

Efficient exact computation of iterated maps

J. Blanck*

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

December 29, 2002

Abstract

It is possible to effectively compute the forward orbit of iterated maps contrary to often held beliefs that rounding errors and sensitivity on inputs make this impossible. However, exact real arithmetic can compute the forward orbit of the logistic map and many other maps using linear space resources and $O(n \log n M(n))$ time, where $M(n)$ is the time it takes to multiply two numbers of n bits, and n is the number of iterations to be computed.

Some insights into implementation issues of exact real arithmetic is arrived at, and tested successfully in actual computations. In particular, it is found that bottom-up propagation of error terms is likely to be preferable in involved computations. This will allow for exact real computations that run within some constant factor of the time for the corresponding floating point computation when the computation is stable. Moreover, the exact real computation correctly handles unstable computations and deliver a correct answer using more time and space resources.

1 Introduction

Systems for general purpose exact real arithmetic have been implemented by several people, see for example [14, 10]. A competition between such systems was held in 2000 [2]. Most problems for the competition were to compute some moderately sized expression with very high precision. However, one problem was the computation of a thousand iterations of an iterated map to

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

modest accuracy. The latter problem seemed to point to important questions regarding implementation techniques and is therefore the basis for the study made herein.

We will use iterative maps as a test of the applicability of exact real arithmetic. In particular, we will use exact real arithmetic to compute the forward orbit of an iterative map.

An *iterative map* is a map f on some space X . The *forward orbit* of a point x_0 are all the points $x_n = f^n(x_0)$, where n is a natural number and f^n is defined by $f^0(x) = x$ and $f^{n+1}(x) = f(f^n(x))$.

Iterative maps was chosen as a case study since they are notoriously hard to compute because of their *chaotic* behaviour. Moreover, they require many basic operations to be performed. This matches the use of floating point arithmetic and will hopefully give a more useful indication of the performance of exact real arithmetic compared to evaluating small expressions to enormous precision. The comparison of exact real arithmetic and floating point arithmetic will nevertheless be halting, since floating point arithmetic cannot be used to calculate many iterations reliably. In fact, using floating point, even a hundred iterations might generate completely unreliable results.

The map we will try to evaluate is the quadratic map

$$f_c(x) = cx(1 - x),$$

defined on the unit interval, also known as the *logistic map*. The chosen map is extremely simple but exhibits the chaotic behaviour typical of iterative maps [8, 15]. It is known that the map is *chaotic* on the unit interval for $c = 4$. Periodic points are dense within the unit interval, but the map is sensitive to inputs and is topologically transitive.

The simplicity of the chosen map makes it possible to study the map without too much trouble, and since it only uses elementary operations it is also fairly easy to implement programs that do the computations. Generalisations of this study to other iterative maps should not be hard to do, but would probably not affect the claims made here.

Exact real arithmetic can be implemented in various ways, two of which seem to be more promising. Choosing between the two implementation styles presented later is clearly important but has so far not been investigated carefully. We have a strong indication that one of the algorithms will be more efficient in general computations.

We will also challenge the view that it is infeasible to compute orbits of chaotic functions. For example, Devaney expresses this view in [8, p. 49].

If a map possesses sensitive dependence on initial conditions, then for all practical purposes, the dynamics of the map defy numerical

computation. Small errors in computation which are introduced by round-off may become magnified upon iteration. The results of numerical computation of an orbit, no matter how accurate, may bear no resemblance whatsoever with the real orbit.

We claim that exact arithmetic can be used for such computations. The cost is just not linear in the number of iterations that are to be computed.

Section 2 will give a short explanation of the notion of exact real arithmetic. Section 3 will explain the approximations used in the implementations. Section 4 will explain two basic algorithms to perform exact computations. Sections 2–4 are more fully explained in [3]. The following sections will contain the particulars of the case study.

2 Exact real arithmetic

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 [7].

The theoretical foundations for this approach to a real data type is the subject Computable Analysis [16, 18, 1]. There exist several flavours of computable analysis; we will use the one based on the definitions of Grzegorzczuk and Lacombe [11, 13]. 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, the Blum–Shub–Smale model of computations on the reals [5]. The most notable difference is that equality and the ordering are not computable operations in computable analysis.

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.

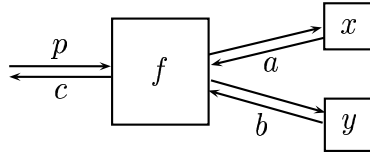


Figure 1: A binary exact real operation.

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 1. The user would input/supply the boxes x and y together with the desired precision p to the operation f , 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 .

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

3 Approximations

The approximations used in any exact real computations must be carefully selected. The approximations chosen here are dyadic approximations of the form

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

where $m, s \in \mathbb{Z}$, and $e \in \mathbb{N}$. In our programs for iterated maps later on, the value e is actually bounded. In fact, e may be fixed to be 1, but we will see that the more general form is useful. These approximations can either be seen as dyadic intervals or as floating point numbers with error bounds. The inclusion of the error terms in the approximation is what will give us the ability to claim that we are doing exact real arithmetic.

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

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

e.g., if $k = \lceil \log_2 e \rceil$ then $(m + e)2^{-p-k}$ is a p -approximation. Approximations are often implicitly assumed to be no better than stated, i.e., a p -approximation is in general not a $(p + 1)$ -approximation.

In order to compare the precision of the input and the output of operations the following terminology is used.

Definition 3.1. If a unary operation takes an p -approximation as input and returns a q -approximation of the output, then the operation resulted in a *loss of $p - q$ bits*. If $p - q$ is negative it is more natural to write a *gain of $q - p$ bits*.

