The Design of Core 2: A Library for Exact Numeric Computation in Geometry and Algebra

Jihun Yu, Chee Yap, Zilin Du, Sylvain Pion, Hervé Brönnimann

The Design of Core 2: A Library for Exact Numeric Computation in Geometry and Algebra*

Jihun Yu[1], Chee Yap[1], Zilin Du[2], Sylvain Pion[3], and Hervé Brönnimann[4]

[1]Courant Institute
New York University
New York, NY 10012, USA
{jihun,yap}@cs.nyu.edu

[2]Google Inc
2400 Bayshore
Mountain View, CA, USA
zilin@courant.nyu.edu

[3]INRIA Sophia Antipolis
2004 route des Lucioles, BP 93
06902 Sophia Antipolis, FRANCE
Sylvain.Pion@sophia.inria.fr

[4]CIS Department, NYU Poly
Six MetroTech Center
Brooklyn, NY 11201, USA
hbr@poly.edu

Abstract. There is a growing interest in numeric-algebraic techniques in the computer algebra community as such techniques can speed up many applications. This paper is concerned with one such approach called **Exact Numeric Computation** (ENC). The ENC approach to algebraic number computation is based on iterative verified approximations, combined with constructive zero bounds. This paper describes `Core 2`, the latest version of the `Core Library`, a package designed for applications such as non-linear computational geometry. The adaptive complexity of ENC combined with filters makes such libraries practical.
`Core 2` smoothly integrates our algebraic ENC subsystem with transcendental functions with ε-accurate comparisons. This paper describes how the design of `Core 2` addresses key software issues such as modularity, extensibility, efficiency in the a setting that combines algebraic and transcendental elements. Our redesign preserves the original goals of the `Core Library`, namely, to provide a simple and natural interface for ENC computation to support rapid prototyping and exploration. We present examples, experimental results, and timings for our new system, released as `Core Library` 2.0.

1 Introduction

Most algorithms involving numbers are designed in the Real RAM model of computation In this model (e.g., [28]) real numbers can be directly manipulated, comparisons are error-free, and basic arithmetic operations are exact. But

* Yap, Du and Yu are supported by NSF Grants #CCF-043836, #CCF-0728977 and #CCF-0917093, and with partial support from Korea Institute of Advance Studies (KIAS). Brönnimann is supported by NSF Career Grant 0133599. Brönnimann, Pion, and Yap are supported by an Collaborative Action GENEPI grant at INRIA and an NSF International Collaboration Grant #NSF-04-036, providing travel support for Du and Yu to INRIA.

in actual implementations, real numbers are typically approximated by machine doubles and this leads to the ubiquitous numerical nonrobustness issues that plague applications in scientific and engineering applications. In Computational Geometry, these numerical errors are exacerbated by the presence of discrete geometric relations defined by numbers. The survey articles [15,31] give an overview of nonrobustness issues in a geometric setting.

Now suppose P is a `C++` program using only standard libraries. When compiled, it suffers the expected nonrobustness associated with numerical errors. Imagine a software library with the property that when it is included by P, the compiled program (magically) runs like a real RAM program because all numerical quantities[1] behave like true real numbers. Such a library would be a boon towards eliminating numerical nonrobustness. The `Core Library` [13] was designed to approximate this dream: the program P only needs to insert the following two directives:

```
#include "CORE.h"
#define CORE_LEVEL 3                    (1)
```

Our library (`CORE.h`) will re-interpret the standard number type `double` as an `Expr` object (a directed acyclic graph representing a numerical expression). Indeed, by changing the `CORE_LEVEL` to 1 or 2 in (1), the program P can be compiled into other "accuracy levels", corresponding to machine precision (Level 1) or arbitrary multiprecision (Level 2). Although Levels 1 and 2 fall short of a Real RAM, the ability for a single program P to compile into different accuracy levels has interesting applications in the debug-exploration-release cycle of program development [34]. The purpose of this paper is to present the rationale and design of `Core Library` 2.0 (or `Core 2`). Towards this end, it will be compared to our original design, which refers to `Core Library` 1.7 (or `Core 1`).

§1. On implementing a Real RAM. How do we implement a Real RAM? This dream in its full generality is impossible for two fundamental reasons. First, real numbers are uncountably many while any implementation is no more powerful than Turing machines which can only access countably many reals. The second difficulty is the general impossibility of deciding zeros (equivalently, making exact comparisons) [35]. The largest class beyond algebraic zeros for which zero is decidable are the elementary constants of Richardson [29, 30, 35]; this result depends on the truth of Schanuel's conjecture. What is possible, however, is to provide a Real RAM for interesting subsets of the reals. If program P uses only the rational operations $(\pm, \times, \div)$ then such a library could be a BigRational number package; such a solution may have efficiency issues (e.g., [37]). If P also use the square-root operation, then no off-the-shelf library will do; our precursor to `Core Library` [15] was designed to fill this gap. Since many many basic problems involve at most irrationalities of the square-root kind, such a library is already quite useful. The natural goal of supporting all real algebraic

[1] We are exploiting the ability of `C++` to overload operators. Otherwise, we can use some preprocessor.

numbers was first attained in `Core Library` 1.6 [34]. The other library that supports exact comparisons with algebraic numbers represented by floating-point approximations is `LEDA` [18,19]. Another major library that is premised on exact comparison is `CGAL` [10]. Although `CGAL` does not have its own engine for general exact algebraic computation, its generic programming design supports number kernels such as the `Core Library`. Thus `Core Library` is bundled with `CGAL`, and commercially distributed by `Geometry Factory`. In the last decade, such libraries have demonstrated that the exact comparison approach is a practical means for eliminating nonrobustness in many applications.

The computation of our algebraic program P could, in principle, be carried out by computer algebra systems (CAS). Why is there a need for something like `Core Library`? First of all, if we may use a retail business analogy, many CAS systems adopt the "department store" approach to providing services while `Core Library` takes the "boutique" approach: our main service is a number type `Expr` that allows the simulation of a Real RAM. Our system is aimed at geometric applications that have salient differences from typical CAS applications. CAS are often used for one-of-a-kind computation which might be very difficult. These computation seeks to elucidate the algebraic properties of numbers while geometric applications are interested in their analytic properties [35]. Inputs for geometric algorithms have some combinatorial size parameter n that can be moderately large. The algebraic aspects of its computation is normally encapsulated in a handful of algebraic predicates $Q(\mathbf{x})$ (e.g., orientation predicate) or algebraic expressions $E(\mathbf{x})$ (e.g., distance between two points) where $\mathbf{x} = (x_1, \ldots, x_k)$ represents the input. Evaluating $Q(\mathbf{x})$ or $E(\mathbf{x})$ is easy from the CAS viewpoint, but we must repeat this evaluation many times (as a function that grows with n). See [35,36] for other differences.

