val}(e)|$ from above. If $\div \in \Omega$, these two types of bounds are essentially interchangeable. Since $|\mathtt{val}(e)|$ can be very large or very small, and these bounds need not be very accurate in practice, it is more efficient to bound only the bit size, i.e., $\lg |\mathtt{val}(e)|$. Define

$$\mu(e) := \lg |\mathtt{val}(e)|. \tag{2}$$

We write $\mu^+(e)$ and $\mu^-(e)$ for any upper and lower bound on $\mu(e)$. Note that $\mu^+(e), \mu^-(e)$ are not functional notations (the value the denote depends on the context). In a sense, relative approximation of x amounts to computing the $\mu(x)$, plus an absolute approximation of x. For, $\widetilde{x} \in \mathtt{Rel}(x, p)$ iff $\widetilde{x} \in \mathtt{Abs}(x, p - \mu(x))$.

Closely related to $\mu(e)$ is the **most significant bit function**: for any real number x, let $\mathtt{msb}(x) := \lfloor \lg |x| \rfloor$. By definition, $\mathtt{msb}(0) = -\infty$. Thus,

$$2^{\mathtt{msb}(x)} \leq |x| < 2^{1+\mathtt{msb}(x)}.$$

If x in binary notation is $\cdots b_2 b_1 b_0 . b_{-1} b_{-2} \cdots$ $(b_i = 0, 1)$ then $\mathtt{msb}(x) = t$ iff $b_t = 1$ and $b_i = 0$ for all $i > t$.

§2.1. Propagating Precision in Basic Operators

Let Ω be a set of operators and $f \in \Omega$ where $f : \mathbb{R}^n \to \mathbb{R}$. We assume the existence of algorithms to compute $f(x_1, \ldots, x_n; \pi)$ where $\pi = \mathtt{Rel}[p]$ or $\mathtt{Abs}[p]$. Such algorithms are given and analyzed by Ouchi[2] for the rational operators and square root. The question we now address is the following: suppose we want to compute $f(x_1, \ldots, x_n; \mathtt{Abs}[p])$. But x_i are not explicitly known, and our goal is to determine $n + 1$ parameters

$$q_1 = q_1(p), \ldots, q_n = q_n(p), q = q(p)$$

such that if

$$wtx_i \sim x_i \ (\mathtt{Abs}[p]) \qquad (i = 1, \ldots, n) \tag{3}$$

then

$$f(\widetilde{x_1}, \ldots, \widetilde{x_n}; \mathtt{Abs}[q]) \sim f(x_1, \ldots, x_n) \ (\mathtt{Abs}[p]). \tag{4}$$

This relation for the operator + is visually illustrated in Figure 1

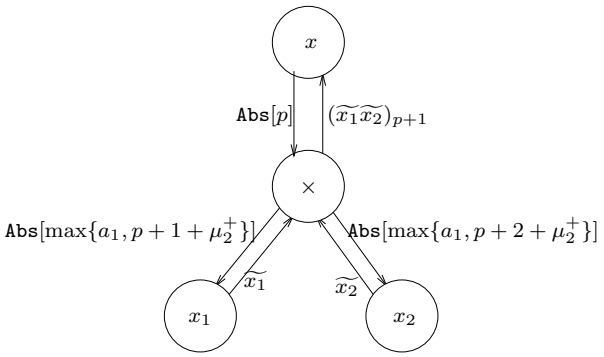

Figure 1: Propagation Rules for Absolute Precision Multiplication

Note that $q_1, \ldots, q_n, q$ can be negative. Also, the problem is trivial when $n = 0$ (just choose $q = p$). The following table summarizes the results:

The proofs can be found in [3]. We could have given this table in terms of relative error bounds although the entry for $\ln(e_1)$ cannot be filled. Even more generally, we can consider composite precision bounds [2]. But it seems that propagating absolute error bounds leads to a nice isolation of the "crucial issue" in approximation of expressions, as we will point out below.