Since efficient algorithms exist for all common operations on arbitrary precision floating point numbers [6, 12], and since the approximations are very similar to floating point numbers, the existing algorithms can be used to efficiently compute on the chosen approximations. For example, the Schönhage–Strassen method [17] can be used to compute multiplication in time $O(M(k))$, where $M(k) = k \log k \log \log k$ and k is the size of the approximations.

4 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 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 the desired precision of $f(x)$, that is, a p -approximation is sought.

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

1. Choose q .
2. Compute a q -approximation $a = g(q) = (m \pm e)2^{-s}$ 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 1. 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 4.1. 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 them to be first approximations.

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 Algorithm 1 has in 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 [14] 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.

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 but finding a first approximation can be expensive in certain cases. Another drawback of using Algorithm 2 is that the computed q , although it may be optimal for arbitrary input, might be greater than necessary for the actual inputs as shown in the example below.

Example 4.2 (Addition). Let us assume that all approximations have an error term of 1. As observed, the real line is a first approximation, so the value q may be computed directly from p . 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 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.

Thus, if many additions are performed then the size needed from the input arguments might be largely overestimated. Similar behaviour exist for 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. However, this does not solve the entire problem.

The conclusion to be drawn 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.

5 Generalised error terms

The error term of an approximation can always be chosen to be 1, often giving an approximation that covers a larger interval. An error term of 0 is

useful to represent exact dyadic numbers whenever they occur. Letting the error terms take on other positive values is an interesting option.

Generalising the error terms reduce the number of times that rounding unnecessarily lose bits. Consider again Example 4.2. There it was claimed that if the computed sum was odd, then rounding of 2 bits 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 $e + e' \leq 2^{s-p+1}$. 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. Thus, there no longer exist an efficient way of demanding an approximation that is (at least) of a certain precision. Furthermore, the handling of the error terms may add significantly to the cost of each operation.

These considerations, make it desirable to bound the error terms. For example, to bound the error terms to fit within a fixed number of bits. Consider now addition where error terms may occupy 3 bits, i.e., range between 0 and 7. Adding $(m \pm e)2^{-s}$ and $(n \pm e')2^{-s}$ we have that if $e + e' \leq 7$ then

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

is an approximation of the sum, and if $e + e' \leq 13$ then

$$\left(\text{round} \left(\frac{m + n}{2} \right) \pm \left\lceil \frac{e + e' + b}{2} \right\rceil \right) 2^{-s+1}$$

(where b is 1 if rounding of the mantissa is needed, i.e., if $m + n$ is odd, and 0 otherwise) is an approximation of the sum. Thus, losing 0 or 1 bit. The remaining case $e = e' = 7$ lose 1 bit if $m + n$ is even since then

$$\left(\frac{m + n}{2} \pm 7 \right) 2^{-s+1}$$

is an approximation of the sum, otherwise 2 bits are lost. The probability of losing 2 bits is clearly reduced compared to having a fixed error term of 1. Assuming a uniform distribution of the values of error terms, it is clear that the number of unnecessarily lost bits due to rounding decreases rapidly with increasing bounds on the error terms.

With bounded generalised error terms some extra cost for operations is still incurred, but it ought to be balanced by the reduction of lost bits due to rounding, i.e., reducing the size of the mantissas used in computing a result of a specified precision. Also, bounding the size of the error terms only adds constant additional memory space for the approximations.

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.

Generalised error terms are only feasible in a system that propagates error terms bottom-up in the expression tree. When propagating required precision top-down in the expression tree it makes little sense to be able to demand an approximation with a generalised error term. A best possible approximation of the value is desirable, but there is no a-priori reason that approximations with a particular error term of, say 5, should be better than approximations with an error term of 1.

6 Implementing the logistic map

An iterative map sensitive to inputs will lose precision during a computation of a forward orbit of the map. Thus, it is important to investigate the loss of precision that each iteration may result in. We start with general bounds (that is, across all approximations of a given precision). The general bounds are needed if first approximations are to be used to compute the necessary input precision to guarantee an appropriate approximation of the orbit. General bounds are therefore needed for Algorithm 2. Looking at particular input values the bounds may be improved. This is of interest in conjunction with Algorithm 1 if the error terms in the approximations are dynamically computed to be as tight as possible.

In general, we must consider approximations of the iterative map, since the function may not map (dyadic) approximations to (dyadic) approximations. For the logistic map, rational (dyadic) numbers are mapped to rational (dyadic) numbers, if the constant c is rational (dyadic). Thus, we may compute the logistic map exactly on the approximations. Therefore, we will avoid the discussion of approximating the function.

6.1 Static analysis

Let us look at the computation of f_4 in terms of accuracy needed from the argument in order to get a p -approximation of the output. Recall that the

maps are defined on the unit interval and that therefore the centre of any approximation is assumed to be within the closed unit interval.

We start by considering the operations $x \mapsto 4x$ and $x \mapsto 1 - x$ in the general case where e is an arbitrary (natural) number.

Lemma 6.1. *The map $x \mapsto 4x$ loses 2 bits.*

Proof. Given any approximation $(m \pm e)2^{-s}$ of a number x we have that $(m \pm e)2^{-s+2}$ is a correct approximation of $4x$. \square

Lemma 6.2. *The map $x \mapsto 1 - x$ loses zero bits.*

Proof. Given any approximation $(m \pm e)2^{-s}$ of a number x we have that $(2^s - m \pm e)2^{-s}$ is a correct approximation of $1 - x$. \square

The lemma above holds for other constants than 1. In fact, it holds for any dyadic number with denominator less than or equal to the denominator of the approximation of x .

