Math. Struct. in Comp. Science (2007), *vol.* 17, *pp.* 81–98.
doi:10.1017/S0960129506005822 Printed in the United Kingdom

RealLib: An efficient implementation of exact real arithmetic

BRANIMIR LAMBOV†

BRICS, University of Aarhus, IT Parken, 8200 Aarhus N, Denmark
Email: barnie@brics.dk

Received 1 December 2005; revised 9 June 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 with 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 that is useful in a wide variety of contexts. For this, the library must 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, perhaps 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 as entities (more information about the performance problems of this approach will be given below: examples of packages implementing this include ICReals (Edalat 2001), XRC (Briggs to appear), Few Digits§ 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 behaviour of the user's programs, as demonstrated by the iRRAM, which was created by Norbert Müller (Müller 2001).

Our exact real number implementation takes an approach that provides two levels of access to real numbers aiming to satisfy conflicting parts of the requirements. One of

† Currently at FB Mathematik, AG 1, Technical University Darmstadt, Schlossgartenstrasse 7, 64289 Darmstadt, Germany.
‡ RealLib, available at http://www.brics.dk/~barnie/RealLib/.
§ See Russell OConnor's website at http://r6.ca/FewDigits/ for details.

the levels operates on real numbers as complete entities and is easy to integrate into 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 that operates on the level of approximations and is free of performance issues, but for which the control flow may become more complicated as the objects operated on do not represent complete reals. Both levels have well-defined behaviour and do not use any non-standard programming.

The two levels interact with each other, more specifically, the layer that operates on complete entities (which will be called the 'top', or 'number' layer in the rest of the paper) can use objects written in the layer that operates on approximations (which will 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 number layer. In an extreme case (which may happen quite often in practice and is very similar to the mode of operation of the iRRAM), all of a computation may be implemented on the approximations layer as what we call a 'nullary' function (that is, a real number constant), using the top layer, for example, just to 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 with the best available alternatives.

2. Performance problems in real number arithmetic

There are a multitude of equivalent theoretical formulations of exact real number computability (see Grzegorczyk (1957), Pour-El and Richards (1989), Ko (1991), Edalat and Sünderhauf (1999) and Weihrauch (2000), among others), which share the ability to operate on all real numbers, including non-computable ones. From a practical point of view the most interesting characteristic of each of these approaches is the type level of the objects used to realise real number functions. Real numbers contain infinite amounts of information, so a higher type concept of computability, where real number arguments are given by function oracles, is the natural choice for the representation of real functions. However, because all computable functions are continuous, it is possible to represent all computable real functions by computable functions of Type 1, that is, ordinary computable functions, and many of the theoretical frameworks employ the latter approach.

In the rest of this paper we will use the term 'type-2' to refer to frameworks where the application of functions to a real number requires the number to be represented along with its complete information, that is, where the representations of functions are type-2 objects. When such a representation is asked for an approximation with a given accuracy, it can decide how much information it needs from its arguments and ask them to provide it, effectively propagating a higher accuracy request to its arguments. Such a scheme is employed in Ko (1991) and also in the most convenient representations ρ_C and ρ_{sd} of Weihrauch (2000). Unlike other schemes that fit into the general concept of type-2 real number computability, this one actually requires the type-2 machinery, thus we will call it 'genuine type-2'. For an implementation to be called just 'type-2', we will require

https://doi.org/10.1017/S0960129506005822 Published online by Cambridge University Press

it to provide the complete information for every real number used in the course of a computation regardless of whether that information is actually used.

In contrast, frameworks where real functions operate on incomplete information that can be encoded in a natural number and thus do not require higher type computability, will be called 'type-1' frameworks. In such approaches the conversion of an argument to the result of the application of a function is accomplished by applying the function pointwise to the infinite sequence that defines a real number (for example, the last two formulations in Grzegorczyk (1957), see also Edalat and Sünderhauf (1999) and Lambov (2005)), in some cases applying additional machinery to shift and clean the resulting sequences (Grzegorczyk 1957; Pour-El and Richards 1989; Ko 1991). A type-1 framework usually fits within the general idea of type-2 real number computability by artificially requesting that more information be available to the objects representing a real function. However, if an implementation relies on a type-1 theoretical approach, it is not required for it to be able to present the complete information about an argument to a function, and thus a 'type-1' implementation is not a type-2 implementation as well.

