Core Library Tutorial

Chen Li, Chee Yap, Sylvain Pion, Zilin Du and Vikram Sharma
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
New York, NY 10012, USA

Nov 12, 2004†

Abstract

The Core Library is a collection of `C++` classes to support numerical computations that have a variety of precision requirements. In particular, it supports the Exact Geometric Computation (EGC) approach to robust algorithms. The implementation embodies our precision-driven approach to EGC. The library is designed to be extremely easy to use. Any `C++` programmer can immediately transform a "typical" geometric application program into fully robust code, without needing to transform the underlying program logic. This tutorial gives an overview of the Core Library, and basic instructions for using it.

†Revised: Jan 18, 1999; Sep 9, 1999; Aug 15, 2000; Sep 1, 2001; Jul 29, 2002; Jun 20, 2003; Nov 12, 2004. This work has been funded by NSF Grants #CCR-9402464, #CCF-0430836, and NSF/ITR Grant #CCR-0082056.

1 Introduction

In programs such as found in engineering and scientific applications, one can often identify numerical variables that require more precision than is available under machine arithmetic[1]. But one is also likely to find other variables that need no more than machine precision. E.g., integer variables used as array indices or for loop control. The Core Library is a collection of `C++` classes to facilitate numerical computation that desire access to a variety of such precision requirements. Indeed, the library even supports variables with irrational values (e.g., $\sqrt{2}$) and allows exact comparisons with them.

Numerical non-robustness of programs is a widespread phenomenon, and is clearly related to precision issues. Two recent surveys are [15, 21]. Non-robustness is particularly insidious in geometric computation. What distinguishes "geometric computation" from general "numerical computation" is the appearance of discrete or combinatorial structures, and the need to maintain consistency requirements between the numerical values and these structures [21]. Our library was originally designed to support the *Exact Geometric Computation* (EGC) approach to robust geometric computation [19, 22]. The EGC approach is one of the many routes that researchers have taken towards addressing non-robustness in geometric computation. Recent research in the computational geometry community has shown the effectiveness of EGC in specific algorithms such as convex hulls, Delaunay triangulation, Voronoi diagram, mesh generation, etc [7, 6, 9, 5, 1, 16]. But programmers cannot easily produce such robust programs without considerable effort. A basic goal of our project is to create a tool that makes EGC techniques accessible to *all* programmers. Through the Core Library, any `C/C++` programmer can now create robust geometric programs *without* any special knowledge of EGC or other robustness techniques. The Core Library, because of its unique numerical capabilities, has other applications beyond EGC. An example is in automatic theorem proving in geometry [17].

A cornerstone of our approach is to define a simple and yet natural numerical accuracy API (Application Program Interface). The Core Library defines four *accuracy levels* to meet a user's 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 underflow 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.

Mixed Accuracy (Level 4) Users can freely intermix the various precisions at the level of individual variables. This level is not fully defined, and only a primitive form is currently implemented.

Level 3 is the most interesting, and constitute the critical capability of EGC. Level 2 is essentially the capability found in big number package, and in computer algebra systems such as `Maple` or `Mathematica`. There is a fundamental gap between Levels 2 and 3 that may not be apparent to the casual user.

[1] In current computers, this may be identified with the IEEE 754 Standard.

One design principle in our library is that a CORE program should be able to compile and run at any of the four accuracy levels. We then say that the program can "simultaneously" access the different levels. The current library development has focused mostly[2] on Levels 1 and 3. As a result of the simultaneous access design, CORE programs can be debugged and run at various levels as convenient. E.g., to test the general program logic, we debug at Level 1, but to check the numerical computation, we debug at Level 3, and finally, we may choose to run this program at Level 2 for a speed/accuracy trade-off.

The mechanism for delivering these accuracy levels to a program aims to be as transparent as possible. In the simplest situation, the user begins with a "standard" C++ program, i.e., a C++ program that does not refer to any CORE-specific functions or classes. We call this a *Level 1 program*. Then the user can invoke Core Library's numerical capabilities just by inserting the line `#include "CORE/CORE.h"` into the program, and compiling in the normal way. In general, a key design objective is to reduce the effort for the general programmer to write new robust programs, or to convert existing non-robust programs into robust ones.

It should be evident that if an "ordinary" C++ program is to access an accuracy level greater than 1, its basic number types must be re-interpreted and overloading of arithmetic operators must be used. In Level 2, the primitive types `double` and `long` are re-interpreted to refer to the classes `BigFloat` and `BigInt`, respectively. Current implementation encloses these values inside a number type `Real`. In Level 3, both `double` and `long` refer to the class `Expr`. Think of an instance of the `Expr` class as a real number which supports exact (error-less) operations with $+, -, \times, \div$ and $\sqrt{}$, and also exact comparisons. Each instance of `Expr` maintains an *approximate value* as well as a *precision*. The precision is an upper bound on the error in the approximate value. Users can freely modify this precision, and the approximate value will automatically adjust itself. When we output an `Expr` instance, the current approximate value is printed.

Our work is built upon the `Real/Expr` Package of Yap, Dubé and Ouchi [22]. The `Real/Expr` Package was the first system to achieve Level 3 accuracy in a general class of non-rational expressions. The most visible change in the transition to Core Library is our new emphasis on ease-of-use. The CORE accuracy API was first proposed by Yap [18]. An initial implementation was described by Karamcheti et al [8]. At about the same time, Burnikel et al [2] introduced the `leda_real` Library that is very similar to Level 3 of our library.

The library has been extensively tested on the Sun UltraSPARC, Intel/Linux and Windows platforms. The main compiler for development is GNU's g++. The base distribution for Version 1.7 is less than 800 KB, including source, extensions and examples. The full distribution, which includes documentation and GMP, is less than 4MB. It can be freely downloaded from our project homepage

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

This tutorial has been updated for Core Library, Version 1.7, released on August 15, 2004.

2 Getting Started

Installing the Core Library. The CORE distribution file is called `core_vX.Y.Z.tgz`, where `X.Y.Z` denotes the library version. Thus, for the initial version 1.7, we have `X.Y.Z =`

[2]Level 1 effort simply amounts to ensuring that a Level 3 program can run at Level 1 as well. A "Level 3 program" is one that explicitly use classes or functions that are specific to Level 3.

1.7.0. Assume that the distribution file has been downloaded into some directory ${INSTALL_PATH}. In Unix, you can extract the files as follows:

```
% cd ${INSTALL_PATH}
% gzip -cd core_vX.Y.Z.tgz | tar xvf -
```

where % is the Unix prompt. This creates the directory core_vX.Y containing all the directories and files. Let ${CORE_PATH} be the full path name of this newly created directory: thus ${CORE_PATH} expands to ${INSTALL_PATH}/core_vX.Y. The Core Library directory structure is as follows:

${CORE_PATH}/doc:	Documentation
${CORE_PATH}/inc:	The header files
${CORE_PATH}/src:	Source code for the Core Library
${CORE_PATH}/lib:	The compiled libraries are found here
${CORE_PATH}/ext:	Extensions for linear algebra and geometry, etc
${CORE_PATH}/progs:	Demo programs using the Core Library
${CORE_PATH}/tmp:	Temporary directory
${CORE_PATH}/win32:	Director for Windows files
${CORE_PATH}/gmp:	gmp installation directory (may be a link)

The link ${CORE_PATH}/gmp is not present after unpacking, but will be created in the first three steps of the installation below. The README file in ${CORE_PATH} describes the easy steps to compile the library, which are as follows:

```
% cd ${CORE_PATH}
% make first       // determine system configurations for gmp
% make second      // make gmp libraries
% make third       // install gmp
% make testgmp     // check if gmp is properly installed
% make fourth      // make core library, extensionns, demo programs
% make fifth       // run sample programs
```

These five steps are equivalent to a simple "make all". The first make will determine the system configurations (platform, compilers, available files, etc). This information is needed for building the GMP library, which is the object of the second make. These first two makes are the most expensive of the installation, taking between 10–30 minutes depending on the speed of your machine. But you can skip these steps in subsequent updates or recompilation of the Core Library. The third make will install the gmp library. Before the fourth make, we do a simple check to see if gmp has been properly installed ("make testgmp"). The fourth make is equivalent to three separate makes, corresponding to the following targets: corelib, corex, demo. Making corelib creates the core library, that is, it compiles the files in ${CORE_PATH}/src resulting in the file libcore++.a which is then placed in ${CORE_PATH}/lib. Make corex creates the Core Library extensions (COREX's), resulting in the files libcorex++_level*.a being placed in ${CORE_PATH}/lib. Note that we currently create levels 1, 2 and 3 of the COREX. Make demo will compile all the sample programs in ${CORE_PATH}/progs. The fifth make will test all the sample programs. The screen output of all the above makes are stored in corresponding files in ${CORE_PATH}/tmp. An optional "make sixth" will run the fifth test with a single timing number. The above steps assume that you downloaded the full distribution (with GMP). Variant installations (e.g., for a base distribution, without GMP) is described in the README file.

Programming with the Core Library. It is simple to use the Core Library in your C/C++ programs. There are many sample programs and Makefiles under ${CORE_PATH}/progs. These could be easily modified to compile your own programs. A simple scenario is when you already have a working C++ program which needs to be converted to a CORE program. The following 2 steps may suffice:

1. Modifying your program: add one or two instructions as preamble to your program. First, use a define statement to set the CORE accuracy level:

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

   Here <level_number> can be 1, 2, 3 or 4. Next, include the Core Library header file CORE.h (found in ${CORE_PATH}/inc)

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

   To avoid potential name conflict, all header files are stored under ${CORE_PATH}/inc/CORE[3] This include line should appear *before* your code which utilizes Core Library arithmetic, but *after* any needed the standard header files, e.g. <fstream>. Note that CORE.h already includes the following:

   ```
   <cstdlib>, <cstdio>, <cmath>, <cfloat>, <cassert>, <cctype>,
   <climits>, <iostream>, <iomanip>, <sstream>, <string>.
   ```

2. Quick start to compiling and running your own programs. When compiling, make sure that ${CORE_PATH}/inc and ${CORE_PATH}/gmp/include are among the include paths (specified by the -I compiler flag) for compiling the source. When linking, you must specify the libraries from CORE and GMP and the standard math library m, using the -l flag. You also need to use the -L flag to place ${CORE_PATH}/lib and ${CORE_PATH}/gmp/lib among the library paths. E.g., to compile the program foo.cpp, type:

   ```
   % g++ -c -I${CORE_PATH}/inc -I${CORE_PATH}/gmp/include foo.cpp -o foo.o
   % g++ -o foo -L${CORE_PATH}/lib -L${CORE_PATH}/gmp/lib -lcore++ -lgmp -lm
   ```

   in this order.

The easy way to use the Core Library is to take advantage of the Core Library directory structure. This can be seen in how we compile all the demo programs. First, create your own directory under ${CORE_PATH}/progs and put your program foo.cpp there. Then copy one of the Makefiles in ${CORE_PATH}/progs. E.g., ${CORE_PATH}/progs/generic/Makefile. You can modify this make file to suit your needs. To compile foo.cpp, just modify the Makefile by adding the following line as a new target:

```
foo: foo.o
```

To compile your program, you simple type "make foo" in this directory. The examples in this tutorial are found in ${CORE_PATH}/progs/tutorial/.

[3]Before Core Library 1.6, header files were in ${CORE_PATH}/inc. For backward compatibility, you still can use `#include "CORE.h"`

Namespace CORE. The library uses its own namespace called `CORE`. Therefore classes and functions of `CORE` are accessible by explicitly prefixing them by `CORE::`. E.g., `CORE::Expr`. You can also use the global statement :

```
using namespace CORE;
```

In fact, this is automatically added by the include file `CORE.h` unless the compilation flag `CORE_NO_AUTOMATIC_NAMESPACE` is defined.

Basic Numerical Input-Output. Input and output of literal numbers come in three basic formats: scientific format (as in `1234e-2`), positional format (as in `12.34`), or rational format (as in `1234/100`). Scientific and positional can be mixed (as in `1.234e-1`) and will be collectively known as the "approximate number format". It is recognized by the presence of an "e" or a radical point. In contrast, the rational format is known as "exact number format", and is indicated by the presence of a "/". For I/O purposes, a plain integer 1234 is regarded as a special case of the rational format. Input and output of exact numbers is pretty straightforward, but I/O of approximate numbers can be subtle.

