

General purpose exact real arithmetic

J. Blanck*

University of Wales Swansea, Singleton Park, Swansea, SA2 8PP, UK

June 20, 2002

Abstract

Implementation techniques for exact computations are described and to some degree evaluated. In particular, two basic algorithms for exact computations are identified and investigated. The very important choice of approximations is examined, in particular for the space of the reals. The proposed approximations contain an error term and it is argued that the error term should be variable, but bounded.

1 Introduction

Exact real arithmetic is an attempt to provide an implementable data type for the reals. The real numbers will be proper real numbers without any rounding off errors or limitations in size. (We will, strictly speaking, incorrectly assume that computers have sufficient memory for any computation, or equivalently, that computers are as powerful as Turing machines.) Since a computer may only represent countably many of the uncountably many real numbers, even if the memory of the computer can be indefinitely extended, there will be real numbers that cannot be represented within a computer. However, these non-representable numbers have in common that they are not approximable, there cannot simultaneously exist procedures giving all upper bounds and all lower bounds respectively. Thus, most common quantities in mathematics, e.g., $\sqrt{2}$, π , e , and $\sin 2$, are representable in exact real arithmetic. Among the exceptions are the Ω -numbers of Chaitin [8].

Systems for general purpose exact real arithmetic have been implemented by several people, see for example [15, 10]. Some of these participated in a

*Supported by STINT, The Swedish Foundation for International Cooperation in Research and Higher Education.

competition between systems for exact real arithmetic [4] in Swansea. We aim here to give a survey of common techniques and some choices that an implementor needs to consider. The main aim is to provide background and describe some ways of achieving exact computations. Some relative comparisons between different solutions will also be made, but the conclusions of these are, in general, very loosely formulated since proper testing has yet to be made. One could in fact say that there is not yet any agreement on what should be reasonable benchmarks for these systems.

The main reason for exact real arithmetic is obviously a hope that computational errors will be avoided without having to perform separate error analysis. The correctness depends on choosing correct algorithms for each function, and on the correct implementation of each of these algorithms. Whereas the former is readily verified in mathematics, the latter concerns the more unwieldy problem of verifying computer programs. So far, no system comes with a proof of its correct implementation. The only verification that has been performed is that different systems agree for a number of problems. Agreement to high precision is not a proof of correctness but is nevertheless a useful indication. Thus, faith in the correctness is limited by our trust of the implementations.

1.1 General purpose exact real arithmetic

A general purpose system for exact real arithmetic is of course necessary before any attempt is made to incorporate a proper data type for the reals into a programming language. For an exact real arithmetic to be general it has to supply a finite set of basic operations that can be combined to evaluate any desired real expression. Thus, the system needs to provide the basic arithmetic operations, logarithms, exponentials, trigonometric functions, and so on.

1.2 The underlying theory

The theoretical foundations for this approach to a real data type is the subject Computable Analysis [16, 22, 1]. There exist several flavours of computable analysis; we will use the one based on the definitions of Grzegorzczuk and Lacombe [11, 14]. The main point about computable analysis is that all operations will be implementable on an ordinary computer (given that it has enough memory). This is in contrast to, for example, Blum–Shub–Smale’s abstract model of computations on the reals [6]. The most notable difference is that equality and the ordering are not computable operations in computable analysis. However, the abstract model of computation given

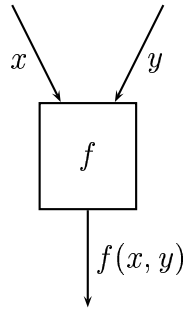


Figure 1: Traditional picture of a binary operation.

by Tucker and Zucker [20, 19], does truly model the behaviour of concrete computations of discontinuous functions such as equality.

1.3 Calling interface

The theoretical model of computable analysis uses special Cauchy sequences (recursive Cauchy sequences with recursive moduli) to represent the reals. The Cauchy sequences are easy to pass around in mathematics, but they are not that easy to handle as input to, and output from, a computer. The traditional dataflow picture (cf. Figure 1, where input and output in our case should be thought of as pairs of a Cauchy sequence and a modulus) of an operation as a box taking arguments and generating output is therefore not the best one to illustrate the calling interface.

