CORE Library Tutorial: A Library for Robust Geometric Computation

Chen Li and Chee Yap
Courant Institute of Mathematical Sciences

January 18, 1999†

Abstract

The CORE package contains a set of C++ classes for exact computation with constructible real numbers. It embodies our precision-driven approach to "exact geometric computation". In this tutorial we give an overview for this package, and basic instructions for using it.

Table of Contents

†Revised: Sep 9, 1999.

1 Introduction

The `CORE` library is a collection of classes written in the `C++` language which facilitate numerical computation which has a variety of precision requirements. The library is comprised of two main parts: the `Real` subpackage which provides a uniform interface to a heterogeneous collection of arbitrary precision real number system (usually called Big Number packages), and the `Expr` subpackage which supports a *precision-driven* approach to high precision computation. In particular, our library is designed to support the *Exact Geometric Computation* (EGC) approach to robust geometric computation [12, 13]. The EGC approach is one of the many routes that researchers have taken towards addressing the seemingly intractable problem of robust algorithms in geometric settings. Recent research have shown the effectiveness of EGC in many specific algorithms (such as convex hulls, Delaunay triangularion, etc) [5, 4, 7, 3, 1, 10]). Our goal in this project aims to make such EGC a useful and widely accessible tool. Through the `CORE` library, any (`C/C++`) programmer can create robust geometric algorithms *without* any prior knowledge of EGC techniques.

The `CORE` library defines three accuracy levels to meet users' different needs:

Machine Accuracy (level 1) This may be identified with the IEEE Floating-Point Standard 754.

Arbitrary Accuracy (level 2) Users can specify any desired accuracy in term of the number of bits used in the computation. E.g. "200 bits" means that the numerical operations will not cause an overflow or under flow until 200 bits are exceeded.

Guaranteed Accuracy (level 3) Users can specify the absolute or relative precision that is guaranteed to be correct in the final results. E.g. "200 relative bits" means that the first 200 significant bits of a computed quantity are correct.

The mechanism for delivering these three *accuracy levels* to the users is almost transparent. The users just need to insert a simple preamble as described in Section 2.2 to "simultaneously" access the different levels. Then the user compiles and executes the resulting `C++` program in the conventional way. Consequently, the effort required to write new robust programs or convert existing non-robust programs into robust ones is minimal.

It should be clear that a conventional `C++` program, if it is to have accuracy levels 2 or 3, must have its basic number types re-interpreted. Specifically, in level 3, the primitive type `double` is now re-interpreted to refer to the class `Expr`. The class `Expr` in turn depends on the class `Real`. The user may identify instances of class `Real` with various types of rational numbers (possibly rational intervals). Similarly, instances of class `Expr` can be identified with algebraic expressions built up from instances of `Real` via the operators $+, -, \times, \div$, and $\sqrt{}$. Each instance of `Expr` maintains an *approximate value* as well a *precision.* Users can freely set and modify the precision of any expression and its approximate value will automatically adjust to satisfy this precision in the evaluations thereafter. Users can compare the values of two instances of `Expr` exactly. Mathematically, our system allows one to perform exact computations with any constructible real numbers (subject to physical limitations).

The recent paper of Burnikel et al [2] proposed a library of `leda_real` that is similar in many aspects to our work. Our work is directly derived from the `Real/Expr` Package; however, there are significant changes in philosophy. Specifically, the user interface represents a major improvement for usability. The `CORE` accuracy API was first proposed in Yap [11], and this was realized in Karamcheti, Li, Pechtchanski and Yap [6]. The initial implementation was based on the `Real/Expr` package designed by Dubé and Yap (circa 1993) and rewritten by Ouchi [9]. Further details about the basic underlying algorithms and their error analysis can be found in the Ouchi's thesis [9]. The current version 1.2 incorporates some techniques to provide significant speedup to the system. Moreover, various bugs have been fixed.

Our library has been tested on the Sun SPARC, SGI and Intel/Linux platforms. The complete source codes, together with examples and documentation, is about 2.5 MB (about half of this are documentation). It can be downloaded from our project homepage

`http://cs.nyu.edu/exact/core`

This tutorial describes the `CORE` Package, version 1.2, released in September 1999.

2 Getting Started

2.1 Installing the CORE library

In the following, we assume that the `CORE` distribution files are extracted in a directory called `core`. The `README` file in `core` will describe the simple steps to compile the library. The directory structure is as follows:

`core/src`:	The source codes for the `CORE` library.
`core/inc`:	The header files.
`core/lib`:	The compiled libraries is found here.
`core/ext`:	The extensions for linear algebra and geometry.
`core/progs`:	Some examples using the CORE library.

The source files for the `CORE` library are stored in `core/src`. The Makefile there builds the core library "libcore.a" and places it in the `core/lib`(but this is automatically done by the Makefile at the top level).

2.2 Programming with the CORE library

It is simple to use the core library in your `C/C++` programs. There are two steps you need to follow:

1. Add a compiler directive to set the constant "Level"

```
#define Level <level_number> // this line can be omitted when
                             // using the default value 3.
```

where the value of `<level_number>` could be 3 for guaranteed accuracy, 2 for arbitrary accuracy and 1 for machine accuracy.

2. Include the header file "CORE.h" (in `core/inc` directory)

```
#include "CORE.h"
```

Of course, you need to set the correct INCLUDE path for compilation, which should include the directory `core/inc`.

