REALLIB 3 MANUAL

BRANIMIR LAMBOV

ABSTRACT. abstract.

1. INTRODUCTION

RealLib3 is a library for exact real number computations. It represents real numbers via descriptions that allow infinite precision computations.

Theoretically, exact real number computations are only possible if the real numbers are represented as functions that can give approximations to the real number with arbitrarily good precision. As a consequence of that, theorists usually represent functions on real numbers as functionals that operate on the functions that represent real numbers. This approach is safe and easy to use theoretically, but does not work well when it is actually implemented, because of the significant overheads that handling function representations requires.

In contrast to this approach, the RealLib3 library uses a theory that treats functions on real numbers as simpler objects, operating on approximations of the arguments and yielding approximations to the outputs. This allows much more efficient, although possibly more complicated, implementations for functions, which can be run very close to the computer hardware and can lead to computing speeds in the order of magnitude of hardware floating point, something that had never been achieved before for exact real number computations.

This manual describes the library from a user's perspective: the interfaces it provides for real numbers, the interfaces used to define efficient real functions, and the methods used to link between the two levels. The manual also discusses the current state and some future improvements to the library. None of these improvements is expected to change the interface, meaning that programs written using this manual will work with the current version and continue to work with future versions, making full use of the improved performance that they will provide.

The theoretical background to this library is discussed in a separate paper.

2. CURRENT STATE OF THE LIBRARY

The library has recently undergone a serious redesign to allow for the implementation of very fast machine precision computations. With this change the library has reached Version 3.

The Version 3 interface is now defined and is described in this text. The implementation of the library is currently under heavy development to improve the implementations of

1991 *Mathematics Subject Classification.* *.
Key words and phrases. *.
*.

machine precision functions and to speed up other aspects, but the interface is not likely to change in the near future.

The current version cannot be fully verified to be accurate because of some dependencies on external functions. We are working on handling this issue, which is described in more detail in Section 6.

3. THE REAL NUMBERS INTERFACE

The class Real is the main class in the system. It contains the description of a real number and can be used to extract properties of this real number, as well as to apply operations to it.

A real number can be constructed in several different ways:

- from a constant entered as a double, taken to be exact, e.g. Real(1.5) would define the number 1.5 exactly, while Real(0.1) would define the real number which matches the double precision representation of 0.1, which is different from 0.1 by about $5.55 \cdot 10^{-18}$;
- from a string, e.g. Real("0.1") would define the number 0.1 exactly;
- via operations applied to Real arguments, e.g. Real(1)/Real(3) will define $\frac{1}{3}$ exactly;
- via functions applied to Real arguments, e.g. sqrt(Real(2)) for $\sqrt{2}$;
- some real constants are predefined (Pi and Ln2), others can be created using the interfaces described in Section 4;
- from oracle functions, i.e. functions that can return arbitrarily good decimal approximations to a number.

The use the term "constructing a real number" instead of "assigning a value to a real variable" here is intentional. Real numbers are represented via structures that describe the computation through which they were computed. Every time a new operation is performed, a new object is being created that describes the operation and contains references to the objects that were arguments for the operation. In this sense, updating the value of a variable of type Real usually does not mean that the objects that described the earlier value are destroyed. The latter only happens if the variable had not been used in other operations.

The main purpose of the real numbers layer is to be able to extract properties of a number that is constructed. These can be:

- a best[1] approximation to the value in double precision, different from the actual value by at most 1 ulp;
- a decimal representation of the real number, which is at most different by 1 in the least significant position[2];
- strict comparisons, which will loop indefinitely if the numbers are equal. It is impossible to test two real numbers for equality, thus the system does not provide the operators $==$, $<=$ or $>=$.

Let us take a closer look at how the system behaves with a few simple program fragments:

[manual/fragments.cpp]

```
001 #include <iostream>
```

[1] A correctly rounded approximation according to the IEEE-754 standard is not possible to achieve because of the undecidability of the equality test for real numbers

[2] correct rounding is impossible to achieve, this may mean all digits are incorrect: in different scenarios the system may print either of 1.000 and 0.999 for the real number 0.9995

```
002 #include <iomanip>
003 #include "Real.h"
004 using namespace std;
005 using namespace RealLib;
006
007 void main() {
008    InitializeRealLib();
009    Real a, b(4);
010    b = Pi / b;
011    a = sin(b);
012    cout << "sin(Pi/4) is " << a << endl;
...
031    FinalizeRealLib();
032    return 0;
033 }
```

Lines 1 to 5 include the necessary headers, and lines 8 and 31 perform the initialization and finalization of the library.

Line 9 declares two variables of type Real. One of them is initialized to the default value (which is 0), the other to the constant 4. Line 10 updates b with the result of the division of the constant Pi and b. If we look at this with more detail, the system constructs a new real number which is the result of the application of the operation to its arguments (in this case $\frac{\pi}{4}$), keeping a reference to the constant Pi and the object that b was assigned to, after which it removes the link between b and the constant 4 object. Since this object is still needed as argument to the division operator, it will not be deleted.

Line 11 updates a with the result ($\sin\frac{\pi}{4} = \frac{\sqrt{2}}{2}$) of the function sin applied to the argument b. That is, the system creates a new object that describes the operation "sin applied to the object that b links to", and links a to it. The link is not to the variable b, but to the object that it linked, i.e. the real number it held, $\frac{\pi}{4}$. The previous value of a (the zero it was implicitly initialized to) was not used anywhere, thus it is not needed any more and will be deleted by the system.

Line 12 prints out the result. In order to do this, the system will run through the objects linked to a and generate an approximation that can be printed on the screen. This approximation will be cached for possible later use. This fragment prints

```
sin(Pi/4) is 0.707107
```

The next fragment demonstrates a little bit more complicated case:

```
...
014    Real c(a * 2 / sqrt(Real(2)));
015    for (int i=0; i<1000; ++i)
016       c = sin(c);
017    cout << "sin(sin(...(1)...)) (1000 times) is " << c << endl;
...
```

Line 14 declares c to be 1 using the fact that we know about the value of b. In a usual floating point environment such a roundabout definition would certainly introduce a significant accuracy loss. This is not the case here: performance may suffer, but the accuracy will still be full.