We are, in general, more interested in getting some approximation of the real number. Hence, the interface to exact real arithmetic is not simply the input numbers, but also a specified precision for the resulting approximation. Similarly, the input is not regarded as sequences which are read element by element, but rather as functions taking an accuracy and returning an element of the Cauchy sequence within that accuracy. The situation is depicted in Figure 2. The user would input/supply the operation f with the boxes x and y , and the desired precision p , the output is the approximation c of $f(x, y)$. It might be necessary for the operation f to compute several different approximations of the arguments x and y . There is a semblance of lazy operations in that arguments are not evaluated until an approximation is needed. However, there is more than laziness to the calling scheme since expressions may be evaluated multiple times in order to generate sufficiently good approximations.

Note that neither the Cauchy sequence nor the modulus is depicted. This

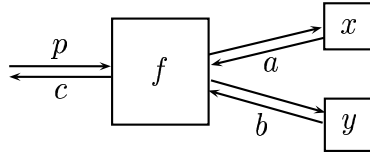


Figure 2: A binary exact real operation.

is reasonable given the calling interface suggested. The Cauchy sequence and the modulus is a part of the implementation of the operation, not part of the interface.

It has become common to say that implementations use *fast* Cauchy sequences. The interpretation is then that successive calls will be made to the operation with accuracy 2^{-n} for the n th call. The resulting sequence of approximations is indeed a Cauchy sequence which has a fast convergence to the limit.

Having established that input and output of real numbers will not be done by sending around the sequences themselves but rather some approximation, we have to decide on the form of these approximations, which is the topic of the next section.

2 Approximations

In general, approximations are based on the existence of a countable dense set within the space. We will consider here mainly the case of approximations of real numbers as it is the most basic case. In particular, the rationals are dense in the reals. If one takes the rationals as the basis for approximations there are two equivalent ways of constructing approximations. To approximate a real point x either take a closed interval $[a, b]$ with rational endpoints that contains x , or take a pair of rational numbers (c, e) such that $|x - c| \leq e$. The two approaches are obviously equivalent. An approximation of the form $[a, b]$ can also be given as $((a+b)/2, (b-a)/2)$, and conversely, (c, e) can be given as $[c - e, c + e]$. However, from an implementational point of view the midpoint-error representation is preferable since in practice the representation of the rational number e can be kept very small compared to the representations of the numbers a , b and c .

Using rational numbers to approximate real numbers and performing computations on these approximations entails a need to represent exactly rational numbers with huge denominators. For example, the simple opera-

tion of squaring a rational approximation p/q results in a number p^2/q^2 that requires double the memory to store.

By approximating these rationals in turn it is however possible to keep the size of the approximations down. In fact, within every interval there exists a simplest rational. The simplest rational is the one with a minimal positive denominator and among these the one with the smallest absolute value of the numerator. Finding the simplest rational approximation available may be time consuming. Therefore, it is often convenient to choose a *dyadic* approximation, numbers of the form $m \cdot 2^{-s}$, where m and s are integers, since these are more readily arrived at, see Example 4.1.

For the reasons explained above an often suitable choice for approximations of real numbers are *dyadic approximations* of the form

$$a = (m \pm e)2^{-s},$$

where $m, s \in \mathbb{Z}$, and $e \in \mathbb{N}$. The value e is usually bounded. In fact, e may be fixed to be 1, but we will see that the more general form is preferable. We will use a and b to denote approximations, m and n to denote mantissas (or significands), s and t to denote exponents, and e to denote error terms.

We will say that a real x is *approximated* by an approximation $a = (m \pm e)2^{-s}$ if

$$x \in [(m - e)2^{-s}, (m + e)2^{-s}].$$

An approximation of the form $(m + e)2^{-s}$ is a *p-approximation* if

$$e2^{-s} \leq 2^{-p},$$

that is, if $k = \lceil \log_2 e \rceil$ then $(m + e)2^{-p-k}$ is a *p-approximation*.

To avoid an increase in size of the dyadic approximations during the course of a computation rounding of intermediate results needs to be done rather often. We will assume that rounding takes place after each elementary operation. For example, let x be approximated by $a = (m \pm e)2^{-s}$. An approximation of x^2 is $b = (m^2 \pm (2me + e^2))2^{-2s}$, which takes double the storage space compared to a . But rounding b will often give an approximation that lose very little precision (relative to the magnitude of x) and avoids increasing the size.

