

RealLib: An Efficient Implementation of Exact Real Arithmetic

Branimir Lambov

BRICS, University of Aarhus

IT Parken, 8200 Aarhus N

Denmark;

currently at

FB Mathematik, AG 1

Technical University Darmstadt

Schlossgartenstrasse 7

64289 Darmstadt

Germany

email: barnie@brics.dk

Received 5 January 2006

This paper is an introduction to the *RealLib* package for exact real number computations. The library provides certified accuracy, but tries to achieve this at performance close to the performance of hardware floating point for problems that do not require higher precision. The paper gives the motivation and features of the design of the library and compares it to other packages for exact real arithmetic.

1. Introduction

In developing a library[†] for exact real number computations, our main objective has been to create a tool which is useful in a wide variety of contexts. For this, the library needs to be able to replace standard floating point with a minimum of extra programming, stay clear of bad programming practices to decrease the possibility of the library introducing bugs in existing code and to facilitate the understanding of the mechanisms of the library, and, maybe most importantly, the library must be able to reach performance comparable to that of hardware floating point in the cases where it is actually possible to compute meaningful results using low precision.

Combining these requirements presents a very difficult task. The performance requirement clearly excludes higher-level approaches that manipulate the real numbers

[†] *RealLib*, available at <http://www.brics.dk/~barnie/RealLib/>

as entities (more information about the performance problems of this approach will be given below; examples of packages implementing this include *ICReals* (Edalat), *XRC* (Briggs), *Few Digits* (O'Connor) and others). On the other hand, a minimal user effort in replacing floating point arithmetic with real number arithmetic appears to require such an approach. It may seem that a user interface that pretends to act on complete objects while in reality it does something else is an answer to the problem, but such an approach appears to require unorthodox programming and non-standard behavior of the user's programs as demonstrated by Norbert Müller's *iRRAM* (Müller 2001).

Our exact real number implementation takes an approach which provides two levels of access to real numbers aiming to satisfy conflicting parts of the requirements. One of the levels operates on real numbers as complete entities and is able to be easily integrated in existing code, but has poor performance when a multitude of simple operations is to be performed. For the latter case, our design provides an interface which operates on the level of approximations, is free of performance issues, but where the control flow may become more complicated as the objects operated on do not represent complete reals. Both levels have well defined behavior and do not use any non-standard programming.

The two levels interact with each other, more specifically, the layer that operates on complete entities (to be called the "top", "numbers" or "direct" layer in the rest of the paper) can use objects written in the layer that operates on approximations (to be called the "bottom", "function" or "approximations" layer). With the help of this mechanism, a user may encapsulate large computations in a function on the bottom layer, and use it once or repeatedly at the direct layer. In an extreme case (which may happen quite often in practice and is very similar to the mode of operation of *iRRAM*), all of a computation may be implemented on the approximations layer as what we call a "nullary" function (i.e. a real number constant), using the top layer, for example, to only create a real number linking to that object and print out an approximation to it.

The rest of this paper will explain the obstacles in designing efficient real number systems along with suggestions for solving them, explain the way these solutions are used in *RealLib*, and compare the library to the best available alternatives.

2. Performance problems in real number arithmetic

The approach to real number computability most often attempted in practice is some kind of type-2 approach using the TTE model of Weihrauch (Weihrauch 2000) or an equivalent formulation using type-2 computability. The approach looks pretty easy to implement, especially if one uses a functional language such as *SML*, *ocaml* and others. Unfortunately, the type-2 character of the approach presents some barriers to efficiently implementing the ideas. Some of them can be avoided, but some of the problems are so serious that a clean type-2 implementation free of severe performance problems is impossible to achieve.

Indeed, clean[‡] type-2 implementations have proven to be extremely slow, being at least a

[‡] The term will be properly defined below.

magnitude slower[§] than mixed solutions such as *RealLib* and *iRRAM*[¶], even in cases where higher precisions are used and the bookkeeping overheads are low. What are the reasons for this poor performance?

Since every real number used in the course of a computation needs to be present as a function object, a type-2 approach requires the library to use representations of the way in which a real number was obtained, which has to be implemented in some structure that stores a term representation of every object used in the computation.

This leads to a wide variety of performance problems which will be discussed below.

2.1. Problems with common subexpressions

The type-2 implementation of a binary operation computes an approximation based on approximations to its operands at higher precision. Every time a binary operation is called, it asks its first argument to compute an approximation, then it asks its second argument to compute an approximation and uses them to compute an approximation to the output.