Thus, the simplest preamble you need to add to your usual `C/C++` program is

```
#define Level <level_number>    // defaults to 3 if omitted
#include "CORE.h"
```

Please see Section A.4 for an example. This preamble should appear *before* your code in which you want to utilize the exact arithmetic, but *after* all the standard header files, e.g. `<iostream.h>`, etc.

At levels 2 and 3, in all the codes after this preamble, the built-in machine types `double` and `long` will be replaced with our number classes. But you can still access the original `double` and `long` types by using `machine_double` and `machine_long` instead.

Some examples are collected in `core/progs`. There are also Makefile templates which could be easily modified to compile your own programs.

2.3 I/O

I/O streams understand `CORE` expressions. Simply writing "`cout << e`" would output an approximate value of e to the stream *cout*. In our implementation, that value is always a `BigFloat`. Please see Section A.1.6 for more details about `BigFloat`'s output.

There is a global variable `defPrtDgt` which specifies at most how many significant *decimal* digits would be printed out. The users can set this variable by calling the function `setDefaultPrintDigits(int)`.

Our system can read arbitary long numbers from streams. Currently numbers read from streams are internally represented as `BigFloat`s. Beware that only binary fractions can be exact else the `BigFloat` is correct save the last bit. E.g., to get an exact rational input, you may need to read in 2 `BigInt`s and then divide them.

An alternative method is to read the number into a variable of machine build-in types e.g. `int` or `float`at first, and then convert them to the `CORE` objects by either object initialization or assignment.

2.4 A Simple Example

We give a simple program to compare the following two expressions, numerically:

$$\sqrt{x} + \sqrt{y} : \sqrt{x + y + 2\sqrt{xy}}.$$

Of course, these expressions are algebraically identical, and hence the comparison should result in equality regardless of the values of x and y.

```
#ifndef Level
# define Level 3
#endif
#include <iostream.h>
#include "CORE.h"
main() {
    double x = Rational(12345, 56789);
    double y = "1234567890.0987654321";      // string input

    double e = sqrt(x) + sqrt(y);
    double f = sqrt(x + y + 2 * sqrt(x*y));

    cout << "e == f ?  " << ((e == f) ?
        "yes (CORRECT!)" :
        "no (INCORRECT!)") << endl;
}
```

3 Composite Precision Bound

Given a real number X, integers a and r, we say that a real number $\widehat{X}$ is an *approximation* of X to *(composite) precision* $[r, a]$, denoted

$$\widehat{X} \simeq X\,[r, a]\,,$$

provided either

$$\left|\widehat{X} - X\right| \le 2^{-r}\,|X|$$
$$\text{OR}$$
$$\left|\widehat{X} - X\right| \le 2^{-a}.$$

Intuitively, r and a bound the number of "bits" of relative and absolute error (respectively) when $\widehat{X}$ is used to approximate X. Note that we use the "or" semantics (either the absolute "or" relative error has the indicated bound). In the above notation, we view the combination "$X[r, a]$" as the given data (although X is really a blackbox, not an explicit number representation) from which our system is able to generate an approximation $\widehat{X}$. For any given data $X[r, a]$, we are either in the "absolute regime" (if $2^{-a} \ge 2^{-r}|X|$) or in the "relative regime" (if $2^{-a} \le 2^{-r}|X|$).

To force a relative precision of r, we can specify $a = \infty$. Thus $X[r, \infty]$ denotes any $\widehat{X}$ of X which satisfies $\left|\widehat{X} - X\right| \le 2^{-r}\,|X|$. Likewise, if $\widehat{X} \simeq X[\infty, a]$ then $\widehat{X}$ is an approximation of X to the absolute precision a, $|\widehat{X} - X| \le 2^{-a}$.

In our implementation, r must be `unsigned long` or `INFTY`, and a is `long` or $\pm$`INFTY`. We also use two global variables to specify those two precisions

$$[\texttt{defRelPrec}, \texttt{defAbsPrec}]$$

whose default values are initialized to $[35, \infty]$. The user can change these values at compile time as well as run time by calling the functions:

```
long setDefaultRelPrecision(long r);    // returns previous value
long setDefaultAbsPrecision(long a);    // returns previous value
void setDefaultPrecision(long r, long a);
```

4 Introduction to the CORE Classes

There are three main classes in the `CORE` package: `Expr`, `Real` and `BigFloat`. Although users do not have to directly access these classes, understanding them is still useful for understanding the behavior of the `CORE`. Advanced users may want to program with these classes directly. Here we briefly introduce them. Users can find more details in Appendix A.

`BigFloat` is an arbitrary precision floating point number representation that we built on top of `GNU`'s `BigInt` and `Rational`.

`Real` is a "heterogeneous" number system that incorporates the following six subclasses: `int`, `long`, `double`, `BigInt`, `Rational`, and `BigFloat`.

`Expr` , the most interesting class in the package, supports level 3 accuracy. This class depends on `Real` and `BigFloat` and represents a subset of the real numbers and the expressions over them.