Lines 15 and 16 run a loop in which sin is consecutively applied to c 1000 times. In reality, this means that every run through Line 16 a new object is created that describes an application of the function sin to the previous object. In the end of this loop, c will point to a structure that looks like this:

$$\mathsf{c} \longrightarrow \mathsf{sin} \longrightarrow \mathsf{sin} \longrightarrow \ldots \longrightarrow \mathsf{operator}/ \longrightarrow \begin{cases} \mathsf{operator}* \longrightarrow \begin{cases} \mathsf{sin} \longrightarrow \mathsf{operator}/ \longrightarrow \begin{cases} \mathsf{Pi} \\ 4 \end{cases} \\ 2 \end{cases} \\ \mathsf{sqrt} \longrightarrow 2 \end{cases}$$

At Line 17, this structure is traversed (using the cached value of $\sin\frac{\pi}{4}$) to construct an approximation to the number which is good enough to be displayed:

```
sin(sin(...(1)...)) (1000 times) is 0.054593
```

The necessity for the structure comes from the possibility that the approximation may be not good enough to ensure that the requested number of digits be displayed. This will happen when we run through the next line, which requests 120 decimal digits of accuracy, well beyond the accuracy of the default initial precision of the system:

```
...
019    cout << " or " << scientific << setprecision(120) << c <<
endl;
...
```

Here the system will check if the cached value of c (computed in Line 17) is good enough for the new request. Since it isn't, this will cause the system to clear all cached values and run through the description c points to again in order to get better precision. The new starting precision may again be insufficient, which will trigger another iteration and this process will continue until the precision is enough or the maximum precision[3] is reached[4]. At the end, we see

```
 or 5.45929715101851774031536067239940104010575369617410161283520
70820777006979734673648566210735738887296043552263141817177780e-2
```

Line 19, along with the following lines, also demonstrates some of the formatting manipulators that the system's output functions respect.

```
...
021    cout << "the double representation of 0.1 is "
022       << fixed << noshowpoint << Real(0.1) << endl;
023    cout << "and its distance from 0.1 is "
024       << showpos << scientific << uppercase
025       << showpoint << Real(0.1) - Real("0.1") << endl;
...
```

The output of this fragment:

```
the double representation of 0.1 is 0.10000000000000000555111 5123
1257827021181583404541015625000000000000000000000000000000000000
000000000000000000000000000 and its distance from 0.1 is
+5.5511151231257827021181583404541 01
5625000000000000000000000000000000000000000000000000000000000000
000000000000000000000E-18
```

The other ways to extract properties of the real number are shown in the next few lines, where you can also see that double cannot even print its own value correctly:

```
...
026    cout << "double(0.1) < Real(\"0.1\"): "
027       << boolalpha << (Real(0.1) < Real("0.1")) << endl;
028    cout << "0.1 converted to double is "
```

---

[3]specified as an argument of InitializeRealLib(), see Subsection 5.1 for details

[4]which will cause an exception that can be caught by the user, see Subsection 5.1

```
029       << noshowpos << fixed << Real("0.1").AsDouble() << endl;
...
```

The output of this fragment:

```
double(0.1) < Real("0.1"): false
0.1 converted to double is 0.10000000000000005551115123125782702
11815800000000000000000000000000000000000000000000000000000000000
0000000000000000000
```

The following is a complete program written using only the real numbers interface to compute the sum of the harmonic series for 1000000 members. We have added some code to track the time it takes for every stage of the computation to complete. [`manual/harm.cpp`]

```
001 #include <iostream>
002 #include <iomanip>
003 #include <ctime>
004 #include "Real.h"
005
006 using namespace RealLib;
007 using namespace std;
008
009 #define LENGTH 1000000
010 #define MACROTOSTRING2(x) # x
011 #define MACROTOSTRING(x) MACROTOSTRING2(x)
012
013
014 Real Harmonic(const int mcnt)
015 {
016    Real one(1);
017    Real s;   // initialized to 0
018    for (int i=1; i<=mcnt; ++i) {
019       s += one/i;
020    }
021    return s;
022 }
023
024
025
026 int main()
027 {
028    clock_t starttime, endtime;
029
030    cout << "Computing the sum for " MACROTOSTRING(LENGTH) " members"
<< endl;
031
032    starttime = clock();
033    InitializeRealLib();
034    {
035       Real h(Harmonic(LENGTH));
036       endtime = clock();
037       cout << "construction time: " <<
038          double(endtime - starttime) / CLOCKS_PER_SEC << endl;
039
040       for (int n=10; n<500; n*=6) {
041          starttime = clock();
```

```
042          cout << unitbuf << fixed << setprecision(n);
043          cout << n <<" digits: \t" << h << endl;
044          endtime = clock();
045          cout << fixed << setprecision(6);
046          cout << "prec: " << GetCurrentPrecision() << " time elapsed:
" <<
047             double(endtime - starttime) / CLOCKS_PER_SEC <<
endl;
048       }
049       starttime = clock();
050    }
051    FinalizeRealLib();
052    endtime = clock();
053    cout << "destruction time: " <<
054       double(endtime - starttime) / CLOCKS_PER_SEC << endl;
055
056
057    return 0;
058 }
059
```

The clock() pairs surround the regions of code that do the interesting work. First we measure the time it takes to initialize the system and to construct the representation of h, the real number that represents the 1000000-member sum, then we consecutively time the extraction of representations with different number of decimal digits, and finally we measure the time needed to destroy the representation when it goes out of scope. The extraction is the place where the actual computation is performed, and the accuracies are chosen to require a single new iteration through the representation.

The result of the execution[5] of this program:

```
Computing the sum for 1000000 members
construction time: 3.845
9 digits:   14.3927267
prec: 4 time elapsed: 24.806
63 digits:  14.392726722865723631381127493188587676644800013744311
1653418433
prec: 9 time elapsed: 20.049
441 digits:     14.392726722865723631381127493188587676644800013744
443116534184330458129585075179950035682981759472191007083599521360
798129002641641025869300946330062005496116666391427558432665415721
219730782928819514121133122033133043828972712951321469882948594551
104755079764875032609612144070163003538369161116798217677091946824
417163326372248859422898758102848526351896600065279756908532436952
455327427912589432571939166589739628482163578405644674173550690758
6
prec: 48 time elapsed: 23.463
destruction time: 0.942
```

As you can see from the timings, the computation time did not change much when we went from the initial precision to 9 32-bit words, and when we quintupled the precision for the 441-digit approximation. This shows that something is wrong, i.e. that too much time is being spent somewhere else, not in performing the actual computation.

[5] machine used: Pentium M 1.8GHz, 2MB Level-2 cache, 512 MB DDR-333 main memory, GCC 3.3.3 in Cygwin environment

When the class Real is used to perform computations, the real numbers are represented as functions and the system acts as a type-2 machine to transform functions into functions. This is the traditional approach for real number computations, which suffers from serious efficiency problems.

To deal with these efficiency problems, the system offers the real functions interface where the user can create functions that work on the more efficient approximations level. In the next section we will see how this program can be changed to make use of this and obtain a dramatic performance improvement.