These dyadic approximations are just floating point numbers with error terms. Recall that a floating point number is of the form

$$(-1)^S \cdot m \cdot 2^s,$$

where S is the sign bit, m is the mantissa, and s is the exponent. The mantissa in double precision floating point numbers (ANSI/IEEE Standard

754/854) is between 2^{52} and $2^{53} - 1$ (which fits into 52 bits since the leftmost bit, always 1, need not be stored). The exponent ranges from -1074 to 971 (the remaining two possibilities for the exponent covers some special cases like zero, non-normalised numbers and infinity). Thus, floating point numbers are special cases of the approximations above where both the mantissa and the exponent are bounded and the error term is missing. The inclusion of the error terms in the approximations is what will give us the ability to claim that we are doing exact real arithmetic.

This set of dyadic approximations is not necessarily the best for all applications but it does have two important properties that many other choices do not possess in conjunction. Firstly, efficient algorithms have been developed to compute all common operations, see, for example, [7]. Multiplication can be computed in $O(n \log n \log \log n)$, where n is the size of the operands, by, e.g., the Schönhage–Strassen method [13, 17]. Secondly, for any bounded interval (assuming that the error term is bounded) the storage space needed for a p -approximation of any point in the interval is merely $p + \log p + c$, where c is some constant. For unbounded intervals this is clearly not true since the representation of the integer part becomes unbounded. However, with a change of semantics of the notion of precision to be the number of significant digits, as often done in science, a similar statement is true even for unbounded intervals.

Approximations based on arbitrary rational numbers does not have any bound on the amount of storage needed for p -approximations even on compact intervals. The process of finding the simplest rational for any point will give bounds on the storage but will also incur a lot of computation.

Various signed digit representations of the reals can also be considered. These representations share the limited space requirement of dyadic representation. However, sofar, there seems to be no complete implementation of all fundamental operations (including transcendental functions) using directly the signed digit representation. Thus, though an elegant representation, it should not outperform the dyadic representations, at least as long as the hardware does not directly support signed bits. Yet other alternatives are approximations that uses continued fractions, either purely as [21], or as linear fractional transformation [9]. While both methods may require even less storage than dyadic representations, there still does not exist methods for many elementary operations that are as efficient as those for dyadic approximations. For linear fractional transformations this is investigated by Heckmann [12].

Approximations for other metric spaces can easily be constructed. Choosing among different kinds of approximations for general metric spaces will be even more difficult than choosing approximations for real computations.

In the real case, some subset of rational intervals are the obvious way to approximate reals, so we essentially had to consider how to represent the chosen dense set of rationals. For arbitrary metric spaces one furthermore has to decide on the dense set to use and sometimes even on which metric to use. Consider, for example, finding approximations of the real continuous functions on a compact interval. Approximations could be closed spheres (according to some norm) with centres in some dense set, e.g., polynomials or piecewise linear functions. Selecting the best approximations will probably depend on what operations are to be performed. Hence, an extensive study is needed in order to decide on a set of approximations for general computations.

2.1 Structure of approximations

The dyadic approximations above can be given a notion of how good they are in terms of an ordering. An approximation b is above an approximation a (or b is better than a), $a \sqsubseteq b$, if the interval approximated by b is included in the interval approximated by a . It is customary to reverse the ordering notion in this way.

The reverse inclusion ordering \sqsupseteq introduced above is only a pre-order. The order is not antisymmetric since, for example, $(m \pm e)2^{-s}$ approximates the same interval as $(2m \pm 2e)2^{-s-1}$. The least upper bound of two approximations would be the intersection of the intervals approximated, which again is a dyadic interval. However, in the case of a bounded generalised error term, this dyadic interval might not be representable with an error term within the bound. For example, suppose the error term is to be strictly bounded by 2^j . The least upper bound of $(0 \pm (2^j - 1))2^0$ and $(1 \pm (2^j - 1))2^0$ is the interval represented by $(1 \pm (2^{j+1} - 3))2^{-1}$, which clearly cannot be represented with an error term within the bound. Hence, the approximations do not form a csl, and the ideal completion of the pre-order is therefore not a Scott-Ershov domain. There is, however, a complete set of minimal upper bounds, $(0 \pm (2^j - 2))2^0$ and $(1 \pm (2^j - 2))2^0$. In fact, the ideal completion of the pre-order is an algebraic cpo that satisfies the conditions of an SFP-domain. Thus, the computability on the set of computable reals can be modelled as in [18, 2, 3].