The approach to real number computability most often attempted in practice is some variation of the genuine type-2 framework defined by the representation ρ_C in the Type-2 Theory of Effectivity (TTE) (Weihrauch 2000). The approach looks pretty easy to implement, especially if one uses a functional language such as SML and ocaml.

Unfortunately, the type-2 character of the approach presents some barriers to implementing the ideas efficiently. Some of them can be avoided, but some of the problems are so serious that it is impossible to achieve a type-2 implementation free of severe performance problems.

Indeed, genuine type-2 implementations have proved to be extremely slow, being at least a magnitude slower[†] than mixed solutions such as RealLib and the iRRAM[‡], even in cases where higher precisions are used and the bookkeeping overheads are low. We will now look at the reasons for this poor performance.

Since every real number used in the course of a computation must 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 in the following subsections.

2.1. *Problems with common subexpressions*

An implementation of a binary operation computes an approximation based on approximations to its operands with higher accuracy. When the result of a type-2 binary operation is evaluated, the function is called with an argument telling it how good an approximation

[†] 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 the iRRAM with XRC, Few Digits and BigNum.

[‡] There is a common misconception that the iRRAM is a type-2 implementation of exact real computability, but, as we will see in Section 6.3, this is not the case.

https://doi.org/10.1017/S0960129506005822 Published online by Cambridge University Press

is required. The function in turn asks its arguments to compute (better) approximations and uses the information obtained to compute its 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 may build a representation of the computation as a tree, which would cause the argument to appear 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 to avoid any unnecessary growth in 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 combination is highly non-trivial to do in a genuine type-2 implementation, where the problem of a gradual increase in the precision can easily destroy the intended benefits of caching and reintroduce the complexity explosion.

2.2. *Unnecessary precision growth*

A genuine type-2 implementation of a function requests higher accuracy from its arguments in order to be sure about the accuracy of its result. This may lead to unnecessarily large precision growth.

Consider the following 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, so in this code fragment the accuracy needed will grow by at least one bit at every iteration. Thus, if we want to compute an approximation to `a` that is 32-bit accurate, we will end up performing many of the additions at very high precision, some of them with 1,000,000 or more bits of precision. Clearly, this is not acceptable from a performance point of view, since in reality fewer than 60 bits suffice to carry out the whole computation.

https://doi.org/10.1017/S0960129506005822 Published online by Cambridge University Press

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 reliably in practice, even less so if one wants to combine it with a solution to the previous problem. For this particular example, one may ask the user to change their code to use special implementations of balanced accumulation, but we do not want to impose such requirements on the user.

The approach described so far is sometimes called 'top-down evaluation' to indicate that a node controls the accuracy of its children. The name also hints at the possibility of a 'bottom-up' approach where the accuracy of the children determines the accuracy in a node, an approach which does not suffer from the problem at hand and in practice turns out to be far superior although there are cases where the top-down evaluation scheme may be theoretically better.

The bottom-up approach evaluates everything at an (almost) constant precision, starting from the leaves of the dag and 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 dag is not accurate enough. In such a case the whole computation is restarted at higher precision until the process leads to a result that is accurate enough (that is, in which the accumulated error is sufficiently small). The reiterations do not add much to the complexity of the process, since, for example, by doubling the precision for each new iteration we can be certain that the 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 get more accurate as the input interval shrinks. 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 centre 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, since an implementation of a function now takes only an approximation, and does not have the ability to request a better one from its arguments. In other words, it operates on incomplete information that can be encoded in a natural number and is thus of Type 1. An implementation that contains this modification is certainly not genuinely type-2, but if we still keep the computation dags, that is, complete information about all real numbers as a function representation, it may be called a (fake) type-2 implementation.

2.3. *Bookkeeping requirements*

Even with the modifications discussed above, a type-2 implementation (genuine, fake, or any other type of 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. This restriction is clearly unacceptable since this number of operations can be reached easily in, for example, linear algebra.

https://doi.org/10.1017/S0960129506005822 Published online by Cambridge University Press

One needs to design some machinery that permits the encapsulation of user-defined functions in the nodes of the term representation, thus reducing the growth of the dag at the expense of having a more complicated traversal scheme. Still, in any useful scheme that we have seen or can imagine, the presentation of any form of complete representations of the arguments to a type-2 implementation of an operation requires every step in the computation to allocate memory, which is known to be a painfully slow process in modern computers, and is significantly slower than the time it takes actually to 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 only possible in a type-1 approach, since the basic property of real functions in a type-2 model is the access to full function representations of their real arguments.

