Tutorial for `Real/Expr` Package

T. Dubé,[*] K. Ouchi and C. Yap[†]
Courant Institute of Mathematical Sciences
New York University
New York, NY 10012
U.S.A.

February 10, 1997

Abstract

The `Real/Expr` 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 a overview for this package.

1 Introduction

The `Real/Expr` package is a collection of classes written in the `C++` language to facilitate the *exact computation* philosophy for geometric computation. The package is comprised of two parts: the `Real` subpackage which embodies computational knowledge for arbitrary precision real numbers, and the `Expr` subpackage which allows *precision-driven* approach to exact geometric computation. This package is a tool for implement the "exact approach" to geometric algorithms (see [Yap97, YD95]).

There are three main classes in the package: `Expr`, `Real` and `BigFloat`. The `Expr` class depends on the other two classes. The basic functionalities of our package can be obtained if users restrict themselves to the class `Expr`. However, understanding the classes `Real` and `BigFloat` is useful for understanding the behavior of the class `Expr`. Some users may want to directly access the classes `Real` or `BigFloat`.

The user may identify instances of class `Real` with various types of rational numbers. 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. Users can compare the values of two instances of `Expr`, exactly. Mathematically, our system allows one

[*]The College of the Holy Cross, Worcester, MA 10610.

[†]Supported by an National Science Foundation Grant #CCR-9402464.

to perform exact computations with any constructible real numbers (subject to physical limitations).

This tutorial describes the `Real/Expr` Package, version 1.0, released in January 1997. Full details about this implementation, including the algorithms and their error analysis, can be found in the Kouji Ouchi's masters thesis [Ouc97]. This was a complete rewrite of the original system written by Tom Dubé (circa 1993). The package is written in `GNU g++`. The complete source code, together with examples, documentation and this tutorial, can be downloaded from our project homepage

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

2 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

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

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 $2^{-a} \geq 2^{-r}|X|$) or in the "relative regime" (if $2^{-a} \leq 2^{-r}|X|$).

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

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 a default precision

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

that we initialize to $[40, 40]$. The user can change these values at compile time as well as run time. As we will see below, this precision is used in various capacities, including to control the output precision.

3 The Class `BigFloat`

The class `BigFloat` is an arbitrary precision floating point number representation that we built on top of long 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 [Ouc97] 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.

3.1 Class Constructors for `BigFloat`

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

```
BigFloat B;
// declares B with an undefined value.

BigInt I(5);
BigFloat B(I);
// declares B to have a bigInt value 5.
```

3.2 `BigFloat` Approximation of Rationals

Another important source of `BigFloat` numbers is via the approximation of rational numbers. We provide the member function `approx` which takes a

`Rational` number R and a precision p and assigns to the `BigFloat` a value that approximates R to precision p:

```
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.
```

3.3 `BigFloat` Output

Output streams understand `BigFloat` numbers. If o is an ostream and x a `BigFloat`, then "$o << x$" puts out into o a string of the form

$$d_0.d_1d_2\cdots d_\ell\ \mathbf{e} \pm c_1c_2\cdots c_m$$

where the d's and c's are in decimal notation and e is a literal character indicating the start of the exponent. 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 d_ℓ really lies in the range $d_\ell \pm 1$. 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 8, 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]`. 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`:

```
defPrtDgt= 4;
// sets the number of output digits to 4.
cout << B;
// prints the value 3.333e-1.
defPrtDgt= 6;
cout << B;
// prints the value 3.33333e-1.
defPrtDgt= 20;
cout << B;
// prints the value 3.333333e-1. This is less than 20 digits since
// that is all the available number of significant digits.
// That is, the 7 printed digits is roughly the 16 bits of
// precision specified for B.
BigFloat B1 = sqrt(B);
cout << B1;
// prints the value 5.77350e-1.
```

The last output has only 5 digits of significance; it means (roughly) that the square root of `B` has been computed to about 5 significant digits. The section on class Real explains what determines this. We say "roughly" because our output algorithm may not determine the optimal number of significance that B has – it does a conservative estimate. If the conservative estimate leads to no digits of significance being output, the output is "small":

```
BigFloat x=2;
defPrtDgt= 100;
// gives us generous amount of output precision.

defAbsPrec= 1;
cout << "Sqrt(2)=" << sqrt(x) << endl;
// prints: Sqrt(x)=Small

defAbsPrec= 8;
cout << sqrt(x) << endl;
// prints: Sqrt(x)= 1.4

defAbsPrec= 32;
cout << sqrt(x) << endl;
// prints: Sqrt(x)= 1.41421356
```

4 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"
```

4.1 Class Constructors for `Real`

The default constructor declares an instance with an undefined value. 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`.

