Transcendental and Algebraic Computation made Easy: Redesign of `Core Library`

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

December 3, 2006

Abstract

Expression packages constitute a key technique for achieving the fundamental goals of Exact Geometric Computation (EGC). In this paper, we address the redesign of the Expression Package in the Core Library, addressing the following issues:

- Transcendental computation: the current library is developed for algebraic computation but there is need for transcendental functions in many applications. The integration of algebraic and transcendental computations has major impact on library design.
- Extensibility and modularity: as our library evolve and take on more functionality, these issues have become more pressing for maintainability and flexibility. In our redesign, two major subsystems (filters, root bounds) are modularized.
- Efficiency: expression evaluation is the central algorithm in the package. The optimal design of this algorithm is open, but there are various inefficiencies which we can address. We separate it into four mutually recursive subroutines (Approx, Sign, uMSB, lMSB) to improve efficiency.
- Bigfloat Kernel: this is the engine for expression evaluation. The dual-role of the current bigfloat class leads to inefficiencies, calling for a complete overhaul. We now split up into two distinct bigfloat classes, building upon the `MPFR` package.

Our redesign preserves the original goals of `Core Library`, namely, to provide a simple and natural interface for EGC computation. We present experimental results and timings for our new system, which is released as `Core Library` 2.0.

1 Introduction

Exact Geometric Computation (EGC) has proven to be a highly effective approach for solving numerical nonrobustness problems in algorithms. By definition, an EGC number library is one that guarantees the sign evaluation of expressions over the supported operations. We assume the supported operations include at least $\pm, \times, \div, \sqrt{\cdot}$. More generally, EGC libraries can compute any expression to user-specified relative accuracy [28]. Two EGC libraries for general algebraic expressions are `Core Library` [15] and `LEDA Real` [3, 20]. Because of such libraries, programmers today can routinely write fully robust geometric algorithms. This was inconceivable just over a decade ago. In fact, after the basic principles of EGC were initially understood, one had to manually construct each robust algorithm on a case-by-case basis (e.g., [6, 10]).

In `Core Library`, guaranteed accuracy is known as Level 3 accuracy. In contrast, the ability to compute *each* operation to any specified accuracy is known as Level 2 accuracy. This is basically a generalization of the IEEE Standard to arbitrary precision. There are many Level 2 libraries available, and in their simplest form, they simply provide big number arithmetic. Level 3 requires additional sophistication. It is a major open problem whether Level 3 accuracy is possible for any general class of transcendental computation [29]. This fact underlies some redesign issues of this paper.

Another context for understanding EGC libraries is the general area of real computation. Practically, the construction of real number libraries pose considerable challenge, and there are various effort currently going on (e.g., [14]). We consider EGC libraries to be a new breed in this diverse landscape of real number libraries. Theoretically, we still do not have a consensus about the foundations of a theory of real computation [1, 26, 16, 30]. Such a foundation would be helpful to guide our practical development.

There are three main motivations for the present work. The first is the desire to incorporate transcendental functions in our expressions. In many computation, we need expressions containing transcendental constants ($\pi, e, \ln 2$, etc), elementary functions ($\sin x, \exp x, \ln x$, etc), or more generally hypergeometric functions [8, 9]. Because we can no longer guarantee sign of such expressions, we must distinguish transcendental nodes from algebraic ones. Essentially, transcendental values must be computed at Level 2 while algebraic ones can be computed at Level 3, but a systematic integration is needed. The second motivation is to make the `Expr` class more flexible, extensible and modular. There is interest in introducing new operators into `Expr`, and we would like to introduce mechanisms to support this. Invisible to users of `Expr`, the expression system must provide two critical functions: **filters** [12, 5, 2] and **root bounds** [4, 23]. These are currently integrated into expressions which makes them hard to maintain and extend. The third motivation is the perpetual quest for improved efficiency. There are two major sources of inefficiency that we address. First, we reexamine the central algorithm of `Expr`, its **expression evaluation algorithm**. This is, in fact, several co-recursive subroutines, but the optimal way to implement them is unclear. The current system has some clear inefficiencies that we want to address. We refer to [19] for a survey of the topics mentioned on here: filters, root bounds, expression evaluation.

A background constraint on our redesign effort is to preserve our basic philosophy to provide a very simple and natural user interface. This is encoded in the Core Numerical Accuracy API [27]. A contrasting philosophy is found in `CGAL` [11], another major EGC library which offers users a choice of various specialized number types ("kernels"). Such an approach is usually more efficient, if harder to use.