§2. Exact Numerical Computation. There are four ingredients in our real RAM implementation:
(a) certified approximation of basic real functions (e.g., [3]),
(b) the theory of constructive zero bounds [5,20],
(c) a precision-driven evaluation mechanism [15], and
(d) filter mechanism [4].

The first two ingredients are essential for any Real RAM implementation; the last two ingredients are key to making the system efficient and practical. The certified approximations in (a) are ultimately dependent on interval techniques [24]. The **constructive zero bound** in (b) is a systematic way to compute a bound $B(E)$ for a numerical expression E such that if E is defined and non-zero, then $|E| > B(E)$. Using this, we are able to do exact comparisons. We can view (c) as a pro-active kind of lazy evaluation – this is expanded below in Section 3.3. Finally, a simplified view of "filters" in (d) is to regard them as certified machine arithmetic. Using them we can cheaply perform exact comparisons in the majority of input instances, despite the fact that exact comparisons are very difficult in the worst case. This form of computation is[2] characterized as

[2] Also known as Exact Geometric Computation (EGC) in the context of geometric applications.

Exact Numeric Computation (ENC) in [35,36]. Computer algebra textbooks (e.g., [6]) list several alternatives for computing with algebraic numbers; to this list, we may now add the ENC approach.

There are many other libraries (e.g., [22, 23, 33]) for arbitrary precision real computation but they do not support exact comparison. They lack the critical ingredient (b). As substitute for exact comparison, they use "ϵ-comparison" that compares numbers up to any desired $\epsilon > 0$ accuracy. Brattka and Hertling [2] provides a theoretical study of Real RAM with ϵ-comparisons. Note that numerical analysts also use this ϵ-accuracy approach. In this paper, we will need to integrate an exact subsystem for algebraic numbers with a new ϵ-accurate part for transcendental numbers.

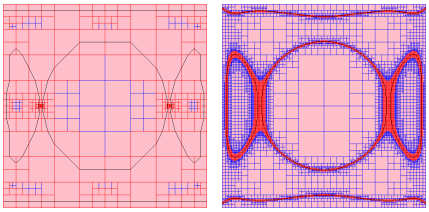


Fig. 1. Isotopic Approximation of curve $\sin_3(x^2) - \cos_3(y^2) = 0$ with `Core 2`.

As an illustration of ENC applications, Figure 1 shows the curve $f(x, y) = \sin_3(x^2) - \cos_3(y^2) = 0$ approximated by `Core 2`, using a recent algorithm [16]. Here $\sin_n, \cos_n$ means we use the first n terms of their Taylor expansions. Our computation in the left figure stops once the isotopy-type is determined; in the right figure, we continue to a user-specified Hausdorff distance. Until recently, most exact computation on algebraic curves and surfaces are based on strong algebraic techniques such as resultant computation (e.g., [1]). In ENC, our main techniques are evaluation and domain subdivision (such subdivision boxes are seen in Figure 1). Superficially, this resembles the traditional numerical approaches, but ENC can provide the topological guarantees [16, 27] that is normally only associated with algebraic algorithms. ENC algorithms have many advantages: adaptive complexity, relatively easy to implement, and locality (i.e., we can restrict computational effort to a local region, as in Figure 1).

§3. Goals of this Paper. There are three main motivations. The first is the desire to increase the modularity, extensibility and maintainability of `Core Library`. These are standard concerns of software engineering, but we shall discuss their unique manifestations in an ENC software. The second is efficiency: the centerpiece of any ENC library is a complex "poly-algorithm" (i.e., a suite of complementary algorithms that work together to solve a problem) to evaluate a numerical expression [15]. The optimal design of this poly-algorithm is far from understood, as we will see much room for improvement. The third is the desire to introduce transcendental computation into the context of ENC. As there are no constructive zero bounds for general transcendental expressions,

we must intermix Level 2 with Level 3 accuracy (cf. (1)). Finally, our redesign is constrained to preserve the simple numerical accuracy API of `Core Library` as illustrated y (1), and to be backward compatible with `Core 1`.

§4. Overview. Section 2 reviews the original design of `Core Library` and discusses the issues. Sections 3 and 4 present (resp.) the new design of the main `C++` classes for expressions and bigFloats. Section 5 describes new facilities to make `Expr` extensible. We conclude in Section 6. Many topics in this paper appear in greater detail in the Ph.D. thesis [8] of one of the authors. The source code for all experiments reported here are found in the subdirectory `progs/core2paper`, found in our open source (QPL license) `Core 2` distribution [7]. Experiments are done on an Intel Core Duo 2.4 GHz CPU with 2 GB of memory. The OS is Cygwin Platform 1.5 and compiler is `g++-3.4.4`.

2 Review of `Core Library`, Version 1

The `Core Library` features an object-oriented design, implemented in `C++`. A basic goal of the `Core Library` is to make ENC techniques transparent and easily accessible to (non-specialist) programmers through a simple numerical accuracy API (illustrated by (1)). User convenience is high priority because we view `Core Library` as a tool for experimentation and rapid prototyping.

There are three main subsystems in `Core 1`: the expression class (`Expr`), the real number class (`Real`) and the big float number class (`BigFloat`). These are number classes, built over standard big number classes (`BigInt`, `BigRat`) which are wrappers around corresponding types from `GNU`'s multiprecision package `GMP`. The `Expr` class provides the critical functionalities of ENC. An instance of `Expr` is a directed acyclic graph (DAG) representing a numerical constant constructed from arbitrary real algebraic number constants. In the following, we raise some issues in the old design of the `Expr` and `BigFloat` classes.

- Some critical facilities in `Expr` should be modularized and made extensible. Specifically, the filter and root bound facilities have grown considerably over the course of development and are now hard to maintain, debug, or extend.
- The main evaluation algorithm (the "poly-algorithm" in the introduction) of `Expr` has three co-recursive subroutines. The old design does not separate their roles clearly, and this can lead to costly unnecessary computations.
- `Core 1` supports only algebraic expressions. An overhaul of the entire design is needed to add support for non-algebraic expressions.
- Currently, users cannot easily add new operators to `Expr`. E.g., it is useful to add diamond operator [5], product, summation (see below), etc.