2.2 Computer representations of the approximations

To compute with the dyadic approximations considered here they need to be mapped to existing data types. The error term would normally be bounded so this may be represented by any integer data type. The exponent can be arbitrarily large in principle, however, there is no point in being able to

represent exponents that would require the mantissas to be larger than the available memory. A 64-bit exponent is therefore enough for most practical implementations today. The mantissas are arbitrarily large integers, and hence they must be represented with some form of big integer package unless a data type of this sort is included in the language.

An alternative that merits mention is the choice made by Müller [15] in his implementation. There, the mantissa together with the exponent is represented by an arbitrary precision floating point number. The error term then needs to be handled separately. We have chosen the form here with explicit exponents since we have some theoretical points to explain, there might also be some minor gains to be made by having the error term as well-defined as it is in our approach. On the other hand, the floating point approach can take advantage of already implemented transcendental functions and so on.

3 Computing an exact real operation

Let f be an operation on the reals. We would like to implement this operation as a part of exact real arithmetic. We will for simplicity often assume that f is a unary function.

The call to the operation f will have as arguments a function g that computes arbitrarily good approximations of the input x and an integer p specifying which element in the Cauchy sequence is to be returned, i.e., that a p -approximation of the result is desired.

Algorithm 1. To compute a p -approximation of $f(x)$.

1. Choose q .
2. Compute a q -approximation $a = (m \pm e)2^{-s}$ of x .
3. Compute $m' = f(m)$ and an error term e' from a .
4. If a p -approximation $a' = (m'' \pm e'')2^{-t}$ of $f(x)$ can be constructed from m' and e' , then return a' .
5. Increase q and repeat from 2.

Compare this algorithm with Figure 2. We note that the operations may return an approximation where the scaling factor, t above, is greater than p (and, in fact, has to be larger if the error e'' is allowed to be greater than 1).

The choice of q in the first step of the algorithm is arbitrary, and if care is not taken, it may result in very poor performance. If q is taken larger than necessary and the computation of x is expensive compared to

the computation of f (for example, $x = e^{1000}$ and f the identity function) then much time may be wasted in computing a good approximation of x . Similarly, if f is hard to compute compared to x (for example, $x = 2$, and f the logarithm function) and q is chosen too small, and later increased in too small steps in step 5, then much time may be wasted computing f on approximations where the result has to be thrown away.

Is there a way to compute a good value of q ? Yes, in some cases, there is. Given some initial approximation of the inputs it is often possible to compute a sufficient accuracy of the input to guarantee that the output will have the desired accuracy. For example, if an approximation of the input to the reciprocal function is away from zero we can, using our knowledge of the reciprocal function, compute an input accuracy that will guarantee that the answer is accurate enough.

Definition 3.1. A *first approximation* for a function f is an approximation a of the input such that $f'(x)$ is bounded for all x approximated by a .

For example, any approximation a such that 0 is not approximated by a is a first approximation of the reciprocal function.

First approximations are very common for real functions. The real line, and hence any approximation, is a first approximation for sine and cosine. Likewise for the two arguments of addition. For multiplication it is sufficient that both arguments are bounded for it to be a first approximation.

Using first approximations we can modify Algorithm 1.

Algorithm 2. To compute a p -approximation of $f(x)$.

0. Generate approximations of x until a first approximation is found.
1. Compute q from the first approximation.
2. Compute a q -approximation $a = (m \pm e)2^{-s}$ of x .
3. Compute $m' = f(m)$ and an error term e' from a .
4. Construct a p -approximation $a' = (m'' \pm e'')2^{-t}$ of $f(x)$, and return it.

The unbounded iteration that may occur in the Algorithm 1 has in the Algorithm 2 been confined to the search for a first approximation in step 0.

Algorithms 1 and 2 are the basis for those used in the exact real arithmetic systems constructed by Müller [15] and Lester [10] respectively. Müller applies his algorithm bottom-up in the expression tree while Lester applies his algorithm top-down. The choice of bottom-up or top-down is natural given the nature of the algorithms. In fact, Müller's implementation makes

use of having found a first approximation in the computation of the new bound q in step 5, thus often avoiding more than one recomputation of the function f . Hence, the boundary between the two algorithms need not be as strict as presented.