OVERVIEW. In section 2, we review the original design of the Core Library (hereafter called `Core Library` 1) and address the issues we faced. In Section 3, we present our new design, which forms the basis of our new library release (hereafter called `Core Library` 2). Various experimental benchmark are reported in Section 4. We conclude and propose some open problems in Section 5. Most of the topics in this paper appear in greater detail in the thesis [7]. The experiments reported here are available with the `Core Library` 2 distribution, which is open source. The experimental times are based on a Power Book...

2 Review of Current `Core Library`

The `Core Library` features an object-oriented design and is implemented in C++. A basic goal in the design of the `Core Library` is to make EGC techniques transparent and easily accessible to non-specialist programmers [27, 18]. Built upon the `Real/Expr` package from Yap, Dubé and Ouchi [31, 22], it facilitates the rapid development of robust geometric applications.

There are three main subsystems in the current `Core Library` 1.7: expressions class (`Expr`), real number

class (`Real`) and big float number class (`BigFloat`). These are number classes, built up in a layered fashion. Beneath these three classes are standard big number classes (`BigInt`, `BigRat`) which are wrappers around corresponding number types from `GNU`'s multiprecision package `gmp`. The `Expr` class provides the basic functionalities of exact geometric computation. In theory, this is the only number interface that users need to use. But experienced users can also access the underlying number classes directly, mainly for efficiency.

In the following, we raise design and efficiency issues in the current design of the `Expr` and `BigFloat` classes. The `Expr` class represents expressions that are constructed from real algebraic number constants by repeated application of four basic arithmetic operations $\{+, -, *, /\}$ and radical ($\sqrt[k]{\cdot}$). Structurally, expressions are directed acyclic graphs (DAGs). The expression DAG and most functionalities of `Expr` are implemented by an internal class called `ExprRep`. So `Expr` is mainly a wrapper for `ExprRep`. The following issues arise:

- There are critical facilities in `Expr` that should be modularized and made extensible. In particular, the filter facility and the root bound facility have grown considerably over the course of library development and is now hard to maintain, debug or to extend.
- The main evaluation algorithm of `Expr` has essentially three co-recursive subroutines. The current design does not separate their roles clearly, and this can lead to costly unnecessary computations. For example, the subroutine for computing the sign of a node is always called even though this may not be necessary.
- We currently support 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`. For instance, it is desirable to add determinants, summation and product, and diamond operators.
- It can be very inefficient to build huge expressions. For example, a naive implementation of the summation $H_n = \sum_{i=1}^{n}(1/i)$ would build an unbalanced tree of depth n. For large n, this can easily cause stack overflow. But for expressions with regular structure such as H_n, we want to automatically generate and evaluate such expressions on the fly.

Next we address the `BigFloat` class. It is used by `Expr` to approximate real numbers, and is critical for the efficiency of the entire `Core Library`. It implements an arbitrary precision floating-point number system on top of `BigInt`. We represent a `BigFloat` by the triple $\langle m, err, e\rangle$ of integers where m is the mantissa, $err \geq 0$ is the error bound and e is the exponent. The triple represents 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 interval representation of `BigFloat` has performance penalty because the current design requires frequent error normalization, practically after each arithmetic operation. There are important applications that do not need to maintain error. For example, in Newton-type iterations which are self-correcting, this is not only wasteful, but actually fails to converge if we do not zero out the error (by calling `makeExact`). Although users can manually call `makeExact`, this process is error-prone.
- Our `BigFloat` assumes that for exact bigfloats, the ring operations $(+, -, \times)$ are computed exactly. There are situations where exact ring operations are not needed, and this can be a source of inefficiency. E.g., in Newton-type iterations.
- The current `BigFloat` class only implements $\{+, -, *, /, \sqrt[2]{\ }\}$. Lacking implementations for the elementary functions such as $\exp, \log, \sin, \cos$ etc, limits the applications of current `Core Library`. Of course, this calls for implementation of these functions, not necessarily redesign.

To illustrate these performance penalty, we compare our current implementation of `sqrt` which uses a standard Newton iteration with `BigFloat`s with `MPFR` [17, 21]. Our implementation is about 25 times slower as shown in Figure 5. In fact, the `MPFR` package satisfies all three criteria above, and so it is natural that we consider adopting it. Nevertheless, our current `BigFloat` has important functionality (error bounds and exact ring operations) that are missing in `MPFR`. Our solution is to build on top of `MPFR`.

3 Redesign of Expr Package

The goal of this redesign is to increase modularity, introduce extensibility of expression nodes, and also improve efficiency. A major design requirement is to ensure that these changes do not change the user-level interface. In particular, the simple and natural original Core Number API must be kept intact. For instance, any current CGAL code using Core Library as its number kernel should be able to run after recompilation for Core Library 2.