5 Converting Existing C/C++ Programs

Besides inserting the preamble mentioned before, there are a few more things you may need to pay attention to:

a) Beware that your code appearing after the following preamble

```
#define Level 3
        #include "CORE".h
```

will have the machine built-in types `double` and `long` replaced by the `CORE` class `Expr`. An analogous promotion occurs at level 2. If you do not want such promotions to occur, please use `machine_double` and `machine_long` instead.

b) All the objects implicitly (e.g. automatically promoted from double) or explicitly declared to be of type Expr *must* be initialized appropriately. In most cases, this is not a problem since a default Expr constructor is defined. However if your existing code declares any Expr object, or container object which includes, either implicitly or explicitly, one or more Expr objects, and such object is dynamically allocated using `malloc()`, then it is usually not initialized properly. You probably want to use the "new" operator instead.

```
double *pe, *pf;

// The following is incorrect at Levels 2 and 3:
pf = (double *)malloc(sizeof(double));
cout << *pf << endl;
// prints:  Segmentation fault
// because the object pointed by pf was not initialized properly!

// This is the correct way:
pe = new double();
cout << *pe << endl;
// prints:  zero (the default initial value)
```

c) The system's built-in printf and scanf functions cannot be used to output/input the Expr objects directly. You need to use `C++` stream I/O instead.

d) The variables of machine type `int` or `float` are never promoted to Expr objects. For example,

```
// set Level 3
int i;
double d = sqrt(i);
```

Here the "sqrt" actually refers to the standard C function defined in math.h, and not our precision sensitive sqrt in the Expr class. Hence d holds only a fixed approximation to $\sqrt{i}$, and the exact value can not be recovered. Here is a fix:

```
// set Level 3
int i;
double e = i; // promote to an Expr object
double d = sqrt(e); // the precision sensitive sqrt is called.
```

e) Type promotion does not affect arithmetic of the constant literals appearing in the programs. For example,

```
long la = 1/3; // the value of la would be zero.
// To receive the exact value of 1/3, do this instead:
long lb = Rational(1, 3);
```

f) Note that since all the `double` and `long` variables would be promoted to Expr class during the `C/C++` preprocessing, certain `C/C++` semantics does not work in level 3 anymore. For example,

```
double e;
if (e) { ...}
```

The usual semantics of this code says that if the value of e is not zero, then do .... Since e is now an Expr object, you should write instead:

```
double e;
if (e != 0) { ...}
```

6 Linear Algebra and Geometry Extensions

We plan to provide various basic extensions (`COREX` for short) for our library. In the current distribution, we include a basic linear algebra and geometry extension. The header files for these `COREX`'s are found in the files `linearAlgebra.h`, `geometry2d.h` and `geometry3d.h` under the `core/inc`. To use any of these `COREX`'s, just insert the appropriate include statements: e.g.,

```
#include "linearAlgebra.h"
```

Note that `geometry3d.h` and `geometry2d.h` already include `linearAlgebra.h`. The source codes for the extensions are found under the sub-directory `core/ext` and some examples under `core/progs`.

`linearAlgebra` defines two classes: `Matrix` for general $m \times n$ matrices, and `Vector` for general n-dimension vectors. They support basic matrix and vector operations. Gaussian elimination with pivoting is implemented here. `Geometry3d` defines classes such as 3-dimensional `Point`, `Line` and `Plane` based on the linear algebra API, while `geometry2d` defines the analogous 2-dimensional objects.

The makefile at the top level automatically builds two versions of the `COREX` libraries, named `libcorex_level1.a` and `libcorex_level3.a` respectively. If you use the `COREX` classes in your own program, it is vital to link with the correct library depending on the accuracy level you choose for your program. There are some examples under `core/progs` which use both versions of the `COREX` library.

7 Difference from Version 1.1

New root bound. An improved root separation bound based on the paper of Burnikel et al [8] has been implemented. Users can still build a version of our library based on the degree-length bound which was used in the previous version of the library. [See the Makefile in `core/src` for how to do this.]

Dynamic error bound checking. The current system utilizes the error bound of the computed approximate value to decide whether or not to re-evaluate a node in an expression. This is faster than just looking at the precision bound at that node because the precision bound is only a lower bound on the computed precision.

Progressive and non-progressive evaluation Users can dynamically toggle a flag to instruct the system to turn off progressive evaluation, just by calling `setIncrementalEvalFlag(false)`. This feature might conceivably be useful in comparisons in which there is a strong possibility of an equality outcome. To turn back progressive evaluation, call the same function with a `true` argument.

Expression constructor from string. We can now initialize an expression object using a string representation. Consider the declaration:

```
Expr e = 0.023;
Expr f = "0.023";
```

Note that 0.023 cannot be exactly represented as a `BigFloat`. For `Expr` e, the input value will first be converted into machine double with relative precision of u $(= 2^{-53}$, the machine precision). For `Expr` f, we approximate 0.023 to within some absolute precision which we now explain. Intuitively, if a decimal number has $k \geq 1$ positions to the right of the decimal point, we want to guarantee an absolute error of $\leq 10^{-k}$. For 0.023, $k = 3$ so that the error should be at most 10^{-3}. However, since we work with binary representation internally, we choose an error bound of $2^{-k \cdot C}$ where C is any constant $\geq \log_2 10$. The default value of C is 4. In `core/src`, the file Defs.cc provides the constant `defInputPrec` whose value specifies C. Note: if the input is an integer (i.e., $k \leq 0$) then of course the input is read without error. Thus, to to avoid any error in the input numbers, you need to read in two integers and form a rational value.