To have a natural way of constructing functions from existing ones, 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 achieve full soundness and completeness as well, that is, equivalence to the popular notions of exact real computation, a type-1 theoretical approach must be used. Combined with the requirement for bottom-up evaluation, an interval approach such as Grzegorczyk's interval definitions (Grzegorczyk 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 the 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 999,999 temporary values. The evaluation of a tree is recursive, and a naive implementation may evaluate the argument on the left-hand side to additions first, before diving into the recursion to compute the argument on the right-hand side, 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 thrash 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 that chooses the better pattern of evaluating arguments does not seem trivial. A heuristic, such as choosing to evaluate first the argument that is defined by the tree of higher depth, must be used, but it will most probably fail in a large number of cases.

To solve the problem completely, we would prefer to use the order and locality information that is given by the programmer and compiler in the program code, instead of losing it in the process of generating the dag and then trying to recreate something similar through a heuristic. For this example, it would mean that only a single object, a, needs to be stored to complete the evaluation.

To achieve this, one must again 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 optimisations*

Whenever a computation dag determines the order of operations it is impossible to perform any compiler optimisations 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 achieve a 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 allow this tool to be used to achieve optimal performance. If, instead, the choice is done at run time, the branching necessary to choose the more efficient program path nullifies the benefit of in-lining functions by forbidding the compiler to reorder code around the branch and to optimise as aggressively as it does with floating point code.

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 genuine 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

https://doi.org/10.1017/S0960129506005822 Published online by Cambridge University Press

results and delete them exactly when they are no longer needed[†]. Additionally, in an attempt to minimise the effect of the loss of locality information, the library also tries to optimise the order of evaluation of the children 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 those aspects that have been proved to be undecidable: because of the non-computability of the equality test, all comparisons of real numbers are undefined for equal arguments; in consequence of this, the library does not provide non-strict versions of comparisons (<= and >=) 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.

Once a function is written for the bottom layer of the library, a dag for the operations used in this function will never be constructed, the temporary real numbers used in the computation of the function only appear as approximations and the computations are carried out 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, and, if that does not lead to a sufficiently accurate result, the second approximation uses a different instance capable of performing arbitrary precision simplified interval arithmetic.

This 'function' layer is used to define functions that can be used directly on the number 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 at the top layer and a request for a property of it is made (such as, for example, 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

[†] The released RealLib 3 contains a programming error that interferes with this principle in the case of multiple requests made by the user. This error has already been fixed in the development code.

https://doi.org/10.1017/S0960129506005822 Published online by Cambridge University Press

requesting a reiteration 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 there 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 genuine 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 the application 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 at the level of approximation can add to the amount of uncertainty in an approximation to cover 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 that is to be added to the error in the approximation of the result. Since this function gives improved approximations to the number as `prec` grows, the third condition is also satisfied.

The same method is used in the implementations of real functions that need to compute a limit. The only difference is that prec is no longer given explicitly as an argument to the

https://doi.org/10.1017/S0960129506005822 Published online by Cambridge University Press

function. In exact correspondence with the theoretical model, this parameter is recovered from the approximation to one of the real arguments of the function.

Examples of defining a function that requires limitation are given in the library's manual†.

Another operation that is a requirement for the completeness of a real number package, is the presence of approximate comparisons, that is, 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 that can be used to choose more efficient execution paths. A weak operation only evaluates properties of the centre of an approximation and is not guaranteed to give consistent results in consecutive 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 for which it is known *a priori* that the computations cannot be handled correctly in machine precision, and finishing with a case that 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 that all exact real number systems share: the possibility of requesting arbitrarily precise approximations to a number. In this case, we choose to display a 10,000-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 manual is available as part of the library or at the internet address `http://www.brics.dk/~barnie/RealLib/`

https://doi.org/10.1017/S0960129506005822 Published online by Cambridge University Press

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 initialisation and finalisation 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 10,000 digits that are no more 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 complete this program and compare it to two other exact real number systems, XRC by Keith Briggs (Briggs to appear) and the iRRAM by Norbert Müller (Müller 2001):