4. The real functions interface

The function Harmonic is the only part of the program we need to change. In particular, Lines 13-23 change to the following: [manual/harmfun.cpp]

```
013 template <class TYPE>
014 TYPE Harmonic(const long prec, const long mcnt)
015 {
016    TYPE one(1);
017    TYPE s;   // initialized to 0
018    for (int i=1; i<=mcnt; ++i) {
019       s += one/i;
020    }
021    return s;
022 }
023 CreateIntRealFunction(Harmonic);
```

When we execute this program, we see identical approximations, but the timings differ significantly:

```
Computing the sum for 1000000 members
construction time: 0
9 digits:   14.3927267
prec: 4 time elapsed: 0.17
63 digits:  14.392726722865723631381127493188587676644800013744311
1653418433
prec: 9 time elapsed: 6.079
441 digits:     14.392726722865723631381127493188587676644800013744
443116534184330458129585075179950035682981759472191007083599521360
798129002641641025869300946330062005496116666391427558432665415721
219730782928819514121133122033133043828972712951321469882948594551
104755079764875032609612144070163003538369161116798217677091946824
171633263722488594228987581028485263518966000652797569085324369524
5532742791258943257193916658973962848216357840564467417355069075860
prec: 48 time elapsed: 9.363
destruction time: 0
```

You can see that the time it takes to obtain the more accurate approximations was reduced at least in half. The more important difference is the change in the time it takes to compute the least accurate approximation: instead of more than 20 seconds, this computation now takes 170 milliseconds! This is significantly less than even the time the original program needs to construct or destroy the number's representation.

In fact, when the machine precision is sufficient for the computation, this modified program runs at a speed in the order of magnitude of an identical computation implemented in double. While it is not easy to evaluate the accuracy of the latter, the results obtained using

RealLib3 can be trusted, and the cost of obtaining them is not as dramatically different as it is with earlier exact real number systems[6].

The functions layer in RealLib3 avoids the complex processing needed to construct term descriptions of the real numbers by using the user's code as the description. In order to do this, the user needs to extract the bulk of the computation into a function that is written in a way appropriate for the system.

Let us look with more detail at the changes we needed to apply to use the real functions interface:

- the function Harmonic is now a function template;
- its signature is changed to accommodate one extra argument (precision) which is not used in the function;
- the CreateIntRealFunction macro is used to create a mapping on the level of real numbers linked to that function.

To make fully efficient use of specialized precision code, the system requires the user's functions to be defined as function templates. The template gets instantiated for two[7] different approximation classes that share the same interface, described in detail in Subsection 5.3. Defining the user's function as a template allows the system to make full use of the compiler's optimization abilities, especially in the very fast machine precision stage of the computation.

To be able to compute transcendental functions or numbers, the user functions are supplied with an additional argument that specifies the precision the system expects from the user's function. The programmer can use this parameter to decide e.g. the length of the series that approximates a number. A use of this will be shown later– our current example ignores this parameter as it computes the needed value exactly[8].

Our function is a function that takes one integer argument and returns a real number. In the system such functions are called "nullary real functions" (as they do not take a real argument), and are declared using this signature:

```
template < class TYPE >
TYPE name(unsigned int precision, UserInt userarg)
```

And finally, a function written for the function interface is not very useful unless it has a representation on the level of the real numbers, where it can be applied and its results can be examined. This representation is created by the linking macro, in this case CreateIntRealFunction, which maps a nullary real function into a function of this signature:

```
Real name(UserInt arg)
```

In the rest of the program, this function is used in the same manner as the original version written on the real numbers layer.

Let us now look at the implementation of a transcendental function that takes one real argument: [manual/exp.cpp]

```
001 #include <iostream>
002 #include <iomanip>
003 #include "Real.h"
004
```

[6] an updated version of this manual will contain a comparison

[7] in the current version, the number may change in the future

[8] one may safely assume that the built-in operations on the function level compute exactly; although in reality they only provide approximations, the error in these approximations is accounted for and the user can safely ignore it

```
005 using namespace RealLib;
006 using namespace std;
007
008 template <class TYPE>
009 TYPE myexp(const TYPE &arg)
010 {
011    unsigned int prec = arg.GetPrecision();
012    TYPE s(0.0);
013    TYPE m(1.0);
014
015    if (abs(arg) > 1.0) throw DomainException("myexp");
016    if (!(abs(arg) < 1.0)) throw PrecisionException("myexp");
017
018    for (unsigned i=1; i<=prec; ++i) {
019       s += m;
020       m = m*arg/i;
021    }
022    return s.AddError(m*3);
023 }
024 CreateUnaryRealFunction(myexp)
025
026 int main()
027 {
028    InitializeRealLib();
029    {
030       Real a(myexp(Real(0.5)));
031       Real b(exp(Real(1)));
032
033       cout << fixed << setprecision(10);
034       cout << "a(myexp(0.5)) =\t" << a << endl;
035       cout << "a*a =\t\t" << a*a << endl;
036       cout << "b(exp(1)) =\t" << b << endl;
037       cout << "a*a/b =\t\t" << setprecision(300) << showpoint
038          << a*a/b << endl;
039    }
040    cout << "precision used: " << FinalizeRealLib();
041
042    return 0;
043 }
044
```

The template function myexp defined on Lines 8-23 is the definition of a function that computes the exponent of numbers in the range $(-1, 1)$.

At Line 11 we extract the precision, given as the number of correct 32-bit words, that we need to achieve from the argument. For this first attempts, we will not try to match this request, but will only make sure that our approximation's accuracy increases as prec increases.

Line 15 makes sure we reject arguments that are clearly outside the domain of our function. If the argument is on the boundary, we cannot recognize that and reject it, but we can wait until the argument is inside our domain. Since our domain is an open interval, to any real number in it there will always be an approximating interval that fits inside the domain (when the precision is high enough). Line 16 makes sure we do not compute the result until we have such an approximation.

After this, Lines 18-21 compute the Maclaurin expansion of the exponent function with prec members, and Line 22 adds an error that covers the remainder sum, provided that the argument is within $(-1, 1)$, which lines 15 and 16 have ensured.

Line 24 creates a mapping for the function on the level of real numbers that is in the form

```
Real myexp(const Real& arg);
```

When the main function creates a variable of type Real initialized to myexp(0.5) on Line 30, the system constructs an object that describes an application of the template function myexp to an object that describes the real number 0.5. There is no need for the creation of objects describing the steps of the computation of the Maclaurin sequence: this description is already present as the code of the template function myexp.

When the value of a is requested on Line 34, the library creates an approximation to the argument and runs an instantiation of the template function myexp to it. It then examines the error in the approximation the function returned. If the error is small enough to allow the display of a 10-digit representation, this representation is printed.

If it is not, the system will create a better approximation to the argument and will pass it to myexp (possibly a different instantiation of it), which will give a better approximation to the end result, because the parameter prec has increased and the argument is bounded in absolute value by 1. Eventually an approximation that is good enough will be obtained and will be printed.