If both arguments are the same object, a straightforward implementation would require that the same computation (probably with different accuracy) be carried out twice. Moreover, a naive implementation would build a representation of the computation as a tree and will contain the argument twice.

Consider this simple example using a hypothetical class *Real* that implements type-2 real number arithmetic:

```
Real a(1);
for (i=0;i<100;++i)
    a = a+a;
```

If the class *Real* uses a tree to store the term representations of the real numbers, at the end of this fragment *a* will refer to a tree with $2^{101} - 1$ nodes.

A better implementation of the class should prefer to use a directed acyclic graph (dag) in the representation of real numbers to allow multiple references to the same node and avoid the unnecessary growth of the size of the structure.

Even after switching to dags to represent expression trees, the better implementation may run into the problem of complexity explosion, because the straightforward implementation would still require the computation of an approximation to each node twice for every addition, or the computation of 2^{100} approximations to 1 and $2^{100} - 1$ additions on approximations. Naturally, such an explosion cannot be permitted.

The problem can be solved by careful caching of the approximations, making sure that the approximations to a node are asked with the same precision, and that the cached approximations are deleted when they are no longer needed. This is highly non-trivial to do in a clean type-2 implementation.

[§] This statement is based on the results of section 5 and the “Many Digits” friendly competition, <http://www.cs.ru.nl/~milad/manydigits/>, in comparing the exact real number packages *RealLib* and *iRRAM* to *XRC*, *Few Digits* and *BigNum*.

[¶] There is a common misconception that *iRRAM* is an implementation of the TTE model of real number computability. This is not true, as the reader will see later in subsection 6.3.

2.2. Unnecessary precision growth

A type-2 implementation of a function requires higher accuracy from its arguments in order to be sure about the accuracy of its result. This may lead to unnecessary big precision growth.

Consider this code fragment:

```
Real a(0);
for (int i=1;i<=1000000;++i)
    a = a+Real(i);
```

Addition requires approximations to its arguments of accuracy at least one bit higher. Thus in this code fragment the accuracy needed will grow by at least one bit in every iteration. Thus, if we want to compute an approximation to a which is 32-bit accurate, we will end up performing many of the additions at very high precision, some of them with 1000000 or more bits of precision. Clearly this is not acceptable from a performance point of view, since in reality less than 60 bits suffice to carry out the whole computation.

One may wonder if it is possible to balance the precision request so that a binary operation requires less precision from its more difficult arguments. We do not see how this could be implemented in practice.

The approach described so far is sometimes called “top-down evaluation” to indicate that a node controls the accuracy of its siblings. The name also hints at the possibility of a “bottom-up” approach where the accuracy of the siblings determines the accuracy in a node, an approach which does not suffer from the problem at hand.

The bottom-up approach evaluates everything at an (almost) constant precision starting from the leaves of the tree, keeping track of the errors that are introduced at every step. In certain cases it may turn out that the end result of the computation on a tree is not accurate enough. In such a case the whole computation is restarted at higher precision until the process leads to a result which is accurate enough (i.e. in which the accumulated error is sufficiently small). The reiterations do not add much to the complexity of the process, since e.g. by doubling the precision for each new iteration we can be certain that time taken by the evaluation is dominated by the last iteration.

In this case the implementations of functions use interval arithmetic and approximations to transcendental functions that accept arbitrary precision requests. It may seem that such an approach requires at least twice the amount of work, when one considers that an interval is represented as a pair of bounds and every function must be evaluated at least twice. This is not required, as one can use what is sometimes called “simplified interval arithmetic”, where the functions are evaluated only at the center and the size of the resulting interval is estimated from the size of the input interval.

It becomes very unclear whether we can still call this a type-2 approach after this modification. We will leave this question open, and will keep using the term “clean type-2 implementation” to mean a type-2 implementation that uses the normal top-down evaluation, and will reserve “type-2” as an indication of an approach where every real number is available as a function representation and a real number function has full access to that representation.

2.3. Bookkeeping necessary

Even with the modifications discussed above, a type-2 implementation requires a new object to be created every time an operation is performed. If we assume that this object takes at least 4 bytes of memory (to store the 32-bit address of an argument), the absolute maximal number of operations that can be carried out in a computation on a 32-bit machine is about 1 billion. Since this number of operations can be easily reached e.g. in linear algebra, this restriction is clearly unacceptable.

Moreover, every operation must allocate memory, which is known to be a painfully slow process in modern computers, significantly slower than the time it takes to actually perform the operation at low precisions. If we want to be able to reach the performance level of hardware floating point, this is clearly unacceptable.