```
Real X;
// declares X with an undefined value.

Real X = 2;
// declares X to have the value the machine int 2.

Real X = long(2147483647);
// declares X to have the value the machine long 2147483647,
// assuming 2147483647 = 2^31 - 1 is in the range of long.
// long 2147483648 would give an overflow.

Real X = 2.0;
// declares X whose value is the machine double 2.0.

Real X = bigInt(4294967295);
// declares X whose value is the bigInt 4294967295,
// assuming 4294967295 = 2^32 - 1 can be parsed by the compiler.
// Note: bigInt 4294967296 would give a compiler error.

BigFloat B=1;
Real X = B;
// declares X with BigFloat value 1.

Real X = Rational(1, 3);
// declares X with Rational value 1/3.
```

4.2 Assignment

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.
```

4.3 Real Output

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 `BigFloat`.

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

```
Rational R(1, 3);
BigFloat B(R, 10, 10);
BigFloat C(R, 100, 100);
defPrtDgt = 8;

Real X = R;
Real Y = B;
Real Z = C;

cout << X << endl;
// prints: Real(Rational 1/3)

cout << Y << endl;
// prints: Real(BigFloat 3.33e-1)
// only 3 digits are known

cout << Z << endl;
// prints: Real(BigFloat 3.3333333e-1)
// more than 8 digits are known, but the output is truncated

defPrfDgt = 16;
cout << Z << endl;
// prints: Real(BigFloat 3.3333333333e-1)
// Only 11 digits are known.
```

4.4 Binary and Unary Operators

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 4. Note that overflow or underflow may occur.

```
Real X, Y, Z;

X = 2;
cout << "X = " << X << endl;
// prints: X = Real(int 2);

Rational R(1, 3);
Y = R;
cout << "Y = " << Y << endl;
// prints: Y = Real(Rational 1/3);

Z = X + Y;
cout << "Z = " << Z << endl;
// prints: Z = Real(Rational 7/3);

X = 1;
Y = 4294967295; // 2^32 - 1
cout << "Z = " << Z << endl;
// prints: Z = Real(int 0)
// Overflow!!
```

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 $16\sqrt{err}$ [Ouc97]. If $err = 0$, then the absolute error of the result is at most 2^{-a} where a=`defAbsPrec`.

```
Real X, Y;
X=Rational(2,1);

defAbsPrec = 8;
cout << sqrt(X) << endl;
// prints: Real(BigFloat 1.4)
defAbsPrec = 16;
cout << sqrt(X) << endl;
// prints: Real(BigFloat 1.4142)
```

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`.

```
Real X=1;
Real Y=7;
BigFloatB=1;
BigFloatC=7;
Real V=B;
Real W=C;

cout << X/Y << endl;
// prints: Real(Rational 1/7);

defPrtDgt=100;
defRelPrec=15;
cout << V/W << endl;
// prints: Real(BigFloat 1.428571-1);
defRelPrec=30;
cout << V/W << endl;
// prints: Real(BigFloat 1.1.4285714286e-1);
```

4.5 Other Member Functions

To know the size of the value of an instance of `Real`, the member `LG` is provided.

```
Real X=1024;
cout << X.LG() << endl;
// prints: 10
X=1023;
cout << X.LG() << endl;
// prints: 9
X=1025;
cout << X.LG() << endl;
// prints: 10

X=BigFloat(1024.0);
cout << X.LG() << endl;
// prints: 10
X=X+1;
cout << X << " has LG = " << X.LG() << endl;
// prints: Real(BigFloat 1.025000e+3) has LG = 10
X=X-2;
cout << X << " has LG = " << X.LG() << endl;
// prints: Real(BigFloat 1.023000e+3) has LG = 9

X = Rational(1,8);
cout << X << " has LG = " << X.LG() << endl;
// prints: Real(Rational 1/8) has LG = -3
X = Rational(1,7);
cout << X << " has LG = " << X.LG() << endl;
// prints: Real(Rational 1/8) has LG = -2
X = Rational(1,9);
cout << X << " has LG = " << X.LG() << endl;
// prints: Real(Rational 1/8) has LG = -3
```