For the multiplication of the two factors x and $1 - x$ we consider here only the case when approximations have an error term equal to 1. This is acceptable since generalised error terms is difficult to use in the top-down approach.

For multiplication we need a first approximation. We have assumed that the logistic map only is defined on the unit interval, hence the unit interval itself can be used as a first approximation.

$$(m \pm 1)2^{-s} \cdot (n \pm 1)2^{-s} = (mn \pm m \pm n \pm 1)2^{-2s},$$

where $0 \leq m, n \leq 2^s$, hence the error term is $|m| + |n| + 1 \leq 2^s + 2^s + 1 \leq 2^{s+2}$. This can be rounded into a correct approximation of the form $(m' \pm 1)2^{-s+3}$.

To compute a p -approximation $(m' \pm 1)2^{-p}$ of f_4 it is sufficient to have a $(p + 5)$ -approximation $(m \pm 1)2^{-p-5}$ of x , resulting in a loss of 5 bits per iteration.

Applying more external knowledge it is easy to see that either m or n in the multiplication of x by $1 - x$ is bounded by 2^{s-1} . Thus the rounding can in fact be done correctly losing only 2 bits instead of 3.

Even more can be achieved by considering the operation $x \mapsto x(1 - x)$ as one basic operation.

Proposition 6.3. *The logistic map f_4 loses 3 bits.*

Proof. Let $(m \pm 1)2^{-s}$ be an approximation of x . Then

$$(m \pm 1)2^{-s} \cdot (2^s - m \mp 1)2^{-s} = (m2^s - m^2 - 1 \mp 2m \pm 2^s)2^{-2s},$$

where the error term is bounded by $|2^s - 2m| \leq 2^s$. An $(s-1)$ -approximation can therefore be found of $x(1-x)$. Hence, only 1 bit is lost in computing $x(1-x)$ and by Lemma 6.1 the total loss for the map is 3 bits. \square

There exist cases where a loss of 3 bits is unavoidable, for example, for an approximation of the form $(2 \pm 1)2^{-s}$ (an s -approximation of a number close to zero, but non-zero). The image of the interval represented by such an approximation is $[(2^s - 1)2^{-2s}, (3 \cdot 2^s - 9)2^{-2s}]$ which does not fit into any s -approximation (with an error term of 1), hence a loss of 1 bit. The multiplication by 4 lose a further 2 bits resulting in a total loss of three bits. Thus, the above result is sharp.

Note, that implementing a combined operation, as in the proof of Proposition 6.3, saves 1 bit compared to using the basic operations, regardless of whether all available external knowledge is used. This situation is unfortunate with regard to implementing a data type for exact real arithmetic, since implementing combined operations would have to be left to the programmer.

Consider the map f_c for other values of $c \leq 4$. The only alteration is in the analysis of the multiplication by c . Assume that $(n \pm 1)2^{-t}$ is an approximation of c , and that the approximation of x is $(m \pm 1)2^{-s}$. Furthermore, assume that $t \geq s$, since this is the case when the constant c is evaluated to at least the accuracy of the argument x . Then, cx is approximated by

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

The error term is bounded by

$$|n| + |m| + 1 \leq 4 \cdot 2^t + 2^s + 1 \leq 5 \cdot 2^t + 1 \leq 2^{t+3},$$

since $t \geq s$. Thus, a correct approximation of the form $(m' \pm 1)2^{-s+4}$ can be found, i.e., a loss of 4 bits. The logistic map f_c lose at most 5 bits for any constant $c \leq 4$.

The analysis of the iterative map f_c above can be substantially improved if good approximations of x already has been computed. For example, if it is already known that $(2^{s-1} \pm 1)2^{-s}$ is an approximation of x then $f'(x) = c - 2cx$ is bounded by $c2^{-s+1}$ on that interval. If $c \leq 4$ we have that to compute a p -approximation of $f_c(x)$ in the presence of such an approximation of x it is sufficient to provide a $p - s + 4$ -approximation of x , which corresponds to a gain of $s - 4$ bits. Hence, a much tighter error term can be computed compared to the error term computed above. However, the assumption that such approximations already exist is clearly not very plausible, since it more or less require the forward orbit to be precomputed. A much more plausible method is to do such an error computation dynamically after each iteration step, rather than do the error computation before each iteration step. This will be investigated in the next section.

6.2 Dynamical analysis

Here we will study how tight the error term can be made for particular values of the input to an operation. This is in contrast to the analysis made above of sufficient error terms for any input.

Example 6.4. Consider the computation of $x \mapsto x(1-x)$ if $(2^{s-1} + 1 \pm 1)2^{-s}$ is an approximation of x . Then

$$\begin{aligned} (2^{s-1} + 1 \pm 1)2^{-s} \cdot (2^{s-1} - 1 \mp 1)2^{-s} &= ((2^{2s} - 1) - 1 \mp 2)2^{-2s} \\ &= (2^{2s} - 2 \pm 2)2^{-2s} \\ &= (2^{2s-1} - 1 \pm 1)2^{-2s+1}, \end{aligned}$$

where the last equality holds by dividing through by 2. Note that the product of ± 1 and ∓ 1 is computed to -1 since it is known that the error in x and $1-x$, if non-zero, will have opposite signs. It is in fact the proper correction so that the computed midpoint of the approximation lies exactly on the midpoint of the image interval (except when the midpoint of the input approximation is on the critical point $\frac{1}{2}$). Thus instead of losing 1 bit which is the general case for the unit interval, this particular approximation results in a gain of $s-1$ bits.