Therefore one must consider an approach that does not store the history of a computation at every step. This is not possible in a type-2 approach, since the basic property of real functions in a type-2 model is the access to full function representations of their real arguments.

The functions must operate on approximations and be modular in the approximations in the sense that the representation of the composition of two functions must be achieved by composing the representations of the two functions. In order to also achieve full soundness and completeness, i.e. equivalence to the popular notions of computable analysis, a type-1 theoretical approach must be used. Combined with the requirement for bottom-up evaluation, an interval approach such as Grzegorzczuk's interval definitions (Grzegorzczuk 1957) of real function computability must be used.

2.4. Loss of locality information

Even if we disregard the time and space problems of creating a term representation, which may be negligible at higher precisions, a type-2 representation of a computation lacks the locality information that a programmer or compiler gives in an implementation of a function.

Let us take a look again at a slight modification of the last example:

```
Real a(0);
for (int i=1;i<=1000000;++i)
    a = Real(i)+a;
```

Depending on the actual way that the process of evaluating the generated expression is implemented, this computation may require the storage of up to 999999 temporary values. The evaluation of a tree is recursive, and a naive implementation may evaluate the left hand side argument to additions first before diving into the recursion to compute the right hand side argument, which will lead to computing and storing a value for every iteration. If the precision is high, this process will waste huge amounts of memory. Since this will also destroy all data locality and trash all cache levels, the performance in such a case is very far from acceptable.

While it is easy to find a solution to the problem for this concrete example, finding an approach which chooses the better pattern of evaluating arguments does not seem trivial. A heuristic must be used which will most probably fail in a large number of cases.

To solve the problem completely, we would prefer to execute the operations with the

order and locality that the programmer and compiler give. For this example, it would mean that only a single object, a , needs to be stored to complete the evaluation.

To achieve this, one again must use a bottom-up evaluation scheme combined with a modularity requirement on the level of approximations, so that the loop above can start with an approximation to a , update it at every iteration, and finish with an approximation to the end result. If that end result is not accurate enough, we should be able to rerun the code to achieve a better approximation.

2.5. Impossible compiler optimizations

Whenever a computational tree determines the order of computations, it is impossible to perform any compiler optimizations on the code.

In the case of very low precisions the overhead of a function call and inability to execute computations in parallel may result in two-digit factors of slowdown. We certainly do not want this if we want to be able to achieve performance close to hardware floating point when the problem to be solved is easy.

To achieve the best possible performance, the programmer must be able to write function code that is compiled specifically for fast instantiations of interval arithmetic and additionally for slow but generic multiple precision floating point. In *C++* this is achieved using template functions.

The library must permit this tool to be used to achieve optimal performance.

3. Design of the *RealLib* library

The *RealLib* library provides two interfaces to the user. One of them behaves like a type-2 implementation of exact real arithmetic, while the other operates on approximations of numbers but still implements exact real arithmetic by being an implementation of the model of Partial Approximation Representations for computable analysis (Lambov 2005). The top-level interface uses a bottom-up approach to evaluating real numbers and is thus not a clean type-2 implementation. It does this to avoid the first two performance problems discussed in the previous section.

Our bottom-up approach uses fixed precision for all the computations on an expression dag, which not only lets us avoid the precision growth associated with top-down evaluation, but also makes it easier to avoid the complexity explosion by ensuring that once an approximation to an object is computed, this approximation will have the exact precision needed for all other references to the same object. By counting their number and the number of requests already made, we can maintain efficient caching of all temporary results and delete them exactly when they are no longer needed. Additionally, in an attempt to minimize the effect of the loss of locality information, the library also tries to optimize the order of evaluation of the siblings of nodes with multiple arguments. The top layer of the library behaves like a built-in type for floating point arithmetic with the exception of the aspects that have been proven to be undecidable: because of the non-computability of the equality test, all comparisons of real numbers are undefined for equal arguments; in consequence to this, the library does not provide non-strict versions of comparisons (\leq and \geq) because they coincide with their strict counterparts; additionally, the rounding in the library is always faithful (up to a distance of 1 unit in

the last place) as correct rounding (such as rounding to a double precision number according to the IEEE-754 rules) is not achievable.

The top layer of the library is free of the first two performance issues, but suffers badly from the rest of the problems discussed in the previous section. In particular, there is a need to maintain full information about the way in which a number was constructed.

This is very inefficient when simple operations are involved.