Exception handling. We introduced some simple facility for `C++` exception handling. They are defined in CoreExceptions.h under `core/inc`.

8 Bugs and Future Work

A big integer is limited to `MAXIntRep_SIZE` $(= 2^{64}-1)$ shorts. (see src/Integer.hP). Users can increase this if necessary.

Some future plans include floating point filters, improved big number packages, real algebraic numbers, more efficient root bounds (including common subexpression detection), optimized implementation of absolute precision bounds and optimized level 2.

We would like to hear your comments, suggestions and bug reports at `exact@cs.nyu.edu`.

A The `CORE` Classes

A.1 The Class `BigFloat`

The class `BigFloat` is an arbitrary precision floating point number representation that we built on top of big integers.

Fix $B = 2^{14}$. A `BigFloat` number is given as a triple $x = \langle m, err, exp \rangle$ where m is an integer, $err \in \{0, 1, \ldots, B-1\}$ and exp is an integer. The "number" x really represent the interval

$$[(m - err)\, B^{exp}, (m + err)\, B^{exp}] \tag{1}$$

We say that a real number X *belongs* to x if X is contained in this interval. In our implementation of `BigFloat`, m is `BigInt`, err is `unsigned long`, and exp is `long` for efficiency. Here `BigInt` is the class of big integers from `GNU`.

We call err the *error-bound* of x. If $err = 0$ then we say the `BigFloat` x is *error-free*. When we perform the operations $+, -, *, /$ and $\sqrt{}$ on `BigFloat` numbers, the error-bound is automatically propagated subject in the following sense: *if* X *belongs to* `BigFloat` x *and* Y *belongs to* `BigFloat` y*, and we compute* `BigFloat` $z = x \circ y$ *(where* $\circ \in \{+, -, \times, \div\}$*) then* $X \circ Y$ *belongs to* z*. A similar condition holds for the unary operations.* In other words, the error-bound in the result z must be "large enough."

There is leeway in the choice of the error-bound in z. Basically, our algorithms try to minimize the error-bound in z subject to efficiency and algorithmic simplicity. This usually means that the error-bound in z is within a small constant factor of the optimum error-bound (see [9] for full details). But this may be impossible if both x and y are error-free: in this case, the optimum error-bound is 0 and yet the result z may not be representable exactly as a `BigFloat`. This is the case for the operations of *div* and $\sqrt{\cdot}$. In this case, our algorithm ensures that the error in z is within some default precision. This is discussed under the class `Real` below.

A practical consideration in our design of the class `BigFloat` is that we insist that the error-bound err is at most B. To achieve this, we may have to truncate the number of significant bits in the mantissa m in (1) and modify the exponent exp appropriately.

A.1.1 Class Constructors for `BigFloat`

```
BigFloat();
BigFloat(int);
BigFloat(long);
BigFloat(double);
BigFloat(const BigInt&, unsigned long = 0, long = 0);
BigFloat(const char *);
```

The default constructor declares an instance with a value zero. The instances of `BigFloat` can also be constructed from `int`, `long`, `float`, `double`, `BigInt` and `string`.

```
BigFloat B;

BigInt I(5);
BigFloat B(I);

BigFloat bf1("0.023");
BigFloat bf2("1234.32423e-5");
```

A.1.2 Assignment

```
BigFloat& operator =(const BigFloat&);
// arithmetic and assignment operators
BigFloat& operator +=(const BigFloat&);
BigFloat& operator -=(const BigFloat&);
BigFloat& operator *=(const BigFloat&);
BigFloat& operator /=(const BigFloat&);
```

A.1.3 Arithmetic Operations

```
friend BigFloat operator +(const BigFloat&, const BigFloat&);
friend BigFloat operator -(const BigFloat&, const BigFloat&);
friend BigFloat operator *(const BigFloat&, const BigFloat&);
friend BigFloat operator /(const BigFloat&, const BigFloat&);
friend BigFloat sqrt(const BigFloat&);
```

A.1.4 Comparison

```
friend int operator ==(const BigFloat&, const BigFloat&);
friend int operator !=(const BigFloat&, const BigFloat&);
friend int operator <(const BigFloat&, const BigFloat&);
friend int operator <=(const BigFloat&, const BigFloat&);
friend int operator >(const BigFloat&, const BigFloat&);
friend int operator >=(const BigFloat&, const BigFloat&);
```

A.1.5 `BigFloat` Approximations

```
void approx(const BigInt& I, const extULong& r, const extLong& a);
void approx(const BigFloat& B, const extULong& r, const extLong& a);
void approx(const Rational& R, const extULong& r, const extLong& a);
```

Another important source of `BigFloat` numbers is via the approximation of `BigInt`, `BigFloat` and `Rational` numbers. We provide the member functions `approx` which take such a number, and precision bounds r and a and assigns to the `BigFloat` a value that approximates the input number to the specified precision bounds:

```
Rational R(1,3);
// declares R to have value 1/3.

BigFloat B;
BigFloat B.approx(R,16,16);
// now B contains an approximation of 1/3 to precision [16,16].
// Note that B is in the absolute regime.
```

A.1.6 `BigFloat` I/O

```
ostream& operator <<(ostream&) const;
istream& operator >>(istream&);
friend ostream& operator <<(ostream&, const BigFloat&);
friend istream& operator >>(istream&, BigFloat&);
```

Stream I/O operators are defined. The numbers read in are represented exactly if and only if the value can be represented exactly in binary. Otherwise, it is correct to at least the last digit (for more details, see the discussion in section 7 in connection to the constant `defInputPrec`).