The advantage of using Algorithm 2 is that reevaluations using f need not be made. However, there may still occur reevaluations of x within step 0. The risk of getting something which is not a first approximation is often very low, in fact, in one sense the probability is often zero. For example, allowing only approximations where the error is 1, for the case of the reciprocal function there are only finitely many (three) approximations with scaling factor p that contain 0, whereas infinitely many approximations avoid 0. Of course, one can also argue that the three approximation with scaling factor $p = 1$ that contain 0 covers half the Riemann circle, and therefore that the probability is about one half for approximations with scaling factor 1. Still, for approximations with scaling factor $p = n$ one would then get a probability $O(2^{-n})$, which is very small for reasonable n .

One drawback of using Algorithm 2 is that the computed q , although it may be optimal for arbitrary input, might be greater than necessary for certain inputs as shown below.

4 Constructing approximations

An important part in the algorithms presented is constructing approximations that are as good as possible given some intervals of reals that must be approximated by the approximations. We start with the simplest operation of constructing an approximation with a bounded error term from an approximation that has a too large error term.

Example 4.1 (Rounding). Suppose we are given a dyadic number $m \cdot 2^{-s}$ and an error term $e \cdot 2^{-s}$. What is the best possible approximation of the form $(n \pm e')2^{-t}$, where e' is bounded, that can be returned? (By *best* we mean the largest possible value for the scaling factor t .)

Consider first the case where e' is to be 1. If $e \leq 2^{k-1}$ then $(n \pm 1)2^{-s+k}$, where $n = \text{round}(m/2^k)$, that is, the nearest integer to $m/2^k$, is a correct approximation. Since n is the nearest integer to $m/2^k$ the difference is less than or equal to $\frac{1}{2}$. The error $e/2^k$ is also less than or equal to $\frac{1}{2}$ so the total error is less than $\frac{1}{2} + \frac{1}{2} = 1$ which is the error margin given in $(n \pm 1)2^{-s+k}$.

The worst case is when the fractional part of $m/2^k$ is exactly $\frac{1}{2}$. It is clear that for any $e > 2^{k-1}$ the interval approximated by $(m \pm e)2^{-s}$ is not covered by $(n \pm 1)2^{-s+k}$ for any n . On the other hand, if m is a multiple of 2^k then $(m/2^k \pm 1)2^{-s+k}$ is a correct approximation for any $e \leq 2^k$.

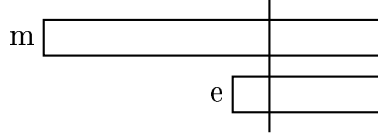


Figure 3: Rounding an approximation $(m \pm e)2^{-s}$.

Thus, if the error is less than or equal to $\frac{1}{2}$ “in bit k ” then we can always find a correct approximation by rounding k bits. Sometimes, this is possible even if the error is between $\frac{1}{2}$ and 1 “in bit k ”.

Consider now the case where e' is to be strictly bounded by 2^j , i.e., fit within j bits. Figure 3 indicates that what needs to be done is to cut both m and e off so that the new error term is small enough. A cut of at least $k = q - j$ bits is required, where q is the number of bits in the representation of e . Let n be the result of rounding $m/2^k$ to the nearest integer. Then an error is introduced that must be added to e . We may now create e' by rounding upwards the new error term divided by 2^k . The rounding may result in e' needing more bits to represent than e , therefore, it is sometimes necessary to repeat the process after increasing the value of k .

Computing the actual rounding made in the computation of n above is important if j is small, 1 or 2. For larger j (> 10), it is often better to approximate this rounding by the worst case of rounding one half, since this very seldomly will be the cause of an overflow in e' .

We now move on to the construction of approximations of the result of operations. The first example is addition.

Example 4.2 (Addition). Let us assume that all approximations have an error term of 1. As already stated above the value q in step 1 of Algorithm 2 may be computed from p without references to first approximations. In fact, letting $q = p + 2$ is sufficient.

Suppose the approximations for x and y are $(m \pm 1)2^{-q}$ and $(n \pm 1)2^{-q}$ respectively. The sum of these approximations is

$$(m \pm 1)2^{-q} + (n \pm 1)2^{-q} = ((m + n) \pm 2)2^{-p-2}.$$

If $m+n$ is even then the above is equal to $(\frac{1}{2}(m+n) \pm 1)2^{-p-1}$ which actually can be used as a $(p+1)$ -approximation of the sum. If $m+n$ is odd then the best possible approximation with an error term of 1 is if $\frac{1}{4}(m+n)$ is rounded to the nearest integer; this yields the p -approximation $(\text{round}(\frac{1}{4}(m+n)) \pm 1)2^{-p}$. Thus, about half of the additions “lose” one bit of precision and the other

half will “lose” two bits of precision. The computation of q in Algorithm 2 will have to allow for a loss of two bits.