3.1 Incorporation of Transcendental Nodes

We now consider expressions (DAG's) with transcendental operators. We classify a node (and the corresponding subexpression) as **transcendental** if any of its descendant nodes has a transcendental operator. For example, a transcendental node may be a leaf node such as π, or a unary node such as $\sin(\cdot)$. In current Core Library, we classify nodes into rational or irrational (this information is used for certain operations). We now refine this classification into integer, dyadic, rational, algebraic, transcendental. Since transcendental numbers do not have constructive root bounds, we need to introduce a user-definable global value called **escape bound**. This bound is used as the root bound at transcendental nodes. Note that there is also another bound called **cutoff bound** which is used for a different purpose, as explained below.

Operator op	A op B
$+/-/\times$	$\max\{A.\texttt{\_numType}, B.\texttt{\_numType}\}$
$\div$	$\max\{\texttt{NODE\_NT\_RATIONAL}, A.\texttt{\_numType}, B.\texttt{\_numType}\}$
$\sqrt[k]{\cdot}$	$\max\{\texttt{NODE\_NT\_ALGEBRAIC}, A.\texttt{\_numType}\}$
Transcendental	$\max\{\texttt{NODE\_NT\_TRANSCENDENTAL}, A.\texttt{\_numType}\}$

Table 1: Rules for computing number type

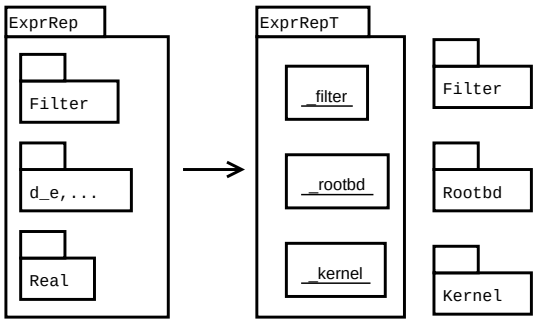
3.2 New Template-based Design of ExprRep

Our redesigned Expr is still a thin wrapper around ExprRep, which is what we focus on.

§1. ExprRep and ExprRepT As noted in the review section, the filter facility and root bound facility are embedded in the old ExprRep class. Now we factor them out from ExprRep into two functional modules: Filter and Rootbd. The Real class, 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 approximations to the exact value. Typically we use bigfloats. Thus, we want to parametrize an expression class with these three modules. So our ExprRep class is redesigned as a template class with new name ExprRepT (see Figure 1(a)). This applies the "delegation pattern" from [25]: delegate certain behaviors of main class to other objects. The benefit of this new design is, now we can replace Filter, Rootbd or Kernel without any changes in ExprRepT. The C++ prototype of ExprRepT is shown as follows:

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

§2. Memory layout of ExprRep Three type parameters are used to define 3 data fields _filter, _rootbd and _kernel inside ExprRepT. However, the nature of their definitions are different: _filter is defined as a variable while _rootbd and _kernel are defined as pointers. We design it in this way because the filter computation will always be done first, but root bound information and high precision computation (based on _kernel) may not be needed. No memory will be allocated for them when they are not needed. The memory layout of ExprRepT is shown in Figure 1(b). For a binary node, it uses a total of 48 bytes on a 32-bit architecture system. Note that for efficiency, a data field named _cache is added to cache some important and costly information such as sign, uMSB, lMSB once they become available. 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)

Figure 1: (a) Comparing `ExprRep` and `ExprRepT`. (b) Memory Layout in 32-bit architecture

§3. `ExprRepT` class hierarchy `ExprRepT` only defines the abstract structures and operations, the actual implementations are delegated to the derived classes. The new design still keep the same class hierarchy of `ExprRepT` and its derived classes as in `Core Library` 1.x except that a "T" is appended at each class names. A full hierarchy diagram is shown in Figure 2.

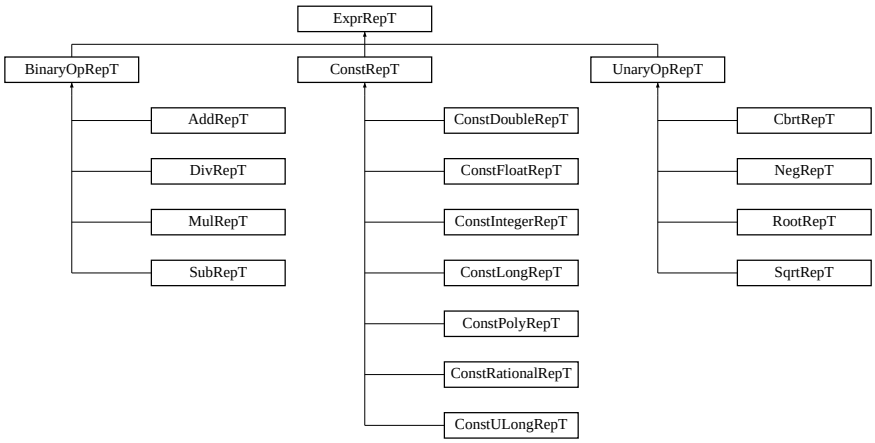


Figure 2: `ExprRepT` Class Hierarchy

§4. `Expr` with plug-gable Filter, Root Bound and Kernel A new template class `ExprT` is defined as:

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

`Expr` is now a typedef:

```
typedef ExprT<BfsFilter, BfmssRootbd, BigFloat2> Expr;
```

Thus, the default `Expr` class uses BFS filter, BFMSS root bound and our `BigFloat2` kernel. However, now the users are free to plug in their own filter, root bound or kernel classes. For example, we can use a better filter and root bound class for division-free expressions.

3.3 Improved Evaluation Algorithm

There are two key evaluation subroutines `computeExactSign()` and `computeApprox()` in `Core Library` 1. `computeExactSign()` is used for computing the sign, upper and lower bound on the most significant bits of the current node (actually it even computes the root bound information). `computeApprox()` is used to propagate the composite precision and compute the approximation. These two subroutines recursively call each other. Now we divided them into five subroutines to be more efficient.

§5. Algorithms for `sgn()`, `uMSB()` and `lMSB()` The sign of an expression node is critical in many places in `Core Library`. For example, division operator must check the sign of the child nodes to avoid "divided by zero" error. The upper and lower bound on the most significant bits of an expression node E, denoted by E^+ and E^- respectively, are also important. They are used to convert the precision from an absolute bound to a relative one or vice-versa. These three parameters are maintained in `ExprRepT`. The subroutine `computeExactSign()` computes them simultaneous using the rules shown in Table 2.

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 $E_1.sgn() = \pm E_2.sgn()$	$E_1.sgn()$	$\max\{E_1^+, E_2^+\} + 1$	$\max\{E_1^-, E_2^-\} + 1$
	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	unknown	unknown
$E_1 \times E_2$		$E_1.sgn() * E_2.sgn()$	$E_1^+ + E_2^+$	$E_1^- + E_2^-$
$E_1 \div E_2$		$E_1.sgn() * E_2.sgn()$	$E_1^+ - E_2^-$	$E_1^- - E_2^+$
$\sqrt[k]{E_1}$		$E_1.sgn()$	$\lceil E_1^+/k \rceil$	$\lfloor E_1^-/k \rfloor$

Table 2: Recursive Rules for computing sign, umsb, lmsb

We can see those rules are recursive. In order to compute those information of the current node, `computeExactSign()` has to compute its children's `computeExactSign()` first, which consequently compute those information for all its below nodes in the DAG even we do not need them all.

In contrast to this, in the new design three separated subroutines: `sgn()`, `uMSB()` and `lMSB()` are introduced and they are only called when it is necessary. The algorithms for them are very similar. Basically, each algorithm consists the following steps (using `sgn()` as an example):

> Sign Evaluation Algorithm
> 1. Ask the filter if it knows the sign;
> 2. Else if the cache exists, ask if it cached the sign;
> 3. Else if the approximation (`_kernel`) exists, ask if it can give the sign;
> 4. Else if the virtual function `compute_sgn()` return `true`, return `sgn()` (the sign in the cache);
> 5. Else call `refine()` (will present next) to get the sign.

For efficiency, we use five levels of computation here: filter, cache, kernel, recursive rules, `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 succeed because a `Cache` structure is large and we try to avoid costly memory allocation.

In above algorithm, we adapt another design pattern which is called "template method pattern" [13]: define the skeleton of an algorithm in terms of abstract operations which subclasses will override to provide concrete behavior. In the derived classes of `ExprRepT`, it is sufficient to just override the virtual function `compute_sgn()` when appropriate. For example, `MulRepT` may override the default `compute_sgn()` function as follows:

```
1  virtual bool compute_sgn() {
2    sgn() = first.sgn() * second.sgn();
3    return true; }
```

§6. Algorithms for `r_approx()` and `a_approx()` For approximation algorithms, we made two improvements: First, unlike the original algorithm, the new approximation algorithm does not always compute the sign before approximating. Second, while the `Core Library` 1 uses one subroutine `computeApprox()` to do approximation which requires a composite precision, the new design use two subroutines, `r_approx()` for relative approximation and `a_approx()` for absolute approximation. This can avoid unnecessary conversion.