Next consider the `BigFloat` class. It is used by `Expr` to approximate real numbers, and is the workhorse for the library. It is implemented on top of `BigInt` from `GMP`. The old `BigFloat` is represented by a triple $\langle m, err, e\rangle$ of integers, representing the interval $[(m - err)B^e, (m + err)B^e]$ where $B = 2^{14}$ is the base. We say the bigfloat is **normalized** when $err < B$ and **exact** when $err = 0$. The following issues arise:

- The above representation of `BigFloat` has performance penalty as we must do frequent error normalization. Some applications do not need to maintain error. E.g., in self-correcting Newton-type iterations, it is not only wasteful but may fail to converge unless we zero out the error by calling `makeExact`. Users can manually call `makeExact` but this process is error-prone.
- Our `BigFloat` assumes that for exact bigfloats, the ring operations $(+, -, \times)$ are computed exactly. This is important for ENC but we see situations below where this is undesirable and the IEEE model of round-off is preferable.
- The old `BigFloat` supports only $\{+, -, *, /, \sqrt[2]{}\}$. For `Expr` to support transcendental functions such as exp, sin, etc., we need their `BigFloat` analogues (recall ingredient (a) in ¶2). This implementation is a major effort, and the correct rounding for transcendental functions is quite non-trivial [21].

To bring out these performance penalties, we compare our old `BigFloat` implementation of `sqrt` against `MPFR` [11]: `Core 1` was 25 times slower as seen in Figure 8. The `MPFR` package satisfies all three criteria above. A key feature of `MPFR` is its support of the IEEE rounding modes (the "`R`" in `MPFR` refers to rounding). Hence a critical decision of `Core 2` was to capitalize on `MPFR`.

3 Redesign of the `Expr` Package

We first focus on expressions. The goal is to increase modularity and extensibility of expression nodes, and also to improve efficiency.

3.1 Incorporation of Transcendental Nodes

What is involved in extending expressions to transcendental operators? In `Core 1`, we classify nodes in `Expr` into rational or irrational ones as such information is critical for root bound computation. We now classify them into `integer`, `dyadic`, `rational`, `algebraic`, and `transcendental`. A node is transcendental if any of its descendants has a transcendental operator (e.g., a leaf for $\pi = 3.1415\ldots$, or a unary node such as $\sin(\cdot)$). This refined classification of nodes are exploited in root bounds. There is a natural total ordering on these types, and the type of a node is the maximum of the types in descendant nodes. As transcendental expressions do not have root bounds, we introduce a user-definable global value called **escape bound** to serve as their common root bound.

3.2 New Template-based Design of `ExprRep`

The `Expr` class in `Core 2` is templated, unlike in `Core 1`. It remains only a thin wrapper around a "rep class" called `ExprRep`, which is our focus here.

§5. `ExprRep` and `ExprRepT`. The filter and root bound facilities were embedded in the old `ExprRep` class. We now factor them out into two functional modules: `Filter` and `Rootbd`. The `Real` class (see Section 2), which was already an independent module, is now viewed as an instance of an abstract number module called `Kernel`. The role of `Kernel` is to provide approximate real values. We introduce the templated classes `ExprT` and `ExprRepT`, parametrized by these three modules:

```
template <typename Rootbd,
          typename Filter, typename Kernel>
   class ExprT;

template <typename Rootbd,
          typename Filter, typename Kernel>
   class ExprRepT;
```

Now, `Expr` and `ExprRep` are just typedefs:

```
typedef ExprT<BfmssRootBd<BigFloat2>,
          DummyFilter, BigFloat2> Expr;
typedef ExprRepT<RootBd,
          Filter, Kernel> ExprRep;
```

The actual template arguments for `Rootbd`, `Filter`, and `Kernel` for `Expr` are passed to `ExprRep`, `ExprT`, and `ExprRepT`. The benefit of this new design is that now we can replace `Filter`, `Rootbd` or `Kernel` at the highest level without any changes in `ExprRep`, `ExprT` or `ExprRepT`. We see here that the default `Expr` class in `Core 2` uses the k-ary BFMSS root bounds [5, 26] and the new `BigFloat2` kernel (below). But users are free to plug in other modules. E.g., one could substitute a better filter and root bound for division-free expressions. This design of `Expr` follows the "delegation pattern" in Object-Oriented Programming [32]: the behavior of `Expr` is delegated to other objects (filters, etc).

§6. `ExprRepT` class hierarchy. The class `ExprRepT` defines abstract structures and operations which are overridden by its subclasses. This hierarchy of subclasses is shown in Figure 2.

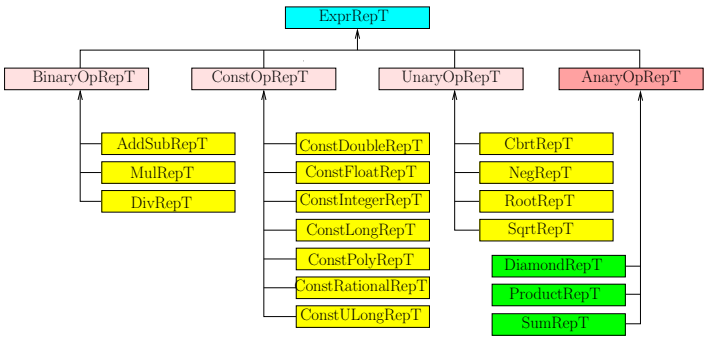
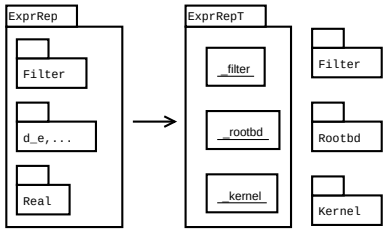


Fig. 2. `ExprRepT` Class hierarchy

There are four directly derived classes, corresponding to the arities of the operator at the root of the expressions: constants, unary, binary, and anary operators. Each of them has further derived classes – for instance, the binary operator class is further derived into three subclasses corresponding to the four arithmetic operators (`AddSubRepT`, `MulRepT`, `DivRepT`). The introduction of anary operators is new. An anary operator is one without a fixed arity, such as

summation $\sum_{i=1}^{n} t_i$. Another is the diamond operator $\diamond(a_0, \ldots, a_n, i)$ to extract the ith real root of a polynomial $p(x) = \sum_{i=0}^{n} a_i x^i$ [5] where a_i's are expressions.