To solve this problem, the library provides a bottom layer, the level of approximations where the objects on which the user's code operates are interval approximations. The objective of the bottom layer is to represent big chunks of a computation tree as single nodes by providing a function that encapsulates a lower-level version of the code of the same operation. This is made possible by the bottom-up approach for evaluation and the theoretical model, which also gives us representations of all computable real functions on the level of approximations in a modular way. The latter allows the approximations layer to look as if it is working on complete real numbers.

The bottom layer does not store any additional information about the temporary real numbers other than their approximations and performs the computations exactly in the order and locality given by the programmer and compiler. Moreover, the bottom layer functions are always defined as template functions, so that a very fast double precision step can be used for the first approximation.

This "function" layer is used to define functions that can be used directly on the numbers layer. For example, it is easy to define a new function computing the Riemann ζ of a complex number, and use this function repeatedly with both the top and bottom layer interface.

An actual computation starts when an object is created on the top layer and a request for a property of it is given (such as e.g. a 10-digit decimal approximation). The request triggers a recursive evaluation on the description of the number at a chosen precision, which in turn executes the bottom layer functions used in the definition. They may be executed more than once, since an end result may turn out to be insufficient to show the property, or the functions may be abruptly terminated by an exception requesting higher precision from a function used in their body (such as division when the current approximation of the divisor does not separate it from zero).

With the combination of the two layers we have a mixed implementation in which the results can be extracted via the more convenient type-2 interfaces, while the bulk of the computations can be carried on the much more efficient approximations level. We still have full descriptions of all temporary objects used on the numbers level, but they do not need to be as many since a single node can encompass a multitude of simple operations. In this case, the program code written on the function layer of the system becomes part of the description of the term used to obtain a real number.

In clean type-2 implementations of real arithmetic, the single nodes can only be representations of the functions that are built into the system. Even if the system allows it, adding a new function to the set is not a trivial task, since the approach requires careful control over the accuracy which is not automatically available. With *RealLib* in many cases adding a function to the set of objects working on the fast layer of approximations is achieved by simply changing a function's header and using a linking macro (as the examples in section 5 show).

A type-2 interface is the most convenient method of working with real numbers, but unfortunately it is not very efficient unless a huge library of predefined functions is

available. While we are not able to provide that library for *RealLib*, we have made it as easy as possible for the user to add new functions that run as close to the hardware as possible.

4. Limit computation and approximate comparisons

The results of applications of most interesting functions in analysis are given numerically as limits of computably converging sequences of computable numbers. One of the ways to define a computably converging sequence is by giving a sequence together with an estimation of the amount of error in all the approximations of the sequence. This corresponds to giving a partial approximation representation (see (Lambov 2005)) of the limit.

The method used in *RealLib* to define limits follows this idea. A function written on the level of approximation can add to the amount of uncertainty in an approximation to cover for uncalculated portions of the result. Every such function is given a parameter *prec* that specifies a precision; if a function needs to compute the limit of a sequence it needs to

- generate members of the sequence for different values of *prec*,
- indicate the distance within which the limit must be contained for every member of the sequence,
- make sure that for every target accuracy ε as *prec* grows there will be a point after which the distance is always smaller than ε .

For example, if one tries to compute the number e by evaluating its Taylor series expansion, one can choose to evaluate the first *prec* number of members and find a bound for the remainder sum which is to be added to the error in the approximation of the result. Since as *prec* grows this function gives improved approximations to the number, the third condition is also satisfied.

The same method is used in the implementations of real number functions that need to compute a limit. The only difference is that *prec* is no longer explicitly given as an argument to the function. In exact correspondence with the theoretical model, this parameter is recovered from the approximation to one of the real number arguments of the function.

Examples of defining a function that requires limitation are given in the library's manual^{||}.

Another operation which is a requirement for the completeness of a real number package, is the presence of approximate comparisons, i.e. a method of showing either $x < b$ or $a < x$ for an arbitrary x when $a < b$. This is not directly given in our library, but the approximations level of *RealLib* includes comparisons that only evaluate to true if the current approximation is sufficiently accurate to prove the fact. Using a combination of two such comparisons and forced reiteration if neither of them is true, one can easily achieve the approximate comparison operation.

In addition to this, the library provides “weak” operations which can be used to choose more efficient execution paths. A weak operation only evaluates properties of the center of an approximation and is not guaranteed to give consistent results in consecutive

^{||} available as part of the library or at the internet address <http://www.brics.dk/~barnie/RealLib/>

iterations through the same code. A *weak_round* operation, for example, can be used to reduce an argument to a periodic function to its primary domain. Although the same real number may round to different values in different iterations, this does not lead to problems as all possibilities will ultimately lead to the same result as a real number.