The upward rules in Table 1 are straightforward and uniform: assuming each operator $f \in \Omega$ computes the value of $f(\widetilde{x_1}, \ldots, \widetilde{x_n})$ to at most $p + 1$ bits of absolute precsion. The downward rule can be view as

e	Downward Rules	Upward Rules
$e_1 \pm e_2$	$q_1 = p+2$, $q_2 = p+2$	$\widetilde{e} = (\widetilde{e_1} \pm \widetilde{e_2})_{p+1}$
$e_1 \times e_2$	$q_1 = \max\{a_1, p+1+\mu_2^+\}$, $q_2 = \max\{a_2, p+1+\mu_1^+\}$ where $a_1 + a + 2 = p+2$	$\widetilde{e} = (\widetilde{e_1} \times \widetilde{e_2})_{p+1}$
$e_1 \div e_2$	$q_1 = p+2-\mu_2^-$, $q_2 = \max\{1-\mu_2^-, p+2-2\mu_2^- + \mu_1^+\}$	$\widetilde{e} = (\widetilde{e_1}/\widetilde{e_2})_{p+1}$
$\sqrt[k]{e_1}$	$q_1 = \max\{p+1, 1-(\mu_2^-/2)\}$	$\widetilde{e} = (\sqrt[k]{\widetilde{e_1}})_{p+1}$
$\exp(e_1)$	$q_1 = \max\{1, p+2+2^{\mu_1^+ + 1}\}$	$\widetilde{e} = \exp(\widetilde{e_1}; \texttt{Abs}[p+1])$
$\ln(e_1)$	$q_1 = \max\{1-\mu_1^-, p+2-\mu_1^-\}$	$\widetilde{e} = \ln(\widetilde{e_1}; \texttt{Abs}[p+1])$

Table 1: Downward and Upward Rules for Absolute Precision

§2.2. Precision-Driven Approximation Algorithm

Let us consider the problem of guaranteeing absolute error bounds. We assume the availability of interval arithmetic for all our basic operators. That is, for any $f \in \Omega$, if interval arguments are given to f, then it can compute a reasonably tight interval output. To be specific, suppose f is a binary operator. Write $x \pm 2^p$ for the interval $[x - 2^p, x + 2^p]$. approximations of f, then $f(x\pi 2^{p_y}, y\pi 2^{p_y}) \in f(x,y)\pi 2^{p_z}$ where p_z is as small as possible.

The rules to propagate α_E to the children of E are presented in column 2 of Table 2. If E_1, E_2 are the children of E, we want to define α_{E_i} $(i = 1, 2)$ in such a way that if $\widetilde{E_i}$ satisfies $|E_i - \widetilde{E_i}| \leq \alpha_{E_i}$ then $\widetilde{E}$ satisfies $|E - \widetilde{E}| \leq \alpha_E$. For simplicity, we write α_{E_i} for α_i. Here, $\widetilde{E}$ is obtained by applying the operator at E to $\widetilde{E_1}, \widetilde{E_2}$, computed to some specified precision. The rule for computing this approximation is given in column 3 of Table 2. A notation used in column 3 is that, for any real X and $\alpha > 0$, $(X)_\alpha$ refers to any approximation $\widetilde{X}$ for X that satisfies the bound $|X - \widetilde{X}| \leq \alpha$.

In summary, column 2 tells us how to propagate absolute precision bounds downward towards the leaves, and column 3 tells us how to compute approximations from the leaves upward to the root. At the leaves, we assume an ability to generate numerical approximations to within the desired error bounds. At each node F, our rules guarantee that the approximate value at F satisfies the required absolute precision bound α_F. Since we only propagate absolute precision, and not composite precision, the underlying algorithms for each of the primitive operations $(+, -, \times, \div, \sqrt{}$, etc) is greatly simplified (cf. [2]).