When outputing, the error bits in a `BigFloat` representation are first truncated. Next, the output routine will print out at most `defPrtDgt` digits using a rounding to the nearest digit rule. If the absolute value of the exponent is equal to or larger than `defPrtDgt`, the output strings use the scientific notation of the form

$$m_1.m_2m_3\cdots m_\ell \ \mathbf{e} \pm e_1e_2\cdots e_m$$

where the m's and e's are in decimal notation and **e** is a literal character indicating the start of the exponent. Otherwise, the usual decimal representation would be adopted. Note that we may actually print out less than `defPrtDgt` many digits.

This output represents decimal floating point number whose value approximates the value of x correctly to the last digit. That means that the last significant digit m_ℓ really lies in the range $m_\ell \pm 1$. The number of significant digits in this output is ℓ, and it is controlled by the global variable `defPrtDgt`. The value of `defPrtDgt` is defaulted to 10, but the user can change this at run time. The value ℓ is equal to the minimum of `defPrtDgt` and the number of significant digits in the internal representation of x.

It is interesting to see the interplay between `defPrtDgt` and the composite precision `[defAbsPrec, defRelPrec]`.

Keep in mind that `defAbsPrec` and `defRelPrec` refer to binary bits; while `defPrtDgt` refers to decimal digits. For instance, if `B` is the `BigFloat` in the preceding example (an approximation of 1/3 to precision [16,16]) then here is the output for B:

```
setDefaultPrintDigits(4);
// sets the number of output digits to 4.
cout << B;
// prints the value 0.333.
setDefaultPrintDigits(6) ;
cout << B;
// prints the value 0.33333.
setDefaultPrintDigits(20);
cout << B;
// prints the value 0.33333333320915699005.
// the precision is not high enough to get all printed digits right.
setDefaultPrecision(67, CORE_INFTY);
// set adequate precision for 20 correct significant decimal digits.
cout << B;
// prints the value 0.33333333333333333333
```

Our output algorithm may not determine the optimal number of significance that `B` has. It is programmers' responsibility to set the precisions high enough to have all printed digits correct.

A.1.7 Miscellaneous

Get sign of a BigFloat

```
int sign() const;
friend int sign(const BigFloat&);
```

Absolute value

```
BigFloat abs() const;
friend BigFloat abs(const BigFloat&);
```

A.2 The Class `Real`

The class `Real` is a "heterogeneous" number system that incorporates subclasses of the real number system. We currently support the following six subclasses:

`int`, `long`, `double`, `BigInt`, `Rational`, and `BigFloat`.

Here `int`, `long` and `double` are standard `C++` types, `BigInt` and `Rational` are classes of arbitrary long numbers (from `GNU`), `BigFloat` is our arbitrary precision floating point number representation.

There is a natural type coercion among these types as one would expect. It is as follows:

$$\texttt{int} \prec \texttt{long} \prec \texttt{double} \prec \texttt{BigFloat} \prec \texttt{Rational}\ ,$$
$$\texttt{int} \prec \texttt{long} \prec \texttt{BigInt} \prec \texttt{Rational}\ .$$

To use the class Real, a program simply includes the file Real.h.

```
#include "Real.h"
```

A.2.1 Class Constructors for Real

```
Real();
Real(int);
Real(long);
Real(double);
Real(BigInt);
Real(BigFloat);
Real(Rational);
Real(const Real&);
```

The default constructor declares an instance with an default RealInt value zero. Also, being consistent with the C++ language, an instance can be initialized to be any subtype of Real: int, long, double, BigInt, Rational, and BigFloat.

A.2.2 Assignment

```
inline Real& operator =(const Real&);

// arithmetic and assignment operators
inline Real& operator +=(const Real&);
inline Real& operator -=(const Real&);
inline Real& operator *=(const Real&);
inline Real& operator /=(const Real&);

// post- and pre- increment and decrement operators
inline Real operator ++();
inline Real operator ++(int);
inline Real operator --();
inline Real operator --(int);
```

Users can assign values of type `int`, `long`, `double`, `BigInt`, `Rational`, and `BigFloat` to any instance of `Real`.

```
Real X;

X = 2;
// assigns the machine int 2 to X.

X = BigInt(4294967295);
// assigns the BigInt 4294967295 to X.

X = Rational(1, 3);
// assigns the Rational 1/3 to X.
```

A.2.3 Arithmetic Operations

```
// unary minus
inline virtual Real operator -() const;
// addition
inline virtual Real operator +(const Real&) const;
// subtraction
inline virtual Real operator -(const Real&) const;
// multiplication
inline virtual Real operator *(const Real&) const;
// division
inline virtual Real operator /(const Real&) const;
// square root
friend Real sqrt(const Real&);
```

The class `Real` supports binary operators `+, -, *, /` and the unary operators `-`, and `sqrt()` with standard operator precedence.