§7. Memory layout of `ExprRepT`. Size of expression nodes can become an issue (see Section 5). Our design of `ExprRepT` optimizes the use of space (see Figure 3(a)). Each `ExprRepT` node has three fields `_filter, _rootbd` and `_kernel`. Here, `_filter` is stored directly in the node, while `_rootbd` and `_kernel` are allocated on demand, and only pointers to them are stored in the node. This is because the filter computation will always be done, but root bound and high precision approximations (from `_kernel`) may not be needed. No memory will be allocated when they are not needed. The memory layout of `ExprRepT` is shown in Figure 3(b). E.g., a binary `ExprRepT` node uses a total of 48 bytes on a 32-bit architecture. The field `_cache` is added to cache important small but potentially expensive information such as `sign, uMSB, lMSB`. The field `_numType` is used for node classification.



(a)

Field name	Size in bytes
Dynamic type information	4
Reference counter	4
Operands : Node * [arity]	4×arity
Filter _filter	16
Kernel *_kernel	4
Rootbd *_rootbd	4
Cache *_cache	4
int _numType	4

(b)

Fig. 3. (a) Comparing `ExprRep` and `ExprRepT`. (b) Layout in 32-bit architecture.

§8. Some Timings. We provide two performance indicators after the above redesign. In Figure 4(a), we measure the time to decide the sign of determinants, with and without (w/o) the filter facility, in `Core 1` and `Core 2`. The format '$N \times d \times b$' in the first column indicates the number N of matrices, the dimension d of each matrix and the bit length b of each matrix entry (entries are rationals). Interestingly, for small determinants, the filtered version of `Core 1` is almost twice as fast. All times are in microseconds.

MATRIX	Core 1.7 Time with (w/o) filter	Core 2.0 Time with (w/o) filter	Speedup
1000x3x10	9 (621)	19 (232)	0.5 (2.7)
1000x4x10	26 (1666)	43 (530)	0.6 (3.1)
500x5x10	449 (1728)	204 (488)	2.2 (3.5)
500x6x10	1889 (3493)	597 (894)	3.2 (3.9)
500x7x10	4443 (6597)	1426 (1580)	3.1 (4.2)
500x8x10	8100 (11367)	2658 (2820)	3.0 (4.0)

(a)

bit length L	`Core 1`	`Core 2`	Speedup
1000	0.82	0.59	1.4
2000	6.94	1.67	4.2
8000	91.9	11.63	7.9
10000	91.91	30.75	3.0

(b)

Fig. 4. (a) Timing filter facility (b) Timing root Bound facility

In Figure 4(b), we test the new root bound facility by performing the comparison $\sqrt{x} + \sqrt{y} \;:\; \sqrt{x + y + 2\sqrt{xy}}$ where x, y are b-bit rational numbers. As this expression is identically zero, filters do not help and root bounds will always be reached.

3.3 Improved Evaluation Algorithm

Since the evaluation algorithm is the centerpiece of an ENC library, it is crucial to tune its performance.

§9. Algorithms for `sign()`, `uMSB()` and `lMSB()`. `Core 1` has two main evaluation subroutines, `computeApprox()` and `computeExactSign()` (see [15]). The former computes an approximation of the current node to some given (composite) precision bound. The latter computes the sign, upper and lower bounds on the magnitude of the current node. These three values are maintained in `Expr` as `sign()`, `uMSB()` and `lMSB()`. In `Core 1`, `computeExactSign()` computes them simultaneously using the rules in Table 5.

E	Case	$E.sgn()$	E^+	E^-
Constant x		$sign(x)$	$\lceil \log_2 x \rceil$	$\lfloor \log_2 x \rfloor$
$E_1 \pm E_2$	if $E_1.sgn() = 0$	$\pm E_2.sgn()$	E_2^+	E_2^-
	if $E_2.sgn() = 0$	$E_1.sgn()$	E_1^+	E_1^-
	if $E_1.sgn() = \pm E_2.sgn()$	$E_1.sgn()$	$\max\{E_1^+, E_2^+\} + 1$	$\max\{E_1^-, E_2^-\}$
	if $E_1.sgn() \neq \pm E_2.sgn()$ and $E_1^- > E_2^+$	$E_1.sgn()$	$\max\{E_1^+, E_2^+\}$	$E_1^- - 1$
	if $E_1.sgn() \neq \pm E_2.sgn()$ and $E_1^+ < E_2^-$	$\pm E_2.sgn()$	$\max\{E_1^+, E_2^+\}$	$E_2^- - 1$
	otherwise	unknown	$\max\{E_1^+, E_2^+\}$	unknown
$E_1 \times E_2$		$E_1.sgn() * E_2.sgn()$	$E_1^+ + E_2^+$	$E_1^- + E_2^-$
$E_1 \div E_2$		$E_1.sgn() * E_2.sgn()$	$E_1^+ - E_2^-$	$E_1^- - E_2^+$
$\sqrt[k]{E_1}$		$E_1.sgn()$	$\lceil E_1^+/k \rceil$	$\lfloor E_1^-/k \rfloor$

Fig. 5. Recursive rules for computing `sign`, `uMSB`, `lMSB`.

There are two "unknown" entries in Table 5. In these cases, `computeApprox()` will loop until the sign is determined, or up to the root bound. To compute such information, we recursively compute sign and other information over the children of this node, whether needed or not. This can be unnecessarily expensive. In `Core 2`, we split the 2 routines into five co-recursive routines in `ExprRepT`: `get_sign()`, `get_uMSB()`, `get_lMSB()`, `refine()` and `get_rootBd()`. Depending on the operator at a node, these co-routines can better decide which information from a child is really necessary. The structure of these algorithms are quite similar, so we use `get_sign()` as an example:

Sign Evaluation Algorithm, `get_sign()`:
1. Ask the filter if it knows the sign;
2. Else if the cache exists, ask if sign is cached;
   Note: the cache may contain non-sign information
3. Else if the approximation (`_kernel`) exists, ask if it can give the sign;
4. Else if the virtual function `compute_sign()` returns `true`, return `sgn()` (sign is now in the cache);
5. Else call `refine()` (presented next) to get the sign.