§7. Algorithm for `refine()` The override-able `compute_sgn()`, `compute_uMSB()` and `compute_lMSB()` return `false` when they cannot determine the sign, upper and lower bound on most significant bits of the current node. Then a refinement of the current approximation is required. The abstract algorithm for such refinement is as follows:

Refinement Algorithm
1. If an approximation exists, use its precision as the start precision, otherwise use 52 bits instead of since this is the relative precision of the floating-point filter can provide.
2. Check if the Rootbd is non-constructive or the current node is non-algebraic. If so, use the global escape bound. Otherwise, compute the degree bound if the Rootbd is degree based. Then compute the root bound.
3. Compare the bound that we get in step 2 with the global cutoff bound. Take the minimum of them.
4. Run a loop which doubles the precision each time until hits the precision that we obtained in step 3. In the loop, compute a better approximation (absolute) on the current node. Once the refined approximation can give the correct sign, return immediately.
5. Upon the loop termination, if we use either escape bound or cutoff bound, print out a warning message in a diagnostic file, otherwise we set the current node to be zero.

§8. Conditional Guarantees The cutoff bound in the above algorithm is a global variable that is set to `CORE_INFTY` by default. Unlike the escape bound, the cutoff bound simply restricts the precision in root approximation. For instance, to speedup programs during testing, users can set a small cutoff bound using `set_cut_off_bound()`. If either bounds are used to terminate an approximation, the sign computations is not guaranteed any more. However, `Core Library` will issue a "conditional certificate" in a standard Core Diagnostic file, saying that a certain value is assumed to be zero. Our computation is correct, conditioned on the truth of these certificates.

3.4 Improved Propagation of Precision

`Core Library` uses a *precision-driven* algorithm with composite precisions [31]. The precision propagation for various bigfloat operations were worked by Ouchi [22]. Using this analysis to propagate composite precision, various small constants appear in the code which made the current code hard to understand and maintain. Here we offer a simpler and more intuitive solution, where we propagate either absolute or relative precision, but not both. It basically follows the ideas in [28] but with some simplification.

It may be noticed from the above error analysis that exact ring operations are not necessary although this is assumed in [28]. To see the the difference, we did the following experiment:

Experiment 1 Precision Propagation for Multiplication

We use two methods to compute $z = \sqrt{2} \cdot \sqrt{3}$ to relative precision p.
Method 1: Approximate $\sqrt{2}$ and $\sqrt{3}$ to $(p+2)$ relative bits, then perform exact multiplication on multiplying the approximated values.
Method 2: Do the same as the Method 1 except performing multiplication in $(p+1)$ relative precision.

The timing (in microseconds) is shown in the Figure 3. We use loops to repeat the experiment since the time for a single run is short. Method 2 is generally more efficient.

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%

Figure 3: Timing for computing $\sqrt{2} \cdot \sqrt{3}$ w/ and w/o exact multiplication

4 Redesign of `BigFloat` system

In `Core Library`, the `BigFloat` system is the "engine" of `Expr` that performs actual numerical approximations. Currently, we implement our own `BigFloat` using `BigInt` from `GMP`. As we want to bring transcendental functions into `Core Library`, this would require us to provide the corresponding functions in `BigFloat`. Our decision to adopt `MPFR` automatically solves this issue. Next, the interface of `BigFloat` in `Core Library`

has certain requirements that are not directly available in MPFR. Another issue is that our original BigFloat has two roles: to support interval arithmetic, and to implement an exact number ring. Both properties are important in EGC: (a) Interval arithmetic are necessary to provide guaranteed bounds. (b) Exact ring operations are needed in some applications. E.g., to locate implicit curve intersections reliably, we may need to evaluate polynomials with exact BigFloat coefficients, at exact BigFloat values, using exact ring operations.

For efficiency, we create two new BigFloat classes with the respective roles: (1) The exact ring class is (still) called BigFloat, obtained by a C++ wrapper around MPFR. (2) The interval arithmetic class is called BigFloat2, with each interval represented by two MPFR bigFloats. This explains the "2" in its name.

4.1 Design of Class BigFloat

The new class BigFloat is an enhanced version of the type mpfr_t provided by MPFR.

§9. Precision in MPFRversus Relative Precision MPFR follows the IEEE standard for arithmetic. Thus the result of arithmetic operations are rounded according to user-specified output precision and rounding mode. If the result can be exactly represented, then it guarantees to output this result. E.g., a call of `mpfr_mul(c, a, b, GMP_RNDN)` in MPFR will compute the product of a and b with rounding to nearest and put the result into c. The user must explicitly set the number of bits in the mantissa of c before calling `mpfr_mul()`. The following lemma shows how to set this precision:

LEMMA 1 *To guarantee relative p-bit precision in an arithmetic operation in MPFR, it is sufficient that the mantissa of result variable has at least $p+1$ bits (using any of the four rounding modes).*