The rule for the binary operators $bin\_op \in \{+, -, \times\}$ is as follows: let typ_X and typ_Y be the underlying types of `Real X` and `Y`, respectively. Then the type of `X` bin_op `Y` would be $MGU = max\{typ_1, typ_2\}$ where the order $\prec$ is defined as in the type coercion in the Section A.2. Note that overflow or underflow may occur.

```
Real X, Y, Z;

X = 1; // X is of type RealInt
Y = 4294967295; // 2^32 - 1, RealInt too!
Z = X + Y;
cout << "Z = " << Z << endl;
// prints: Z = -2147483648
// Overflow in the int (RealInt) addition.!!
```

Square Root. The result of `sqrt()` is always a `BigFloat`. There are two cases: in case the original error-bound is $err > 0$, then the result of the `sqrt()` operation has error-bound at most $16\sqrt{err}$ [9]. If $err = 0$, then the absolute error of the result is at most 2^{-a} where a=`defAbsPrec`.

```
Real X = 2.0;

cout << sqrt(X) << endl;
// prints: 1.414213562
```

Division. The type typ_Z of `Z`=`X / Y` is either `Rational` or `BigFloat`. If both typ_X and typ_Y are not `float`, `double` or `BigFloat`, then typ_Z is `Rational`; otherwise, it is `BigFloat`.

If the output type is `Rational`, the output is exact. For output type of `BigFloat`, the error bound in `Z` is determined as follows. Inputs of type `float` or `double`are considered to be error-free, so only `BigFloat` can have positive error. If the error-bounds in `X` or `Y` are positive, then the relative error in `Z` is at most $12 \max\{relerr_X, relerr_Y\}$ where $relerr_X, relerr_Y$ are the relative errors in `X` and `Y`, respectively. If both `X` and `Y` are error-free then the relative error in `Z` is at most `defRelPrec`.

A.2.4 Comparison

```
inline virtual int operator ==(const Real&) const;
inline int operator !=(const Real&) const;
inline virtual int operator <(const Real&) const;
inline int operator <=(const Real&) const;
inline int operator >(const Real&) const;
inline int operator >=(const Real&) const;
```

A.2.5 Real Output

```
virtual ostream& operator <<(ostream&) const;
friend ostream& operator <<(ostream&, const Real&);
```

To output the value of an instance of `Real`, we overload the standard `C++` output stream operator `<<`. Output values are in the decimal representation. There are two kinds of decimal outputs: for `int`, `long`, `BigInt` and `Rational`, this is the exact value of `Real`. But for `double` and `BigFloat` we use the decimal floating point notation described under `BigFloat` output. For the latter output format, the number of output precision is controlled by the global variable `defPrtDgt` described under Section A.1.6.

The operator `<<` simply outputs the current value of the instance.

```
Rational R(1, 3);
BigFloat B(R);
BigInt I = 2147483647;
setDefaultPrintDigits(8); // set defPrtDgt to be 8

Real Q = R;
Real X = B;
Real Z = I;

cout << Q << endl;
// prints: 1/3

cout << X << endl;
// prints: 0.3333333

cout << Z << endl;
// prints: 2147483647
```

A.2.6 Approximation

```
inline virtual Real approx(const extULong& r, const extLong& a) const;
```

Force the evaluation of the approximate value to the composite precision $[r, a]$. The returned value is always a `RealBigFloat` value. Actually it is built upon the `BigFloat` functions described in Section A.1.5.

A.2.7 Miscellaneous

```
// get the sign of a Real value
inline virtual int sgn() const;
friend int sgn(const Real&);
```

A.3 The Class Expr

To use the class `Expr`, a program simply includes the file `Expr.h`. (The file `Real.h` is automatically included then.)

```
#include "Expr.h"
```

As will be seen below, we can use an `Expr` instance almost exactly as we would use a `Real` instance. Indeed, for most users, the ideal way to use our package is to have the user access only the class `Expr` indirectly by setting the accuracy level to 3 so that `double` and`long` will be prompted to `Expr`.

An instance of the class `Expr` E is formally a triple

$$E = (T, P, A)$$

where T is an *expression tree*, P a composite precision, and A is some real number or is $\uparrow$ (undefined value). The internal nodes of T are labeled with one of those operators

$$+, -, \times, \div, \sqrt{\cdot}, \tag{2}$$

and the leaves of T are labeled by `Real` values or is $\uparrow$. $P = [r, a]$ is a pair of integers or infinity, with r non-negative. If all the leaves of T are labeled by `Real` values, then there is a real number V that is the value of the expression T; otherwise, if at least one leaf of T is labeled by $\uparrow$, then $V =\uparrow$. Finally, the value A satisfies the relation

$$A \simeq V[r, a].$$

This is interpreted to mean either $V = A =\uparrow$ or A approximates V to precision P.

In implementation, the value A is always a `BigFloat`. The nodes of expression trees are instances of the class `ExpRep`. More precisely, each instance of `Expr` has a member `rep` that points to an instance of `ExpRep`. Each instance of `ExpRep` is allocated on the heap (so it goes away after a function call) and has a type, which is either one of the operations in (2) or type "constant". Depending on its type, each instance of `ExpRep` has zero, one or two pointers to other `ExpRep`. For instance, a constant `ExpRep`, a $\sqrt{\cdot}$-`ExpRep` and a $+$-`ExpRep` has zero pointers, one and two pointers, respectively. The collection of all `ExpRep`s together with their pointers constitute a directed acyclic graph (DAG). Every node N of this DAG defines an expression tree $E(N)$ in the natural way: the expression tree is obtained by taking the sub-DAG D_N comprising all nodes reachable from N, and for any node N' in D_N whose in-degree $indeg(N')$ is > 1, we duplicate the expression tree $E(N')$ for $indeg(N')$ times. Note that $E(N')$ is, recursively, the expression tree defined by N'.