For output of approximate numbers, users can choose either scientific or positional format, by calling the methods `setScientificFormat()` or `setPositionalFormat()`, respectively. The output precision is manipulated using the standard `C++` stream manipulator `setprecision(int)`. The initial default is equivalent to

```
setprecision(6); setPositionalFormat();
```

Note that the term "precision" in `C++` streams is not consistent with our use of the term (see Section 4). In our terminology, precision are measured in "bits", while input or output representation are in "decimal digits". In contrast, precision in `C++` streams refers to decimal digits.

```
Expr e1 = 12.34; // constructor from C++ literals
Expr e = "1234.567890"; // constructor from string
        // The precision for reading inputs is controlled by defInputDigits
        // This value is initialized to be 16 (digits).
cout << e << endl;
        // prints 1234.57 as the output precision defaults to 6.
cout << setprecision(10) << e << endl; // prints 1234.567890
cout << setprecision(11) << e << endl; // prints 1234.5678900
setScientificFormat();
cout << setprecision(6) << e << endl; // prints 1.23457e+4
```

Program 1

Program 1 uses `Expr` for illustrating the main issues with input and output of numerical values. But all CORE number classes will accept string inputs as well. Of course, in Level 3, `Expr` will be our main number type.

For input of approximate numbers, two issues arise. As seen in Program 1, expression constructors accept two kinds of literal number inputs: either standard `C++` number literals

(e.g., `12.34`, without any quotes) or strings (e.g., `"1234.567890"`). You should understand that the former is inherently inexact: E.g., `12.34` has no exact machine double representation, and you are relying on the compiler to convert this into machine precision. Integers numbers with too many digits (more than 16 digits) cannot be represented by `C++` literals. So users should use string inputs to ensure full control over input precision, to be described next.

Yet another issue is the base of the underlying natural numbers, in any of the three formats. Usually, we assume base 10 (decimal representation). However, for file I/O (see below) of large numbers, it is important to allow non-decimal representations.

I/O streams understand expressions (`Expr`). The output precision in both scientific and positional formats is equal to the number of digits which is printed, provided there are that many correct digits to be printed. This digit count does not include the decimal point, the "`e`" indicator or the exponent value in scientific format. But the single 0 before a decimal point is counted.

In two situations, we may print less than the maximum possible: (a) when the approximate value of the expression does not have that many digits of precision, and (b) when the exact output does not need that many digits. In any case, all the output digits are correct except that the last digit may be off by ± 1. Note that 19.999 or 20.001 are considered correct outputs for the value 20, according to our convention. But 19.998 or 20.002 would not qualify. Of course, the approximate value of the expression can be improved to as many significant digits as we want – we simply have to force a re-evaluation to the desired precision before output.

Three simple facts come into play when reading an approximate number format into internal representation: (1) We normally prefer to use floating point in our internal representation, for efficiency. (2) Not all approximate number formats can be exactly represented by a floating point representation when there is a change of base. (3) Approximate number format can always be represented exactly by a big rational.

We use the value of the global variable `defInputDigits` to determine the precision for reading literal input numbers. This variable can be set by calling the function `setDefaultInputDigits(extLong)`. Here, the class `extLong` is basically a wrapper around the machine `long` type which supports special values such as $+\infty$, denoted by `CORE_INFTY`. If the value of `defInputDigits` is $+\infty$, then the literal number is internally represented without error, as a rational number if necessary. If `defInputDigits` is a finite integer m, then we convert the input string to a `BigFloat` whose value has absolute error at most 10^{-m}, which in base 10 means the mantissa has m digits. The initial default value of `defInputDigits` is 16.

A Simple Example. Consider 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. Running the following program in level 1 will yield incorrect results, while level 3 is always correct.

```
#ifndef CORE_LEVEL
# define CORE_LEVEL 3
#endif
#include "CORE/CORE.h" // this must come after the standard headers
int main() {
    setDefaultInputDigits(CORE_INFTY);
    double x = "12345/6789";       // rational format
    double y = "1234567890.0987654321";      // approximate format

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

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

Program 2

Terminology. We use the capitalized "CORE" as a shorthand for "Core Library" (e.g., a CORE program). Note that "Core" is not an abbreviation; we chose this name to suggest its role as the "numerical core" for robust geometric computations.

3 Expressions

The most interesting part of the Core Library is its notion of expressions, embodied in the class `Expr`. This is built on top of the class `Real`, which provides a uniform interface to the following subtypes of real numbers:

`int`, `long`, `float`, `double`, `BigInt`, `BigRat`, `BigFloat`.

Instances of the class `Expr` can be thought of as algebraic expressions built up from instances of `Real` and also real algebraic numbers, via the operators $+, -, \times, \div$ and $\sqrt{}$. Among the subtypes of `Real`, the class `BigFloat`, has a unique and key role in our system, as the provider of approximate values. It has an additional property not found in the other `Real` subtypes, namely, each `BigFloat` keeps track of an error bound, as explained next.

The simplest use of our library is to avoid explicit references to these classes (`Expr`, `Real`, `BigFloat`). Instead use standard `C++` number types (`int`,`long`,`float`,`double`) and run the program in level 3. Nevertheless, advanced users may find it useful to directly program with our number types. Appendix A serves as a reference for these classes.

Level 2 and Level 3 numbers. There are two inter-related concepts in the Core Library: *precision* and *error*. One may view them as two sides of the same coin – a half-empty cup versus a half-full cup. Within our library, they are used in a technical sense with very different meanings. Let f be an instance of the class `BigFloat`, and e be an instance of the class `Expr`. We call f a "Level 2 number" and e a "Level 3 number". Basically, we can compute with

Level 2 numbers[4] to any desired error bound. But unlike Level 3 numbers, Level 2 numbers cannot guarantee error-less results. The instance f has a *nominal value* Val_f (a real number) as well as an *error bound* Err_f. One should interpret Err_f as an upper bound on the difference between Val_f and some "actual value". The instance f does not know the "actual value", so one should just view f as the interval $\mathrm{Val}_f \pm \mathrm{Err}_f$. But a user of Level 2 numbers may be able to keep track of this "actual value" and thus use the interval properly. Indeed, Level 3 numbers uses Level 2 numbers in this way.

The idea of Level 3 numbers is novel, and was first introduced in our `Real/Expr` package [22]. The expression e also has a *value* Val_e which is exact[5]. Unfortunately, the value Val_e is in the mathematical realm ($\mathbb{R}$) and not directly accessible. Hence we associate with e two other quantities: a *precision bound* Prec_e and an *approximation* Approx_e. The library guarantees that the approximation error $|\mathrm{Approx}_e - \mathrm{Val}_e|$ is within the bound Prec_e. The nature of Prec_e will be explained in the next section. What is important is that Prec_e can be freely set by the user, but the approximation Approx_e is automatically computed by the system. In particular, if we increase the precision Prec_e, then the approximation Approx_e will be automatically updated if necessary.

In contrast to Prec_e, the error bound Err_f should not be freely changed by the user. This is because the error is determined by the way that f was derived, and must satisfy certain constraints (basically it is the constraints of interval arithmetic). For instance, if $f = f_1 + f_2$ then the error bound in f is essentially[6] determined by the error bounds in f_1 and f_2. Thus, we say that error bounds are *a posteriori values* while precision bounds are *a priori values.*

Error-less Comparisons. While we can generate arbitrarily accurate approximations to a Level 3 number, this does not in itself allow us to do exact comparisons. When we compare two numbers that happen to be equal, generating increasingly accurate approximations can only increase our confidence that they are equal, but never tell us that they are really equal. Thus, there is a fundamental gap between Level 2 and Level 3 numbers. To be able to tell when two Level 3 numbers are equal, we need some elementary theory of algebraic root bounds [20]. This is the basis of the Exact Geometric Computation (EGC) approach to non-robustness.

4 Numerical Precision and Input-Output

Numerical input and output may be subtle, but they should never be ambiguous in our system. A user should know how input numbers are read, but also know how to interpret the output of numbers.

For instance, confusion may arise from the fact that a value may be *exactly* represented internally, even though its printout is generally an approximation. Thus, the exact representation of $\sqrt{2}$ is available internally in some form, but no printout of its approximate numerical value can tell you that it is really $\sqrt{2}$. For this, you need to do a comparison test.

NOTE: Precision is always given in base 10 or base 2. Generally, we use base 2 for internal precision, and use base 10 for I/O precision. `CORE` has various global variables such as `defAbsPrec` and `defInputDigits` that controls precision in one way or other. Our naming

[4]Level 2 numbers ought to refer to any instance of the class `Real`, if only they all track an error bound as in the case of the `BigFloat` subtype. Future implementations may have this property.

[5]And real, for that matter.

[6]In operations such as division or square roots, exact results may not be possible even if the operands have no error. In this case, we rely on some global parameter to bound the error in the result.

convention for such variables tells you which base is used: precision variables in base 10 have the substring `Digit`, while precision variables in base 2 have the substring `Prec`.

The Class of Extended Longs. For programming, we introduce an utility class called `extLong` (extended long) which is useful for expressing various bounds. In our system, they are used not only for specifying precision bounds, but also root bounds as well. Informally, `extLong` can be viewed as a wrapper around machine `long` and which supports the special values of $+\infty, -\infty$ and NaN ("Not-a-Number"). These values are named `extLong::CORE_posInfty`, `extLong::CORE_negInfty` and `extLong::CORE_NaN`, respectively. For convenience, `CORE_INFTY` is defined to be `extLong::CORE_posInfty`. The four arithmetic operations on extended longs will never lead to exceptions such as overflows or divide-by-zero[7] or undefined values. This is because such operations can be detected and given the above special values. A user may use and assign to `extLong`'s just as they would to machine `long`'s.

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

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

provided either

$$\left|\widetilde{X} - X\right| \leq 2^{-r}\,|X| \qquad \text{or} \qquad \left|\widetilde{X} - X\right| \leq 2^{-a}.$$

Intuitively, r and a bound the number of "bits" of relative and absolute error (respectively) when $\widetilde{X}$ is used to approximate X. Note that we use[8] 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 black-box, not an explicit number representation) from which our system is able to generate an approximation $\widetilde{X}$. 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 $\widetilde{X}$ which satisfies $\left|\widetilde{X} - X\right| \leq 2^{-r}\,|X|$. Likewise, if $\widetilde{X} \simeq X[\infty, a]$ then $\widetilde{X}$ is an approximation of X to the absolute precision a, $|\widetilde{X} - X| \leq 2^{-a}$.

In implementation, r and a are `extLong` values. We use two global variables to specify the global composite precision:

$$[\texttt{defRelPrec}, \texttt{defAbsPrec}]. \tag{1}$$

It has the default value $[60, \texttt{CORE\_INFTY}]$. The user can change these values at run time by calling the functions:

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

[7] To delay the onset of overflows, it may be useful to extend `extLong` to implement a form of level arithmetic. E.g., when a value overflows machine `long`, we can keep track of $\log_2$ of its magnitude, etc.

[8] Jerry Schwarz, "A C++ library for infinite precision floating point" (Proc. USENIX C++ Conference, pp.271–281, 1988) uses the alternative "and" semantics.

How does the default precision in 1 control your computation? Say you perform arithmetic operations such as `z = x/y;` The system will ensure that the computed value of `z` satisfies the relation `z`$\sim x/y$[`defRelPrec`, `defAbsPrec`].

Sometimes, we want to control this precision for individual variables. If `e` is an `Expr`, the user can invoke `e.approx(rel, abs)` where `rel, abs` are extended longs representing the desired composite precision. The returned value is a `Real` instance that satisfies this requested precision. If `approx` is called without any arguments, it will use the global values [`defRelPrec`, `defAbsPrec`].

In Section 2 ("Getting Started"), we gave the basics for numerical input and output. In particular, we have 3 formats: positional (e.g., `3.14159`), scientific (e.g., `314159 e-5`), or rational (e.g., `314159/100000`). These formats can be read as a `C++` literal or as a string. But there are important differences related to precision.

Precision of Numerical Input. Consider the following input of numerical values:

```
Expr e = 0.123;      // position format in machine literal
Expr f = "0.123";    // positional format in string
Expr g = 123e-3;     // scientific format in machine literal
Expr h = "123e-3";   // scientific format in string
Expr i = 12.3e-2;    // mixed format in machine literal
Expr j = "12.3e-2";  // mixed format in string
```

The input for expressions `e`, `g` and `i` are `C++` number literals, and you may expect some error when converted into the internal representation. But the relative error of the internal representation is at most 2^{-53}, assuming the IEEE standard. In contrast, the values of the expressions `f`,`h` and `j` are controlled by the global variable `defInputDigits`. If `defInputDigits` has the value `CORE_INFTY` then `f`,`h` and `j` will have the exact rational value 123/1000. Otherwise, they will be represented by `BigFloat` numbers whose absolute error is at most $10^{-\mathtt{m}}$ where `m = defInputDigits`.

Instead of using constructors, we can also read input numbers from streams. E.g.,