Thus it is seen that, for efficiency, we use five levels of computation to determine sign: filter, cache, kernel, recursive rules (called `compute_sign()`), and

`refine()`. Note that we do not put the cache at the first level. We do not even cache the `sign`, `uMSB` and `lMSB` information when the filter succeeds because a `Cache` structure is large and we try to avoid costly memory allocation.

The object oriented paradigm used by the above design is called the "template method pattern" [12, p. 325]: define the skeleton of an algorithm in terms of abstract operations which is to be overridden by subclasses to provide concrete behavior. In the derived classes of `ExprRepT`, it is sufficient to just override the virtual function `compute_sign()` when appropriate. For example, `MulRepT` may override the default `compute_sign()` function as follows:

```
1 virtual bool compute_sign() {
2   sign() = first->get_sign() * second->get_sign();
3   return true; }
```

§10. Algorithm for `refine()`. As seen in the `get_sign()` algorithm, if the first four levels of computation fail, the ultimate fall-back for obtaining sign (and also for lower bounding magnitude) is the `refine()` algorithm. We outline this key algorithm to obtain sign via refinement:
1. If the node is transcendental, get the global escape bound. Otherwise, compute the constructive root bound.
2. Take the minimum of the bound from step 1 and the global cutoff bound.
3. Compute an initial precision. If an approximation exists, use its precision as the initial precision. Otherwise use 52 bits instead which is the relative precision that a floating-point filter can provide.
4. Initialize the current precision to the initial precision. Then enter a for-loop that doubles the current precision each time, until the current precision exceeds twice the bound computed in step 2.
5. In each iteration, call `a_approx()` (see below) to approximate the current node to an absolute error less than the current precision. If this approximation suffices to give a sign, return the sign immediately (skip the next step).
6. Upon loop termination, set the current node to be zero.
7. Check if the termination was caused by reaching the escape bound or cutoff bound. If so, append **zero assertion** to a diagnostic file in the current directory. This assertion says that "the current node is zero".

§11. Conditional Correctness. The cutoff bound in the above `refine()` algorithm is a global variable that is set to `CORE_INFTY` by default. While escape bound affects only transcendental nodes, the cutoff bound sets an upper bound on the precision in `refine()` for all nodes. Thus it may override computed zero bounds and escape bounds. During program development, users may find it useful to set a small cutoff bound using `set_cut_off_bound()`. Thus, *our computation is correct, conditioned on the truth of all the zero assertions in the diagnostic file.*

§12. Computing Degree Bounds. In the `refine()` algorithm above, the first step is to compute a constructive root bound. Most constructive root bounds need an upper bound on the degree of an algebraic expression [15]. For

radical expressions, a simple upper bound is obtained as the product of all the degrees of the radical nodes (a radical node $\sqrt[k]{E}$ has degree k). A simple recursive rule can obtain the degree bound of E from the degree bounds of its children (e.g., [14, Table 2.1]). But this bound may not be tight when the children share nodes. The only sure method is to traverse the entire DAG to compute this bound. To support this traversal, in `Core 1` we store an extra flag `visited` with each `ExprRep`. Two recursive traversals of the DAG are needed to set and to clear these flags, while computing the degree bound. To improve efficiency, we now use the `std::map` data structure in `STL` to compute the degree bound: we first create a map M and initialize the degree bound D to 1. We now traverse the DAG, and for each radical node u, if its address does not appear in M, we multiply its degree to the cumulative degree bound D and save its address in M. At the end we just discard the map M. This approach requires only one traversal of the DAG.

3.4 Improved Propagation of Precision

An essential feature of precision-driven evaluation is the need to propagate precision bounds [15]. Precision propagation can be illustrated as follows: if we want to evaluate an expression $z = x+y$ to p-bits of absolute precision, then we might first evaluate x and y to $(p+1)$-bits of absolute precision. Thus, we "propagate" the precision p at z to precision $p+1$ at the children of z. This propagation is correct provided x and y have the same sign (otherwise, $p+1$ bits might not suffice because of cancellation). In general, we must propagate precision from the root to the leaves of an expression. In `Core 1`, we use a pair $[a, r]$ of real numbers that we call "composite precision" bounds. If $x, \widetilde{x} \in \mathbb{R}$, then we say $\widetilde{x}$ is an $[a, r]$**-approximation** of x (written, "$\widetilde{x} \approx x[a, r]$") if $|\widetilde{x} - x| \leq 2^{-a}$ or $|\widetilde{x} - x| \leq |x|2^{-r}$. If we set $a = \infty$ (resp., $r = \infty$), then $\widetilde{x}$ becomes a standard **relative** r**-bit** (resp., an **absolute** a**-bit**) **approximation** of x. It is known that a relative 1-bit approximation would determine the sign of x; so relative approximation is generally infeasible without zero bounds. The propagation of composite bounds is tricky, and various small constants crop in the code, making the logic hard to understand and maintain (see [25]). Our redesign offers a simpler and more intuitive solution in which we propagate either absolute or relative precision, not their combination.

§13. Algorithms for `r_approx()` and `a_approx()`. `Core 1` has one subroutine `computeApprox()` to compute approximations; we split it into two subroutines `a_approx()` and `r_approx()`, for absolute and relative approximations (respectively). Above, we saw that `refine` calls `a_approx()`. There are two improvements over `Core 1`: first, propagating either absolute or relative precision is simpler and can avoid unnecessary precision conversions. Second, the new algorithms do not always compute the sign (which can be very expensive) before approximation.

§14. Overcoming inefficiencies of Computational Rings. Another issue relates to the role **computational rings** in ENC (see §16 in [35]). This

is a countable set $\mathbb{F} \subseteq \mathbb{R}$ that can effectively substitute for the uncountable set of real numbers. To achieve exact computation, $\mathbb{F}$ needs a minimal amount of algebraic structures [35]. We axiomatize $\mathbb{F}$ to be a subring of $\mathbb{R}$ that is dense in $\mathbb{R}$, with $\mathbb{Z}$ as a subring. Furthermore, the ring operations together with division by 2, and comparisons are effective over $\mathbb{F}$. BigFloats with exact ring operations is a model of $\mathbb{F}$, but IEEE bigFloats is not. For computations that do not need exactness, the use of such rings may incur performance penalty. To demonstrate this, suppose we want to compute $\sqrt{2} \cdot \sqrt{3}$ to relative p-bits of precision. We describe two methods for this. In Method 1, we approximate $\sqrt{2}$ and $\sqrt{3}$ to relative $(p+2)$-bits, then perform the exact multiplication of these values. In Method 2, we proceed as in Method 1 except that the final multiplication is performed to relative $(p+1)$-bits. The timings (in microseconds) are shown in Figure 6. We use loops to repeat the experiment since the time for single runs is short. It is seen that Method 2 can be much more efficient; this lesson is incorporated into our refinement algorithm.