5 The Class Expr

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

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

As will be seen below, we can use an `Expr` instance almost exactly as we would use a `Real` instance. Indeed, the ideal way to use our package is to have the user access only the class `Expr`, and not to have to directly handle the class `Real`.

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 acylic 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.

5.1 Class Constructors for `Expr`

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).

```
Expr e;
// declares a parameter e whose rep points to
// the instance ExpRep holding some undefined Real value.

Real X=1.0;
Expr e(X);
cout << e << endl;
// prints: Real(BigFloat 1.0000000)

Expr f=Expr(1.0);
cout << f << endl;
// prints: Real(BigFloat 1.0000000)
// If we had used "Expr f=Expr(1)", it would prints Real(BigFloat Small).
// declares B whose rep points to
// the same instance of ExpRep as e does.

Expr e = ⟨expr⟩;
// ⟨expr⟩ is any algebraic expression
// constructed from instances of Real or Expr
// by applying operators +, −, ∗, / or sqrt().
// E.g. B + C * sqrt(D) or ( X * C ) - ( D * Y ).
```

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".
```

5.2 Assignments

Two distinct instances of the class `Expr` may share their `rep` s. Assignments to instances of `Expr` must therefore be handled carefully.

Depending on what is on the right hand side of the assignment, there are three distinct meanings for assignments.

If the RHS is a `Real` value, then the assignment is called *value assigning*; the LHS becomes a parameter.

```
Expr e, f;
Real X;

e = X;
f = 5;
// value assigning;
// e and f are now parameters.
```

If the RHS is an algebraic expression, then the assignment is called *constructing*; the LHS becomes a variable.

```
Expr e, f, g;

e = f * g;
// e is now a variable.
```

If the RHS is another instance of `Expr`, then the assignment is called *aliasing*; the LHS becomes a variable.

```
Expr e, f, g, h;
Real X;
h = X;
// h is a parameter.
e = f * g;

h = e;
// aliasing (we say "h is aliased to e").
// h is now a variable.
```

Once `D` is aliased to `A`, then any change which is performed on `A` also affects `D`. However, the converse is not true; `D = A` is different from `A = D`. Mathematically, aliasing is an antisymmetric, transitive binary relation over instances of `Expr`.

```
Expr e, f, g, h;
Real X;
e = f * g;

h = e;
// h is aliased to e and becomes a variable.

e = X;
// Not only e but also h becomes a parameter.
```

```
Expr e, f, g, h;
Real X;
h = X;
e = f * g;
// e is a variable

e = h;
// e is aliased to h and becomes a parameter.
// This is different from h = e.

e = f * g;
// e becomes a variable but h remains a parameter.
// e is no longer aliased to h.
```

So the assignment operator `=` may cause side effects on an `Expr` instance other than the obvious one on the LHS of the assignment. Users who want to make a new copy of an instance of `Expr` must use the function `duplicate()`.

```
Expr e, f;
Real X;

duplicate(e, f);
// e gets a brand new copy of f;

e = pow(X, 3);
// e becomes a variable, but f remains a parameter.
```

Another tool to avoid unnecessary side-effects is the `reset` member function. Executing `e.reset()` for the `Expr e` will make the exact value of `e` undefined without affecting any other instance of `Expr`. Similarly, `e.reset(f)` sets the exact value of `e` to be the exact value of `f` without other side-effects:

```
Expr f=2;
Expr g=3;
Expr e= f+g; // the exact value of e is 5.
f = 5; // the exact value of e is 8.
e.reset(); // the exact value of e is undefined.
e.reset(f); // the exact value of e is 5.
```

5.3 Comparisons

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".