We will use the terminology that k bits are *lost* when the result after a computation is a $(p - k)$ -approximation given that the inputs were p -approximations.

If many additions are performed then the size needed from the input arguments might be largely overestimated. However, for a simple sum of the form $\sum_{i=1}^{1000} x_i$ a p -approximation of the sum can be computed from $(p+1)$ -approximations of the inputs. The error is $1000 \cdot 2^{-11} < \frac{1}{2}$ for the sum which guarantees correct rounding to a p -approximation. Thus, a real problem only arises when the additions are interspersed with other operations.

Allowing the error to be different from 1 in the approximation of the sum is clearly one way to avoid the rounding that is otherwise necessary for odd results.

Example 4.3 (Multiplication). As for addition, we assume that all error terms are 1. The product of the approximations $a = (m \pm 1)2^{-s}$ and $b = (n \pm 1)2^{-t}$ is

$$(mn \pm (m + n + 1))2^{-s-t}.$$

Thus, if we can find first approximations so that m and n are bounded by 2^k and $2^{k'}$ respectively, then the error term $m + n + 1$ is bounded by $2^k + 2^{k'} + 1$. So, if $k = k'$ then the best bound for the error term is $2^{k+1} + 1$, which means that rounding has to be made of the last $k + 3$ bits in the result mn to get an error term of 1. Let $k = \min(s, t)$, then the result after rounding is a $(k - 3)$ -approximation, so we have a loss of 3 bits. Note, that in almost all cases rounding $k + 2$ bits is sufficient, but the single worst case when $m = n = 2^k$ requires us to always round $k + 3$ bits.

If $k \neq k'$ then rounding of $\max(k, k') + 2$ bits is always sufficient. Again, for many examples it is actually sufficient to round $\max(k, k') + 1$ bits.

4.1 Generalised error terms in approximations

Sofar, we have often assumed that the error term in an approximation is 1. By generalising the error term to take on other values we may reduce the number of times rounding make it necessary to throw bits away. The generalised error terms are related to the notion of guard bits in floating point computations. Another advantage of generalising the error term is that we may use an error term of 0 to denote an exact dyadic number.

Consider again Example 4.2. There we claimed that if the computed sum was odd, then rounding was necessary to get an approximation with an error

term of 1. Given two p -approximations $a = (m \pm e)2^{-s}$ and $b = (n \pm e')2^{-s}$, their sum is

$$a + b = (m + n \pm (e + e'))2^{-s},$$

which is a $(p - 1)$ -approximation since $e + e' \leq 2^{s-p+1}$ and may be a p -approximation or better if at least one of a and b is a $(p + 1)$ -approximation or better. So by generalising the error term we never lose more than one bit regardless of whether the result is odd or even.

However, this must come at a cost, and it does. The error term may eventually grow to be large compared to the mantissa of the approximation. Hence, we have lost the bound on the storage size of an approximation even if we only consider a compact interval. Furthermore, the handling of the error term may add significantly to the cost of each operation.

These considerations, make it desirable to bound the error term. For example, the error term may be required to fit within one computer word. With this scheme some extra cost for operations is still incurred, but it ought to be balanced by the reduced size of the mantissas used to compute a result of a certain precision. Also, bounding the size of the error term limits the exponent s needed in a p -approximation $a = (m \pm e)2^{-s}$ to $p + \lceil \log_2 e \rceil$, solving our first problem.

It may be feared that bounding the error terms would eliminate the gains demonstrated above. However, the benefits are not entirely lost. For example, adding one thousand $(p + 11)$ -approximations will automatically give a p -approximation if the bound on the error terms is at least 2^{11} . This holds even if the naive method of using a loop and a partial sum is used to compute the sum. On the other hand, there will always be situation when it turns out that having a bound results in a loss of an “unnecessary” bit occasionally. Computing the effect of generalised but bounded error terms is a statistical problem that depends on the bit patterns of the partial results in the computation.

The bound for the error term might be taken to depend on the size of the mantissas. This has not been considered here. Also, the optimal value of the bound should be investigated.