Precision	Loops	Method 1	Method 2	Speedup
10	1000000	345	191	45%
100	100000	60	46	23%
1000	10000	72	71	1%
10000	1000	267	219	18%
100000	100	859	760	12%

Fig. 6. Timing for computing $\sqrt{2} \cdot \sqrt{3}$ w/ and w/o exact multiplication.

4 Redesign of the `BigFloat` system

The `BigFloat` system is the "engine" for `Expr`, and `Core 1` implements our own `BigFloat`. In Section 2, we discussed several good reasons to leverage our system on `MPFR`, an efficient library under active development for bigFloat numbers with directed rounding. Our original `BigFloat` plays two roles: to implement a computational ring [35] (see section section 3.4), and to provide arbitrary precision interval arithmetic [24]. Computational ring properties are needed in exact geometry: e.g., to compute implicit curve intersections reliably, we can evaluate polynomials with exact `BigFloat` coefficients, at exact `BigFloat` values, using exact ring operations. Interval arithmetic is necessary to provide certified approximations. For efficiency, `Core 2` splits the original BigFloat class into two new classes: (1) A computational ring class, still called `BigFloat`. (2) An interval arithmetic class called `BigFloat2`, with each interval represented by two `MPFR` bigFloats. This explains[3] the "2" in its name.

4.1 The `BigFloat` Class as Base Real Ring

The new class `BigFloat` is based on the type `mpfr_t` provided by `MPFR`. `MPFR` follows the IEEE standard for (arbitrary precision) arithmetic. The results of

[3] Happily, it also coincides with the "2" in the new version number of `Core Library`.

arithmetic operations are rounded according to user-specified output precision and rounding mode. If the result can be exactly represented, then `MPFR` always outputs this result. E.g., a call of `mpfr_mul(c, a, b, GMP_RNDN)` will compute the product of a and b, rounding to nearest `BigFloat`, and put the result into c. The user must explicitly set the precision (number of bits in the mantissa) of c before calling `mpfr_mul()`. To implement the computational ring `BigFloat`, we just need to automatically estimate this precision. E.g., we can use the following:

Lemma 1. *Let $f_i = (-1)^{s_i} \cdot m_i \cdot 2^{e_i}$ (for $i = 1, 2$) be two bigFloats in `MPFR`, where $1/2 \leq m_i < 1$ and the precision of m_i is p_i. To guarantee that all bits in the mantissa of the sum $f = f_1 \pm f_2$ is correct, it suffices to set the precision of f to*

$$\begin{cases} 1 + \max\{p_1 + \delta, p_2\} & \text{if } \delta \geq 0 \\ 1 + \max\{p_1, p_2 - \delta\} & \text{if } \delta < 0 \end{cases}$$

where $\delta = (e_1 - p_1) - (e_2 - p_2)$. Similarly, for multiplication, it suffices to set the precision of f to be $p_1 + p_2$ in computing $f = f_1 \cdot f_2$.

See [8] for a proof. While this lemma is convenient to use, it may over-estimate the needed precision. In binary notation, think of the true precision of c as the minimum number of bits to store the mantissa of c. Trailing zeros in the mantissa contributes to over-estimation. To avoid this, we provide a function named `mpfr_remove_trailing_zeros()` whose role is to remove the trailing zeros. In an efficiency tradeoff, it only removes zeros by chunks (chunks are determined by `MPFR`'s representation). To understand the effect of overestimation, we conduct an experiment in which we compute the factorial $F = \prod_{i=1}^{n} i$ using two methods: In Method 1, we initialize $F = 1$ and build up the product in a for-loop with $i = 2, 3, \ldots, n$. In the i-th loop, we increase the precision of F using Lemma 1, then call `MPFR` to multiply F and i, storing the result back into F. In Method 2, we do the same for-loop except that we call `mpfr_remove_trailing_zeros()` on F after each multiplication in the loop. Instead of F, we can repeat the experiment with the arithmetic sum $S = \sum_{i=1}^{n} i$. The speedup for the second method over the first method is shown in Figure 7 (time in microseconds, precision in bits).

n	$F = \prod_{i=1}^{n} i$		$S = \sum_{i=1}^{n} i$	
	trailing zeros (prec/msec)	no zeros (prec/msec)	trailing zeros (prec/msec)	no zeros (prec/msec)
10^2	575/0	436/0	102/0	31/0
10^3	8979/0	7539/0	1002/0	31/0
10^4	123619/62	108471/47	10002/15	31/16
10^5	1568931/9219	1416270/8891	100002/437	31/110
10^6	timeout	timeout	1000002/57313	63/1078

Fig. 7. Timing for computing F and S w/ and w/o removing trailing zeros.

§15. Benchmarks of the redesigned `BigFloat`. By adopting `MPFR`, our `BigFloat` class gains many new functions such as `cbrt()` (cube root) and the elementary functions (`sin()`, `log()`, etc). The performance of the `BigFloat` is also greatly improved. We compared the performance of `Core 1` and `Core 2` on `sqrt()` using the following experiment: compute $\sqrt{i}$ for $i = 2, \ldots, 100$ with precision p. The timing in Figure 8 show that `Core 2` is about 25 times faster, thanks purely to `MPFR`.

Precision	`Core 1`	`Core 2`	Speedup
1000	25	1	25
10000	716	32	22
100000	33270	1299	25

Fig. 8. Timing comparisons for `sqrt()`.

4.2 The Class `BigFloat2`

`BigFloat2` is the second class split off from the original `BigFloat`. An instance of `BigFloat2` is just an interval represented by a pair of bigFloat numbers. This representation can be less efficient than the original `BigFloat` by a factor of 2 (in the worst case), both in speed and in storage. But this loss in efficiency is compensated by ease of implementation, and in sharper error bounds, which may be beneficial in non-asymptotic situations. In the future, we may also experiment with a bigfloat class using a midpoint-radius representation using `MPFR`. This class is also useful for ENC applications (e.g., in meshing algorithms of the kind producing Figure 1).