```
Expr k;
cin >> k;
```

In this case, the input number literals are regarded as strings, and so the value of the variable `k` is controlled by `defInputDigits`.

Precision of Numerical Output. Stream output of expressions is controlled by the precision variable stored in the stream `cout`. Output will never print inaccurate digits, but the last printed digit may be off by ± 1. Thus, an output of 1.999 may be valid when the exact value is 2, 1.998, 1.9988 or 1.9998. But this output is invalid when the exact value is 1.99 (since the last digit in 1.999 is misleading) or 2.01. Similarly, an output of 1.234 is invalid when the exact value is 1.2 or 1.23.

Output Number Formats. We have two formats for approximate numbers: scientific and positional. But even when positional format is specified, under certain circumstances, this may be automatically overridden, and the scientific format used. For instance, if the output

precision is 3 and the number is 0.0001 then a positional output would be 0.00. In this case, we will output in scientific format as `1.00e-4` instead. Again, if the number is an integer 1234, then we will output in scientific format as `1.23e+3`. In both cases, we see why the positional output (restricted to 3 digits) is inadequate and the scientific format (also restricted to 3 digits) is more accurate. See Appendix A.1.6 for details.

One issue in numerical output is how to tell the users whether there is any error in an output or not. For instance, if you print the value of `Expr("1.0")`, you may see a plain `1.` (and not `1.0`). It is our way of saying that this value is exact. But if you print the value of `Expr(1.0)`, you may be surprised to see `1.0000`. Why the difference? Because in the former case, the internal representation is a `BigRat` while in the latter case is a machine `double`. The latter is inherently imprecise and so we print as many digits as the current output precision allows us (in this case 5 digits). But in printing a `BigRat` we do not add terminal 0's.

Interaction of I/O parameters. It should be clear from the preceding that the parameters `defRelPrec`, `defAbsPrec`, `defInputDigits`, and (stream) output precision interact with each other in determining I/O behavior:

```
setScientificFormat();
setDefaultInputDigits(2);              // defInputDigits = 2
Expr X = "1234.567890";
cout << setprecision(6);               // output precision = 6
cout << X << endl;                   // prints .123457e+4
cout << setprecision(10) << X << endl; // prints .1234567871e+4
cout << setprecision(100) << X << endl; // prints .123456787109375000e+4
```

Program 3

Note that since the input precision is set to 2, the internal value of X is an approximation to 1234.567890 with an error at most 10^{-2}. Thus the second output of X printed some "wrong digits". In particular, the output 1234.567871 contains 8 correct digits. However, notice that our semantic guarantees only 6 correct digits – so this output has given us 2 "bonus digits". In general, it is difficult to predict how many bonus digits we may get. Our convention for expressions is that all leaves are error-free and thus the output may appear strange (although it is not wrong). In fact, if we set the output precision to 100, we see that expression X is assigned the exact value of 1234.56787109375000 (we know this because this output was terminated prematurely before reaching 100 digits). To force exact input, you must set `defInputDigits` to $+\infty$:

```
setScientificFormat();
setDefaultInputDigits(CORE_INFTY);
Expr X = "1234.567890"; // exact input
cout << setprecision(6) << X << endl;   // prints .123457e+4
cout << setprecision(10) << X << endl;  // prints .1234567890e+4
cout << setprecision(100) << X << endl; // prints .12345678899999999999e+4
X.approx(CORE_INFTY, 111); // enough for 33 digits.
cout << setprecision(33) << X << endl;
// prints 33 digits:  .123456789000000000000000000000000e+4
```

Program 4

Output Stream Precision. Besides the output precision parameters for `BigFloat`, the above examples illustrate yet another parameter that controls output: namely the output precision parameter that is associated with output streams such as `cout`. This parameter as set using the standard `setprecision(int)` method of output streams. Core provides another way to do this, which is currently not entirely consistent with `setprecision(int)` method. Namely, you can call the method `setDefaultOutputDigits(long p)` method. This method will perform the equivalent of `setprecision(p)`, but in addition updates the global parameter `defOutputDigits` to `p`. It returns the previous value of `defOutputDigits`. The initial default value of `defOutputDigits` is 10.

Output for Exact and Inexact `BigFloat` Values. A `BigFloat` is represented by a triple of integers (man, exp, err) where man is the mantissa, exp the exponent and err the error. This represents an interval $(man \pm err) \times B^{exp}$ where $B = 2^{-14}$ is the base. When $err = 0$, we say the `BigFloat` value is *exact* and otherwise *inexact*. For efficiency, we normalize error so that $0 \leq err \leq B$. Since we do not want to show erroneous digits in the output, the presence of error ($err > 0$) is important for limiting the number of `BigFloat` output digits. To illustrate this, suppose $B = 2$ instead of 2^{14} and our `BigFloat` is $x = (man, exp, err) = (10, -5, 0)$, written here with decimal integers. Then `cout << setprecision(6) <<` x will show 0.3125. In general, we expect such a floating point number to have an error $B^{exp} = 2^{-5} = 0.03125$. Then $x = 0.3125 \pm 0.03125$. This suggests that we should not output beyond the first decimal place. To ensure this behavior, we can simply set the error component to 1. The method to call is x`.makeInexact()`, and as a result x becomes $(man, exp, err) = (10, -5, 1)$. Now, `cout << setprecision(6) <<` x will output only 0.3. There is a counterpart x`.makeExact()` that makes $err = 0$. The Section on Efficiency Issues has related information on this.

Connection to Real Input/Output. Expression constructor from strings and stream input of expressions are derived from the `Real` constructor from strings. See Appendix A.2.1 and A.2.5 for more details. On the other hand, stream output of expressions is derived from the output of `BigFloat` values. See Appendix A.1.6 for this.

String and File I/O. Instead of input/output from/to a stream, we can also input/output from/to a string. The methods are called `toString` and `fromString`. These methods are available for the number classes `BigFloat`, `BigInt`, and `BigRat`. For `Real` and `Expr` we only

have the method `toString`. In place of `fromString`, you can directly assign (=) a string to `Real` and `Expr` variables.

The directory `${CORE_PATH}/progs` contains examples of both string and file I/O.

The need to read very large numbers from files, and to write them into files, is important for certain computations. Moreover, base for representing these numbers in files should be flexible enough to support standard bases for big integers (base 2, 10 and 16). Such a format is specified in `${CORE_PATH}/progs/fileIO`. In particular, this format assumes that files are ascii based, for flexibility and human readability. Four basic formats are defined:

Integer, Float, Normalized Float (NFloat) and Rational.

The following methods are available:

```
void BigInt::read_from_file(istream is, long maxLen = 0)
void BigInt::write_to_file(ostream os, int base = 10, long lineLen = 70)
void BigFloat::read_from_file(istream is, long maxLen = 0)
void BigFloat::write_to_file(ostream os, int base = 10, long lineLen = 70)
void BigRat::read_from_file(istream is, long maxLen = 0)
void BigRat::write_to_file(ostream os, int base = 10, long lineLen = 70)
```

There are two bases in the representation of a `BigFloat`, the base of the mantissa and the base for raising the exponent. We call these the "mantissa base" and the "exponent base". We view `BigInt` as having only a mantissa base. In the above arguments, `base` is the mantissa base. The exponent base is defaulted to the internal base representation used by our `BigFloat` class (which is 2^{14}). Also `maxLen` indicates the number of bits (not "digits" of the input file base) to be read from the input file. The default value of `maxLen = 0` means we read all the available bits. The `lineLen` tells us how many digits should be written in each line of the mantissa. Note that when we read into a `BigFloat` and the base is 10, then this may incur an approximation but the error is guaranteed to be smaller than half a unit in the last digit.

Conversion to Machine Double. This is strictly speaking not a numerical I/O issue, but one of inter-conversion among number types. Such details are described under individual number classes (Appendix A). However, the conversion of our internal numbers into machine double values is an important topic that merits a place here.

All our number classes has a `doubleValue()` method to convert its value to one of the two nearest machine representable double value. For instance, if `e` is an expression, we can call `e.doubleValue()` a machine double value. We do not guaranteed rounding to nearest but that this value is one of the two closest machine doubles (either ceiling or floor). See `${CORE_PATH}/progstestSqrt.cpp` for some discussion of the conversion errors. It is also useful to convert an exact value into an interval defined by two machine doubles. The method `Expr::doubleInterval()` does this.

It is important to realize that all these conversions to machine doubles may overflow or underflow. It is the user's to check for this possibility. The function `int finite(double)` can be used for this check: it returns a zero if its argument does not represent a finite double value (i.e., the argument is either NaN or infinity).

5 Polynomials and Algebraic Numbers

Beginning in Version 1.6, we introduce arbitrary real algebraic numbers into expressions. For example, suppose we want to define the golden ratio $\phi = 1.618033988749894\ldots$. It can be defined as the unique positive root of the polynomial $P(X) = X^2 - X - 1$. We have a templated (univariate) polynomial class named `Polynomial<NT>` where `NT` is a suitable number type to serve as the type of the coefficients in the polynomial. We allow `NT` to be `BigInt`, `Expr`, `BigFloat`, `BigRat`, `long` or `int`. Some `Polynomial` methods may not be meaningful or obvious for certain choices of `NT`. For instance, many polynomial operations such as polynomial GCD depend on the concept of divisibility in `NT`. But the notion of divisibility for `BigFloat` values may not be so obvious (but see Appendix A under `BigFloat`). For `BigRat`, we may take the position that divisibility is the trivial relation (every nonzero value can divide other values). But this will not be our definition. For `long` and `int`, clearly coefficients may overflow. We regard `BigInt` as the "canonical" number type for polynomial coefficients because all polynomial methods will be fairly clear in this case. Hence, our polynomial examples will usually take `NT`=`BigInt`.

First consider how to input polynomials. There is an easy way to input such a polynomial, just as the string `"x^2 - x - 1"`. Currently, the coefficients from string input are assumed to be `BigInt`. A slower way (which may be more appropriate for large polynomials) is to construct an array of its coefficients.

```
typedef BigInt NT;
typedef Polynomial<NT> PolyNT;
NT coeffs[] = {-1, -1, 1}; // coeffs[i] is the coefficient of X^i
PolyNT P(2, coeffs);      // P = X^2 - X - 1
PolyNT Q = "x^2-x-1";     // Q = X^2 - X - 1
```

We use the polynomial `P` to define an `Expr` whose value is ϕ. There are two ways to identify ϕ as the intended root of `P`: we can say ϕ is the i-th smallest root of `P` (where $i = 2$) or we can give a `BigFloat` interval `I = [1, 2]` that contains ϕ as the unique root of `P`. We may call the arguments $i = 2$ or `I`$= [1.0, 2.0]$ the *root indicators* for the polynomial `P`. The other root of `P` is $\phi' = 1 - \phi$, and it has the root indicators $i = 1$ or `I = [-1.0, 0.0]`.

```
Expr phi1 = rootOf(P, 2);   // phi1 is the 2nd smallest root of P
BFInterval I(-1.0, 0.0);    // I is the interval [-1, 0]
Expr phi2 = rootOf(P, I);   // phi2 is the unique negative root of P
Expr Phi2 = rootOf(P, -1, 0); // Alternative
```

To test that `phi1` and `phi2` have the correct values, we can use the fact that $\phi + \phi' = 1$. This is done in the next code fragment.

Alternatively, we can use the fact that ϕ and ϕ' are still quadratic numbers, and we could have obtained these values by our original `sqrt` operators, e.g., $\phi = (1 + \sqrt{5})/2$. However, we offer a convenient alternative way to get square roots, by using the radical operator. In general, `radical(n, m)` gives the `m`-th root of the number `n`. Compared to the Version 1.6 the current version can handle any positive number type (`BigInt`, `Expr`, `BigFloat`, etc.) for n.