§10. Precision for exact ring operations The need for users to specify the precision for each variable is inconvenient and error-prone. Exact ring operations is convenient but sometimes also necessary. To implement exact ring operations using MPFR, our BigFloat must automatically estimate the output precision for each of the ring operations:

LEMMA 2 *Let $f_1 = (-1)^{s_1} \cdot m_1 \cdot 2^{e_1}$ and $f_2 = (-1)^{s_2} \cdot m_2 \cdot 2^{e_2}$ be two variables in MPFR. If f_1 has precision p_1, f_2 has precision p_2 and $\delta = (e_1 - p_1) - (e_2 - p_2)$, then in order to guarantee all bits in the mantissa of f correct, it is enough to set precision of f to be*

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

for computing $f = f_1 \pm f_2$. Similarly, it suffices to set the precision of f to be $p_1 + p_2$ for computing $f = f_1 \cdot f_2$.

Note that over-estimation could grow very fast (see column 2 of Figure 4). To avoid this, we provide a function named `mpfr_remove_trailing_zeros()` is remove the unnecessary tailing zeros (for efficiency, it only removes zeros by chunks). It is called after each corresponding arithmetic operations. Experiment 2 shows the timing difference between having this call, and without.

Experiment 2 Optimization of Precision Estimation

We compute $\prod_{n=1}^{n} i$ using two methods:

Method1: initialize $s = 1$, then for $i = 1, 2, \cdots, n$, increase the precision of s by the precision of i, and then call MPFR to multiply s and i and put the result in s.
Method2: same as Method 1, but call `mpfr_remove_trailing_zeros()` for s after multiplication in the loop.

A time comparison between them is given in Figure 4 (timings are measured in microseconds and precisions are in bits). From this experiment, we can see that it may be quite useful to trailing zeros.

§11. Benchmark of new BigFloat By adapting MPFR, our BigFloat gain many new functions such as `cbrt()` and the elementary functions (`sin()`, `cos()`, etc). The performance of the BigFloat has also been greatly improved. We compared the performance of `Core Library` 1.7x and `Core Library` 2 on `sqrt()` using the following experiment:

The timing comparison is shown in Figure 5. We have about 25 times speedup with new implementations.

$n =$	Estimated Precision	Timing	Optimized Precision	New Timing
100	469	1	32	0
1000	7599	4	32	4
10000	108458	90	32	32
100000	1414677	16305	32	308
1000000	—	—	64	3118

Figure 4: Timing for computing $\sum_{i=1}^{n} i$ w/ and w/o removing trailing zeros.

Experiment 3 Comparison of the Performance of `sqrt()`

Computing `sqrt(`i`)` for $i = 2, \ldots, 100$ with precision p using `BigFloat` class in `Core Library` 1.x and `Core Library` 2.0:

4.2 Design of Class `BigFloat2`

`BigFloat2` is the new number type split off from `BigFloat`. It is used by `Expr` to perform the underlying approximation. However, the users can use it directly for efficiency (to avoid building expression DAG). In the new design, we gave up the original use representation of bigfloat intervals as a triple (m, err, e), and to use a pair (ℓ, r) of `MPFR` bigfloats as in standard interval arithmetic. When $r - \ell$ is very small, this representation is wasteful compared to our original design. But we gain in ease of implementation and maintainence, and more precise bounds on error. We can also exploit `MPFR`'s rounding modes in our implementation.

5 Extending `Expr` Class

We provide the facilities for adding new operations to `Expr`. We will add the standard elementary functions $(\sin x, \exp x, \ln x, \text{ etc})$ to `Core Library` using these facilities. But users can also use the same facilities for their own purpose, e.g., to introduce orientation or other predicates. Below we will give two examples: adding a summation node and adding transcendental constants.

5.1 How to Add Your Own Operation for `Expr`

Since `Expr` is a wrapper around `ExprRep`, to add a new operation for `Expr` mostly amounts to designing a new type of `ExprRep` node, i.e., derive a subclass of `ExprRep`. In `Core Library` 1.x, this procedure is complicated and error-prone. With the new design, adding a new type of `Expr` node becomes easier. Basically, it includes the following steps:

1. Derive a class from `ExprRepT` or a subclass of `ExprRepT` as given in Figure 2.
2. Implement a constructor for this new class and call the following two functions, which compute the filter value and set the number type:

   `compute_filter()`, `compute_numtype()`

3. Implement the following functions:

   `compute_filter()`, `compute_numtype()`,

   `compute_sign()`, `compute_uMSB()`, `compute_lMSB()`,

   `compute_r_approx()`, `compute_a_approx()`, `compute_rootbd()`

4. Define a function (can be global function/operator or constructor/member functions/operator in `Expr`) which can construct this new type of node.