The difference of losing 1 bit compared to gaining $s-1$ bits is spectacular, but not representative across the unit interval. The average loss of bits is difficult to establish since it depends on the bit pattern of the result, the magnitude of the input, and on the error terms of the input. Based on observations, more than one half of the approximations result in zero lost bits. Thus, it should be worthwhile to implement this scheme.

A further advantage to this scheme is that it is easy to combine with generalised error terms within approximations, that is, the e in $(m \pm e)2^{-s}$ is allowed to take on other values apart from 1. This has the advantage of reducing the number of roundings that destroy useful information.

We now look at the implementation of the logistic map with bottom-up propagation of error terms in order to minimise the number of lost bits per iteration, and hence minimising the size of the approximations used in the computation.

Using Algorithm 1 a constant q is chosen and q -approximations of the input is computed. The operations are performed bottom-up in the expression tree, i.e., for iterated maps the first iteration is the first step to be considered. If the computation at some stage has lost too much precision to be useful for further computations the constant q is increased and the computation restarted.

Instead of asking for q -approximations, our implementation works with a limit on the exponent p of any approximation $a = (m \pm e)2^{-p}$. The limit is universal for the expression tree, in particular, all leaves are evaluated to the same precision. At nodes in the expression tree the operation is performed and if the resulting approximation has an exponent over the limit the result is rounded to an approximation with an exponent not greater than the limit. This rounding keeps the size of approximation from growing. However, the error terms grow because of this.

We present pseudo-code for the central parts of the computation of the logistic map. Note that most integer computations involve arbitrarily large integers and hence must be performed with some package for multiple precision integer arithmetic. The implementation we have is based on the GMP package [9].

The function `mul` multiply two approximations $(m \pm e)2^{-s}$ and $(m' \pm e')2^{-t}$ and returns an approximation of the result.

```

e'' = e + e' + ee';
n = mm';
return round_and_limit_approx((n ± e'')2-s-t);

```

Here, `round_and_limit_approx` implements the rounding of the approximation and limits the exponent to the current limit of exponents.

The function `one_minus` computes $1 - x$, where $(m \pm e)2^{-s}$ is an approximation of x .

```

n = 2s - m;
return round_and_limit_approx((n ± e)2-s);

```

Note that we utilise the fact that the number 1 has an exact dyadic representation in $(2^s \pm 0)2^{-s}$. This avoids losing any precision in the computation in contrast to the case of subtraction (or addition) of arbitrary approximations.

Thus, the logistic map can be computed using the following code where a is an approximation of the input x , and c is the constant in the logistic map.

```

mul(c, mul(a, one_minus(a)))

```

(1)

It is possible to narrow the error term if the operation $g: x \mapsto x(1 - x)$ is implemented as a basic operation. The following is pseudo-code for `g` to compute an approximation of $x(1 - x)$ if x is approximated by $a = (m \pm e)2^{-s}$.

```

if (|2s - 2m| ≥ e) {
  e' = e|2s - m|;
  n = m(2s - m) - e2;
}

```

```

} else {
  e' =  $\lceil \frac{1}{2}e(|2^s - 2m| + e) \rceil$ ;
  n =  $\frac{1}{4}2^{2s} - e'$ ;
}
return round_and_limit_approx((n  $\pm$  e')2-2s)

```

The then-branch is executed when the approximation a does not contain the critical point $1/2$ of the map.

The logistic map can now be computed using the following code.

$$\text{mul}(c, \text{g}(a)) \tag{2}$$

7 Implementing iteration

Let f be the function that is to be iterated within an exact arithmetic framework. We assume that there exist operations taking dyadic numbers as input and give arbitrarily good approximations of the image of the input under f .

7.1 Top-down propagation of required precision

Evaluating an exact expression top-down (which is more in the spirit of Algorithm 2) suggests representing f by a computable operator A_f taking two arguments, the argument x , and a number q , and returning a q -approximation of $f(x)$. Recall that the number x is really given as a process, that compute arbitrarily good approximations of the number. The operator A_f must somehow determine a q' , compute a q' -approximation a' of x , and finally compute a q -approximation a of $f(x)$ using the approximation a' . For an arbitrary f there does not exist any way of computing q' from q . For particular f and for particular ranges of arguments, there may exist ways of computing q' from q . In particular, for the logistic map f_4 considered here and arguments within the unit interval, we have seen above that it is sufficient that q' is taken to be $q + 3$. Let $a' = (m \pm 1)2^{-q-3}$ be a $(q - 3)$ -approximation of x , then

$$a = (\text{round}(f_4(m2^{-q-3})/2) \pm 1)2^{-q}$$

is a q -approximation of $f_4(x)$ by Proposition 6.3.

To evaluate the n -th iteration of A_f bottom-up the following has to be computed.

$$\begin{aligned} x_1 &= A_f(x_0, q_1), \\ x_2 &= A_f(x_1, q_2), \end{aligned}$$

$$\begin{aligned} & \vdots \\ x_{n-1} &= A_f(x_{n-2}, q_{n-1}) . \\ x_n &= A_f(x_{n-1}, q) . \end{aligned}$$

In general the numbers q_i are not known in advance, and if that is the case it is very difficult to evaluate this expression effectively.