```
if (phi1 + phi2 == 1) cout << "CORRECT!" << endl;
else cout << "ERROR!" << endl;
Expr goldenRatio = (1 + radical(5,2))/2:  // another way to specify phi
if (phi1 == goldenRatio) cout << "CORRECT!" << endl;
else cout << "ERROR!" << endl;
```

Program 5

Recall that the constructible reals are those real numbers that can be obtained from the rational operations and the square-root operation. The new constructors `rootOf` and `radical` can give rise to numbers which are not constructible reals.

It is possible to construct invalid `Expr` when we specify an inappropriate root indicator: when `i` is larger than the number of real roots in the polynomial `P`, or when `I` contains no real roots or contains more than one real root of `P`. We can generalize this by allowing root indicators that are negative integers: if `i` is negative, we interpret this to refer to the $(-\texttt{i})$-th largest real root. E.g., if `i` is -2, then we want the 2nd largest real root. Moreover, we allow `i` to be 0, and this refers to the smallest *positive* root. When the root indicator is completely omitted, it defaults to this special case.

To support these methods, we define the Sturm class that implements Sturm techniques and Newton iterations. For more details, see Appendix A.4 (Polynomials) and Appendix A.5 (Sturm).

Beyond univariate polynomials. In Version 1.7, we introduce bivariate polynomials as well as algebraic curves. We provided basic functionality for doing arithmetic on curves as well as graphical display functions (based on openGL) are available. These may be found under `${CORE_PATH}/progs/curves`.

6 Converting Existing C/C++ Programs

Most of the following rules are aimed at making a Level 1 program compile and run at Level 3. So, this section might also be entitled "How to make your C/C++ program robust".

1. There is a *fundamental rule* for writing programs that intend to call the Core Library: all arithmetic operations and comparisons should assume error-free results. In other words, imagine that you are really operating on members of the mathematical domain $\mathbb{R}$ of real numbers, and operations such as $+, -, \times, \div, \sqrt{}$ return exact (error-free) results. Unfortunately, programs in conventional programming language that have been made "numerically robust" often apply tricks that violate this rule. Chief among these tricks is *epsilon-tweaking*. Basically this means that all comparisons to zero are replaced by comparison to some small, program-dependent constant ("epsilon"). There may be many such "epsilons" in the program. This is a violation of our fundamental rule.

   Perhaps the simplest way to take care of this is to set these epsilons to 0 when in Level 3. There is only one main concern here. When comparing against such epsilons, most programmers do not distinguish between "$\leq$" and "$<$". Often, they exclusively use "$<$" or "$>$" in comparisons against an epsilon. E.g., "$|x| < \varepsilon$" is taken as equivalent to $x = 0$.

If you now set ε to 0 in level 3, it is clear that you will never succeed in this test. Note that in C/C++ the usual function for absolute value is `fabs(x)`. This function will also work correctly in Level 3.

2. In your code that follows the preamble

```
#define CORE_LEVEL 3
#include "CORE/CORE.h"
```

   you must remember that the built-in machine types `double` and `long` will be replaced by (i.e., promoted to) the class `Expr`. An analogous promotion occurs at Level 2. If you do not want such promotions to occur, be sure to use `machine_double` and `machine_long` instead of `double` and `long`, respectively.

   If you are including a standard library, it is important to ensure that such promotions are not applied to your library. An example is the standard C++ library `<fstream>`, when you need to perform file I/O in a CORE program. Therefore such libraries should be placed before the inclusion of `CORE.h`.

3. All 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. `Expr` objects which are dynamically allocated using `malloc()` will not be initialized properly. You should use the `new` operator of C++ instead.

```
double *pe, *pf;

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

// This is the correct way:
pf = new double();
cout << *pf << endl;
// prints "0" (the default value)
```

4. 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.

```
double e = sqrt(double(2));
cout << e << endl; // this outputs 1.4142...  depending on current
        // output precision and default precision for evaluation
cin >> e; // reads from the standard input.
```

   Since we can construct `Expr` objects from strings, you can use `scanf` to read a string value which is then assigned to the `Expr` object. Unfortunately, current implementation does not support the use of `printf`.

```
char s[255];
scanf("%s", s);
double e = s;
```

5. Variables of type `int` or `float` are never promoted to `Expr` objects. For example,

```
// set Level 3
int i = 2;
double d = sqrt(i);
double dd = sqrt(2);
```

The two `sqrt` operations here actually refer to the standard `C` function defined in math.h, and not our exact `sqrt` found in the `Expr` class. Hence `d` and `dd` both hold only a fixed approximation to $\sqrt{2}$. The exact value can not be recovered. Here is a fix:

```
// set Level 3
int i = 2;
double e = i;                // promote i to an Expr object
double d = sqrt(e);          // the exact sqrt() is called.
double dd = sqrt(Expr(2)); // the exact sqrt() is called.
```

Users may work around this problem by defining the following macro :

```
#define sqrt(x) sqrt(Expr(x))
```

`CORE` does not define this since it may be risky for some programs.

6. In Level 3, constant literals (e.g., 1.3) or constant arithmetic expressions (e.g., 1/3) are not promoted. Hence they may not give exact values. This can cause some surprises:

```
double a = 1.0/3;   // the value of a is an approximation to 1/3
double b = 1.3;     // the value of b is also approximate
// To input the exact value of 1/3, do this instead:
double c = BigRat(1, 3); // sure way to get exact value of 1/3
double d = "1/3";   // sure way to get exact value of 1/3
double e = "1.3";   // the global defInputDigits should be
                    // +∞ in order for e to be exact.
```

Program 6

7. 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, a typical `C/C++` idiom is the following:

```
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) { ... }
```

8. Use of standard mathematical functions can be handled in two ways. Note that such functions are typically those found in `math.h`. Among these functions, only `sqrt()` is fully supported in Level 3. The other functions such as `sin()`, `cos()`, `exp()`, `log()`, etc, are not available in Level 3. If your program uses these functions, you can invoke the functions of `math.h` by explicitly converting any Level 3 number to machine double as argument:

```
double e = sqrt(2);
double s = sin(e.doubleValue()); // this invokes sin() in math.h
```

Thus we invoke the `Expr::doubleValue()` method to get a machine double value before calling the sine function. All our number types have an equivalent `doubleValue()` methods. The second way to call these mathematical functions is to use our hypergeometric package which computes these functions to any desired absolute precision. These functions are found under `${CORE_PATH}/progs/hypergeom`.

7 Using CORE and CGAL

The `CGAL` library (Computational Geometry Algorithm Library, `www.cgal.org`) provides a rich set of geometric primitives and algorithms. These primitives and algorithms are all parametrized (templated) to receive a variety of number types. In particular, you can use `CORE` number types. We recommend using Level 4, and that you directly plug `CORE::Expr` as template parameter of any CGAL kernel :

```
#include <CGAL/Cartesian.h> // From CGAL
#include "CGAL_Expr.h" // From CORE

typedef CGAL::Cartesian<CORE::Expr> Kernel;
...
```

`CORE` provides some additional functions in the file `inc/CGAL_Expr.h`, which are required by `CGAL`, so you should include this file in your program. This file sets the `Level` to 4 and includes `CORE.h`. Some example programs may be found under `${CORE_PATH}/progs/cgal`. Core Library is also distributed under `CGAL` by `Geometry Factory`, the company that distributes commercial licenses for `CGAL` components.

8 Efficiency Issues

A Level 3 `CORE` program can be much less efficient than the corresponding Level 1 program. This applies to efficiency in terms of time as well as space. First, Level 3 arithmetic and

comparison can be arbitrarily slower than the corresponding Level 1 operations. This is often caused by root bounds which may be far from optimal. Second, `Expr` objects can be arbitrarily larger in size than the corresponding machine number representation.

This inefficiency is partly inherent, the overhead of achieving robustness in your code. But sometimes, part of this overhead not inherent, but caused by the way your code is structured. Like any library, Core Library gives you the freedom to write arbitrarily inefficient programs. In programming languages too, you can also write inefficient code, except that modern optimizing compilers can detect common patterns of suboptimal code and automatically fix it. This is one of our research goals for Core Library. But until now, automatic optimization has not been our primary focus. This forces users to exercise more care. The following are hints and tools in Core Library that may speed up your code. Level 3 is assumed in the following.

Ways to Speed Up your code.

- Share subexpressions. This requires developing an awareness of how expressions are built up, and their dependency structure. Thus, in comparing $E_1 = \sqrt{x} + \sqrt{y}$ with $E_2 = \sqrt{x + y + 2\sqrt{xy}}$ (they are equal for all x, y), you could share the $\sqrt{x}$ and $\sqrt{y}$ in E_1 and E_2 as follows:

```
double x, y, sqrtx, sqrty;
cin >> x; cin >> y; // read from std input
sqrtx = sqrt(x); sqrty = sqrt(y);
double E1 = sqrtx + sqrty;
double E2 = sqrt(x + y + 2*sqrtx * sqrt y);
if (E1 == E2) cout << "CORRECT!" << endl;
else cout << "ERROR!" << endl;
```

  See `prog12.cpp` in in `${CORE_PATH}/progs/tutorial` some timings for shared and non-shared versions of this example.

- Avoid divisions in your expressions if possible. Note that input numbers that are not integers are rational numbers, and they implicitly invoke division. But a special class of rational numbers, the k-ary numbers (typically, $k = 2$ or 10) can be quite effectively handled using some new techniques based on the so-called "k-ary root bounds". `CORE` Version 1.5 has implemented such binary root bounds ($k = 2$).

- Be aware that a high algebraic degree can be expensive. But it is just as important to realize that high algebraic degree by itself is not automatically a problem. You can add a hundred positive square roots of integers, and this algebraic number may have degree up to 2^100:

```
Expr s = 0;
for (int i=1; i<=100; i++) s += sqrt(Expr(i));
```

  You can easily evaluate this number `s`, for example. But you will encounter trouble when trying to compare to such numbers that happen to be identical.

- Sometimes, use of expressions are unnecessary. For instance, consider[9] a piece of code

[9] We are grateful for this example from Professor Siu-Weng Cheng. In his example, the points p_i are 3-dimensional points on the surface of a unit sphere, which has been computed. So the coordinates of p_i are complex expressions involving square roots. Since we are sure to have a true equality comparison in this application, the resulting code was extremely slow using the known root bounds, circa year 2000.

which iterates over members of a circular list of points until it reaches the starting member. If the circular list is $(p_1, p_2, \ldots, p_n)$, and you start at some point $q = p_i$ in this list, you would normally think nothing of checking if $q = p_j$ to check if the entire list has been traversed. But this can be very inefficient. In this case, you should simply compare indexes (check if $i = j$), as this does not involve expressions.

- Avoid unbounded depth expressions. A well-known formula [12] for the signed area of a simple polygon P is to add up the signed areas of each triangle in any triangulation of P. If P has n points, you would end up with an expression of depth $\Omega(n)$. If n is large, you will end up with stack overflow. Even if there is no overflow, comparing this expression to zero (to get its sign) may be too slow. This example arises in an actual[10] software called FIST ("Fast Industrial-Strength Triangulation'). FIST is routinely tested on a database of over 5000 test polygons, some of which have up to 32,000 points. FIST uses the above formula for signed area, and this is too slow for Core Library (version 1.5) on 32,000 points (but 16,000 points can be handled). Although it is possible to avoid computing the signed area when the input polygon is simple, the signed area heuristic is critical for achieving the "industrial strength" properties of FIST. That is, the signed area approach allows FIST to produce reasonable triangulations even for "dirty data" (such as non-simple polygons). One solution in this case is to compute the signed area approximately. In fact, machine precision approximation is sufficient here. If more precision is needed, we can use Level 2 numbers (`BigFloat`).

- Sometimes, `BigFloat` can be an adequate substitute for expressions. For instance, if you are computing very high precision approximations of a number, or maintaining isolation intervals of a number you should use `BigFloat` for this purpose. For examples, see our implementation of Sturm sequences in the distribution. Here are some useful tips. First, it is important to know that `BigFloat` carries an error bound, and you should probably set this error to zero in such applications. Otherwise, this error propagates and you loose more precision than you might think. For this purpose, the following methods are useful to know: If x is a `BigFloat` variable, you can find out the error bits in x by calling x.`err`(). You can test if the error in x is 0 by calling x.`isExact()`, and if you want to set this error to zero, call x.`makeExact()`. Sometimes, you need upper or lower bounds on the interval represented by an inexact `BigFloat` value. In this case, you can call `makeCeilExact()` or `makeFloorExact()`.

  On the other hand, the inverse method x.`makeInexact()` will set the error to 1. This is useful for preventing garbage digits from being printed. For more information, see the file `BF_output.cpp` in the tutorial directory.

  Such error bits are positively harmful for self-correcting algorithms such as Newton iteration – they may even prevent Newton iteration from converging. Note that the expression $x/2$ may not return an exact `BigFloat` value even if x is exact. If you want exact result, you must call the method x.`div2`(). This is useful when you use the `BigFloat` values for interval refinement.

  To get your initial `BigFloat` value, one often computes an expression e and then call e.`getBigFloat()` to get its current `BigFloat` approximation.

  See Appendix A.1.7 for more information.

- A huge rational expression can always be replaced by an expression with just one node. This reduction is always carried out if a global Boolean variable called `rationalReduceFlag`

[10] We are grateful to Professor Martin Held for this example.

is set to true. You can set this flag by calling `setRationalReduceFlag(bool)`, which returns the previous value of the flag. For instance, when this flag is true, the time to run "make test" for CORE 1.5 programs takes 1 minute 46.4 seconds; when this flag is false, the corresponding time is 35.9 seconds. Hence we do not automatically turn on this flag. On the other hand, setting this flag to true can convert an infeasible computation into a feasible one. Prior to CORE 1.6, the pentagon test (`progs/pentagon/pentagon.cpp`) is an infeasible computation unless we use the escape precision mechanism (see below). But with this flag set to true, this test is instantaneous. Another more significant example is Martin Held's FIST program – without this flag, his program will not run with 3D data sets, crashing because of stack size problems.

The Problem of Inexact Inputs. A fundamental assumption in EGC is that inputs are exact. Even when the application does not care for exactness, we must treat the input as "nominally exact". This assumption may fail for some Level 1 programs. Handling such conversions will depend on the application, so it is best to illustrate this. An example is the FIST software above. Although the basic FIST software is for triangulating a 2-dimensional polygon, it is also able to triangulate a 3-dimensional polygon P: FIST will first find the normal of P and then project P along this normal to the xy-plane. The problem is thus reduced to triangulating a 2-dimensional polygon. In practice, the vertices of P are unlikely to lie in a plane. Hence, the "normal" of P is undefined. Instead, FIST computes the average normal for each triple of successive vertices of P. The algorithm will further sort the coordinates of the projected points in order to remove duplicate points. This sorting turns out to be extremely expensive in the presence of duplicates (since in this case, the root bounds of the huge expressions determine the actual complexity). In fact, FIST could not handle this data under CORE 1.5. It turns out that if we apply the simple heuristic (within CORE) of reducing all rational expressions into a single node, the above data could be processed by FIST without any change in their code, albeit very slowly. This heuristic is available from CORE 1.6 onwards.

As an industrial strength software, FIST must introduce such heuristics to handle inexact data. To greatly improve the speed of FIST it probably best to change the logic of its code. For instance, we suggest converting the average normal into an approximate floating point representation. If we further want to preserve the principle that "exact inputs should produce exact solutions", then we can further make FIST to first check that P is actually planar. If so, the normal can be determined from any three non-collinear vertices and hence completely avoid large expressions. Otherwise, it can use the approximate floating point representation.

Precision Escape Mechanism. It is useful to have an escape mechanism to intervene when a program does not return because of high precision. This is controlled by the following two global variables with default values:

```
extLong EscapePrec = CORE_INFTY;
long EscapePrecFlag = 0;
```

When `EscapePrec = CORE_INFTY`, the escape mechanism is not in effect. But when `EscapePrec` has a finite value like $10,000$, then we evaluate the sign of a number, we will not evaluate its value to an absolute precision that is more than past $10,000$ bits. Instead, the `EscapePrecFlag` will be set to a negative number and we will *assume* that the sign is really zero. Users can

check the value of this flag. This mechanism is applied only in the addition and subtraction nodes of an expression. An example of this usage is found `${CORE_PATH}/progs/nestedSqrt`.

When this mechanism is invoked, the result is no longer guaranteed. In practice, there is a high likelihood that the *assumed* zero is really a zero. That is because root bounds are likely to be overly pessimistic.

Floating Point Filter. It is well-established by recent research that floating point filters are extremely effective in avoiding costly big number computations. We implemented the floating point filter of Burnikel, Funke and Schirra (BFS). Note that our implementation, to achieve portability, does not depend on the IEEE floating point exceptions mechanism. This filter can be turned off or turned on (the default) by calling the function `setFpFilterFlag(bool)`.

Progressive and Non-progressive Evaluation. Users can dynamically toggle a flag to instruct the system to turn off progressive evaluation by calling `setIncrementalEvalFlag(false)`. This feature may speed up comparisons that are likely to have equality outcomes. In applications such as theorem proving (see [17]), this may be the case. However, it does not automatically lead to better performance even when the comparison result is an equality. The reason is that when we request a certain number of bits of precision, the system return a higher precision than necessary. Hence progressive evaluation may be able to achieve the desired root bound even though a lower precision is requested, while going straight to the root bound may cause significant overshoot in precision. To turn back to progressive evaluation, call the same function with a `true` argument.

9 Core Library Extensions

We plan to provide useful Core Library extensions (`COREX` for short). In the current distribution, we included two simple `COREX`'s, for linear algebra and for geometry, which the user may extend to suit their needs. The header files for these `COREX`'s are found in the files `linearAlgebra.h`, `geometry2d.h` and `geometry3d.h` under the `${CORE_PATH}/inc`. To use any of these `COREX`'s, just insert the appropriate include statements: e.g.,

```
#include "CORE/linearAlgebra.h"
```

Note that `geometry3d.h` and `geometry2d.h` already includes `linearAlgebra.h`. The source for the extensions are found under `${CORE_PATH}/ext`.

The `linearAlgebra` extension 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 three versions of the `COREX` libraries, named `libcorex++_level1.a`, `libcorex++_level2.a` and `libcorex++_level3.a`. If you use the `COREX` classes in your own program, it is important to link with the correct library depending on the accuracy level you choose for your program. See examples under `${CORE_PATH}/progs` which use both versions of the `COREX` library (in particular, `${CORE_PATH}/progs/geom2d`, `${CORE_PATH}/progs/geom3d`, and `${CORE_PATH}/progs/determinant`).

10 Miscellany

Invalid Inputs. Core will detect the construction of invalid inputs: this include NaN or Infinity for machine floats and doubles, divide by zero and square root of negative numbers. The normal behaviour is to print an error message and abort. But you can set the `AbortFlag` to false if you do not want to automatically abort. In this case, you can check if the `InvalidFlag` is negative. It is your responsibility to reset this `InvalidFlag` to a non-negative value.

Debugging. You can output the innards of an expression by calling the method `Expr::dump()`. But these may not be comprehensible except for the experts. You can print error or warning messages by calling `core_error()`, and these messages will be written into a file `Core_diagnostics`.

Variant Libraries. It is often useful to store variants of the library simultaneously. For instance, besides the normal library `libcore++.a`, we may want to use a variant library called `libcore++Debug.a` which has information for the debugger, or some other variant which implement some new techniques. In our `Makefile`s, we use a variable `VAR` whose value `${VAR}` is normally set to the empty string. This variable is defined in the file `${CORE_PATH}/make.config`. To produce a debugging version of the library, set this variable to the string "`Debug`". Then, in the `${CORE_PATH}/src`, type "`make`" to automatically create the library `libcore++${VAR}.a` which will be put in `${CORE_PATH}/lib` as usual. Finally, to use the debugging version of the library, call `g++` with the library option `-lcore++${VAR}` instead of `-lcore++`.

11 Bugs and Future Work

The ability of a single program to access all of four accuracy levels has not been fully implemented. Currently, support for Levels 2 and 4 is fairly basic. Absolute precision in Level 3 is not optimized: the system always determines the signs of expressions, which is sometimes unnecessary. It would be useful to extend `Expr` to (1) allow interval values in leaves and (2) construct real roots of polynomials whose coefficients are expressions (i.e., diamond operator).

We started to address many of the I/O issues raised in previous versions: number formats (scientific, positional, rational), the ability to read and write from files, especially in hex. But there is room for improvement. Bivariate polynomials and plane algebraic curves were introduced in Version 1.7, and are expected to be developed over the next versions.

Future plans include better floating point filters, special determinant subsystem, optimized Level 2, optimized implementation of absolute precision bounds, complex algebraic numbers, incremental arithmetic, more efficient root bounds, expression optimization (including common subexpression detection), expression compilation for partial evaluation, transcendental functions, templated versions of Linear algebra and geometry extensions, graphical facilities, enriched `COREX`'s.

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

A APPENDIX: CORE Classes Reference

There are three main classes in the CORE package: Expr, Real and BigFloat. The Expr class is built upon the other two classes and provides the basic functionalities of Level 3 accuracy. Although users do not have to directly access the Real and BigFloat classes, they are useful for understanding the behavior of the Core Library. Advanced users may want to program directly with these classes. Here is a brief summary of these classes.

Real is a "heterogeneous" number system[11] that currently incorporates the following six subtypes: int, long, double, BigInt, BigRat, and BigFloat. The first three are standard machine types while the latter three are big number types. Since Version 1.6, BigInt and BigRat are wrapper classes for gmp's mpz and mpq.

Expr is the most important class of the library. It provides the mechanism to support Level 3 accuracy. An Expr object has an approximate value as well as a precision. Users can freely set the precision of the Expr object, and its approximate value will be automatically adjusted to satisfy the precision. Currently, the approximate value is a BigFloat.

BigFloat is an arbitrary precision floating point number representation that we built on top of BigInt. It is used by our library to represent approximate values. A BigFloat is represented by the triple $\langle m, \varepsilon, e\rangle$ where m is the mantissa of type BigInt, ε is the error bound and e is the exponent. It represents the interval $(m \pm \varepsilon)B^e$ where $B = 2^{14}$. These intervals are automatically maintained when performing arithmetic with BigFloat's.

Besides these three classes, the user should know about the class extLong of "extended longs". This is a wrapper around the primitive long type with the special values of extLong::CORE_posInfty, extLong::CORE_negInfty, and extLong::CORE_NaN. For convenience, CORE_INFTY is defined to be extLong::CORE_posInfty. By using these special values, extended longs can handle overflows as well as undefined operations (divide by zero) in a graceful way. This class is extensively used in specifying root bounds and precision.

The rest of this appendix is a reference for the classes BigFloat, Real and Expr.

A.1 The Class BigFloat

A BigFloat number x is given as a triple $\langle m, err, exp\rangle$ where the *mantissa* is m, the *error-bound* is $err \in \{0, 1, \ldots, B-1\}$ and exp is the *exponent*. Here the *base* B is equal to 2^{14}.

The "number" x really represents the interval

$$[(m - err)\, B^{exp}, (m + err)\, B^{exp}] \tag{2}$$

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. You can obtain these components of a BigFloat by calling the member functions m(), err() and exp(). Since Version 1.4, BigInt is gmp's Big Integer.

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

[11] In contrast, most number systems has a "homogeneous" representation. The Real class ought to provide automatic conversions among subtypes, but this capability is not currently implemented.

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 Koji's thesis [13] 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 (the value of global variable `defAbsPrec`). 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 (2) and modify the exponent exp appropriately.

A.1.1 Class Constructors for `BigFloat`

```
BigFloat();
BigFloat(int);
BigFloat(long);
BigFloat(double);
BigFloat(const BigInt& M, unsigned long err = 0, long exp = 0);
BigFloat(const BigFloat&);
BigFloat(const char *);
BigFloat(const std::string&);
BigFloat(const BigRat &, const extLong& r = defRelPrec,
        const extLong& a = defAbsPrec);
BigFloat(const Expr &, const extLong& r = defRelPrec,
        const extLong& a = defAbsPrec);
```

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`. The last two constructors needs some clarification: (a) The constructor from strings is controlled by the global parameter `defBigFloatInputDigits`, and ensures that the `BigFloat` value constructor differs from the string value by an absolute value of at most $10^{-\texttt{defBigFloatInputDigits}}$. (b) The constructors `BigFloat(BigRat R, r, a)` and `BigFloat(Expr e, r, a)` constructs a `BigFloat` that approximates the rational `R` and expression `e` to the composite precision `[r, a]`.

```
BigInt I(5);
BigFloat B(I);
BigFloat bf1("0.023");
BigFloat bf2("1234.32423e-5");
BigRat R(1, 3);
BigFloat br(R, 200, CORE_INFTY);
```

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

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

A.1.4 Comparison

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

A.1.5 Approximations

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

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

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

BigFloat B;
B.approx(R,16,16);
// now B contains an approximation of 1/3 to precision [16,16].
```

A.1.6 Conversion Functions

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

The semantics of these expressions are mostly self-explanatory. For conversion of `BigInt`, we simply use truncation of the mantissa of the `BigFloat`. Users must exercise caution in using these conversions. Overflow or underflow errors occur silently during the conversion. It is the user's responsibility to detect such conditions.

A.1.7 Algebraic Operations

```
bool isDivisible(const BigFloat &a, const BigFloat &b);
BigFloat div_exact(const BigFloat &a, const BigFloat &b);
BigFloat gcd(const BigFloat &a, const BigFloat &b);
```

All algebraic operations are only defined for exact `BigFloat` values. The methods `isDivisible`, `div_exact` and `gcd` are global functions. The definition of divisibility is not obvious: every floating point value can be uniquely written in the form $m2^e$ where m is an odd integer and e is an integer. We say $m2^e$ divides $m'2^{e'}$ if m divides m' and $|e| \leq |e'|$ and $ee' \geq 0$. Once this concept is defined, the meaning of these algebraic operations are standard.