RealLib3	iRRAM 2004_02	XRC 1.1
730 ms	230 ms	364 s

The iRRAM has the reputation of being the fastest exact real number library, using highly optimised GMP[‡] for the higher precisions that are required for this example. Its reputation 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++. On the other hand, XRC 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. The argument `prec` exists to permit the computation of limits, and the declaration creates a function with the signature

```
Real Logistic(UserInt len);
```

[†] The results are for a Pentium-M 1.8GHz and GCC 3.3 in the Cygwin environment.
[‡] The Gnu Multiple Precision arithmetic library: see `http://www.swox.com/gmp/`.

https://doi.org/10.1017/S0960129506005822 Published online by Cambridge University Press

which is the representation of this function on the level of real numbers. This object has a single argument, because the precision is only needed to transform inexact computations into exact ones and has no meaning for the object created.

Apart from the unused argument, the code above is pretty standard code for the iteration, and, being a template, it can also be used to try out the direct implementation of the computation using the type `Real` or any other type. We will make use of this code in the following main function (which is not very 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 on Line 17 from the template function `Logistic`. In the table that follows, this will be reflected in the column 'RealLib3, function'. For the column 'RealLib3, number' 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;
```

The RealLib3 timings[†] and the timings for the corresponding code for the other two exact real number systems are given in the following table:

iterations	RealLib3, function	RealLib3, number	iRRAM 2004_02	XRC 1.1	double
100	1 ms	3 ms	625 ms	383 ms	0.6 μs
1000	150 ms	188 ms	12 ms	143 s	6 μs
10000	48 s	50 s	5.5 s	–	60 μs

For unknown reasons the iRRAM did not compute the 100 iterations as fast as we would expect, so the first value in the iRRAM's column should probably be ignored[‡].

[†] 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 iRRAM's author was unable to supply either an explanation of the problem or a remedy.

https://doi.org/10.1017/S0960129506005822 Published online by Cambridge University Press

XRC was apparently not designed for use with heavily nested computations and its recursive evaluation mechanism failed for more than a few thousand nested operations.

The 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 library's top-level interface is clearly visible when compared with the approximations interface. It may be little or negligible at high precision (only 2 out of the 50 seconds required to compute the 10,000 iterations), but is quite a big portion of the time for a lower precision computation, and dominates the time in the simplest case, taking twice as long 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 way to improve the type-2 overheads much further than 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 the numbers with the efficiency of lower-type user functions. This decision was also influenced by the fact that the iRRAM already employs 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 1,000, 10,000, 100,000 and 1,000,000 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)
```

As with 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 with the corresponding code for the other two libraries. The following table shows how exact real number packages compare with hardware floating point:

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

https://doi.org/10.1017/S0960129506005822 Published online by Cambridge University Press

Again, XRC failed to produce result with 10,000 or more nested operations. The 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 that is not that different from the hardware double precision and significantly faster than any other previous implementation. Moreover, the implementations of the same function at 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 gap between exact real numbers using RealLib and hardware floating point disappears completely:

members	RealLib3, function	RealLib3, number	double
1000	22.5 μs	3 ms	18.5 μs
10000	186 μs	30 ms	185 μs
100000	1.75 ms	1.05 s	1.85 ms
1000000	17.8 ms	5 s	18.5 ms

Unexpectedly, the exact version even surpasses the performance of the double precision one. Since the library's design does not hinder optimisations, the parallelism of current processors can make up for the larger amount of work that needs to be done and apparently the compiler even managed to find better optimisation opportunities for the exact code.

Although the current version of RealLib is slower than the 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, and 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

[†] Pentium-M 1.8, Intel C++ Compiler 8.0, Windows XP.
Unfortunately, GMP, XRC and the iRRAM do not support Windows natively.

https://doi.org/10.1017/S0960129506005822 Published online by Cambridge University Press

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.

6.2. *Boehm and Lee's Program Slicing*

Lee and Boehm (1990) presents an approach to overcoming the inefficiencies of 'programs over the constructive reals', that is, type-2 exact real computations. 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. The code written at the approximations layer of the library can be seen as the chosen program slices, while the objects at the numbers layer are the encapsulations of the results of these slices that can be used to perform the output operations and take care automatically 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 just uses 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 actual code in the program slices can be used at the approximations layer without change.