5. Examples and performance comparison

We will give a few sample programs written for the library, starting with two cases where it is known a-priorily that the computations cannot be correctly handled in machine precision, and finishing with a case which shows the strength of the library in dealing with the situations that more often appear in practice: where the need for high-precision computations may be suspected, but hardware floating point actually suffices.

The first example is a very simple demonstration of a feature all exact real number systems share: the possibility to request arbitrarily precise approximations to a number. In this case, we choose to display a 10000-digit approximation to the value of π :

```
001 #include <iostream>
002 #include <iomanip>
003 #include "Real.h"
004 using namespace std;
005 using namespace Reallib;
006
007 int main(int argc, char **argv) {
008     InitializeReallib();
009     {
010         cout << setprecision(10000) << Pi;
011     }
012     FinalizeReallib();
013     return 0;
014 }
```

The actual work of this code is done at Line 10, the rest of the file includes the appropriate headers, makes the definitions in the *std* and *Reallib* namespaces local, and takes care of the necessary initialization and finalization of the library. At Line 10, *Pi* is a predefined value for the library and represents the exact value of the real number π via a function that computes approximations to it for any given precision. To display the result with the number of digits specified by *setprecision*, the library will call this function, possibly more than once, to get an approximation accurate enough to display 10000 digits which are no further than a unit in the last place from the actual value.

We will not print the result here, instead we will measure the time^{††} it takes to

^{††} using a Pentium-M 1.8GHz and GCC 3.3 in Cygwin environment

complete this program and compare it to two other exact real number systems, *XRC* by Keith Briggs (Briggs) and *iRRAM* by Norbert Müller (Müller 2001):

<i>RealLib3</i>	<i>iRRAM</i> 2004_02	<i>XRC</i> 1.1
730 ms	230 ms	364 s

iRRAM has had the reputation of the fastest exact real number library, using highly optimized *GMP* (GMP) for the higher precisions that are required for this example. It does not fail it in this case, producing the approximation three times faster than *RealLib*, which still uses a portable custom multi-precision library written entirely in *C++*.

XRC, on the other hand, is too slow.

For the next example, we will use the logistic sequence example from (Müller 2001). We will compute the iteration $x_{i+1} = 3.75(1 - x_i)x_i$ with $x_0 = 0.5$ and print 6 digits of x_{100} , x_{1000} and x_{10000} . This time, we will use two different versions of the program. One that uses only *RealLib*'s mechanism for dealing with real numbers as entities, and one that uses *RealLib*'s mechanism for constructing functions that operate on the efficient approximations layer.

We will encapsulate the computation in the following function:

```

007 template <class TYPE>
008 TYPE Logistic(unsigned int prec, UserInt len)
009 {
010     TYPE s(0.5);
011     TYPE coeff(3.75);
012     TYPE one(1.0);
013     for (int i=1;i<=len;++i)
014         s = coeff * (one - s) * s;
015     return s;
016 }
017 CreateIntRealFunction(Logistic)

```

This is a template function that has an unused argument *prec*. This form, along with the declaration at Line 17, is required by the library for functions that work on the approximations layer of the library. This example does not use the argument *prec*, because we are not computing a limit of a sequence. Other than that, it's a pretty standard code for the iteration, and, being a template, it can also be used to try the direct implementation of the computation using the type *Real*, which is the basic type in the library for working with real numbers as entities. We will make use of this in the following main function (not much different from the one in the previous example):

```

019 int main(int argc, char **argv) {
020     InitializeRealLib();
021     {
022         cout << Logistic(1000) << endl;
023     }
024     FinalizeRealLib();
025     return 0;

```

026 }

Line 22 is the important one, which in this case calls the real number function object constructed at Line 17 from the template function *Logistic*. This object has a single argument, because it makes no sense to specify precisions for exact computations. In the table that follows this will be reflected in the column “*RealLib3*, function”. For the column “*RealLib3*, direct” we will use the same code with Line 22 changed to

```
022      cout << Logistic<Real>(0, 1000) << endl;
```

which runs a direct instantiation of the template function to the type *Real*, and for the column “double” we will use

```
022      cout << Logistic<double>(0, 1000) << endl;
```

Timing these^{‡‡} and corresponding code for the other two exact real number systems results in the following table:

iterations	<i>RealLib3</i> , function	<i>RealLib3</i> , direct	<i>iRRAM</i> 2004_02	<i>XRC</i> 1.1	<i>double</i>
100	1 ms	3 ms	625 ms	383 ms	0.6 μ s
1000	150 ms	188 ms	12 ms	143 s	6 μ s
10000	48 s	50 s	5.5 s	–	60 μ s

For unknown reasons *iRRAM* did not want to compute the 100 iterations as fast as we would expect, thus we suggest that the reader ignore the first value in *iRRAM*’s column^{§§}. *XRC* apparently was not expected to be used for heavily nested computations and its recursive evaluation mechanism failed for more than several thousand nested operations.

iRRAM is again the fastest, and *XRC* is disappointingly slow. All libraries compute the correct values in contrast to the double precision implementation, which runs really fast, but is completely wrong.