A.1.8 I/O

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

Stream I/O operators are defined for `BigFloat`. Integer values can be read in exactly. Fractional values are read in correctly within an absolute error of $10^{-\texttt{defBigFloatInputDigits}}$ where `defBigFloatInputDigits` is a global parameter settable by users. Note that `defBigFloatInputDigits` cannot be ∞.

Outputs utilize the precision parameter p associated with output streams. The parameter p is interpreted to be the number of digits printed: in scientific format, this is the number of significant digits but in positional format, leading zeros are counted. E.g., 0.123 and 0.001 both have $p = 4$. When outputting, the error bits in a `BigFloat` representation are first truncated. The output is in one of two formats: *positional* or *scientific*. In scientific notation, the length of the mantissa can at most be p. The extra digits are rounded (to the closest possible output value). We choose scientific notation if at least one of the following conditions is true:

1. The scientific notation flag is on. This flag is associated with output streams in C++ and can be set using standard I/O manipulator.

2. The absolute value of the number is smaller than 10^{-p+1}.
3. The absolute value of the number is bigger or equal to $10^{p-\delta}$ where $\delta = 0$ or 1 depending on whether the number is a whole number or not.

Note that we may actually print out less than p digits if the `BigFloat` value does not have that many digits of precision. If a `BigFloat` x is not error-free, the output is a decimal 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 $d_\ell \pm 1$, where d_ℓ is the last digit of output.

It is interesting to see the interplay between ostreams' `precision` p and the composite precision `[defAbsPrec, defRelPrec]`. Keep in mind that `defAbsPrec` and `defRelPrec` refer to binary bits.

```
double q = BigRat(1, 3);
setDefaultAbsPrecision(67); // about 20 digits
cout << "q = " << setprecision(10) << q << ", in 10 digits" << endl;
// output:  q = 0.33333333, in 10 digits
cout.precision(30); // or use setDefaultOutputDigits(30, cout),
// default to output 30 digits.
cout << "q = " << q << endl;
// output:  q = 0.33333333333333333333, in positional notation.
```

Program 7

It is programmers' responsibility to set the composite precisions high enough to have all requested digits printed correctly.

A.1.9 Miscellaneous

To get the sign of the mantissa in a `BigFloat`, use

```
int BigFloat::sign()
```

which returns one of the values $-1, 0, +1$. Since there may be error in a `BigFloat`, this may not be taken as the sign of the `BigFloat` unless you also verify that the following predicate is false:

```
bool BigFloat::isZeroIn().
```

This predicate is true iff 0 lies in the interval $[mantissa \pm err]$.

Another useful function is

```
BigFloat::isExact()
```

which returns a true Boolean value if the error component of the `BigFloat` is zero. We can set this error component to zero by calling

```
BigFloat::makeExact().
```

There are two variants, `makeCeilExact()` and `makeFloorExact()`. Why would one do this? There are applications where the error bound is not needed. An example is when you implement a Newton iterator for roots. The algorithm is self-correcting, so the error bound is not

necessary. But after an inexact operation (e.g., division), the error bound is nonzero. If you do not set this to zero, our automatic significance arithmetic algorithms may start to truncate the mantissa in order to keep the error bound from growing. This may prevent Newton from converging.

The most significant bit (MSB) of any real number x is basically $\lg|x|$ (log to base 2). When x is an exact `BigFloat`, this value is an integer. In general, x has an error, and it represents an interval of the form $[x-\varepsilon, x+\varepsilon]$. We provide two functions, `BigFloat::uMSB()` and `BigFloat::lMSB()`, each returning an extended long. Assuming that $0 < x - \varepsilon$, the following inequalities must hold

$$\texttt{BigFloat::lMSB()} \leq \lg|x-\varepsilon| \leq \lg|x+\varepsilon| \leq \texttt{BigFloat::uMSB()}.$$

When $x + \varepsilon < 0$, we modify the inequalities approproriately. When `x.isExact()` is true, the inequalities become equalities, and you can simply call the function `x.MSB()`. Here is an application: suppose you have computed a `BigFloat` value x approximates $\sqrt{2}$. To see how close x is to $\sqrt{2}$, you can compute

```
extLong p = (x*x - 2).uMSB() .
```

E.g., to guarantee that $x - \sqrt{2} < 2^{-100}$, it is enough to make sure that this p is less than -97. An alternative approach is to compare $x * x - 2$ to 2^{-97}. You can obtain the value 2^{-97} by calling another helper function

```
BigFloat::exp2(int n)
```

which returns a `BigFloat` whose value is 2^n.

A `BigFloat` is really a wrapper about a `BigFloatRep` object. Sometime, you may like to get at this "rep", using the member function `getRep()`.

A.2 The Class `Real`

The class `Real` provides a uniform interface to the six subtypes of numbers: `int`, `long`, `double`, `BigInt`, `BigRat`, and `BigFloat`. There is a natural type coercion among these types as one would expect. It is as follows:

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

The `BigFloat` in this coercion is assumed to be error-free. To use the class `Real`, a program simply includes the file `Real.h`.

```
#include "CORE/Real.h"
```

A.2.1 Class Constructors for `Real`

```
Real();
Real(int);
Real(long);
Real(double);
Real(const BigInt&);
Real(const BigRat&);
Real(const BigFloat&);
Real(const Real&);
Real(const char *str, const extLong& prec = defInputDigits);
Real(const std::string& str, const extLong& prec = defInputDigits);
```

The default constructor declares an instance with a default `Real` with the value zero. Consistent with the `C++` language, an instance can be initialized to be any subtype of `Real`: `int`, `long`, `double`, `BigInt`, `BigRat`, and `BigFloat`.

In the last constructor from string, the conversion is exact in three cases: (i) when the value is integral (i.e., the string has only digits); (ii) rational (i.e., the string contains one '/' character and digits otherwise); (iii) when `prec` = `CORE_INFTY`. Otherwise, it will convert to a `BigFloat` number within absolute precision 10^{-prec}.

A.2.2 Assignment

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

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

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

Users can assign values of type `int`, `long`, `double`, `BigInt`, `BigRat`, 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 = BigRat(1, 3); // assigns the BigRat 1/3 to X.
```

A.2.3 Arithmetic Operations

```
Real operator-() const;
Real operator+(const Real&) const;
Real operator-(const Real&) const;
Real operator*(const Real&) const;
Real operator/(const Real&) const;
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 rules in Section A.2. For instance, consider Program 8. Although `X` and `Y` are of type `RealInt`, their sum `Real Z` is of type `RealBigInt` since the value of `Z` cannot be represented in `RealInt`.

```
Real X, Y, Z;
unsigned int x, y, z;
int xx, yy, zz;

X = 1; x = 1; xx = 1;
Y = 4294967295; // 2^32 - 1
y = 4294967295; // 2^32 - 1
yy = 2147483647; // 2^31 - 1
Z = X + Y; z = x + y; zz = xx + yy;
cout << "Z = " << Z << endl;  // prints:  Z = 4294967296 (correct!)
cout << "z= " << z << endl;  // prints:  z = 0 (overflow)
cout << "zz= " << zz << endl; // prints:  z = -2147483648 (overflow)
```

Program 8

Square Root. The result of `sqrt()` is always a `BigFloat`. There are two cases: in case the original input has an error $err > 0$, then the result of the `sqrt()` operation has error-bound at most $16\sqrt{err}$, see [13]. If $err = 0$, then the absolute error of the result is at most 2^{-a} where a=`defAbsPrec`.

```
Real X = 2.0;

cout << setprecision(11) << sqrt(X) << endl;
// prints: 1.414213562
```

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

If the output type is `BigRat`, 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

```
bool operator==(const Real&) const;
bool operator!=(const Real&) const;
bool operator< (const Real&) const;
bool operator<=(const Real&) const;
bool operator> (const Real&) const;
bool operator>=(const Real&) const;
```

A.2.5 Real I/O

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

The input string is parsed and if the value is integral, it is read in exactly. Otherwise, it is approximated with absolute precision `defInputDigits` (in decimal). Note that if `defInputDigits` is ∞, the input is read in exactly as a big rational number.

To output the value of an instance of `Real`, we can use the standard `C++` output stream operator `<<`. Output is in decimal representation. There are two kinds of decimal outputs: for `int`, `long`, `BigInt` and `BigRat` subtypes, this is the exact value of `Real`. But for `double` and `BigFloat` subtypes, we use the decimal floating point notation described under `BigFloat` output.

```
BigRat R(1, 3);
BigFloat B(R);
BigInt I = 1234567890;
cout.precision(8); // set output precision to 8

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

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

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

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

cout << Z << endl; // prints:  .12345679e+10
```

Program 9

Other related functions are

```
int BigInt::fromString(const char* s, int base = 0);
std::string BigInt::toString(int base = 10);
```

If `base = 0`, then a prefix `p` in string `s` tself determines the base: `p = 0b` means binary, `p = 0` means octal, `p = 0x` means hexadecimal, `p` is the empty string means decimal.

A.2.6 Approximation

```
Real approx(const extLong& 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.

A.2.7 Miscellaneous

```
// get the sign of a Real value
int Real::sign() const;
int sign(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 with `Expr.h`.

```
#include "CORE/Expr.h"
```

For most users, the ideal way to use our library 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 $\uparrow$ (undefined value). The internal nodes of T are labeled with one of those operators

$$+, -, \times, \div, \sqrt{\cdot}, \tag{3}$$

and the leaves of T are labeled by `Real` values or is $\uparrow$. $P = [r, a]$ is a pair of `extLong`, 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 the current implementation, leaves must hold exact values. Moreover, the value A is always a `BigFloat`. The nodes of expression trees are instances of the class `ExprRep`. More precisely, each instance of `Expr` has a member `rep` that points to an instance of `ExprRep`. Each instance of `ExprRep` is allocated on the heap and has a type, which is either one of the operations in (3) or type "constant". Depending on its type, each instance of `ExprRep` has zero, one or two pointers to other `ExprRep`. For instance, a constant `ExprRep`, a $\sqrt{\cdot}$-`ExprRep` and a $+$-`ExprRep` has zero, one and two pointers, respectively. The collection of all `ExprRep`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. Unlike [13], assignment to `Expr` has the standard semantics. As an example, after the assignment $e = f \odot g$, $\mathrm{Val}(e) = \mathrm{Val}(f) \odot \mathrm{Val}(g)$ and $\mathrm{Val}(e)$ does not change until some other assignment to e. In particular, subsequent assignments to f and g do not affect $\mathrm{Val}(e)$.

A.3.1 Class Constructors for `Expr`

```
Expr();
Expr(int);
Expr(long);
Expr(unsigned int);
Expr(unsigned long);
Expr(float);
Expr(double);
Expr(const BigInt &);
Expr(const BigFloat &);
Expr(const BigRat &);
Expr(const char *s, const extLong& prec=defInputDigits);
Expr(const std::string &s, const extLong& prec=defInputDigits);
Expr(const Real &);
Expr(const Expr &); // copy constructor
template<class NT>
    Expr(const Polynomial<NT>& p, int n=0);
        // this specifies the n-th smallest real root.
template<class NT>
    Expr(const Polynomial<NT>& p, const BFInterval& I);
        // this specifies the unique real root in interval I.
```

The default constructor of `Expr` constructs a constant `Expr` object with the value zero. When a constructor is called with some `Real` value, then a parameter which contains the specified `Real` value is declared.

In Core Library 1.6, we introduced a new constant `Expr` object which is constructed from a polynomial (see Appendix A.4 for the `Polynomial` class). The value is a real root of this polynomial, and so we need an argument to indicate a unique root. This can be an closed interval I comprising a pair of `BigFloat`'s, or an integer n. Collectively, both n and I are called *root indicators*. The interval I must be *isolating* meaning that it contains a unique real root of the polynomial. If $n \geq 1$, then we specify the nth *smallest* real root (so $n = 1$ is the smallest real root). If $n \leq -1$, this refers to the $(-n)$-th *largest* real root.

For convenience, we also provide three global functions to help the user construct such `Expr` node:

```
template<class NT>
    rootOf(const Polynomial<NT>& p, int n=0);
template<class NT>
    rootOf(const Polynomial<NT>& p, const BFInterval& I);
template<class NT>
    radical(const NT& k, int m); // the m-th root of k
```

So for a polynomial `P`, both `Expr e(P, i)` and `Expr e = rootOf(P, i)` are equivalent (where `i` is a root indicator).