For a general f the evaluation will proceed as follows. Since no approximations of the intermediate values exist yet the computation will look for first approximation of every x_{n-1}, \dots, x_1 in turn. Since x_0 is known, a first approximation of x_1 can be found by computing approximations of x_1 for larger and larger values of q_1 . This can now be repeated for x_2, \dots, x_{n-1} in turn. However, if precision is lost every iteration, the computation will repeatedly find that the approximations computed for x_1, \dots, x_{i-1} are, almost, but not quite, good enough to compute x_i , and hence the computation need to restart from the beginning. Using first approximations in this case is actually an Achilles' heel, since it will compute x_i only to the precision that is immediately needed.

Of course, for the logistic map, using the available external knowledge, it is possible to assign values to q_i in advance. In particular, x_i can be computed by $A_f(x_{i-1}, q + 3(n - i))$ without risking any recomputations of intermediate values.

An iterator I that is suitable for top-down evaluation can be given if there exists a computable function g that given a precision computes an appropriate precision for the input. The iterator I takes three arguments, the number n of iterations to be performed, the initial starting value x_0 , and a precision q for the final result. This can now be coded as follows.

```
for (i = 0; i < n; i++) {
    x_{i+1} = A_f(x_i, g^{n-i}(q));
}
```

Lemma 7.1. *If the iterated map f is continuous and maps a bounded interval into itself, then the loss of bits per iteration can be bounded by a constant.*

Proof. The map f is uniformly continuous, and can be arbitrarily well approximated on dyadic numbers. \square

Proposition 7.2. *If the loss of bits per iteration can be bounded by a constant, then $O(n)$ space is sufficient to compute n iterations.*

Proof. Let k be the maximum number of bits lost in one iteration. If a p -approximation of the final value x_n is sought, then a $(p + kn)$ -approximation of the input x_0 is needed. An approximation of the i -th iteration x_i can be computed using only an approximation of x_{i-1} . Hence, all previous approximations can be thrown away after each iteration. \square

Consider the elementary functions consisting of rational functions, log, exp, and any function obtained from these by composition, multiplication, addition, and solutions of algebraic equations.

Theorem 7.3. *Let f be an elementary map without singularities taking a compact interval into itself. Then, the time complexity of computing n iterations of f is $O(n \log n M(n))$.*

Proof. By [6, Theorem 7.3] the bit complexity of the map f is $O(\log n M(n))$. By Lemma 7.1 the loss of bits per iteration can be bounded by a constant. Thus, giving the bound. \square

For the logistic map f_c the above result can be sharpened by noting that the bit complexity of f_c is $O(M(n))$. Thus the time complexity of computing n iterations of the logistic map f_c is $O(n M(n))$.

7.2 Bottom-up propagation of error terms

Let us look at how to implement the iteration using the bottom-up propagation of Section 6.2. This will not affect the complexity bound given in Theorem 7.3, but will hopefully result in better constants.

The possible loss of precision in each iteration prevents implementing a straight loop to compute the forward orbit. However, Algorithm 1 can be used to compute the forward orbit. Thus, a while loop is run as long as the remaining precision of the approximations are sufficient. Then the limit on the exponent of approximations is increased and the computation is started from the beginning.

In pseudo-code this can be done as follows.

```
do {
  for (i=0; i <= n; i++)
    if (precision is too low) {
      exponent_limit *= 1.5;
      break;
    }
   $x_{i+1} = \text{round\_and\_limit\_approx}(\text{mul}(c, f(x_i)))$ ;
}
```

Table 1: Average loss of precision.

No. of iterations	Lost bits per iteration
100	.93
200	.98
500	.99
1000	.994
2000	.997
5000	.998
10000	.9991
20000	.9995
50000	.9998

```
} while (i <= n);
```

Suppose i iterations were computed using the old limit on exponents. Our implementation does not take any advantage of the previous computation of i iterations with the old limit. The first i iterations will take much longer to compute with the new limit on the exponents since the approximations will be much larger.

One could, in principle, mix the two algorithms and compute the first i iterations using Algorithm 2 since first approximations already are computed and then switching back to Algorithm 1. However, the space needed to store the first i approximations of the orbit is prohibitive, and the generalised error terms cannot be utilised effectively.

In our implementation the increment of the limit on the exponents has been fixed as 1.5 times the previous limit. A factor of 1.5 behaves reasonable for the problem at hand but other ways of computing an increment on the limit may be much more suitable for other problems.

7.3 Evaluation

Although the actual loss of precision for a particular iteration depends on the approximation of the input, there seems to be a stable average loss of precision, over a number of iterations, that does not depend on the size of the approximations. Table 1 shows the average loss of bits per iteration for f_4 for our implementation. The slight increase is due to a number of iterations being performed exactly at the beginning of the computation before any rounding is needed.

By the strong indication above we will assume that the loss of bits is uniform over the size of the approximations. Thus, we can say that there is a loss of approximately one bit per iteration for f_4 .

However, the average loss of bits per iteration of f_c varies with the constant c . Figure 2 shows the familiar bifurcation behaviour of f_c for $c \in [3, 4]$, and a graph of the average number of lost bits in the same interval. When the number of lost bits per iteration is very low it is, in fact, often asymptotically zero, that is, the number of lost bits is independent of the number of iterations. This is due to the existence of attracting orbits that have basins that are large enough to contain the approximations of the points in the orbit. However, some bits are lost early in the computation before such a basin has been found.