Note that for the addition, subtraction and multiplication operations, the computation of $\widetilde{E}$ can be performed exactly (as in [1]). But the present rules no longer require exact computation, even in these cases. The new rule is never much worse than the old rule (at most one extra bit at each node), but is sensitive to the actual precision needed. In fact, for all operations, we now allow an absolute error of $\frac{\alpha_E}{2}$. Let us briefly justify the rule for the case $E = E_1 \times E_2$ in Table 2. The goal is to ensure $|E - \widetilde{E}| \leq \alpha_E$. Since $|\widetilde{E} - \widetilde{E_1}\widetilde{E_2}| \leq \alpha_E/2$ by our upward rules, it is sufficient to ensure that $|E - \widetilde{E_1}\widetilde{E_2}| \leq \alpha_E/2$. But $|E - \widetilde{E_1}\widetilde{E_2}| \leq \alpha_1|E_2| + \alpha_2|E_1| + \alpha_1\alpha_2 \leq \frac{\alpha_E}{c} + \frac{\alpha_E}{c} + \frac{\alpha_E^2}{c^2}$. So it is sufficient to ensure that $\frac{\alpha_E}{c} + \frac{\alpha_E}{c} + \frac{\alpha_E^2}{c^2} \leq \alpha_E/2$. Solving for c, we obtain $c \geq 2 + \sqrt{4 + 2\alpha_E}$. See [1] for justifications of the other entries.

Column 2 is not directly usable in implementation; the formulas for α_i $(i = 1, 2)$ are expressed in terms of $|E_1|$ and $|E_2|$ to make the formulas easier to understand. Since it is generally neither possible nor desirable to compute $|E_i|$

E	Downward Rules	Upward Rules
$E_1 \pm E_2$	$\alpha_1 = \alpha_2 = \frac{1}{4}\alpha_E$	$\widetilde{E} = (\widetilde{E_1} \pm \widetilde{E_2})_{\frac{\alpha_E}{2}}$
$E_1 \times E_2$	Let $c \geq 2 + \sqrt{4 + 2\alpha_E}$, and $\alpha_1 = \frac{\alpha_E}{c}\min\{1, 1/\lvert E_2\rvert\}$, $\alpha_2 = \frac{\alpha_E}{c}\min\{1, 1/\lvert E_1\rvert\}$.	$\widetilde{E} = (\widetilde{E_1} \times \widetilde{E_2})_{\frac{\alpha_E}{2}}$
$E_1 \div E_2$	$\alpha_1 = \alpha_E\lvert E_2\rvert/4$, $\alpha_2 = \frac{\alpha_E\lvert E_2\rvert}{4\lvert E\rvert + 2\alpha_E}$	$\widetilde{E} = (\widetilde{E_1} \oslash \widetilde{E_2})_{\frac{\alpha_E}{2}}$
$\sqrt[k]{E_1}$	$\alpha_1 = \alpha_E \sqrt[k]{\lvert E_1\rvert}^{k-1}/2$	$\widetilde{E} = (\widetilde{\sqrt[k]{\widetilde{E_1}}})_{\frac{\alpha_E}{2}}$

Table 2: Rules for (1) Propagating absolute precision α_E and (2) Approximation $\widetilde{E}$

exactly, we will replace these by upper or lower bounds on $|E_i|$. We next address this issue.

If $\alpha_E = 0$, the rules in Table 2 propagates this zero value to all its children. This is not generally impossible in our implementation model, where $\widetilde{E}$ is assumed to be a big float value. Although it is possible to generalize this to allow exact representation at a hugh cost in efficiency, we prefer to simply require $\alpha_E > 0$.

§3. Bounds on the Magnitude of Expressions

In `Real/Expr` and `Core Library`, expression evaluation begins by computing bounds on the absolute value of each node (see [2]). We need such bounds for two purposes: (1) propagating absolute precision bounds using the rules in Table 2, and (2) translating a relative precision bound at root into an equivalent absolute one.