Here is the result of the execution of this program:

```
a(myexp(0.5)) = 1.6487212707
a*a =       2.7182818285
b(exp(1)) = 2.7182818285
a*a/b =     1.000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000
precision used: 272
```

Let us extend this example a bit. First, since the remainder bound 3m (which is an upper bound for me) on Line 22 is also valid for the real numbers 1 and -1, we would like to expand our domain to $[-1, 1]$ to include them. Also, we want the best possible performance, thus we will try to satisfy the precision request. To do so, we will evaluate so many members of the sequence that m$e \leq 2^{-32\mathsf{prec}}$. We have two choices:

- pre-compute a bound for the number of terms needed and run the series this number of times. This will give the best performance for lower precisions, but may require expensive extra computations for higher precisions;
- run the series until the condition is satisfied. This implies extra computations for each iteration, but may be significantly faster in higher precisions if the argument is smaller.

We will use both, choosing by examining the value of the argument. We will use the inequality $\left(\frac{n}{3}\right)^n < n! < \left(\frac{n}{2}\right)^n$ (for $n \geq 6$) to get a the following upper bound for the number of terms:

$$n(\ln n - ln3) > 32\mathsf{prec} \ln 2 + 1,$$

or very roughly $n \geq 23$prec terms will be sufficient.

The modified function follows (the rest of the program does not change):
[manual/expimpr.cpp]

```
008 template <class TYPE>
009 TYPE myexp(const TYPE &a)
010 {
011    unsigned int prec = a.GetPrecision();
012    TYPE s(0.0);
013    TYPE m(1.0);
014
015    TYPE arg(a.TruncateTo(-1.0, 1.0, "myexp"));
016
017    if (abs(arg).weak_le(0.75)) {
018       TYPE err = (TYPE(1) >> (32 * prec)) / 3;
019       for (unsigned i=1; abs(m) > err; ++i) {
020          s += m;
021          m = m*arg/i;
022       }
023    } else {
024       if (prec < 6) prec = 6;
025       unsigned int pc = prec * 23;
026       for (unsigned i=1; i<=pc; ++i) {
027          s += m;
028          m = m*arg/i;
029       }
030    }
031    return s.AddError(abs(m)*3);
032 }
033 CreateUnaryRealFunction(myexp)
```

This code fragment demonstrates the following:

- how we can satisfy the precision request by choosing a suitable length for the approximating sequence;
- how we can use the member function TruncateTo to define functions that have a closed domain;
- how we can use a weak comparison to choose between different control paths that lead to the same result but may have different performance.

Here is the result of running this example file:

```
a(myexp(0.5)) = 1.6487212707
a*a =       2.7182818285
b(exp(1)) = 2.7182818285
a*a/b =     1.000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
precision used: 34
```

The precision that is needed to compute the ratio is now very close to optimal, and thus the computation itself runs much faster.

5. Library reference

All of the library's functions, constants and operators live in the RealLib namespace and are included using the header `Real.h`.

5.1. **Initialization and finalization, exceptions.** The system must be initialized prior to use and finalized afterwards.

```
void InitializeRealLib(
   unsigned precStart = MachineEstimatePrecision,
   unsigned precMax = 100000,
   unsigned numEstAtStart = 1000);
```

Initializes the library. The starting precision is specified in the first argument. If nothing is specified, the system will start with machine precision floating point approximations.

precMax specifies the maximum working precision. If the system cannot decide a property after reaching this maximum precision, it will abort with a PrecisionException that can be caught by the user.

numEstAtStart specifies the amount of space the library will reserve for approximations at initialization. If more space is needed, the library will increase the storage appropriately. In such a case, specifying a higher numEstAtStart may save a few memory reallocations.

```
#define MachineEstimatePrecision 4
```

A value that is used to indicate interval arithmetic with double precision. This is the default initial precision in the system.

```
unsigned FinalizeRealLib();
```

Finalizes the library. All cached approximations are destroyed, the memory allocated is freed and the current precision is returned.

```
unsigned ResetRealLib(
   unsigned precStart);
```

Resets the library, setting a new working precision. Useful when one computation is complete, and another must start from the initial precision to avoid working with unnecessarily high precision.

```
unsigned GetCurrentPrecision();
```

Returns the current precision in 32-bit words. A value of MachineEstimatePrecision means that the system is currently working with interval arithmetic with double precision.

```
class RealLibException : public std::exception {
   char m_what[128];
public:
   RealLibException(const char *what = NULL) throw();
   virtual const char *what() const throw();
};
```

Base class for the exception classes used in the system. The constructor takes one string argument specifying a text message for the place the exception originated, and the what member function returns a pointer to this string. The following two classes share this interface:

```
class PrecisionException : public RealLibException {
public:
   PrecisionException(const char *what = NULL) throw();
};
```

Raised by functions when the current approximation was not sufficient to know anything about the resulting approximation. Indicates the system must start a new iteration with higher precision.

If this exception is passed on to the user, this means that the maximum precision specified in InitializeRealLib has been reached and was not sufficient to extract the wanted property.

```
class DomainException : public RealLibException {
public:
   DomainException(const char *what = NULL) throw();
};
```

Raised by functions to indicate the argument is certainly outside the domain of the function.

5.2. **Class** Real**.** The real numbers interface is realized by the class Real.

5.2.1. *Construction, destruction, assignment.* The following constructors are available to the user:

```
Real::Real(const double src = 0);
```

Constructs a Real from a value in double precision, also acting as a default constructor for the value 0. The argument is taken to be exact, i.e. for example the real number constructed by Real(0.1) is not the same as the real number 0.1, but rather to what the compiler thinks is its the closest double to 0.1.

Use for constants when you are certain they are correctly represented in double precision (e.g. integers up to 2^{53} are all correctly representable). If in doubt, use the string initialization form.

```
Real::Real(const char *src);
```

Constructs a Real from a decimal string. The string is taken to be exact. Real("0.1") will define the correct value, but Real("3.1415926") or even a 1000000-digit decimal representation will not define the number π (and neither would Real(M_PI)).

If a rational number is not correctly representable as a decimal, use division of real numbers to define it. For example, $\frac{1}{3}$ should be constructed as Real(1)/3[9].

```
typedef const char* (*OracleFunction) (unsigned precision);
Real::Real(OracleFunction oracle);
```

This constructor can be used to construct real numbers by an user-supplied function that can give decimal approximations of a real number for any precision (i.e. an "oracle" function).

This constructor is provided to give the system the possibility to work with numbers supplied from an external source (e.g. a human or a random number generator), which can possibly be non-computable. Other uses of the constructor are also possible[10], but those are covered by the more convenient and efficient real functions layer.

The oracle function is being called for increasing values of the precision parameter, which specifies the function should try to present an approximation with this precision in 32-bit words, or roughly 10 decimal digits per unit of precision.