To understand the semantics of assignment for expressions, it is useful to classify all instances of `Expr` as either *parameters* or *variables*. A parameter expression is, by definition, an instance of `Expr` whose `rep` has no pointers (i.e., is a sink in the DAG of `ExpRep`s). A variable expression is, by definition, a non-parameter expression. Hence, parameters are used to hold `Real` values and a variable are used to represent an expression. In particular, all operator nodes must be referenced via variables.

A.3.1 Class Constructors for `Expr`

```
Expr();
Expr(int);
Expr(long int);
Expr(unsigned int);
Expr(unsigned long int);
Expr(float);
Expr(double);
Expr(const BigInt &);
Expr(const BigFloat &);
Expr(const Rational &);
Expr(const char *s);
Expr(const Real &);
Expr(const Expr &); // copy constructor
```

The default constructor of `Expr` declares a parameter which contains an undefined `Real` value.

Users can declare a parameter with some initial value. When a constructor is called with some `Real` value, then a parameter which contains the specified `Real` value is declared.

A variable is declared if a constructor is called with an algebraic expression which involves operators `+`, `-`, `*`, `/` or `sqrt()`, instances of `Expr` (can be either parameters or variables), and `Real` values. However, the expressions must be dags (i.e., cycles are not allowed).

A.3.2 Assignments

```
Expr& operator=(const Expr&);

Expr& operator+=(const Expr&);
Expr& operator-=(const Expr&);
Expr& operator*=(const Expr&);
Expr& operator/=(const Expr&);

Expr& operator++();
Expr operator++(int);
Expr& operator--();
Expr operator--(int);
```

A.3.3 Arithmetic Operations

```
// unary minus
Expr operator-() const;
friend inline Expr operator+(const Expr&, const Expr&); //addition
friend inline Expr operator-(const Expr&, const Expr&); //substraction
friend inline Expr operator*(const Expr&, const Expr&); //multiplication
friend inline Expr operator/(const Expr&, const Expr&); //division
friend Expr sqrt(const Expr&); // square root
friend Expr fabs(const Expr&); // absolute value
friend Expr pow(const Expr&, unsigned long); // power
```

Partly for the convenience, integer powers can be constructed by applying the function `pow()`:

```
Expr e = 3 * pow(B, 5);
// An alternative for "Expr e = 3 * B*B*B*B*B".
```

A.3.4 Comparisons

```
friend int operator==(const Expr&, const Expr&);
friend int operator!=(const Expr&, const Expr&);
friend int operator< (const Expr&, const Expr&);
friend int operator<=(const Expr&, const Expr&);
friend int operator> (const Expr&, const Expr&);
friend int operator>=(const Expr&, const Expr&);
```

The standard `C++` comparison operators `<`, `>`, `<=`, `>=`, `==`, and `!=` perform "exact comparison". When `A < B` is tested, `A` and `B` are evaluated to sufficient precision so that the decision is made correctly. Because of root bounds, such comparisons always terminate. The returned value is a non-negative integer, where 0 means "false" while non-0 means "true".

An example,

```
Expr e[2];
Expr f[2];
e[0] = 10.0; e[1] = 11.0;
f[0] = 5.0; f[1] = 18.0;
Expr ee = sqrt(e[0])+sqrt(e[1]);
Expr ff = sqrt(f[0])+sqrt(f[1]);
if (ee>ff) cout << "sr(10)+sr(11) > sr(5)+sr(18)" << endl;
else cout << "sr(10)+sr(11) <= sr(5)+sr(18)" << endl;
// prints: sr(10) + sr(11) > sr(5) + sr(18)
```

There is another interesting example in Section 2.4 illustrating this feature.

A.3.5 I/O

```
friend ostream& operator<<(ostream&, const Expr&);
friend istream& operator>>(istream&, Expr &);
```

The standard `C++` operator `<<` outputs the stored approximate value (see the Section A.2.5 on `Real` outputs). If there is no approximate value available, it will force an evaluation. It prints as many digits of significance as is currently known as correct (to certain precisions specified), provided that it does not exceed `defPrtDgt` decimal digits. Otherwise, the extra digits would be truncated. See Section A.1.6 for examples.

A.3.6 Approximation

```
Real approx(const extULong& = defRelPrec, const extLong& = defAbsPrec);
```

This function explicitly forces one evaluation to the specified precision bounds. The expression would not be evaluated until the evaluation is requested explicitly (e.g. by calling `approx()`) or implicitly (e.g. by some I/O operations).

Users can force an instance of `Expr` to recompute its approximate value by applying the member function `approx()`. `A.approx(r, a)` evaluates `A` and get its approximate value to precision `[r, a]`. If no argument is passed, then `A` is evaluated to the default global precision [`defRelPrec`, `defAbsPrec`]. If the required precision is less than current, the function just returns the current approximate value for efficiency. The stricter of the old and new precisions is remembered, and the corresponding approximate value stored.

```
Expr e;
Real X;
unsigned r; int a;

X = e.approx(r, a);
// e is evaluated to precision at least [r, a]
// and this value is given to X;
```

A.3.7 Conversion Functions

```
double doubleValue() const; // convert to a machine built-in double
float floatValue() const;   // convert to a machine built-in float
long longValue() const;     // convert to a machine built-in long
int intValue() const;       // convert to a machine built-in int
```

Please be very *cautious* in using these operators. Overflow or underflow errors might happen silently during the conversion, while users may not be aware of them.