If a basin of an attracting orbit is found the computation can continue to run without further loss of any bits. In fact, for $c \leq 3.56$ basins of attracting orbits are found and the computation can continue indefinitely using only linear time resources and constant space resources. The attracting orbits that can “catch” the computation within a basin has to have limited cardinality. However, the cardinality can be surprisingly large, for example, the orbit found for $c = 7311/2048$ has period 64. The same phenomena occurs within the “windows” of the bifurcation plot. It is clearly visible that the window for period three corresponds to a negligible average loss of bits. The same is true also for the other major windows. One can also see that the basins are slightly harder to find near the bifurcations than away from them.

By Proposition 7.2 the space needed to compute n iterations of the logistic map is linear in n . We have noted above, that our implementation actually may run in $O(1)$ space for many values of c . This clearly is an improvement only achievable with bottom-up propagation of error terms. Furthermore, the orbit may be computed in $O(n)$ time for the same values of c . Thus, giving an important improvement on Theorem 7.3. For other values of c the bottom-up propagation gives the same asymptotical behaviour. However, for these cases, the average number of lost bits per iteration is reduced substantially, resulting in better constants hidden by the ordo notation.

As we have observed above, Algorithm 1 with bottom-up propagation of error terms has an average loss of 1 bit per iteration, whereas, the Algorithm 2 with top-down propagation of required precision has a loss of 3 bits per iteration (see analysis in Section 6.1). We can estimate the running time of the latter algorithm to be at least 9 times as long (recall that this uses a lot of external knowledge about the map). This has to be offset by the time Algorithm 1 uses in computations that are later thrown away because the precision has become insufficient, and for having redundant precision due to the way the limit on the exponents of approximations is computed. This can

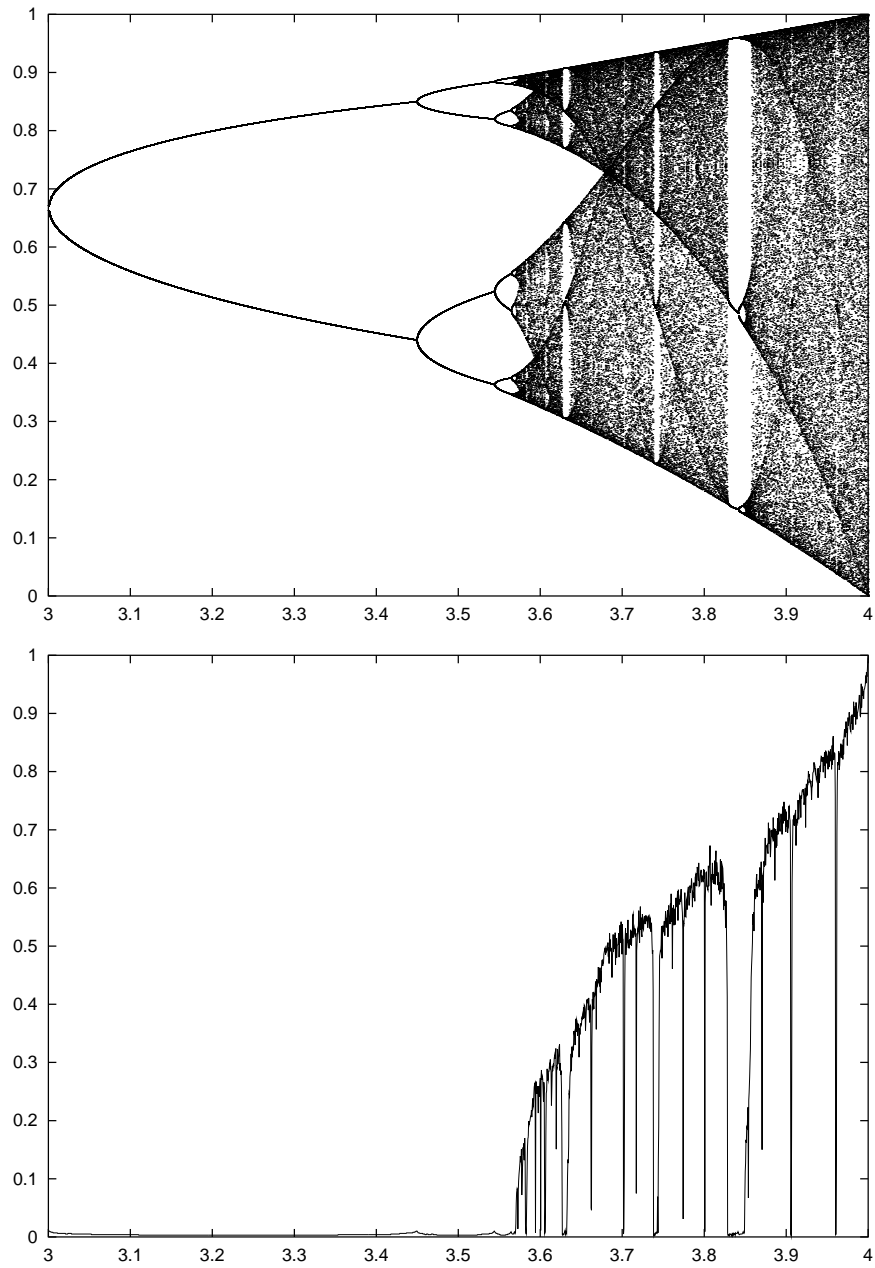


Figure 2: Bifurcation diagram of f_c . Average loss of bits per iteration of f_c .

in bad cases amount to a major fraction of the total time. Compare the two timing charts in Figure 3. The different branches of the graphs correspond to different limits on the exponents in the approximations. Remember that the first iterations are recomputed after every change of the limit. Note that the branches are becoming increasingly flat as the size of the approximations shrink during the computation.