[9] it suffices to have one Real in the division to force division of real numbers. Division of a real number by an integer is faster and will be correct as long as the divisor can be represented as a 32-bit integer

[10] e.g. defining real numbers by limitation

The function may choose to provide more or less correct values, and the system takes them to be correct up to a unit in the last place (*ulp*) of the string the function returns. The requirement for the function is to represent *one* real number, i.e. to make sure that the intervals $[x - ulp, x + ulp]$ overlap for all values x that the function returns with *ulp* defined by the length of the decimal representation after the decimal point, and that *ulp* decreases unboundedly, i.e. the length of the decimal representations increases when precision increases.

The behavior of the system is not defined if the oracle function does not satisfy this requirement.

```
Real::Real(const Real &src);
```
Copy constructor, used to make a copy of a real value.

```
Real::~Real();
```
Destructor, called when a variable goes out of scope or a pointer is deleted.

```
Real& Real::operator = (const Real &rhs);
```
Assignment operator. Updates a real variable with a new value.

5.2.2. *Operators.*

```
Real Real::operator - () const;
```
Negation.

```
Real operator + (const Real &lhs, const Real &rhs);
Real operator - (const Real &lhs, const Real &rhs);
Real operator * (const Real &lhs, const Real &rhs);
```
Addition, substraction and multiplication.

```
Real operator / (const Real &lhs, const Real &rhs);
```
Division. Not defined for rhs $== 0$.

```
Real operator * (const Real &lhs, int rhs);
Real operator * (int lhs, const Real &rhs);
Real operator / (const Real &lhs, int rhs);
Real operator / (int lhs, const Real &rhs);
```
Faster versions of multiplication and division by integer. The division by integer will cause a DomainException if rhs is zero, and the division of integer by real is not defined for rhs $== 0$.

```
Real::Real& operator += (const Real &rhs);
Real::Real& operator -= (const Real &rhs);
Real::Real& operator *= (const Real &rhs);
Real::Real& operator /= (const Real &rhs);

Real::Real& operator *= (int rhs);
Real::Real& operator /= (int rhs);
```
Updating versions of the operators. All of them are just shorthand forms for the operator followed by assignment.

5.2.3. *Built-in constants and functions.*

```
extern const Real Pi;
```
The constant π.

```
extern const Real Ln2;
```
The constant $\ln 2$.

```
Real recip(const Real &arg);
```
Reciprocal, $\frac{1}{\mathsf{arg}}$. Not defined for $\mathsf{arg} == 0$.

```
Real abs(const Real &arg);
```
Absolute value, $|\mathsf{arg}|$. The result is a non-negative real number.

```
Real sqrt(const Real &arg);
```
Square root, $\sqrt{\mathsf{arg}}$. Not defined for negative arguments. The result is a non-negative real number.

```
Real rsqrt(const Real &arg);
```
Reciprocal square root, $\frac{1}{\sqrt{\mathsf{arg}}}$. Not defined for 0 and negative arguments. The result is a positive real number.

```
Real log(const Real &arg);
```
Natural logarithm, $\ln \mathsf{arg}$. Not defined for 0 and negative arguments.

```
Real exp(const Real &arg);
```
Exponent, e^{arg}. The result is a positive real number.

```
Real sin(const Real &arg);
```
Sine, $\sin \mathsf{arg}$. The result is in the range $[-1, 1]$.

```
Real cos(const Real &arg);
```
Cosine, $\cos \mathsf{arg}$. The result is in the range $[-1, 1]$.

```
Real tan(const Real &arg);
```
Tangent, $\tan \mathsf{arg} = \frac{\sin \mathsf{arg}}{\cos \mathsf{arg}}$. Not defined for $\mathsf{arg} == (2k + 1)\frac{\pi}{2}$ for an integer k.

```
Real asin(const Real &arg);
```
Arcsine, $\arcsin \mathsf{arg}$. Defined only for $\mathsf{arg} \in [-1, 1]$. The result is in the range $[-\frac{\pi}{2}, \frac{\pi}{2}]$.

```
Real acos(const Real &arg);
```
Arccosine, $\arccos \mathsf{arg}$. Defined only for $\mathsf{arg} \in [-1, 1]$. The result is in the range $[0, \pi]$.

```
Real atan(const Real &arg);
```
Arctangent, $\arctan \mathsf{arg}$. The result is in the range $(-\frac{\pi}{2}, \frac{\pi}{2})$.

```
Real atan2(const Real &y, const Real &x);
```
Arctangent of $\frac{y}{x}$, using the signs of both arguments to compute the angle. Can be used to compute the argument of a complex number, or the angle between the origin of the plain and the point with coordinates (x, y).

The function is not defined[11] for the ray $y == 0, x <= 0$. The result is in the range $(-\pi, \pi)$.

[11] the original mathematical function has a discontinuity at these points, which makes it non-computable. Excluding the points of discontinuity from the domain of the function makes it computable

5.2.4. *Comparison and truncation.*

```
bool Real::IsNegative() const;
```

Returns true if the number is negative and false if the number is positive. Not defined[12] if the number is zero.

```
bool Real::IsPositive() const;
```

Returns false if the number is negative and true if the number is positive. Not defined if the number is zero.

```
bool Real::IsNonZero() const;
```

Returns true if the argument is non-zero. Not defined if the number is zero. Can be used to cause the computation to be performed at a certain spot in the user's code or to make sure that numbers that are close to zero do not get printed as "probable zero".

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

Comparison between two real numbers. Shorthand forms for substraction followed by resp. IsNegative, IsPositive and IsNonZero. Not defined if the two numbers are equal.

The equality test is undecidable, i.e. no function can exist that can always give a positive answer when two numbers are equal and not complete or give a negative answer if they are not. The system does not provide this test, nor the non-strict inequalities which, without the possibility to recognize equality, coincide with their strict counterparts.

```
const Real& Real::ForceNonZero() const;
```

Uses IsNonZero to force computation until the number can be separated from zero. Will loop indefinitely or cause an exception if the number is an actual zero. Used to make sure that numbers that are close to zero do not get printed as "probable zero".

5.2.5. *Conversion to other types.*

```
double Real::AsDouble() const;
```

Conversion to double. Returns a double precision number which is at most 1 ulp away from the real number. The value returned is not always an IEEE-correct approximation to the number[13].

Numbers that are below the exponent range of double are converted to 0 and numbers that are above the range are mapped to ∞ with the appropriate sign.

```
char* Real::AsDecimal(char *buffer, unsigned len) const;
```

Creates a decimal approximation of the real number that fits in len characters, which is at most 1 ulp away from the number, and returns a pointer to buffer. The representation is not always correctly rounded[14], nor the first len digits of the number's infinite decimal representation[15].