We do not recommend extensive use of these APIs. They were provided only for easier conversion of existing C/C++ programs. More specifically, these operator can be applied on the printf() arguments. See Section 5 for details.

A.3.8 Memory Management

```
static void* operator new( size_t );
static void operator delete( void *p, size_t );
```

To improve performance, we implemented our own customized new and delete operators for the Expr and other CORE classes.

A.4 Another Example

The following is a simple program from O'Rourke's book to "compute" the Delaunay triangulation for n points. The program basically tests all triples of points to see if their interior is empty of other points, and outputs the number of "empty" triples.

In our adaptation of O'Rourke's program below, we generate the input points to be (exactly) co-circular. Hence any level 1 program is bound to fail to compute the correct answer. At level 3 accuracy, our program detects all $\binom{n}{3}$ triples while at level 1 accuracy, we expect to miss many empty triples. For example, when $n = 5$, level 3 gives all the 10 ($= \binom{5}{3}$) triangles, while level 1 produces only 3.

```
#define Level 3
#include <stdio.h>
#include <CORE.h>
main() {
  double x[1000],y[1000],z[1000];/* input points x y,z=x^2+y^2 */
  int    n;                        /* number of input points */
  double xn, yn, zn;               /* outward vector normal to (i,j,k) */
  int    flag;                     /* t if m above of (i,j,k) */
  int    F = 0;                    /* # of lower faces */
  // define the rotation angle to generate points
  double sintheta = 5;  sintheta /= 13;
  double costheta = 12; costheta /= 13;

  printf("Please input the number of points on the circle: ");
  scanf("%d", &n);
  x[0] = 65;  y[0] = 0; z[0] = x[0] * x[0] + y[0] * y[0];
  for (int i = 1; i < n; i++ ) {
    x[i] = x[i-1]*costheta - y[i-1]*sintheta; // compute x-coordinate
    y[i] = x[i-1]*sintheta + y[i-1]*costheta; // compute y-coordinate
    z[i] = x[i] * x[i] + y[i] * y[i];         // compute z-coordinate
  }
  for (int i = 0; i < n - 2; i++ )
    for (int j = i + 1; j < n; j++ )
      for (int k = i + 1; k < n; k++ )
        if ( j != k ) {
          // For each triple (i,j,k), compute normal to triangle (i,j,k).
          xn = (y[j]-y[i])*(z[k]-z[i]) - (y[k]-y[i])*(z[j]-z[i]);
          yn = (x[k]-x[i])*(z[j]-z[i]) - (x[j]-x[i])*(z[k]-z[i]);
          zn = (x[j]-x[i])*(y[k]-y[i]) - (x[k]-x[i])*(y[j]-y[i]);
          if ( flag = (zn < 0) ) // Only examine faces on bottom of paraboloid
            for (m = 0; m < n; m++)
              /* For each other point m, Check if m above (i,j,k). */
              flag = flag &&
                ((x[m]-x[i])*xn + (y[m]-y[i])*yn + (z[m]-z[i])*zn <= 0);
          if (flag) {
            printf("lower face indices: %d, %d, %d\n", i, j, k);
            F++;
          }
        }
  printf("A total of %d lower faces found.\n", F);
}
```

References

[1] C. Burnikel. *Exact Computation of Voronoi Diagrams and Line Segment Intersections*. Ph.D thesis, Universität des Saarlandes, March 1996.

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

[3] C. Burnikel, J. Könnemann, K. Mehlhorn, S. Näher, S. Schirra, and C. Uhrig. Exact geometric computation in LEDA. In *Proc. 11th ACM Symp. Computational Geom.*, pages C18–C19, 1995.

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

[5] 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.

[6] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A core library for robust numeric and geometric computation. In *Proc. 15th ACM Symp. on Computational Geometry*, pages 351–359, June 1999. Miami Beach, Florida.

[7] M. Karasick, D. Lieber, and L. R. Nackman. Efficient Delaunay triangulation using rational arithmetic. *ACM Trans. on Graphics*, 10:71–91, 1991.

[8] K. Mehlhron, C. Burnikel, R. Fleischer, and S. Schirra. A strong and easily computable separation bound for arithmetic expressions involving square roots. *Proc. 8th ACM-SIAM Symp. on Discrete Algorithms (SODA97)*, pages 702–709, 1997. New Orleans, Lousiana.

[9] K. Ouchi. Real/Expr: Implementation of an exact computation package. Master's thesis, New York University, Department of Computer Science, Courant Institute, January 1997.

[10] J. R. Shewchuk. Robust adaptive floating-point geometric predicates. In *Proc. 12th ACM Symp. on Computational Geom.*, pages 141–150. Association for Computing Machinery, May 1996.

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

[12] C. K. Yap. Towards exact geometric computation. *Computational Geometry: Theory and Applications*, 7:3–23, 1997.

[13] C. K. Yap and T. Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, volume 1 of *Lecture Notes Series on Computing*, pages 452–492. World Scientific, Singapore, 2nd edition, 1995.