We refer to Zilin's Thesis [7] for a worked out example for doing this.

Precision	Core Library 1.7	Core Library 2.0	Speedup
1000	25	1	25
10000	716	32	22
100000	33270	1299	25

Figure 5: Timing Comparisons for `sqrt()`.

5.2 Summation Operation for `Expr`

When the DAG constructed by `Expr` is very large, we not only lose efficiency but will often segment fault. For example, we can compute the harmonic series $\sum_{i=1}^{n} \frac{1}{i}$ using the following function:

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

This function will build a deep unbalanced DAG tree for large number n. As seen in column 2 in Figure 6, this can easily cause segmentation faults through stack overflow in most systems. We propose a new type of node: instead of building up many binary operation (+) nodes, we can design an[1] "anary" node to do the summation in one level. See [7] for details of implementing anary node. With this anary `sum`, we can rewrite the following functions for computing harmonic series:

```
1  Expr term(int i) {
2    return Expr(1)/Expr(i); }
3  Expr harmonic(int n) {
4    return sum(term, 1, n); }
```

The timing for this new implementation is shown in column 3 in Figure 6 (see `t_sum.cpp` under `progs/benchmark/sum`). With this new implementation, we can compute the summation for much larger n, and achieve a speedup of 3-58 times.

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

Figure 6: Timing for computing harmonic series $\sum_{i=1}^{n} \frac{1}{i}$(in microseconds).

5.3 Transcendental Constants $\pi, e, \ln 2$ and all that

We present our *first* transcendental node π, but it could equally be applied to e or $\ln 2$. It is a leaf node, derived it from `ConstRepT`:

```
1  template <typename T>
2  class PiRepT: public ConstRepT<T> {
3  public:
4    PiRepT() : ConstRepT<T>() {
5      compute_filter();
6      compute_numtype();
7    }
8    // functions for computing filter and number type
9    void compute_filter() const {
10     filter().set(3.14, true); //true means the value is inexact
11   }
12   void compute_numtype() const
13   { _numType = NODE_NT_TRANSCENDENTAL; }
14
15   // virtual functions for computing sign, uMSB, lMSB
16   virtual bool compute_sign() const
17   { sign() = 1; return true; }
18   virtual bool compute_uMSB() const
19   { uMSB() = 2; return true; }
20   virtual bool compute_lMSB() const
21   { lMSB() = 1; return true; }
```

[1] *anary* means "without (fixed) arity".

```
22
23    // virtual functions for r_approx, a_approx
24    virtual void compute_r_approx(prec_t prec) const
25    { kernel().pi(prec); }
26    virtual bool compute_a_approx(prec_t prec) const
27    { kernel().pi(abs2rel(prec)); }
28  };
```

Now the new π function is given by:

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

6 Benchmarks

We give timing comparisons for two standard test programs in the `Core Library` distribution.

§12. `compare.cpp`. Figure 7 shows the timing for the comparison $\sqrt{x} + \sqrt{y} \ : \ \sqrt{x + y + 2\sqrt{xy}}$ where x and y are L-bit rational numbers.

bit length L	`Core Library` 1.7	`Core Library` 2.0	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

Figure 7: Timing Comparisons for `compare.cpp`

§13. `testFilter.cpp` Figure 8 compares the timing of computing the determinants of matrices, with filter facility is on and off (see `testFilter.cpp` under `benchmark/testFilter`). The numbers in first column in each table has the following format: $N \times d \times b$ where N is the number of matrices, d is the dimension of each matrix and b is the bit length of each entry in the matrix (entries are rationals).

MATRIX	1.7 w/ (w/o) filter	2.0 w/ (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)

Figure 8: Timing `testFilter.cpp` w/ (w/o) filter (in microseconds)

7 Conclusion

In 2003, we introduced arbitrary real algebraic numbers into `Core Library` 1.6. With `Core Library` 2, we offer transcendental numbers and functions as well. Our completely redesigned `Core Library` is more modular, extensible and flexible. We gained efficiency from the improved design, better algorithms, as well as the restructuring `Core Library` to exploit the highly efficient `Mpfr` library. Despite this upheaval, the simple and natural Core Numerical API is preserved so that most application programs need not change.

The major open problem is better understanding of the expression evaluation algorithms. Besides empirical evidence, there is currently no satisfactory theoretical basis for the evaluation of such algorithms. The standard input size measure is unsuitable, as we want some form of precision-sensitivity [24].

The `Core Library` represents a new breed of real number library that guarantees a priori precision bounds. This is made possible and practical by sophisticated built-in functionalities such as filters and root bounds. The engineering of such systems is only beginning. There is ample opportunities for exploring the design space in the construction of such libraries. Our `Core Library` offers one data point. The `CGAL` design, with its menu of specialized EGC number types, represents another data point. Such libraries have many applications that have barely been explored. Besides robust geometric algorithms, they are useful in theorem proving, certifying programs and in mathematical explorations.