6.3. *Müller's iRRAM*

Although it is commonly viewed as the model implementation of the TTE framework for computable analysis, which relies heavily on type-2 computability, it is easy to see that the iRRAM (Müller 2001) cannot be called a type-2 implementation, because it does not record the complete information about a number, nor provide this information to the objects that represent real functions. Theoretically, the TTE framework is broad enough to cover the type-1 theoretical approach that underlies the iRRAM, but key to its performance is the fact that the latter does not need or use the powerful machinery that is an integral part of that framework.

The iRRAM's author defines the library to be a simulation of Brattka and Hertling's feasible Real RAM (Brattka and Hertling 1998), which gives an algebraic definition of the computable real numbers equivalent to the TTE. Because the iRRAM's programs 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 at the level of approximations, using a bottom-up evaluation scheme and modularity at 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 the 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 other than the inability to use compiler optimisations to implement very fast initial approximations.

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

The former can be very difficult to control, because the execution of a user's program under the iRRAM becomes complicated by possible reiterations and abrupt termination. In this mode, *all* user code is executed a multitude of times, not just 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, for example, 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 the iRRAM that 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 realised 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 optimisations and performance very close to hardware floating point. This also makes it easier for RealLib to include additional optimised steps 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; the 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 the methods used in fact only have access to approximations. RealLib provides a type-2 interface to allow exact computations to be used in a program without modifying its behaviour and control flow, while the efficient layer that corresponds

https://doi.org/10.1017/S0960129506005822 Published online by Cambridge University Press

to the mechanisms used in the iRRAM is clearly marked as a layer that operates on approximations and should not be used for anything other than exact computations. 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.

— The 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, which is 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 us to decide equality for algebraic numbers.

— The multiple-precision back-end of the iRRAM is GMP, which is very fast and hand optimised using assembler code. RealLib uses a custom library, which is completely portable, but much slower. The only thing that prevents RealLib from using 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 the iRRAM and are currently not supported in RealLib. They require a theoretical background that has only been developed recently (Lambov 2005), and has not been realised yet.

— Unlike other packages that do not permit the computation of limits in user programs, both RealLib and the iRRAM can be shown to be complete, that is, 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. (1974) A precise numerical analysis program. *Communications of the ACM* **17** (9) 509–513.

Aberth, O. (1988) *Precise Numerical Analysis*, Wm. C. Brown Publishers.

Aberth, O. and Schaefer, M. J. (1992) Precise computation using range arithmetic, via C++. *ACM Trans. Math. Softw.* **18** (4) 481–491.

Brattka, V. and Hertling, P. (1998) Feasible Real Random Access Machines. *Journal of Complexity* **14** (4) 490–526.

Briggs, K. (to appear) 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. (2001) 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`.)

Edalat, A. and Sünderhauf, P. (1999) A domain-theoretic approach to computability on the real line. *Theoretical Computer Science* **210** (1) 73–98.

https://doi.org/10.1017/S0960129506005822 Published online by Cambridge University Press

Grzegorczyk, A. (1957) On the definitions of computable real continuous functions. *Fundamenta Matematicae* **44** 61–67.

Hida, Y., Li, X. S. and Bailey, D. H. (2001) Algorithms for Quad-Double Precision Floating Point Arithmetic. *15th IEEE Symposium on Computer Arithmetic*, IEEE Computer Society, 155–162.

Ko, K.-I. (1991) *Complexity Theory of Real Functions*, Birkhäuser.

Lambov, B. (2005) Complexity and Intensionality in a Type-1 Framework for Computable Analysis. *Springer-Verlag Lecture Notes in Computer Science* **3634** 442–461.

Lee, V. A. Jr. and Boehm, H.-J. (1990) 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 102–111.

Mehlhorn, K. and Schirra, S. (2000) Generalized and improved constructive separation bound for real algebraic expressions. Research Report, Max-Planck-Institut für Informatik.

Müller, N. (2001) The iRRAM: Exact arithmetic in C++. Computability and complexity in analysis (Swansea 2000). *Springer-Verlag Lecture Notes in Computer Science* **2064**. (See also `http://www.informatik.uni-trier.de/iRRAM/`.)

Pour-El, M. B. and Richards, J. I. (1989) *Computability in Analysis and Physics*, Springer-Verlag.

Weihrauch, K. (2000) *Computable Analysis*, Springer-Verlag.