What is interesting in this example, is that the overhead of using the top-level interface of the library in comparison to the approximations interface, is clearly visible. It may be little or negligible at high precision (only 2 out of the 50 seconds required to compute the 10000 iterations), but it is quite a big portion of the time for a lower precision computation, and dominates the time in the simplest case, taking twice as much time as the actual computation!

Because of the problems inherent in a type-2 approach to exact real arithmetic, we do not believe that there is a chance to improve the type-2 overheads much further than what has already been done in *RealLib*. Instead, seeing results similar to the ones above, we opted for incorporating user functions that use an interval approach as an option that could combine the ease-of-use of higher-type access to

^{‡‡} To measure execution times in the order of microseconds, the timing is done using a modification of this code that executes Line 22 many times.

^{§§} The author of *iRRAM* could not supply an explanation or remedy for the problem.

the numbers with the efficiency of lower-type user functions. This decision was also influenced by the fact that *iRRAM* already employed an approach that works on approximations and was displaying its strengths.

For the next example, we will do a computation that does not require high precision: the first 6 digits of the sum of the first 1000, 10000, 100000 and 1000000 members of the harmonic series $\sum_{i=1}^n \frac{1}{i}$. These values can be correctly computed in double precision. We will use the following function:

```
007 template <class TYPE>
008 TYPE Harmonic(unsigned int prec, UserInt len)
009 {
010     TYPE s(0.0);
011     TYPE one(1.0);
012     for (int i=1;i<=len;++i)
013         s += one / i;
014     return s;
015 }
016 CreateIntRealFunction(Harmonic)
```

Similarly to the previous example, we measure the time needed when a function object is used and the time needed when the function is directly instantiated to the types *Real* and *double*, as well as corresponding code for the other two libraries. Let us see how exact real number packages compare to hardware floating point:

members	<i>RealLib3</i> , function	<i>RealLib3</i> , direct	<i>iRRAM</i> 2004_02	<i>XRC</i> 1.1	<i>double</i>
1000	91 μ s	27 ms	1.3 ms	22 ms	21 μ s
10000	580 μ s	275 ms	12.5 ms	–	212 μ s
100000	5.5 ms	2.85 s	118 ms	–	2.23 ms
1000000	54 ms	28 s	1.2 s	–	22 ms

Again, *XRC* failed to produce result with 10000 or more nested operations. *iRRAM* was about 60 times slower than hardware floating point.

The two layers of *RealLib* show radically different results. While the high level approach has performance as disappointing as a factor of 1000 slower, the implementation of the computation as a function using the library's approximation interfaces achieves a performance which is not that different from the hardware double precision and significantly faster than any other previous implementation. Moreover, the implementations of the same function on the two levels only differ in the types used in the definition of the function (*Real* vs. a template argument), an unused argument in the header of the function, and the use of a declaration to link the two layers.

If we switch to a better compiler^{¶¶¶}, the distance between exact real numbers using *RealLib* and hardware floating point disappears completely:

members	<i>RealLib3</i> , function	<i>RealLib3</i> , direct	<i>double</i>
1000	22.5 μ s	3 ms	18.5 μ s
10000	186 μ s	30 ms	185 μ s
100000	1.75 ms	1.05 s	1.85 ms
1000000	17.8 ms	5 s	18.5 ms

Although the current version of *RealLib* is slower than *iRRAM* when higher precisions are needed, running in a general context it will achieve significantly better performance on average, because the majority of the problems that show up in practice are easy, and the performance of the library in the error-sensitive cases is of the order of magnitude of the fastest alternative, much closer to it than any other implementation.

6. Related work

6.1. Aberth's Range Arithmetic

In Aberth's work on precise numerical computations (Aberth 1974; Aberth 1988; Aberth and Schaefer 1992), the term *Range Arithmetic* refers to performing computations with simplified interval arithmetic of increasing precision until certain accuracy requirements are met. When the precise computation of a number that cannot be represented exactly is required, range arithmetic calls for the computation of an approximation of the value and an upper bound for the error of this approximation. As an example, the computation of a Taylor polynomial would require the computation of a certain number of members of the sum via range arithmetic and the expansion of the resulting interval by an upper bound for the remainder sum.