5 Extending the `Expr` Class

We provide facilities for adding new operators to `Expr`. `Core 2` uses such facilities to implement the standard elementary functions. Future plans include extending elementary functions to all hypergeometric functions, following the analysis in [8, 9]. We give two examples of how users can use these facilities for their own needs. We refer to Zilin Du's thesis [8] for more details about these facilities.

5.1 Summation Operation for `Expr`

When an `Expr` is very large, we not only lose efficiency (just to traverse the DAG) but we often run out of memory. Consider the following code to compute the harmonic series $H = \sum_{i=1}^{n} \frac{1}{i}$:

```
Expr harmonic(int n) {                                  1
  Expr H(0);                                            2
  for (int i=1; i<=n; ++i)                              3
    H = H + Expr(1)/Expr(i);                            4
  return H; }                                           5
```

This function builds a deep unbalanced DAG for large n. This can easily cause segmentation faults through stack overflow (column 2 in Figure 9). In ¶6, we said that `Core 2` supports a new class of **anary** (i.e., "without arity") nodes. In particular, we implemented the `summation` and `product` operators. Using `summation`, we can rewrite the harmonic function:

```
Expr term(int i) {
  return Expr(1)/Expr(i); }
Expr harmonic(int n) {
  return summation(term, 1, n); }
```

The improvements from this new implementation is shown in Figure 9. We can now compute the harmonic series for a much larger n, and achieve a speedup as well.

n	Time w/o summation	Time w/ summation	Speedup
1000	24	7	3.4
10000	3931	67	58.6
100000	(segmentation fault)	752	N/A
1000000	(segmentation fault)	12260	N/A

Fig. 9. Timings for computing harmonic series $\sum_{i=1}^{n} \frac{1}{i}$ (in microseconds).

Similarly, the use of `product` operator leads to speed-ups.

5.2 Transcendental Constants π, e and all that

We present our *first* transcendental node π, which is a leaf node derived from `ConstRepT`:

```
template <typename T>
class PiRepT: public ConstRepT<T> {
public:
  PiRepT() : ConstRepT<T>() {
    compute_filter();
    compute_numtype();
  }
  // functions to compute filter and number type
  void compute_filter() const {
    filter().set(
      3.14159265358979323846264338327950288F);
      /* value is not exact*/
  }
  void compute_numtype() const
  { _numType = NODE_NT_TRANSCENDENTAL; }

  // virtual functions for sign, uMSB, lMSB
  virtual bool compute_sign() const
  { sign() = 1; return true; }
  virtual bool compute_uMSB() const
  { uMSB() = 2; return true; }
  virtual bool compute_lMSB() const
  { lMSB() = 1; return true; }

  // virtual functions for r_approx, a_approx
  virtual void compute_r_approx(prec_t prec) const
  { kernel().pi(prec); }
  virtual bool compute_a_approx(prec_t prec) const
```

```
  { kernel().pi(abs2rel(prec)); }
};
```

Now the new π expression is given by:

```
template <typename T>
ExprT<T> pi()
{ return new PiRepT<T>(); }
```

Note how easy it is to do this extension — it could equally be used to introduce e or $\ln 2$ or any constant, provided the kernel class knows how to approximate it. Such constants can now freely appear in an expression, and our precision-propagation mechanism can automatically approximate the expression to any desired absolute error bound. Figure 10 gives timings for π and other elementary functions after incorporation into `Expr`.

Precision (bits)	Expr.	`Core 2`	`Core 1`	Speedup
10,000	π	20	—	—
	$\sqrt{\pi}$	90	—	—
	e^2	80	—	—
	sin(0.7)	50	1240	25
	cos(0.7)	50	1230	25
	tan(0.7)	110	2490	23
100,000	π	710	—	—
	$\sqrt{\pi}$	2200	—	—
	e^2	830	—	—
	sin(0.7)	4780	—	—
	cos(0.7)	4650	—	—
	tan(0.7)	9450	—	—

Fig. 10. Transcendental Constants and Functions

6 Conclusion

The goal of `Core Library` is to approximate the ideal real RAM. To support rapid prototyping of algorithmic ideas in geometry and algebra, ease of use and functionality is prized above sheer efficiency. With `Core 2`, we combine exact algebraic computation with transcendental functions. Our redesigned package is more modular, extensible, and flexible. We gained efficiency from the design and better evaluation algorithms. We adopt the highly efficient `MPFR` library to improve maintainability and to gain transcendental functions. Despite this overhaul, the original simple Core Numerical API is preserved.

In the transcendental aspects, we are just leveraging the speed of `MPFR` into our environment. What we give back is a new convenient way to access `MPFR`. In 2005, `MPFR` won the **Many Digits Competition** [17] in a field of 9 teams that included `Maple` and `Mathematica`. Their solutions are "hand-coded" in the sense that each algorithm is preceded by an error analysis to determine the needed precision, plus a hand-coded implementation of these error bounds. By incorporating `MPFR` into `Core 2`, we can now do these competition problems *automatically*: `Core 2` provides the automatic error analysis. Another "added

value" that `Core Library` provides `MPFR` is access to a computational ring to support exact geometric computation. A future research is to understand and reduce any performance penalties of this automation.

A major open problem is to better understand the expression evaluation algorithms. There is no satisfactory theoretical basis for the optimal evaluation of such expressions. The second major problem is to provide constructive bounds for non-algebraic constants. Instead of escape bounds, we could use Richardson's algorithm to decide zero [29,30]. This is because all the constants in `Core 2` are elementary constants in the sense of Richardson. This is a win-win situation because if our computation is ever wrong, we would have found a counter-example to Schanuel's conjecture. Unfortunately, current versions of Richardson's algorithm do not appear practical enough for general application.

`Core Library` represents a new breed of real number libraries to support exact numerical computation (ENC). It is made feasible through sophisticated built-in functionalities such as filters and constructive zero bounds. There remain ample opportunities for exploring the design space of constructing such libraries. Our `Core Library` offers one data point. Such libraries have many potential applications. Besides robust geometric algorithms, we can use them in geometric theorem proving, certifying numerical programs, and in mathematical explorations.