The wasted time in the first graph are all but the final branch, about 40%. The wasted time in the second graph correspond to slightly less than the sum of all branches but the penultimate. This is because having had just a slightly higher limit on the exponents in the penultimate run would have given all iteration at a marginal extra cost. In this case, the wasted effort amounts to about 75%. Increasing the factor 1.5, used to scale the limit for the exponents in the approximations, would make the good cases better and the bad cases worse. Tuning of this factor is possible but it is not in the scope of this paper. It is even possible to use the average loss per iteration and make an educated guess at the best possible limit after some initial run, thereby avoiding almost all wasted effort. This, however, has not been tried since it is based on an assumption that is not known to be true for other involved computations.

Nevertheless, the running times that we are getting very strongly suggests that Algorithm 1 is the better choice for computations involving many operation.

Moreover, the average loss of bits is often effectively zero if f_c has an attracting periodic orbit of limited length. For example, compare the timing chart for $f_{3.830078125}$ in Figure 4. For this value of c the periodic orbit is:

0.1560550000
0.5044283249 .
0.9574444232

This was computed using a limit on the exponent of only 80. But since only 4 bits were lost during the computation of 100000 iterations it was still sufficient to guarantee the accuracy of the final iteration. The total time is only 2.5s, which is very fast indeed, the corresponding floating point computations took .46s. The timing chart is linear in this case since we are computing with approximations of the same size throughout.

As a test of the importance of generalised error terms the iteration has been run with different bounds on the error terms. Table 2 shows the average numbers of bits lost per iteration and execution time for the chosen sizes of error terms. The table also gives a comparison between the two ways of implementing the iterated map of Section 6.2, i.e., using only primitive operation (1), and using a special operation for the quadratic map (2).

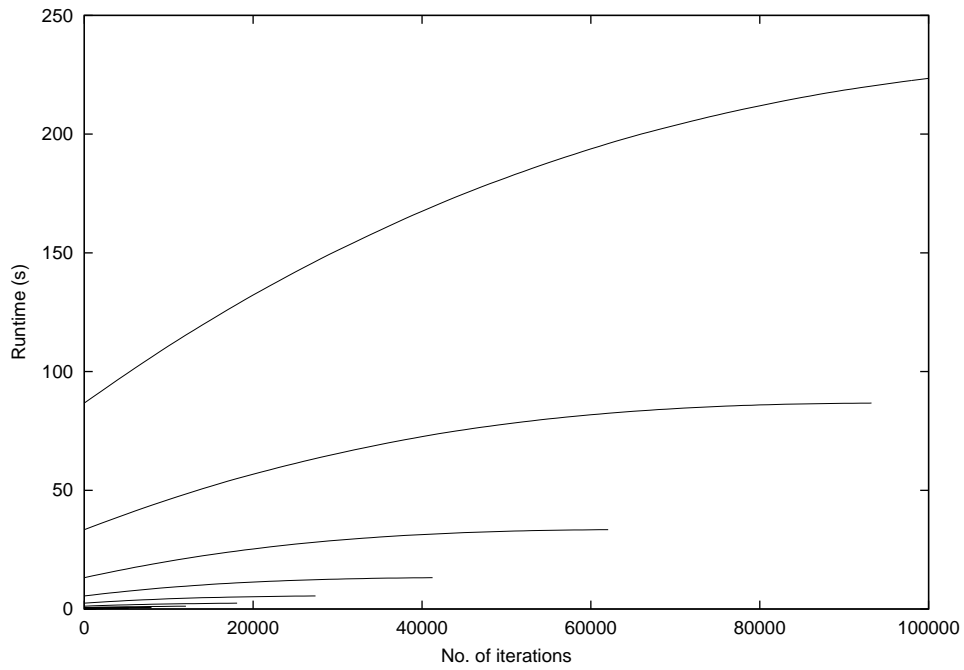
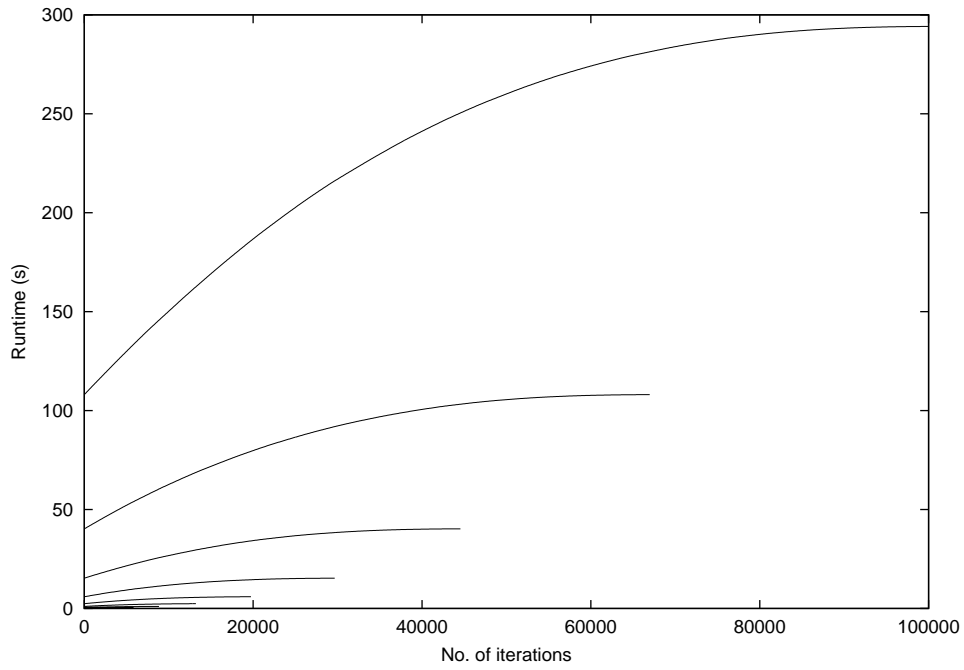


Figure 3: Timing charts of f_4 and $f_{3.59375}$.