We review this magnitude bounds computation. For any expression E, we define its **most significant bit** $\texttt{msb}(E)$ to be $\lfloor \lg(|E|) \rfloor$. Intuitively, the `msb` of E tells us about the location of the most significant bit of E. By definition, the $\texttt{msb}(0) = -\infty$. For efficiency purpose in practice, we will compute a bounding interval $\left[\mu_E^-, \mu_E^+\right]$ that contains $\texttt{msb}(E)$, instead of computing its true value. The rules in Table 3 are used to maintain this interval.

E	μ_E^+	μ_E^-
rational $\frac{a}{b}$	$\lceil \lg(\frac{a}{b}) \rceil$	$\lfloor \lg(\frac{a}{b}) \rfloor$
$E_1 \pm E_2$	$\max\{\mu_{E_1}^+, \mu_{E_2}^+\} + 1$	$\lfloor \lg(\lvert E\rvert) \rfloor$
$E_1 \times E_2$	$\mu_{E_1}^+ + \mu_{E_2}^+$	$\mu_{E_1}^- + \mu_{E_2}^-$
$E_1 \div E_2$	$\mu_{E_1}^+ - \mu_{E_2}^-$	$\mu_{E_1}^- - \mu_{E_2}^+$
$\sqrt[k]{E_1}$	$\lceil \mu_{E_1}^+/k \rceil$	$\lfloor \mu_{E_1}^-/k \rfloor$

Table 3: Rules for upper and lower bounds on $\texttt{msb}(E)$

The main subtlety in this table is the entry for μ_E^- when $E = E_1 \pm E_2$. We call this the **special entry** of this table because, due to potential cancellation, we cannot derive a lower bound on $\texttt{msb}(E)$ in terms of the bounds on E_1 and E_2 only. If the MSB bounds cannot be obtained from the fast floating-point filtering techniques, there are two possible ways to determine this entry in practice. First, we could approximate E numerically to obtain its most significant bit, or to reach the root bound in case $E = 0$. Thus, this method really determines the true value of $\texttt{msb}(E)$, and provides the entry $\lfloor \lg |E| \rfloor$ shown in the table. This numerical approximation process can be conducted in a progressive and

adaptive way (e.g., doubling the absolute precision in each iteration, until it finds out the exact MSB, or reaches the root bit-bound). The second method is applicable only under certain conditions: either when E_1 and E_2 have the same (resp., opposite) sign in the addition (resp., subtraction) case, or their magnitudes are quite different (by looking at their `msb` bounds). In either case, we can deduce the μ_E^- from the bounds on E's children. For instance, if $\mu_{E_1}^- > \mu_{E_2}^+ + 1$ then $\mu_{E_1}^- - 1$ is a lower bound for μ_E^-. This approach could give better performance if the signs of both operands are relatively easier to be obtained.

In the current implementation, we assume μ_E^- and μ_E^+ are machine integers. Improved accuracy can be achieved by using machine doubles. Furthermore, we could re-interprete μ_E^- and μ_E^+ to be upper and lower bounds on $\lg|E|$ (and not $\lfloor \lg|E| \rfloor$). Note that $|\mu_E^+ - \mu_E^-| \leq 2^m$ where m is the number of operations in E.

§4. The Approximation and MSB Algorithms

Two key algorithms will be derived from the above tables: one is APPROX(E, α_E) which computes an approximation of E to within the absolute precision α_E. The other algorithm is ComputeMSB(E), which computes upper and/or lower bounds for `msb`(E), following the rules in Table 3. A third algorithm of interest is **sign determination** for an expression E. Although this can be reduced to APPROX$(E,\ \beta(E)/2)$ where $\beta(E)$ is the root bound, such bound can be overly pessimistic. Instead, we note below how to incorporate the sign determination algorithm into ComputeMSB. The original algorithms for APPROX and ComputeMSB in `Core Library` require the sign of every node in the expression to be determined first. This is no longer required in the algorithm to be presented.