---

[12] again, the discontinuity in the discrete function is avoided by removing the points of discontinuity from its domain

[13] the map between real values and their IEEE-correct representations is discontinuous thus non-computable. Our conversion is computable, but it is not a function in the sense of real analysis, because it can map different representations of the same real number into different double values

[14] for the same reasons as above

[15] e.g. the number $1.01000\ldots$ can be converted to the decimal representation 1.0 as well as to 1.1, same reasons as above

Scientific notation is used if the number does not fit in the space provided. This also includes the cases where the number is smaller than the smallest number that can be written in fixed notation in the space provided (i.e. $10^{-\mathsf{len}}$). Because this can lead to infinite re-iteration if the number is 0, the system will return "probable zero" if the number is smaller than $10^{-2\mathsf{len}}$ and is not distinguishable from zero at the current working precision. If this behavior is not desired (i.e. is the user knows the number is not zero), use ForceNonZero to make sure the computation is performed until number is separated from zero, for example

```
printf("%s", x.ForceNonZero().AsDecimal(buf, len));
```

The buffer must have enough space to accommodate len characters, and len must be at least 10.

5.2.6. *Stream input and output.*

```
std::istream& operator >>(std::istream &in, Real &r);
```

Stream input. Reads a string from the input stream and creates a real number from it. The string can be of arbitrary (finite) length and is taken to be an exact decimal representation of the number.

```
std::ostream& operator <<(std::ostream &out, const Real& r);
```

Stream output. Sends a decimal approximation to r that is at most 1 ulp away to the output stream. The representation need not be correctly rounded nor the beginning of the number's infinite decimal representation[16].

If the number is smaller than $10^{-\mathsf{prec}}$ (for fixed notation) or $10^{-2\mathsf{prec}}$ (for default and scientific notation), where prec is the output precision (set through setprecision below), and at the current working precision cannot be distinguished from 0, the system will print "probable zero" to avoid infinite loops if a real zero needs to be printed. If this behavior is not wanted, use ForceNonZero to force computation until separation from zero can be ensured, for example:

```
cout << x.ForceNonZero();
```

The output is influenced by the following stream manipulators:

- setprecision(x): sets the precision in decimal digits. The digits of the exponent are not counted. The actual count of the digits depends on the notation
- fixed: sets fixed notation, the number is printed with the specified digits of precision after the decimal point, regardless how long the string needs to be to ensure that
- scientific: scientific (exponent) notation will be used, where the mantissa will have one non-zero decimal before the decimal point and the specified digits of precision after the decimal point
- **(notation not set):** (default) prints in fixed notation if the number can fit in the specified digits of precision. If it cannot, prints in scientific notation. In both cases, the total number of digits printed (excluding the exponent) will be as specified
- showpoint: the trailing zeros and decimal point will be displayed
- noshowpoint: (default) the trailing zeros and decimal point will not be shown
- showpos: positive numbers will have a leading '+'
- noshowpos: (default) no leading '+' will be shown
- setw(x): sets the field width (only applies for the next thing printed)

[16] for the same reasons as in AsDecimal above

setfill(x): sets the character for filling the field
left: the number will be left-justified in the field
right: the number will be right-justified (default)
internal: the sign will remain to the left, while the number will be flushed to the right
lowercase: (default) use lowercase 'e' for the exponent
uppercase: use uppercase 'E' for the exponent.

5.3. **The functions interface: Class** Estimate**.** This interface is meant to be used for the implementation of user-defined functions or user-defined real constants. To make the functions useable on the numbers level, they have to be defined according to one of the signatures specified in Subsection 5.4.

On this level the functions are instantiated for different types which share the same interface. The interface is that of the class Estimate, which we are describing in this section.

The functions on this level work with approximations to the number. They return approximations to the result of the application of the function or operator. The nullary functions (constants or functions on integers) take an additional precision argument which indicates how precise the approximation should try to be. This argument is implicit in the functions that take real arguments and can be recovered through Estimate::GetPrecision() on one of the real arguments. The precision indicates the approximation should try to be correct for the specified number of 32-bit words of relative precision. The built-in functions try to achieve this[17].

The correctness requirement for the user functions is that they produce overlapping intervals, and that the error in the produced approximations decreases as the precision increases. In order to achieve the best possible efficiency in the system, the user should try to produce approximations according to the precision specification.

From a user's point of view, functions that satisfy the correctness requirement (this includes the built-in functions), can be assumed to compute real numbers exactly. The errors they produce are handled automatically. The only errors that the user needs to address are the result of finitely approximating an infinite sequence. An upper bound for such an error should be explicitly specified via a call to Estimate::AddError.

Some of the operations on this level are called "weak" operations. This is to signify that they extract properties of the current approximations, which are not necessarily properties of the real number being approximated. Still, the weak operations can be useful to choose a control path when both control paths return the same end result, e.g. a weak_lt is used to switch between a control path that computes $\sin$ in the range $[0, \frac{\pi}{2}]$ and one that computes it for the range $[\frac{\pi}{2}, \pi]$. Both can compute the approximation even if the number is slightly outside that range, but use different algorithms and will yield results with different precision.

5.3.1. *Conversion from other types.*

```
Estimate::Estimate(double v = 1.0);
Estimate::Estimate(const char *val);
```

Conversion from double or a decimal string. The argument is taken to be correct, and the resulting Estimate will be an approximation to that number (exact in the case of double and with decreasing error value as the precision increases in the case of a decimal string).

[17] but lose a few bits due to rounding errors of the basic operators

5.3.2. *Error manipulation.*

```
Estimate Estimate::GetError() const;
Estimate& Estimate::SetError(const Estimate &err);
Estimate& Estimate::AddError(const Estimate &err);
```

Get, set and add to the error value in the approximation. The most often used function is AddError, which can be used to add the error resulting from imperfectly approximating a transcendental number via a finite part of an infinite sequence (see Section 4 for example).

GetError returns an exact approximation (i.e. one having error 0), and the other functions use a value which is not smaller than the largest one within the argument interval.

Example: if a represents the real number interval $[0, 1]$, a.GetError() would return the interval $[0.5, 0.5]$, a.SetError(1) would make a representation of the interval $[-0.5, 1.5]$, and a.AddError(a) would return the interval $[-1, 2]$.

```
i32 Estimate::GetRelativeError() const;
```

Returns a lower bound for the number of digits in the mantissa of the approximation that are within 1-ulp of the real number. Used by the system in the process of obtaining approximations for conversion to double or decimal string.

```
u32 Estimate::GetPrecision() const;
Estimate& Estimate::SetPrecision(u32 prec);
```

Get and set the current working precision of the number in 32-bit words. This value controls how precise functions working on this argument should try to be (see Section 4 for example).