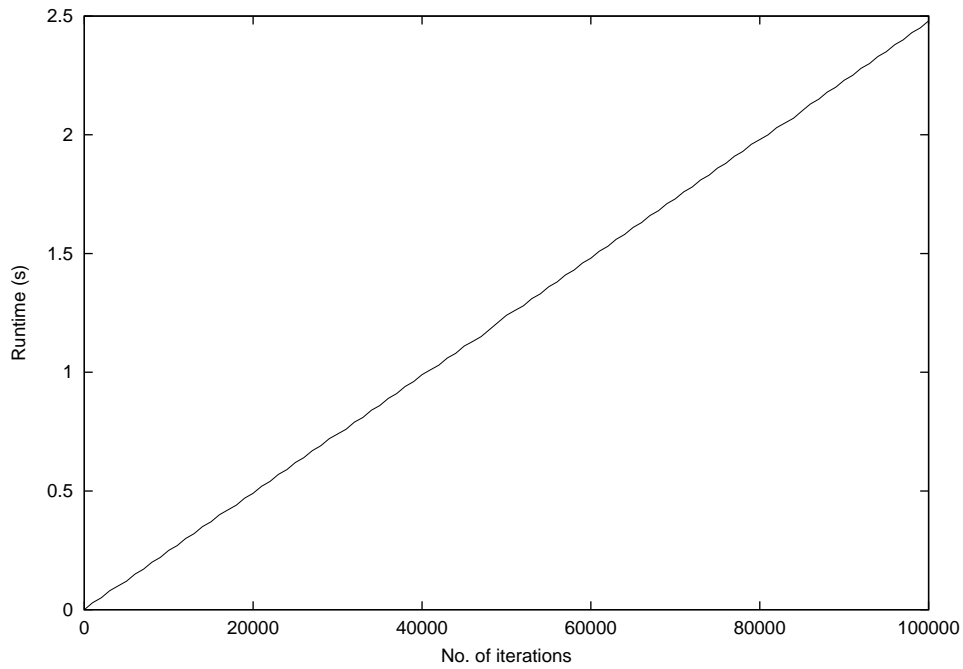


Figure 4: Timing chart of $f_{3.830078125}$.

Table 2: Number of lost bits per iteration for 10000 iterations of the logistic map f_4 with different bounds on the size of error terms.

Size of error terms	(1)		(2)	
	bits	time	bits	time
1	2.9963	39.0	1.9453	22.46
2	2.4969	45.6	1.4326	11.50
3	2.2475	19.3	1.2053	13.33
4	2.1626	20.3	1.0994	14.24
5	2.0602	20.8	1.0484	14.75
10	1.9997	21.6	1.0006	6.53
20	1.9978	21.5	.9991	6.57
30	1.9978	21.5	.9991	6.61

With only 1 bit to represent the error term the number of lost bits per iteration is approximately 3 (using only primitive operation) and 2 (using a special operation for $x \mapsto x(1 - x)$). This is 1 bit better than the best bounds available for top-down propagation of required precision. A further saving of 1 bit per iteration is obtained using generalised error terms. For the logistic map at least, it seems that about 10–20 bits is enough to achieve a good reduction of unnecessary rounding. From the timings provided, one notes that the smaller the error terms are, the faster the computation, as long as no extra reevaluations has to be performed (extra reevaluation can be seen in the table by leaps in execution time).

References

- [1] O. Aberth. *Computable Calculus*. Academic Press, 2001.
- [2] J. Blanck. Exact real arithmetic systems: Results of competition. In Blanck et al. [4], pages 390–394.
- [3] J. Blanck. General purpose exact real arithmetic. CS Report ??, Dept. of Computer Science, University of Wales Swansea, 2002.
- [4] J. Blanck, V. Brattka, and P. Hertling, editors. *Computability and Complexity in Analysis*, volume 2064 of *Lecture Notes in Computer Science*. Springer, 2001.
- [5] 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.
- [6] J. M. Borwein and P. B. Borwein. *Pi and the AGM: A Study in Analytic Number Theory and Computational Complexity*. John Wiley & Sons, 1998.
- [7] G. Chaitin. Incompleteness theorems for random reals. *Adv. in Appl. Math.*, 8:119–146, 1987.
- [8] R. L. Devaney. *An Introduction to Chaotic Dynamical Systems*. Addison-Wesley, 1989.
- [9] GMP. <http://www.swox.com/gmp/>.

- [10] P. Gowland and D. Lester. A survey of exact computer arithmetic. In Blanck et al. [4], pages 30–47.
- [11] A. Grzegorzcyk. Computable functionals. *Fundamenta Mathematicae*, 42:168–202, 1955.
- [12] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, 1969.
- [13] 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.
- [14] N. Müller. The iRRAM: Exact arithmetic in C++. In Blanck et al. [4], pages 223–252.
- [15] H.-O. Peitgen, H. Jürgens, and D. Saupe. *Chaos and Fractals*. Springer-Verlag, 1992.
- [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] K. Weihrauch. *An Introduction to Computable Analysis*. Springer, 2000.