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

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

For the convenience and efficiency, integer powers can be constructed by applying the function `power()`.

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

A.3.4 Comparisons

```
bool operator==(const Expr&, const Expr&);
bool operator!=(const Expr&, const Expr&);
bool operator< (const Expr&, const Expr&);
bool operator<=(const Expr&, const Expr&);
bool operator> (const Expr&, const Expr&);
bool 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".

```
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)
```

Program 10

A.3.5 Expr I/O

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

The input will construct a ConstRep with a Real value read in from the input stream. The input routine for Real is discussed in Section A.2.5.

The standard C++ operator << outputs the stored approximate value which is always a BigFloat number. If there is no approximate value available, it will force an evaluation to the default precisions. It prints as many digits of significance as is currently known as correct (up to the output precision specified). See Section A.1.8 for examples.

A.3.6 Approximation

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

A.approx(r, a) evaluates A and returns 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 already satisfied by the current approximation, the function just returns the current approximate value.

An expression is not evaluated until the evaluation is requested explicitly (e.g., by `approx()`) or implicitly (e.g. by some I/O operations).

```
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;
```

The following helper functions allow you to get at the current approximate value in an `Expr`:

```
Expr e;
⋮
e.sign(); // returns the exact sign of e (note that e.getSign() is deprecated,
             as "sign()" is the uniform interface for all the number classes
e.BigFloatValue(); // returns the current BigFloat approximation
e.getMantissa(); // returns the mantissa of current BigFloat
e.getExponent(); // returns the exponent of current BigFloat
```

A.3.7 Conversion Functions

```
double doubleValue() const;  // convert to machine built-in double
float floatValue() const;    // convert to machine built-in float
long longValue() const;      // convert to machine built-in long
int intValue() const;        // convert to machine built-in int
BigInt BigIntValue() const;  // convert to a BigInt number
BigRat BigRatValue() const;  // convert to a BigRat number
BigFloat BigFloatValue() const;// convert to a BigFloat number
```

The semantics of these operations are clear except for converting into `BigRat` or `BigFloat`. For `BigFloat`, we use the current approximate value of the expression. For `BigRat`, we use the same `BigFloat` value converted into a rational number. Note that users must exercise caution in using these conversions. Overflow or underflow errors occur silently during the conversion. It is the user's responsibility to detect such conditions. Nevertheless, they are useful for converting existing `C/C++` programs. E.g., these operators can be applied on the `printf()` arguments. See Section 6 for details.

A.4 Filters and Root Bounds

The expression class has an elaborate mechanism for computing root bounds, and a floating point filter. Our filters is based on the so-called BFS Filter [4]. Our root bounds are a

combination of several techniques (BFMSS Bound, Measure bound and conjugate bound). In fact, the BFMSS bound is the so-called k-ary version [14]. For more details on these topics, see [10].

A.5 The Template Class `Polynomial`

Class `Polynomial` is a template class, which can be instantiated with the number type `NT` of polynomial coefficients. We support `NT` chosen from `int BigInt BigFloat BigRat` and `Expr`.

Since Version 1.6, the Class `Polynomial` is incorporated into Core Library. In particular, the file `CORE.h` or `Expr.h` automatically include the files `poly/Poly.h` and `poly/Poly.tcc`. The following constructors are available for this class:

```
Polynomial(); // the Zero Polynomial
Polynomial(int n); // the Unit Polynomial of nominal deg n ≥ 0
Polynomial(int n, NT* coef); // coef is the array of coefficients
Polynomial(const VecNT &); // VecNT is a vector of coefficients
Polynomial(int n, const char * s[]);
Polynomial(const Polynomial &);
Polynomial(const string & s, char myX='x' );
Polynomial(const char * s, char myX='x' );
```

The last two constructors takes a string `s`. They are convenient and intuitive to use, and works best for up to moderate size polynomials. For instance, The user can construct a polynomial by calling `Polynomial p("3x^2 + 4*x + 5")` using the default variable name `x`. If you use some other variable name such as `Z`, then you can use the second argument to specify this. E.g., `Polynomial p("3Z ^2 + 4*Z + 5", 'Z')`. The syntax for a valid input string `s` given by the following BNF grammar:

```
[poly] -> [term] | [term] '+/-' [poly] \\
                 | '-' [term] | '-' [term] '+/-' [poly] \\
[term] -> [basic term] | [basic term] [term] | [basic term]*[term]\\
[basic term] -> [number] | 'x'
                 | [basic term] '^' [number] | '(' [poly] ')'
```

The recursiveness in these rules meant that an input string such as `s = "(2x - 1)^2 (x^2 - 2x + 3)"` is valid. See `${CORE_PATH}/progs/poly/parsePoly.cpp` for examples.

When specifying an array `coef` of coefficients, the coefficient of the power product x^i is taken from `coeff[i]`. So the constant term is `coeff[0]`. If we want to reverse this ordering (and treat `coeff[0]` as the leading coefficient), we can first use the above constructor, and then reverse the polynomial (the reverse method is listed below).

An example of how to use these constructors are shown below:

```
typedef BigInt NT;
typedef Polynomial<NT> PolyNT; // convenient typedef
PolyNT P1; // Zero Polynomial
PolyNT P2(10); // Unit Polynomial of degree 10
NT coeffs[] = {1, 2, 3};
PolyNT P3(1, coeffs); // P3(x) = 1 + 2x + 3x**2
const char* s[] = {"123456789", "0", "-1"};
PolyNT P4(1, s) // P4(x) = 123456789 - x**2;
PolyNT P5("u^2(u + 234)^2 - 23(u + 2)*(u+1)", 'u');
```

You can also input these coefficients as strings (this is useful when the coefficients are so large that they may overflow a machine integer).

A.5.1 Assignments

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

Polynomial& operator+=(const Polynomial&);
Polynomial& operator-=(const Polynomial&);
Polynomial& operator*=(const Polynomial&);
```

A.5.2 Arithmetric Operations

```
Polynomial& operator-();

Polynomial& operator+(const Polynomial&, const Polynomial&);
Polynomial& operator-(const Polynomial&, const Polynomial&);
Polynomial& operator*(const Polynomial&, const Polynomial&);
```

A.5.3 Comparisons

```
bool operator ==(const Polynomial&, const Polynomial&);
bool operator !=(const Polynomial&, const Polynomial&);
```

A.5.4 I/O

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

A.5.5 Manipulation and Query Functions

The following methods are used to manipulate (i.e., modify) to query polynomials:

```
int expand(int n); // Change the nominal degree to n
int contract(); // get rid of leading zeros
int getDegree() const; // nominal degree
int getTrueDegree() const; // true degree
const NT& getLeadCoeff() const; // get TRUE leading coefficient
const NT& getTailCoeff() const; // get last non-zero coefficient
NT** getCoeffs() ; // get all coefficients
const NT& getCoeff(int i) const; // Get coefficient of x^i
bool setCoeff(int i, const NT& cc);i // Makes cc the coefficient
    // of x^i; return FALSE if invalid i.
void reverse(); // reverse the coefficients
Polynomial & negate(); //Multiply by -1.
int makeTailCoeffNonzero(); // Divide (*this) by x^k, so that
    // the tail coeff is non-zero. Return k.
```

A.5.6 Algebraic Polynomial Operations

```
Polynomial& differentiate(); // self-differentiation
Polynomial& differentiate(int n); // multi self-differentiation
Polynomial& squareFreePart(); // P/gcd(P,P')
Polynomial& primPart(); // Primitive Part
Polynomial pseudoRemainder (const Polynomial& B, NT& C);
    // The pseudo quotient of (*this) mod B is returned, but (*this) is
    // transformed into the pseudo remainder. If argument C is not not
    // null, then C*(*this) = B*pseudo-quotient + pseudo-remainder.
Polynomial & negPseudoRemainder (const Polynomial& B);
    // Same as the previous one, except negates the remainder.
Polynomial reduceStep (Polynomial& p );
```

All of the above operations are self-modifying. If this is undesirable, the user ought to make a copy of the polynomial first.

A.5.7 Numerical Polynomial Operations

These operations include evaluation and root bounds:

```
Expr eval(const Expr&) const; // polynomial evaluation
BigFloat eval(const BigFloat&) const; // polynomial evaluation
template <class myNT> myNT eval(const myNT&) const; // evaluation at an
    //arbitrary number type.
BigFloat CauchyUpperBound() const; // Cauchy Root Upper Bound
BigFloat CauchyLowerBound() const; // Cauchy Root Lower Bound
BigFloat sepBound() const; // separation bound (multiple roots allowed)
BigFloat height() const; // height function
BigFloat length() const; // length function
```

Note that the `eval` function here is a generic function: it allows you to evaluate the polynomial at any number type `myNT`. The return type is also `myNT`. To do this, we convert each coefficient (which has type `NT`) of the polynomial into type `myNT`. Then all the operations of the evaluation is performed within the class `myNT`. For this to work properly, we therefore require that `NT`≤`myNY` (recall that there is a natural partial ordering among number types). For instance, if `NT`=`BigFloat`, then `myNT` can be `BigFloat`, `BigRat` or `Expr`. In particular, using `myNT` will ensure exact results; but this may be expensive and in many situations, `BigFloat` is the correct choice (e.g., Newton iteration).

A.5.8 Miscellaneous

Some methods in `Polynomial` depend on the choice of `NT`. In particular, some methods need to know whether the coefficient type `NT` supports[12] "general" division. Hence we require all such number types to provide a static method `NT::hasDivision()` that returns a boolean value. Among the supported `NT`, only `BigRat` and `Expr` has general division.

A.6 The Template Class `Sturm`

This class implements the Sturm sequence associated with a polynomial. Starting with Version 1.7 this class can handle `int`, `BigInt`, `long`, `BigRat`, `BigFloat`, and `Expr`. The most important being `BigInt`, `BigFloat`, and `Expr`, although the last one can be inefficient for polynomials with large degree. The constructors are:

```
Sturm(); // null constructor
Sturm(PolyNT pp); // constructor from polynomial
Sturm(int n, NT * c);// constructor from an array of coefficients
Sturm(const Sturm& s); // copy constructor
```

After we have constructed a Sturm object based upon some polynomial, we can use the following functions to get more properties as described below.

[12] We say "general division" to distinguish this from special kinds of division such as division by 2 (this is supported by `BigFloat`) or exact division (this is supported by `BigInt`).

A.6.1 Functions in `Sturm` Class

```
int signVariations(const BigFloat& x, int sx);
    // Gets the sign variations of the Sturm sequence at a given point
int signVariationsAtPosInfty();
int signVariationsAtNegInfty();
int numberOfRoots(const BigFloat& x, const BigFloat& y);
    //Number of roots in the closed interval [x, y]
int numberOfRoots();// number of real roots of the polynomial
int numberOfRootsAbove(const BigFloat &x);
int numberOfRootsBelow(const BigFloat &x);
void isolateRoots(const BigFloat &x, const BigFloat &y, BFVecInterval &v);
    //Isolates all the roots in the interval [x,y] and returns them in v
    //a list of intervals
void isolateRoots(BFVecInterval &v); // Isolates all the roots
BFInterval isolateRoot(int i); // Isolate the i-th smallest
    // root, if i < 0 then we get the i-th largest root
BFInterval isolateRoot(int i, BigFloat x, BigFloat y);
    // Isolate the i-th smallest root in the interval [x,y]
BFInterval firstRootAbove(const BigFloat &e);
BFInterval firstRootBelow(const BigFloat &e);
BFInterval mainRoot();//First root above 0
BFInterval refine(const BFInterval& I, int aprec);
    // Refine the interval I containing the root using bisection
BFInterval refinefirstRootAbove(const BigFloat &e, int aprec);
    //Get an absolute approximation to aprec of the first root above e.
    //Achieved using the refine method above.
BFInterval refinefirstRootBelow(const BigFloat &e, int aprec);
    // Similar to previous method, except refines the first root below e
void refineAllRoots( BFVecInterval &v, int aprec);
    //Refines all the roots to absolute precision aprec (based upon refine)
```

A main feature of the Sturm Class is that it provides standard Newton iteration using which we can converge rapidly to any root of the underlying polynomial. The following methods provide the desired functionality.