5.3.3. *Interval truncation.* The truncation functions that follow are useful to compute functions that have domains with closed ends. They remove the specified parts of the approximating interval, leaving only the part that is a valid argument for the function. They raise an exception if the argument lies entirely in the unwanted part of the real line.

```
Estimate TruncateNegative(const char *origin = "Truncate") const;
```

Truncates the negative part of the interval. For example, $[-1, 2]$ will be truncated to $[0, 2]$, $[-2, -1]$ will raise a DomainException(origin), and $[1, 2]$ will remain unchanged.

```
Estimate TruncateBelow(double l,
   const char *origin = "Truncate") const;
Estimate TruncateBelow(const Estimate &l,
   const char *origin = "Truncate") const;
```

Truncates the parts below a certain lower limit. The error in l will appear in the result, e.g. if a is $[0, 2]$ and l is $[0.75, 1.25]$, a.TruncateBelow(l) will be $[0.75, 2.25]$. To make sure the limit is exact, use a double value for l.

```
Estimate TruncateAbove(double u,
   const char *origin = "Truncate") const;
Estimate TruncateAbove(const Estimate &u,
   const char *origin = "Truncate") const;
```

Truncates the parts above a certain upper limit. Use a double constant for u to avoid introducing extra error in the result.

```
Estimate TruncateTo(double l, double u,
   const char *origin = "Truncate") const;
Estimate TruncateTo(const Estimate &l, const Estimate &u,
```

```
    const char *origin = "Truncate") const;
```
Truncates the input interval to fit into the specified interval.

5.3.4. *Operators.*
```
Estimate operator - (const Estimate &arg);

Estimate operator + (const Estimate &lhs, const Estimate &rhs);
Estimate operator - (const Estimate &lhs, const Estimate &rhs);
Estimate operator * (const Estimate &lhs, const Estimate &rhs);
Estimate operator / (const Estimate &lhs, const Estimate &rhs);

Estimate operator * (const Estimate &lhs, i32 rhs);
Estimate operator * (i32 lhs, const Estimate &rhs);
Estimate operator / (const Estimate &lhs, i32 rhs);
Estimate operator / (i32 lhs, const Estimate &rhs);
```
Negation, addition, substraction, multiplication and division, the last two also in a faster form with one integer argument.

Division needs the argument to be non-zero, thus it will raise a PrecisionException if the right-hand side is an interval that contains zero.

The division by integer form will raise a DomainException if the integer divisor is zero.

```
Estimate Estimate::operator << (i32 howmuch) const;
Estimate Estimate::operator >> (i32 howmuch) const;
```
Binary shift by howmuch bits, i.e. a $<< n = a2^n$ and a $>> n = a2^{-n}$. Very fast multiplication by a power of two.

5.3.5. *Built-in constants and functions.*
```
Estimate pi(unsigned int prec);
```
The constant π. Returns an approximation which has close to prec correct 32-bit words.

```
Estimate ln2(unsigned int prec);
```
The constant $\ln 2$.

```
Estimate recip(const Estimate &arg);
```
Reciprocal, $\frac{1}{\mathsf{arg}}$. Raises a PrecisionException if the argument is an interval that contains 0.

```
Estimate abs(const Estimate &arg);
```
Absolute value, $|\mathsf{arg}|$. The resulting interval may contain zero, but no negative real number.

```
Estimate sqrt(const Estimate &arg);
```
Square root, $\sqrt{\mathsf{arg}}$. If the argument interval does not intersect the domain of the function, raises a DomainException. Otherwise, the interval is truncated only to its valid part, i.e. the intersection of the domain of the function and the argument interval.

```
Estimate rsqrt(const Estimate &arg);
```
Reciprocal square root, $\frac{1}{\sqrt{\mathsf{arg}}}$. Raises a DomainException for argument intervals that do not intersect the domain, and a PrecisionException for arguments that contain zero.

```
Estimate log(const Estimate &arg);
```

Natural logarithm, $\ln$ arg. Raises a DomainException for argument intervals that do not intersect the domain, and a PrecisionException for arguments that contain zero.

```
Estimate exp(const Estimate &arg);
```
Exponent, e^{arg}.

```
Estimate sin(const Estimate &arg);
```
Sine, $\sin$ arg. The resulting interval may contain numbers outside the range $[-1, 1]$.

```
Estimate cos(const Estimate &arg);
```
Cosine, $\cos$ arg. The resulting interval may contain numbers outside the range $[-1, 1]$.

```
Estimate tan(const Estimate &arg);
```
Tangent, $\tan \text{arg} = \frac{\sin \text{arg}}{\cos \text{arg}}$. Raises a PrecisionException for arguments that contain $(2k + 1)\frac{\pi}{2}$ for an integer k.

```
Estimate asin(const Estimate &arg);
```
Arcsine, $\arcsin$ arg. Raises a DomainException for intervals that do not intersect $[-1, 1]$ and truncates the argument interval to fit the domain.

```
Estimate acos(const Estimate &arg);
```
Arccosine, $\arccos$ arg. Raises a DomainException for intervals that do not intersect $[-1, 1]$ and truncates the argument interval to fit the domain.

```
Estimate atan(const Estimate &arg);
```
Arctangent, $\arctan$ arg.

```
Estimate atan2(const Estimate &y, const Estimate &x);
```
Arctangent of $\frac{y}{x}$, using the signs of both arguments to compute the angle. Can be used to compute the argument of a complex number, or the angle between the origin of the plain and the point with coordinates (x, y).

Raises a PrecisionException if the arguments contain points with $y == 0$ and $x <= 0$.

5.3.6. *Strong comparisons.* Strong comparisons return true if the comparison is true for any real number in the interval, i.e. if the approximated real number satisfies the inequality.

```
bool Estimate::IsPositive() const;
```
Returns true if this $\subseteq (0, +\infty)$, and false otherwise.

```
bool Estimate::IsNegative() const;
```
Returns true if this $\subseteq (-\infty, 0)$, and false otherwise.

```
bool Estimate::IsNonZero() const;
```
Returns false if $0 \in$ this, and true otherwise.

```
bool Estimate::operator < (const Estimate &rhs) const;
bool Estimate::operator > (const Estimate &rhs) const;
bool Estimate::operator != (const Estimate &rhs) const;
```
Comparison operators, shorthand forms for substraction followed by resp. IsNegative, IsPositive, IsNonZero.

If a value of true is returned, the real numbers satisfy the inequality, but a value of false means that either they do not satisfy it, or that this cannot be shown from the current approximation.

5.3.7. *Weak discrete functions.* Weak functions work with the current approximation, more specifically, with the center of the approximating interval. They ignore the error information and may give information that would be wrong for the approximated real number.

They are all discrete functions that would be non-computable on the real numbers level, but have a clearly defined meaning for its current approximation.