References

[1] L. Blum, F. Cucker, M. Shub, and S. Smale. *Complexity and Real Computation*. Springer-Verlag, New York, 1998.

[2] 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.

[3] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. Exact efficient geometric computation made easy. In *Proc. 15th ACM Symp. Comp. Geom.*, pages 341–450, New York, 1999. ACM Press.

[4] C. Burnikel, S. Funke, K. Mehlhorn, S. Schirra, and S. Schmitt. A separation bound for real algebraic expressions. In *9th ESA*, volume 2161 of *Lecture Notes in Computer Science*, pages 254–265. Springer, 2001. To appear, Algorithmica.

[5] C. Burnikel, S. Funke, and M. Seel. Exact geometric computation using cascading. *Int'l. J. Comput. Geometry and Appl.*, 11(3):245–266, 2001. Special Issue.

[6] C. Burnikel, K. Mehlhorn, and S. Schirra. How to compute the Voronoi diagram of line segments: Theoretical and experimental results. In *Lecture Notes in Computer Science*, volume 855, pages 227–239. Springer, 1994. Proceedings of ESA'94.

[7] 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/`.

[8] 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`.

[9] Z. Du and C. Yap. Absolute approximation of the general hypergeometric functions. In S. il Pae and H. Park, editors, *Proc. 7th Asian Symposium on Computer Mathematics (ASCM 2005)*, pages 246–249, 2005. Korea Institute for Advanced Study, Seoul, Dec 8–10, 2005.

[10] T. Dubé and C. K. Yap. A basis for implementing exact geometric algorithms (extended abstract), September, 1993. Paper from `http://cs.nyu.edu/exact/`.

[11] 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; Proc. 1st ACM Workshop on Applied Computational Geometry (WACG), Federated Computing Research Conference 1996, Philadelphia, USA.

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

[13] 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.

[14] P. Gowland and D. Lester. A survey of exact arithmetic implementations. In J. Blank, V. Brattka, and P. Hertling, editors, *Computability and Complexity in Analysis*, pages 30–47. Springer, 2000. 4th International Workshop, CCA 2000, Swansea, UK, September 17-19, 2000, Selected Papers, Lecture Notes in Computer Science, No. 2064.

[15] 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.

[16] K.-I. Ko. *Complexity Theory of Real Functions*. Progress in Theoretical Computer Science. Birkhäuser, Boston, 1991.

[17] V. Lefèvre. The generic multiple-precision floating-point addition with exact rounding (as in the mpfr library). In *Proceedings of the 6th Conference on Real Numbers and Computers*, pages 135–145, 2004.

[18] 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/`.

[19] 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".

[20] 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.

[21] MPFR Homepage, Since 2000. URL `http://www.mpfr.org/`. The MPFR Library is a C-library for multi-precision floating-point computation with exact rounding modes.

[22] 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/`.

[23] S. Pion and C. Yap. Constructive root bound method for k-ary rational input numbers. *Theor. Computer Science*, 369(1-3):361–376, 2003. Also: Proc.19th SoCG, 2003, 256–263.

[24] J. Sellen, J. Choi, and C. Yap. Precision-sensitive Euclidean shortest path in 3-Space. *SIAM J. Computing*, 29(5):1577–1595, 2000. Also: 11th ACM Symp. on Comp. Geom., (1995)350–359.

[25] B. Stroustup. *The Design and Evolution of C++*. Addison Wesley, April 1994.

[26] K. Weihrauch. *Computable Analysis*. Springer, Berlin, 2000.

[27] C. K. Yap. A new number core for robust numerical and geometric libraries. In *3rd CGC Workshop on Computatioanl Geometry*, 1998. Invited Talk. Brown University, Oct 11–12, 1998.
Abstracts, `http://www.cs.brown.edu/cgc/cgc98/home.html`.

[28] C. K. Yap. On guaranteed accuracy computation. In F. Chen and D. Wang, editors, *Geometric Computation*, chapter 12, pages 322–373. World Scientific Publishing Co., Singapore, 2004.

[29] C. K. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 41, pages 927–952. Chapmen & Hall/CRC, Boca Raton, FL, 2nd edition, 2004.

[30] C. K. Yap. Theory of real computation according to EGC, 2006. To appear in LNCS Volume based on the Dagstuhl Seminar "Reliable Implementation of Real Number Algorithms: Theory and Practice", Jan 8-13, 2006.

[31] C. K. Yap and T. Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, pages 452–492. World Scientific Press, Singapore, 2nd edition, 1995.