It should be noted that APPROX and ComputeMSB are mutually recursive algorithms: this is because ComputeMSB will need to call APPROX to compute the special entry (for $\mu_{E_1 \pm E_2}^-$) in Table 3. Clearly, APPROX needs ComputeMSB for the downward rules (except for addition or subtraction) in Table 2. It is not hard to verify that this mutual recursion will not lead to infinite loops based on two facts: (1) the underlying graph of E is a DAG, and (2) for the addition/subtraction node, the ComputeMSB may need to call APPROX, while the APPROX will not call ComputeMSB at the same node again. Although it is conceivable to combine APPROX and ComputeMSB into one, it is better to keep them apart for clarity as well for as their distinct roles: ComputeMSB is viewed as a one-time pre-computation while APPROX can be repeatedly called.

APPROX The APPROX algorithm has two phases: (1) distributing the precision requirement down the DAG, and (2) calculating an approximate value from leaves up to the root. Step (2) is straightforward, as it just has to call the underlying algorithms for performing the desired operation ($+, -, \times$, etc) to compute the result to $\alpha_E/2$, the desired absolute precision. We refer the reader to Koji [2] for implementation of these underlying algorithms – it amounts to a multiprecision library with knowledge of precision bounds. As for step (1), we see from Table 2 that we need both upper bounds and lower bounds on `msb` . However, these are not required by every operation. Instead of computing both the upper and lower `msb` bounds at every node, we propose to compute them as needed. The following is immediate from column 2 of the Table 2:

- Addition and subtraction $E = E_1 \pm E_2$. No `msb` bounds on E_1, E_2 are needed to propagate precision bounds.

- Multiplication $E = E_1 \times E_2$. Only the μ^+'s of E_1, E_2 are needed.
- Division $E = E_1 \div E_2$. Only the bounds $\mu^+_{E_1}$ and $\mu^-_{E_2}$ are needed.
- Root extraction $E = \sqrt[k]{E_1}$. Only $\mu^-_{E_1}$ is needed.

Briefly, APPROX(E,α_E) proceeds as follows. It first checks to see if there is already a current approximation to E, and if so, whether this approximation has error less than α_E. If so, it immediately returns. Otherwise, depending on the nature of the operator at E, it checks to see if it already the current upper and/or lower bounds on $|E_i|$ (where E_i are the children of E). This follows the rules noted above. If any of these bounds are not known, it will call ComputeMSB($E, b, b', false$) where b, b' are the appropriate Boolean flags (see below). When this returns, it can compute α_i and recursively call APPROX(E_i, α_i) for the ith child. Finally, it takes the approximate values $\widetilde{E_i}$ available at its children and computes $\widetilde{E}$ from them using the upward rules.

ComputeMSB We describe a refined ComputeMSB algorithm that take three additional arguments: ComputeMSB(E, needUMSB, needLMSB, needSIGN) where the need-variables are Boolean flags. The needUMSB (resp., needLMSB) flag is set to true when an upper (resp., lower) bound on `msb`(E) is needed. The meaning of needSIGN is clear. We see that in all but one case, the sign of E is completely determined by the signs of E's children. Hence we just have to propagate needSIGN to the children. The exception is when E is a $\pm$-node. So we could now define `getSign(E)` $\equiv$ ComputeMSB`(E, false, false, true)`.