They are to be used to differentiate between control paths that compute the same thing via different algorithms (possibly with different accuracy), or for debugging or progress reports.

```
bool Estimate::weak_IsPositive() const;
bool Estimate::weak_IsNegative() const;
bool Estimate::weak_IsNonZero() const;

bool Estimate::weak_lt(const Estimate &rhs) const;
bool Estimate::weak_eq(const Estimate &rhs) const;
bool Estimate::weak_gt(const Estimate &rhs) const;

bool Estimate::weak_le(const Estimate &rhs) const;
bool Estimate::weak_ne(const Estimate &rhs) const;
bool Estimate::weak_ge(const Estimate &rhs) const;
```

Weak comparisons: positivity test, negativity test, non-zero[18], less-than, equal, greater-than, less-than-or-equal, not-equal, greater-than-or-equal.

```
Estimate Estimate::weak_round() const;
```

Rounding. Returns an exact Estimate, which is the integer closest to the center of the interval. Can be used to compute the value of periodic functions.

```
i32 Estimate::weak_normalize() const;
```

Normalization: returns an integer such that $a >> a.\mathrm{Normalize}()$ is within $[0.5, 1)$. Used to compute logarithms.

```
double Estimate::weak_AsDouble() const;
```

Returns a double approximation to the center of the interval, at most $\frac{1}{2}$ ulp away from it, correctly rounded according to the IEEE-754 specification[19].

```
char *Estimate::weak_AsDecimal(char *buffer, u32 buflen) const;
```

Returns a decimal representation of the center of the interval which fits in buflen characters and is at most $\frac{1}{2}$ ulp away. Fixed notation is normally used, switched to scientific if the number cannot fit in the space provided (buflen has to be 10 characters minimum).

```
std::ostream& operator <<(std::ostream &os, const Estimate &e);
```

[18] some implementations define this to be always true, because they do not allow a zero to be the center of an approximation

[19] this specification is not correctly implemented in the current version

Stream output, prints a number which is at most $\frac{1}{2}$ ulp away from the center of the interval. Influenced by the same set of format manipulators as the stream output for real numbers.

5.4. **Macros linking the functions and numbers interfaces.** In order to use functions defined on the functions layer on Real arguments, the user must create a mapping of the function using one of the linking macros.

There are linking macros for the following types of functions:

- nullary functions, i.e. real constants
- nullary functions with int, i.e. functions taking one integer argument and returning a real number
- unary real functions
- unary functions with int, i.e. functions taking one real and one integer arguments
- binary real functions
- binary functions with int, i.e. functions taking two real and one integer arguments
- real functions on arrays
- real functions on arrays with an additional integer argument

The user's functions have to be function templates parameterized by the type of the approximation object. The approximation objects for which they will be instantiated all share the interface of Estimate, described in Subsection 5.3.

5.4.1. *Nullary functions (constants).* Defined using this form:

```
template <class TYPE>
TYPE name(unsigned int prec);
```

The macro CreateNullaryRealFunction(*name*) maps such a function to the following real function:

```
Real name ();
```

Alternatively, the macro CreateRealConstant(*const_name*, *fun_name*) maps the function *fun_name* into the constant *const_name* defined as

```
const Real const_name;
```

For example, CreateRealConstant(Pi, pi) is used in the code of the system to define the built-in constant Pi from the function pi.

5.4.2. *Nullary functions with integer argument.* Defined using this form:

```
template <class TYPE>
TYPE name(unsigned int prec, UserInt uint);
```

The macro CreateIntRealFunction(*name*) maps such a function to the following real function:

```
Real name (UserInt uint);
```

5.4.3. *Unary functions.* Defined using this form:

```
template <class TYPE>
TYPE name(const TYPE &arg);
```

The macro CreateUnaryRealFunction(*name*) maps such a function to the following real function:

```
Real name (const Real& arg);
```

For example, CreateUnaryRealFunction(sin) is used in the source code of the system to define the real number function sin from the function-layer sin.

5.4.4. *Unary functions with integer argument.* Defined using this form:

```
template <class TYPE>
TYPE name(const TYPE& arg, UserInt uint);
```

The macro CreateUnaryAndIntRealFunction(*name*) maps such a function to the following real function:

```
Real name (const Real& arg, UserInt uint);
```

5.4.5. *Binary functions.* Defined using this form:

```
template <class TYPE>
TYPE name(const TYPE &lhs, const TYPE &rhs);
```

The macro CreateBinaryRealFunction(*name*) maps such a function to the following real function:

```
Real name (const Real& lhs, const Real& rhs);
```

For example, CreateBinaryRealFunction(atan2) is used in the source of the system to define the real number function atan2 from the function-layer atan2.

5.4.6. *Binary functions with integer argument.* Defined using this form:

```
template <class TYPE>
TYPE name(const TYPE& lhs, const TYPE &rhs, UserInt uint);
```

The macro CreateBinaryAndIntRealFunction(*name*) maps such a function to the following real function:

```
Real name (const Real& lhs, const Real& rhs, UserInt uint);
```

5.4.7. *Array functions.* Defined using this form:

```
template <class TYPE, class ARRAY>
TYPE name(ARRAY &arg);
```

where ARRAY is a type with the following interface:

```
template <class TYPE>
class ArrayInterface {
public:
    long size();
    TYPE& operator[] (long index);
};
```

(indexed array elements can be retrieved or updated; no pointer or prev/next operations are supported and no bounds checking is performed)

The macro CreateArrayRealFunction(*name*) maps such a function into the triple:

```
void name(Real *ptr, long count);
void name(std::valarray<Real> &arr);
void name(std::vector<Real> &arr);
```

For an example of the use of this, see `examples/linear.cpp`.

5.4.8. *Array functions with integer argument.* Defined using this form:

```
template <class TYPE>
TYPE name(ARRAY &arg, UserInt int);
```

where ARRAY is as above.

The macro CreateArrayAndIntRealFunction(*name*) maps such a function into the triple:

```
void name(Real *ptr, long count, UserInt uint);
void name(std::valarray<Real> &arr, UserInt uint);
void name(std::vector<Real> &arr, UserInt uint);
```

6. FUTURE PLANS

The current version of the machine precision layer depends on stdlib implementations of the elementary functions sin, cos, log, exp, asin, ftoa, whose accuracy cannot be guaranteed. We are working towards removing all dependencies on the accuracy of external functions. This is the current most important priority.

In parallel to this, we will be completing machine-specific optimizations of the machine precision layer starting with Intel SSE2 implementations of the layer. The basic operations are currently in place and offer outstanding performance, but the elementary functions specified in the last paragraph require custom versions which will be implemented as part of the removal of the external accuracy dependencies.

As a latter objective we aim at further improving the topmost layer, Real, where real numbers are represented as terms. Memory handling, which constitutes a fair portion of the processing time on that layer, should be considerably improved.

Another further objective would be the introduction of a layer between pure machine precision and the arbitrary precision which uses double-double or similar arithmetic. This will be done without any modifications to the library's interface.

*

E-mail address: *