```
Expr e[2];
Expr f[2];
e[0] = 10; e[1] = 11;
f[0] = 5; f[1] = 18;
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)
```

5.4 Outputs

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;
```

Both the standard `C++` output stream operator `<<` and the member function `print()` output the stored approximate value (see the section on `Real` outputs).

The operator `<<` outputs the current approximate value of `e`. It prints as many digits of significance as is currently known, but the number of output significance will be truncated to `defPrtDgt` many digits if necessary.

```
Rational R(1, 7);
Expr e = R;

defPrtDgt= 1;
cout << e << endl;
// prints: Real(BigFloat 1.0e-1)
// this is a truncation of what is known.

defPrtDgt= 10;
cout << e << endl;
// prints: Real(BigFloat 1.428571429e-1)
// again, this is a truncation.

defPrtDgt= 100;
cout << e << endl;
// prints: Real(BigFloat 1.4285714286e-1)
// no truncation: it knows much less than 100 digits.
```

The member function `print(ostream, n)` outputs the specified number of digits. It always converts the value of the expression to `BigFloat`. More precisely, if `e` has been computed to more than `n` digits, then we truncate the output to `n` digits; else if `e` has been computed to less than `n` digits, then the expression will be re-evaluated to at least `n` digits and `n` digits will be printed. displayed.

```
Rational R(1, 7);
Expr e = R;

e.print(cout, 5);
// prints: Real(BigFloat 1.4286e-1)

e.print(cout, 10);
// prints: Real(BigFloat 1.428571429e-1)
```

6 Example

The following attempts to compare the expressions

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

Of course, the expressions are algebraically identical. Hence comparing these two expressions always gives an equality regardless of the values assigned to x and y.

```
#include "Expr.h"

int main()
{
    Expr x;
    Expr y;

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

    BigInt x1, x2, y1, y2;

    for (;;)
    {
        cout << " Enter numerator of x :  ";
        cin >> x1;
        cout << " Enter denominator of x :  ";
        cin >> x2;
        Rational R(x1,x2);

        cout << " Enter numerator of y :  ";
        cin >> y1;
        cout << " Enter denominator of y :  ";
        cin >> y2;
        Rational S(y1,y2);

        x=R; y=S;
        cout << " e == f ?  " << (e == e ?  " yes " :  " no ") << endl;
    }

    return 0;
}
```

7 Bugs and Future Work

In the current system, we assume that the leaves of expressions are `Rational`. This is taken too literally, and you may get some surprises. Executing

`Expr e=1/3; cout << e;`

will produce the output "zero". You probably want to set up the `Expr e` as follows:

`Rational R(1,3); Expr e=R;`.

Likewise, assigning an expression to a `bigInt` will cause surprise. A future version will remove this inconvenience.

There ought to be functions to examine the error bounds in `BigFloat` and also to assign error bounds in `BigFloat`. Currently, one can look at these values but they are not very instructive without doing some calculations. Also, a `BigFloat` member function `relError()` to return the number of relative bits that are actually known will be useful (this was provided in the previous version of this package).

Adding the determinant operator as a primitive would be very useful for many applications. Currently, users can construct the determinant expression by writing its explicit polynomial form that has $n!$ terms. The Gaussian elimination approach can be implemented with some difficulty.

In Section 4.1, we indicated the upper limits of big integers that can be explicitly read into the system. There ought to be an ASCII file format for all the big number constants, and such string can be read directly. For instance, `Real R = "1234567890"`, `Real R = "bigInt(1234567890)"` or `Real R = "1234567890/987654321"`.

References

[Ouc97] Kouji Ouchi. `Real/Expr`: Implementation of an exact computation package. Master's thesis, New York University, Department of Computer Science, Courant Institute, January 1997.

[Yap97] Chee K. Yap. Towards exact geometric computation. *Computational Geometry: Theory and Applications*, 7:3–23, 1997. Invited talk, Proceed. 5th Canadian Conference on Comp. Geometry, Waterloo, Aug 5–9, 1993.

[YD95] Chee K. Yap and Thomas Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, pages 452–486. World Scientific Press, 1995. 2nd edition.