In the algorithm, we prevent re-computation of bounds or signs by first checking whether it has been computed before. This is important as nodes are shared. We simply present a self-explanatory ComputeMSB algorithm here with a `C++`-like syntax:

```
Algorithm
  ComputeMSB(E, needUMSB, needLMSB, needSIGN) {

  // Check if any requested computation is necessary:
    if (E.umsb was computed) needUMSB = false;
    if (E.lmsb was computed) needLMSB = false;
    if (E.sign was computed) needSIGN = false;
    if (needUMSB or needLMSB or needSIGN)=false return;

    switch (E.operation_type) {
    case 'constant':
        if (needUMSB) E.umsb = ceilLog(E.value));
                            // ceilLog is ceiling of log_2
        if (needLMSB) E.lmsb = floorLog(E.value));
        if (needSIGN) E.sign = sign(E.value));
        break;
    case '+' or '-':
        if (needLMSB or needSIGN) {     // some optimization omitted
            for (i=1; i<= ceilLog(E.root_bound) ; i++)  {
                APPROX(E, 2^{-i});      // adaptive precisio
                if (log_2 (|E.approx|)  < -i)
                        E.sign = sign(E.approx).
            }
            E.lmsb = floorLog(E.value);
```

```
            E.umsb = ceilLog(E.value);
            E.sign = sign(E.value);
        } else {
            ComputeMSB(E.first, true, false, false);
            ComputeMSB(E.second, true, false, false);
            E.umsb = max{E.first.umsb, E.second.umsb} + 1;
        }
        break;
    case '*':
        ComputeMSB(E.first,  needUMSB, needLMSB, needSIGN);
        ComputeMSB(E.second, needUMSB, needLMSB, needSIGN);
        if (needUMSB) E.umsb = E.first.umsb + E.second.umsb;
        if (needLMSB) E.lmsb = E.first.lmsb + E.second.lmsb;
        if (needSIGN) E.sign = E.first.sign * E.second.sign;
        break;
    case '/':
        ComputeMSB(E.first,  needUMSB, needLMSB, needSIGN);
        ComputeMSB(E.second, needLMSB, needUMSB, needSIGN);
        if (needUMSB) E.umsb = E.first.umsb - E.second.lmsb;
        if (needLMSB) E.lmsb = E.first.lmsb - E.second.umsb;
        if (needSIGN) {
            if (E.second.sign=0)  E.sign = undefined;
            else E.sign = E.first.sign * E.second.sign;
        }
        break;
    case 'k-th root extraction':
        ComputeMSB(E.first, needUMSB, needLMSB, needSIGN)
        if (needUMSB) E.umsb = E.first.umsb / k;
        if (needLMSB) E.lmsb = E.first.lmsb / k;
        if (needSIGN) E.sign = E.first.sign;
        if (needSIGN) {
            if (E.first.sign = -1)  E.sign = undefined;
            else E.sign = E.first.sign.
        }
        break;
     } //switch
} //ComputeMSB
```

This completes our description of a precision-driven evaluation. It should be noted that on top of this evaluation mechanism, the `Core Library` also has a floating-point filter. This, plus several other minor improvements, have been omitted for clarity. for instance, to get the sign of E_1E_2, we first determine the sign of E_1. If $E_1 = 0$, we do not need the sign of E_2. While our new design is an improvement over an older one [2], it still seems to be suboptimal. For instance, to determine the sign of E_1E_2, we always determine the sign of E_1. This is unnecessary if $E_2 = 0$. One possibility is to "simultaneously determine" the signs of E_1 and E_2, stopping this determination when either one returns a 0. The idea is to expend equal amounts of effort for the 2 children, but this may be complicated to implement in the presence of shared nodes.

In summary, we pose as a major open problem to design some precision-driven evaluation mechanism which is provably optimal, in some suitable sense.

Using the results there, we have a new table for downward propagation.

References

[1] C. Li. *Exact Geometric Computation: Theory and Applications*. Ph.d. thesis, New York University, Department of Computer Science, Courant Institute, Jan. 2001. Download from `http://cs.nyu.edu/exact/doc/`.

[2] K. Ouchi. Real/Expr: Implementation of an exact computation package. Master's thesis, New York University, Department of Computer Science, Courant Institute, Jan. 1997. Download from `http://cs.nyu.edu/exact/doc/`.

[3] C. K. Yap. On guaranteed accuracy computation. In F. Chen and D. Wang, editors, *Geometric Computation*. World Scientific Publishing Co., Singapore, 2004. To appear.

[4] 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.