References

1. E. Berberich, M. Kerber, and M. Sagraloff. Exact geometric-topological analysis of algebraic surfaces. In *24thACM Symp. on Comp. Geometry*, pages 164–173, 2008.
2. V. Brattka and P. Hertling. Feasible real random access machines. *J. of Complexity*, 14(4):490–526, 1998.
3. R. P. Brent. Fast multiple-precision evaluation of elementary functions. *J. of the ACM*, 23:242–251, 1976.
4. H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. *Discrete Applied Mathematics*, 109(1-2):25–47, 2001.
5. C. Burnikel, S. Funke, K. Mehlhorn, S. Schirra, and S. Schmitt. A separation bound for real algebraic expressions. *Algorithmica*, 55(1):14–28, 2009.
6. H. Cohen. *A Course in Computational Algebraic Number Theory*. Springer, 1993.
7. Core Library homepage, since 1999. Software download, source, documentation and links: `http://cs.nyu.edu/exact/core/`.
8. Z. Du. *Guaranteed Precision for Transcendental and Algebraic Computation made Easy*. Ph.D. thesis, New York University, Department of Computer Science, Courant Institute, May 2006. Download from `http://cs.nyu.edu/exact/doc/`.
9. Z. Du, M. Eleftheriou, J. Moreira, and C. Yap. Hypergeometric functions in exact geometric computation. In V.Brattka, M.Schoeder, and K.Weihrauch, editors, *Proc. 5th Workshop on Computability and Complexity in Analysis*, pages 55–66, 2002. Malaga, Spain, July 12-13, 2002. In Electronic Notes in Theoretical Computer Science, 66:1 (2002), `http://www.elsevier.nl/locate/entcs/volume66.html`.
10. A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schoenherr. The CGAL kernel: a basis for geometric computation. In M. C. Lin and D. Manocha, editors,

Applied Computational Geometry: Towards Geometric Engineering, pages 191–202, Berlin, 1996. Springer. Lecture Notes in Computer Science No. 1148.
11. L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. Mpfr: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2):13, 2007.
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.
13. V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A Core library for robust numerical and geometric computation. In *15th ACM Symp. Computational Geometry*, pages 351–359, 1999.
14. 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/`.
15. C. Li, S. Pion, and C. Yap. Recent progress in Exact Geometric Computation. *J. of Logic and Algebraic Programming*, 64(1):85–111, 2004. Special issue on "Practical Development of Exact Real Number Computation".
16. L. Lin and C. Yap. Adaptive isotopic approximation of nonsingular curves: the parametrizability and non-local isotopy approach. In *Proc. 25th ACM Symp. on Comp. Geometry*, pages 351–360, June 2009. Aarhus, Denmark, Jun 8-10, 2009. Accepted for Special Issue of SoCG 2009 in DCG.
17. Many digits friendly competition, Oct 3-4 2005. The details of the competition, including final results, are available from `http://www.cs.ru.nl/~milad/manydigits/`.
18. K. Mehlhorn and S. Näher. LEDA: a platform for combinatorial and geometric computing. *Comm. of the ACM*, 38:96–102, 1995.
19. K. Mehlhorn and S. Schirra. Exact computation with `leda_real` – theory and geometric applications. In G. Alefeld, J. Rohn, S. Rump, and T. Yamamoto, editors, *Symbolic Algebraic Methods and Verification Methods*, pages 163–172, Vienna, 2001. Springer-Verlag.
20. M. Mignotte. Identification of algebraic numbers. *J. of Algorithms*, 3:197–204, 1982.
21. J.-M. Muller. *Elementary Functions: Algorithms and Implementation*. Birkhäuser, Boston, 1997.
22. N. T. Müller. The iRRAM: Exact arithmetic in C++. In J. Blank, V. Brattka, and P. Hertling, editors, *Computability and Complexity in Analysis*, pages 222–252. Springer, 2000. 4th Int'l Workshop, CCA 2000, Swansea, UK. LNCS No. 2064.
23. N. T. Müller, M. Escardo, and P. Zimmermann. Guest editor's introduction: Practical development of exact real number computation. *J. of Logic and Algebraic Programming*, 64(1), 2004. Special Issue.
24. A. Neumaier. *Interval Methods for Systems of Equations*. Cambridge University Press, Cambridge, 1990.
25. K. Ouchi. Real/Expr: Implementation of an Exact Computation Package. Master's thesis, New York University, Department of Computer Science, Courant Institute, Jan. 1997. From `http://cs.nyu.edu/exact/doc/`.
26. S. Pion and C. Yap. Constructive root bound method for k-ary rational input numbers. *Theor. Computer Science*, 369(1-3):361–376, 2006.
27. S. Plantinga and G. Vegter. Isotopic approximation of implicit curves and surfaces. In *Proc. Eurographics Symposium on Geometry Processing*, pages 245–254, New York, 2004. ACM Press.

28. F. P. Preparata and M. I. Shamos. *Computational Geometry*. Springer-Verlag, 1985.
29. D. Richardson. How to recognize zero. *J. of Symbolic Computation*, 24:627–645, 1997.
30. D. Richardson. Zero tests for constants in simple scientific computation. *Mathematics in Computer Science*, 1(1):21–38, 2007. Inaugural issue on Complexity of Continuous Computation.
31. S. Schirra. Robustness and precision issues in geometric computation. In J. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*. Elsevier Science Publishers, B.V. North-Holland, Amsterdam, 1999.
32. B. Stroustrup. *The Design and Evolution of C++*. Addison Wesley, April 1994.
33. J. van der Hoeven. Computations with effective real numbers. *Theor. Computer Science*, 351(1):52–60, 2006.
34. C. Yap, C. Li, S. Pion, Z. Du, and V. Sharma. Core Library Tutorial: a library for robust geometric computation, 1999–2004. Version 1.1 was released in Jan 1999. Version 1.6 in Jun 2003. Source and documents from `http://cs.nyu.edu/exact/`.
35. C. K. Yap. In praise of numerical computation. In S. Albers, H. Alt, and S. Näher, editors, *Efficient Algorithms*, volume 5760 of *Lecture Notes in Computer Science*, pages 308–407. Springer-Verlag, 2009. Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday.
36. C. K. Yap. Tutorial: Exact numerical computation in algebra and geometry. In *Proc. 34th Int'l Symp. Symbolic and Algebraic Comp. (ISSAC'09)*, pages 387–388, 2009. KIAS, Seoul, Korea, Jul 28-31, 2009.
37. 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.