Both characteristics exactly match *RealLib*'s approximations layer. The library thus contains an implementation of Range Arithmetic, which is also complemented by a layer that encapsulates precise computations in higher-type objects that can be easily manipulated in a user's program and contains an implementation of the mechanism of reiterations that is needed for Range Arithmetic to form a library for exact real computations.

¶¶ Pentium-M 1.8, Intel C++ Compiler 8.0, Windows XP

¶¶¶ unfortunately, *GMP*, *XRC* and *iRRAM* do not support Windows natively

6.2. Boehm and Lee's Program Slicing

In (Lee and Boehm 1990), an approach to overcome the inefficiencies of “programs over the constructive reals”, i.e. type-2 exact real computations, is given. The main idea of the approach is the selection of suitable “slices” of the program code that can be computed using Range Arithmetic, thus transforming a program using real numbers as entities to a version that makes use of lower complexity objects for large portions of the computation.

The central idea of the efficiency of the *RealLib* library is the same: avoid the big expression graphs by moving chunks of the computation to a lower level where the history of a computation is not recorded. Code written on the approximations layer of the library can be seen as the chosen program slices, while the objects on the numbers layer are the encapsulations of the results of these slices that can be used to perform the output operations and take automatic care of the intricacies in handling open values, caching their intermediate results and making sure that neither mundane or irrelevant operations are reiterated, nor any already computed approximations are lost.

This makes the library a very suitable framework for the application of the program slicing method. Given a program for *RealLib* that uses only the numbers layer, one can apply the method to select slices of the program that can be transferred to the approximations layer. In fact the program slices chosen by the method of (Lee and Boehm 1990) do not contain any output operation (including comparisons) and thus the code does not need to change to operate on the approximations layer.

6.3. Müller's *iRRAM*

Although it is commonly viewed as an implementation of the TTE model for computable analysis, which is a typical type-2 model, one can easily see that the *iRRAM* (Müller 2001) cannot be called a type-2 implementation in any sense used in this paper. We do not believe it can be called type-2 in any sense whatsoever, since the only connection is using a theoretical foundation which is *equivalent* to TTE.

The author of the *iRRAM* defines the library to be a *simulation* of Brattka and Hertling's feasible Real RAM (Brattka and Hertling 1998), an algebraic definition of the computable real numbers equivalent to the TTE. Because the programs of the *iRRAM* actually operate on simplified interval approximations and must permit reiterations, the library is a simulation rather than an implementation of the model.

Programs written for the *iRRAM* operate only on the level of approximations, using a bottom-up evaluation scheme and modularity on the level of approximations. As such, they only simulate operations on complete objects, and at no point in time do they have access to the functions that define real numbers. Thus *iRRAM* is, indeed, a type-1 implementation of exact real arithmetic.

Because of this main characteristic of the *iRRAM*, it does not suffer from the performance problems discussed in this paper except the inability to use compiler optimizations to implement very fast initial approximations.

There are two ways of using *iRRAM*. Either the user's program is completely under the control of the system, or the program uses *iRRAM* occasionally for separate computations.

The former can be very difficult to control, because the execution of a user's program under *iRRAM* becomes complicated by possible reiterations and abrupt termination. In this mode, all user code is executed a multitude of times, not only the portions that use exact real computations.

The latter mode of using the system is far more inconvenient than having a type-2 interface to the real numbers. We may be forced to redo the same computation more than once if, e.g. we want to print a variable as an absolute value and its base-10 logarithm.

RealLib takes the ideas of the *iRRAM* and pushes them further. Here is a comparison of some of the key features of the two libraries:

- Both operate on interval approximations, but the version of *iRRAM* which was available at the time of this performance comparison did not include a fast initial step using hardware floating point. As communicated by the author, at least the development versions of the *iRRAM* now include such a step, but it is realized using run-time choices to switch between hardware interval arithmetic and software multi-precision simplified arithmetic. *RealLib* makes that a compile-time choice which allows compiler optimizations and performance very close to hardware floating point. This also makes it easier for *RealLib* to include additional optimized step in the future, such as a double-double or quad-double evaluation (Hida et al. 2001) as a second iteration, or possibly a fast implementation using integer arithmetic with hard-coded precision.
- When incorporated into bigger programs, computations using the *iRRAM* do not integrate well; *iRRAM* has to use tricks such as overriding the standard *C++* input/output streams in order to pretend to operate on real numbers as complete objects, while in fact the methods used only have access to approximations. *RealLib* provides a type-2 interface to allow exact computations to be used in a program without modifying its behavior and control flow, while the efficient layer that corresponds to the mechanisms used in *iRRAM* is clearly marked as a layer that operates on approximations. Still, in the cases where a function is a sequence of already defined operations without conditionals based on the values of real arguments, the program code on the approximations layer looks and behaves as if it works on complete entities, because the functions and operations applied on this layer are in fact implementations of the corresponding real number functions.
- *iRRAM* asks for a rapidly converging Cauchy sequence to define the limit of a computably converging sequence, while *RealLib* relies on accounting for the error of the approximation. This is defined by the theoretical model used in

the two systems, but we feel that the latter gives us a more uniform approach, since the former is in effect a case of top-down evaluation of the limit, something both libraries try to avoid.

- The template approach of *RealLib* and the availability of computation dags allows the library to accommodate separation bounds (Melhorn and Schirra 2000), which in a future version of the library could allow to decide equality for algebraic numbers.
- The multiple-precision back-end of *iRRAM* is *GMP*, which is very fast and hand optimized using assembler code. *RealLib* uses a custom library which is completely portable, but is much slower. The only thing that prevents *RealLib* to use the lowest level functions of the faster *GMP* as back-end is the fact that we give higher priority to the computations at low precision which appear much more often in practice and we have not yet implemented the link between the library and *GMP*.
- Intensional (also known as multi-valued) functions are an integral part of *iRRAM* and currently not supported in *RealLib*. They require a theoretical background which was only recently developed (see (Lambov 2005)) and has not been realized yet.
- Unlike other packages that do not allow the computation of limits in user programs, both *RealLib* and *iRRAM* can be shown to be complete, i.e. able to define all computable real numbers and extensional functions, by showing that all operations in the feasible Real RAM model are defined.

References

- Aberth, O., *A precise numerical analysis program*. Communications of the ACM, 17(9), pp. 509–513 (1974).
- Aberth, O., *Precise Numerical Analysis*. Wm. C. Brown Publishers, Dubuque, Iowa (1988).
- Aberth, O. and Schaefer, M. J., *Precise computation using range arithmetic, via C++*. ACM Trans. Math. Softw. 18, 4, pp. 481–491 (1992).
- Brattka, V., Hertling, P., *Feasible Real Random Access Machines*. Journal of Complexity 14, Issue 4, pp. 490–526 (1998).
- Briggs, K., *Implementing exact real arithmetic in python, C++ and C*, to appear in Journal of theoretical computer science
see also <http://keithbriggs.info/xrc.html>
- Edalat, A., *Exact Real Number Computation Using Linear Fractional Transformations*. Final Report on EPSRC grant GR/L43077/01
Available at
<http://www.doc.ic.ac.uk/~ae/exact-computation/exactarithmeticfinal.ps.gz>
- Grzegorzcyk, A., *On the definitions of computable real continuous functions*. Fundamenta Mathematicae 44, pp. 61–67 (1957).
- Hida, Y., Li, X. S. and Bailey, D. H. , *Algorithms for Quad-Double Precision Floating Point Arithmetic*, 15th IEEE Symposium on Computer Arithmetic, IEEE Computer Society, pg. 155–162 (2001).

- Lambov, B., *Complexity and Intensionality in a Type-1 Framework for Computable Analysis*, Lecture Notes in Computer Science **3634**, pp. 442–461 (2005).
- Lee, V.A. Jr., Boehm, H.-J., *Optimizing Programs over the Constructive Reals*. In: Conference on Programming Language Design and Implementation, Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation. ACM Press, New York, pp. 102-111 (1990).
- Mehlhorn, K. and Schirra, S., *Generalized and improved constructive separation bound for real algebraic expressions*, Research Report, Max-Planck-Institut für Informatik, November (2000).
- Müller, N., *The iRRAM: Exact arithmetic in C++*. Computability and complexity in analysis. (Swansea, 2000). Lecture Notes in Computer Science **2064**. Springer (2001). see also <http://www.informatik.uni-trier.de/iRRAM/>
- O’Conner, R., *Few Digits*, <http://r6.ca/FewDigits/>.
- Weihrauch, K., *Computable Analysis*. Springer, Berlin 2000.
- GMP*, *The Gnu Multiple Precision arithmetic library*, <http://www.swox.com/gmp/>.