A.6.2 Newtons Method in `Sturm` Class

```
BigFloat newtonIterN(long n, const BigFloat& bf, BigFloat& del,
    unsigned long & err);
    // Does n steps of standard Newton's method starting from the initial
    // value bf. The return value is the approximation to the root after
    // n steps. del is an exact BigFloat which is an upper bound on the
    difference between the n-th and n-1-th approximation, say del_{n-1}.
    err is an upper bound |del - del_{n-1}|.
BigFloat newtonIterE(int prec, const BigFloat& bf, BigFloat& del);
    // Does Newton iteration till del.uMSB() < -prec
BFInterval newtonRefine(const BFInterval I, int aprec);
    // Given an isolating interval I for a root x*, will return
    //an approximate root x such that |x - x*| < 2^{-aprec}.
    //Assumes that the interval end points are known exactly.
void newtonRefineAllRoots( BFVecInterval &v, int aprec);
    // Refines all the roots of the polynomial to the desired precision
    // aprec using newtonRefine above
bool smaleBoundTest(const BigFloat& z); // Implementation of
    // Smale's point estimate to determine whether we have reached
    // Newton basin. This is an a posteriori criterion unlike the next.
BigFloat yapsBound(const Polynomial<NT> & p);// An apriori bound
    // to determine whether we have reached Newton zone.
```

A.7 The Template Class `Curve`

Introduced in Version 1.7, this class allows the user to manipulate arbitrary real algebraic curves. The `Curve` class is derived from the `BiPoly` class, so we begin by describing the `BiPoly` class:

```
BiPoly(); //Constructs the zero bi-poly.
BiPoly(int n);// creates a BiPoly with nominal y-degree of n.
BiPoly(std::vector<Polynomial<NT> > vp); // From vector of Polynomials
BiPoly(Polynomial<NT> p, bool flag=false);
    //if true, it converts polynomial p(x) into p(y)
    //if false, it creates the bivariate polynomial y - p(x)
BiPoly(int deg, int *d, NT *C); //Takes in a list of list of
    // coefficients. Each cofficient list represents a polynomial in x
    // deg - ydeg of the bipoly
    // d[] - array containing the degrees of each coefficient
    // (i.e., x poly)
    // C[] - list of coefficients, we use array d to select the
    // coefficients.
BiPoly(const BiPoly<NT>&); //Copy constructor
BiPoly(const string& s, char myX='x', char myY='y');
BiPoly(const char* s, char myX='x', char myY='y');
```

The last two constructors from strings are similar to the ones for `Polynomial`. The syntax of valid input string is determined by a BNF grammar that is identical to the one for univariate polynomials, except that we now allow a second variable `'y'`.

A.7.1 Assignments

```
BiPoly<NT> & operator=( const BiPoly<NT>& P); // Self-assignment
BiPoly<NT> & BiPoly<NT>::operator+=( BiPoly<NT>& P); // Self-addition
BiPoly<NT> & BiPoly<NT>::operator-=( BiPoly<NT>& P); //Self-subtraction
BiPoly<NT> & BiPoly<NT>::operator*=( BiPoly<NT>& P); //Self-multiplication
```

A.7.2 Comparison and Arithmetic

```
bool operator==(const BiPoly<NT>& P, const BiPoly<NT>& Q);
    //Equality operator for BiPoly
BiPoly<NT> operator+(const BiPoly<NT>& P, const BiPoly<NT>& Q);
    //Addition operator for BiPoly
BiPoly<NT> operator-(const BiPoly<NT>& P, const BiPoly<NT>& Q);
    //Subtraction operator for BiPoly
BiPoly<NT> operator*(const BiPoly<NT>& P, const BiPoly<NT>& Q);
    //Multiplication operator for BiPoly
```

A.7.3 I/O

```
void dump(std::ostream & os, std::string msg = "");
void dump(std::string msg="");
```

These dump the `BiPoly` object to a file or standard output as a string.

A.7.4 Functions

We have the following methods to manipulate bivariate polynomials.

```
Polynomial<NT> yPolynomial(const NT & x); // Returns the univariate
    //polynomial obtained by evaluating the coeffecients at x.
Polynomial<Expr> yExprPolynomial(const Expr & x);
    // Expr version of yPolynomial.
Polynomial<BigFloat> yBFPolynomial(const BigFloat & x);
    // BF version of yPolynomial
Polynomial<NT> xPolynomial(const NT & y) ;
    // returns the polynomial (in X) when we substitute Y=y
int getYdegree() const; // returns the nominal degree in Y
int getXdegree(); // returns the nominal degree in X.
int getTrueYdegree();//returns the true Y-degree.
Expr eval(Expr x, Expr y);//Evaluate the polynomial at (x,y)
int expand(int n);
    // Expands the nominal y-degree to n;
    // Returns n if nominal y-degree is changed to n, else returns -2
int contract();
    // contract() gets rid of leading zero polynomials
    // and returns the new (true) y-degree; returns -2 if this is a no-op
BiPoly<NT> & mulXpoly( Polynomial<NT> & p);
    // Multiply by a polynomial in X
BiPoly<NT> & mulScalar( NT & c);
    //Multiply by a constant
BiPoly<NT> & mulYpower(int s);
    // mulYpower: Multiply by Y^i (COULD be a divide if i¡0)
BiPoly<NT> & divXpoly( Polynomial<NT> & p);
    // Divide by a polynomial in X.
    // We replace the coeffX[i] by the pseudoQuotient(coeffX[i], P)
BiPoly<NT> pseudoRemainderY (BiPoly<NT> & Q);
    //Using the standard definition of pseudRemainder operation.
    // –No optimization!
BiPoly<NT> & differentiateY(); //Partial Differentiation wrt Y
BiPoly<NT> & differentiateX(); //Partial Differentiation wrt X
BiPoly<NT> & differentiateXY(int m, int n);
    //m times wrt X and n times wrt Y
BiPoly<NT> & convertXpoly();
    //Represents the bivariate polynomial in (R[X])[Y] as a member
    //of (R[Y])[X]. This is needed to calculate resultants w.r.t. X.
bool setCoeff(int i, Polynomial<NT> p);
    //Set the ith Coeffecient to the polynomial passed as a parameter
void reverse();// reverse the coefficients of the bi-poly
Polynomial<NT> replaceYwithX();
BiPoly<NT>& pow(unsigned int n); //Binary-power operator
BiPoly<NT> getbipoly(string s);
    //Returns a Bipoly corresponding to s, which is supposed to
    //contain as place-holders the chars 'x' and 'y'.
```

There are other useful friend functions for `BiPoly` class:

```
bool zeroPinY(BiPoly<NT> & P);
     //checks whether a Bi-polynomial is a zero Polynomial
BiPoly<NT> gcd( BiPoly<NT>& P ,BiPoly<NT>& Q);
     // This gcd is based upon the subresultant PRS to avoid
     // exponential coeffecient growth and gcd computations, both of which
     // are expensive since the coefficients are polynomials
Polynomial<NT> resY( BiPoly<NT>& P ,BiPoly<NT>& Q);
     // Resultant of Bi-Polys P and Q w.r.t. Y.
     // So the resultant is a polynomial in X
BiPoly<NT> resX( BiPoly<NT>& P ,BiPoly<NT>& Q);
     // Resultant of Bi-Polys P and Q w.r.t. X.
     // So the resultant is a polynomial in Y
     // We first convert P, Q to polynomials in X. Then
     // call resY and then turn it back into a polynomial in Y
```

We now come to the derived class `Curve`. All the methods provided for bivariate polynomials are available for curves as well, but there are two additional functions:

```
int verticalIntersections(const BigFloat & x, BFVecInterval & vI,
          int aprec=0);
     // The list vecI is passed an isolating intervals for y's such
     // that (x,y) lies on the curve.
     // If aprec is non-zero (!), the intervals have with < 2^{-aprec}.
     // Returns -2 if curve equation does not depend on Y,
     // -1 if infinitely many roots at x,
     // 0 if no roots at x,
     // 1 otherwise
int plot( BigFloat eps=0.1, BigFloat xmin=-1.0,
          BigFloat ymin=-1.0, BigFloat xmax=1.0, BigFloat ymax=1.0, int fileNo=1);
     // Gives the points on the curve at resolution "eps". Currently,
     // eps is viewed as delta-x step size.
     // The display is done in the rectangle [xmin, ymin, xmax, ymax].
     // The output is written into a file in the format specified
     // by our drawcurve function (see COREPATH/ext/graphics).
     // Heuristic: the open polygonal lines end when number of roots
     // changes.
```

B APPENDIX: Sample Program

The following is a simple program from O'Rourke's book to compute the Delaunay triangulation for n points. The program 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 input points that are (exactly) co-circular. This highly degenerate set of input points is expected to cause problems at Level 1 accuracy.

```
---------------------------------------------------------------------
#define CORE_LEVEL 3     // Change "3" to "1" if you want Level 1 accuracy
#include "CORE/CORE.h"
main() {                 // Adapted from O'Rourke's Book
  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;                   /* true if m above (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 is 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);
}
---------------------------------------------------------------------
```

You can compile this program at Levels 3 or Level 1. At Level 3 accuracy, our program will correctly detects all $\binom{n}{3}$ triples; at Level 1 accuracy, it is expected 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.

C APPENDIX: Brief History

Version 1.1 (Dec 1998) The initial implementation by Karamcheti, Li, Pechtchanski and Yap [8] was based on the `Real/Expr` package, designed by Dubé and Yap (circa 1993) and rewritten by Ouchi [13]. Details about the underlying algorithms (especially in `BigFloat`) and their error analysis may be found in the Ouchi's thesis [13].

Version 1.2 (Sep 1999) Incorporates the BFMS root bound and other techniques to give significant speedup to the system.

Version 1.3 (Sep 2000) Two improvements (new root bounds and faster big number packages based on `LiDIA/CLN`) gave significant general speedup. More examples, including a hypergeometric function package and a randomized geometric theorem prover.

Version 1.4 (Sep 2001) Introduced a floating point filter based on the BFS filter, incremental square root computation, improved precision-sensitive evaluation algorithms, better numerical I/O support. Our big integer and big rational packages are now based on `GMP`, away from `LiDIA/CLN`.

Version 1.5 (Aug 2002) Improvements in speed from better root bounds (k-ary bounds), CGAL compatibility changes, file I/O for large mathematical constants (BigInt, BigFloat, BigRat), improved hypergeometric package.

Version 1.6 (June 2003) The introduction of real algebraic numbers (a first among such systems). CORE is now issued under the the Q PUBLIC LICENSE (QPL), concurrent with its being distributed with CGAL under commercial licenses by Geometry Factory, the CGAL commercial spin-off. Incorporated Polynomial and Sturm classes into Core Library.

Version 1.7 (Aug 2004) Introduced algebraic curves and bivariate polynomials. An interactive version of Core Library called "InCore" is available. Beginning basic graphic capability for display of curves. Restructuring of number classes (Expr, BigFloat, etc) to have common reference counting and rep facilities.

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, R. Fleischer, K. Mehlhorn, and S. Schirra. A strong and easily computable separation bound for arithmetic expressi ons involving radicals. *Algorithmica*, 27:87–99, 2000.

[4] C. Burnikel, S. Funke, and M. Seel. Exact geometric computation using cascading. *International J. Comp. Geometry and Applications*, 11(3):245–266, 2001. Special Issue.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] C. Li, S. Pion, and C. Yap. Recent progress in exact geometric computation. *Journal of Logic and Algebraic Programming*, 2004. To appear. Special issue on "Practical Development of Exact Real Number Computation".

[11] C. Li and C. Yap. A new constructive root bound for algebraic expressions. In *Proc. 12th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 496–505. ACM and SIAM, Jan. 2001.

[12] J. O'Rourke. *Computational Geometry in C*. Cambridge University Press, second edition edition, 1998.

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

[14] S. Pion and C. Yap. Constructive root bound method for k-ary rational input numbers. In *19th ACM Symp. on Comp.Geometry*, pages 256–263, San Diego, California., 2003.

[15] S. Schirra. Robustness and precision issues in geometric computation. Report MPI-I-98-1-004, Max-Planck-Institut für Informatik, Saarbrücken, Germany, Jan 1998. To appear in *Handbook on Computational Geometry*, edited by J.R. Sack and J. Urrutia.

[16] 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.

[17] D. Tulone, C. Yap, and C. Li. Randomized zero testing of radical expressions and elementary geometry theorem proving. In *International Workshop on Automated Deduction in Geometry (ADG'00), Zurich, Switzerland*, Sept. 2000. Preprint: `ftp://cs.nyu.edu/pub/local/yap/exact/`.

[18] 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`.

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

[20] C. K. Yap. *Fundamental Problems in Algorithmic Algebra*. Oxford University Press, 2000. A version is available at URL `ftp:/Preliminary/cs.nyu.edu/pub/local/yap/algebra-bk`.

[21] 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. Expanded from 1997 version.

[22] 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.