5 Conclusions

The dyadic representations presented here have merits as pointed out in this article. Mainly, limited storage requirement and efficient implementation of all operations. They seem to be the most viable choice, which is also supported by the results in [4]. The generalisation of the error terms may

be important in implementing exact real arithmetic since it may reduce the size of the approximations used in the computation.

The choice between the two algorithms presented is open when looking at the results. But the conclusion here is that for small expressions (few operations to be performed) calculated to high precision, Algorithm 2 should be better. But when many operations are to be performed, in particular if the resulting precision is low, then Algorithm 1 may be the better choice. Utilising generalised error terms is easier in Algorithm 1 and this will to some extent influence the choice algorithm. A further study of this is clearly necessary.

References

- [1] O. Aberth. *Computable Calculus*. Academic Press, 2001.
- [2] J. Blanck. Domain representability of metric spaces. *Annals of Pure and Applied Logic*, 83:225–247, 1997.
- [3] J. Blanck. Effective domain representations of $\mathcal{H}(X)$, the space of compact subsets. *Theoretical Computer Science*, 219:19–48, 1999.
- [4] J. Blanck. Exact real arithmetic systems: Results of competition. In Blanck et al. [5], pages 390–394.
- [5] J. Blanck, V. Brattka, and P. Hertling, editors. *Computability and Complexity in Analysis*, volume 2064 of *Lecture Notes in Computer Science*. Springer, 2001.
- [6] L. Blum, M. Shub, and S. Smale. On a theory of computation and complexity over the real numbers: Np-completeness, recursive functions, and universal machines. *Bulletin of the American Mathematical Society*, 21:1–46, 1989.
- [7] J. M. Borwein and P. B. Borwein. *Pi and the AGM: A Study in Analytic Number Theory and Computational Complexity*. John Wiley & Sons, 1998.
- [8] G. Chaitin. Incompleteness theorems for random reals. *Adv. in Appl. Math.*, 8:119–146, 1987.
- [9] A. Edalat and P. J. Potts. A new representation for exact real numbers. *Electronical Notes in Theoretical Computer Science*, 6:14 pp., 1997.

- Mathematical foundations of programming semantics (Pittsburgh, PA, 1997).
- [10] P. Gowland and D. Lester. A survey of exact computer arithmetic. In Blanck et al. [5], pages 30–47.
 - [11] A. Grzegorzcyk. Computable functionals. *Fundamenta Mathematicae*, 42:168–202, 1955.
 - [12] R. Heckmann. The appearance of big integers in exact real arithmetic based on linear fractional transformations. In *Foundations of Software Science and Computation Structures*, volume 1378 of *Lecture Notes in Computer Science*, pages 172–188. Springer, 1998.
 - [13] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, 1969.
 - [14] D. Lacombe. Extension de la notion de fonction récursive aux fonctions d’une ou plusieurs variables réelles i, ii, iii. *Comptes Rendus de l’Académie des Sciences, Série A*, 1955. vol. 240, 2478–2480, and vol. 241, 13–14, 151–153.
 - [15] N. Müller. The iRRAM: Exact arithmetic in C++. In Blanck et al. [5], pages 223–252.
 - [16] M. B. Pour-El and J. I. Richards. *Computability in Analysis and Physics*. Perspectives in Mathematical Logic. Springer, Berlin, 1989.
 - [17] A. Schönhage and V. Strassen. Schnelle multiplikation grosser zahlen. *Computing*, 7:281–292, 1971.
 - [18] V. Stoltenberg-Hansen and J. V. Tucker. Effective algebra. In S. Abramsky et al., editors, *Handbook of Logic in Computer Science*, volume IV, pages 357–526. Oxford University Press, 1995.
 - [19] J. V. Tucker and J. I. Zucker. Abstract versus concrete models of computation on partial metric algebras. *ACM Transactions on Computational Logic*. To appear.
 - [20] J. V. Tucker and J. I. Zucker. Computable functions and semicomputable sets on many sorted algebras. In S. Abramsky et al., editors, *Handbook of Logic for Computer Science*, volume V, pages 317–523. Oxford University Press, 2000.

- [21] J. Vuillemin. Exact real computer arithmetic with continued fractions. INRIA Report 760, INRIA, France, 1987.
- [22] K. Weihrauch. *An Introduction to Computable Analysis*. Springer, 2000.