Available online at www.sciencedirect.com
SCIENCE DIRECT®
Theoretical Computer Science 351 (2006) 74–81
Theoretical Computer Science
www.elsevier.com/locate/tcs

Implementing exact real arithmetic in python, C++ and C

Keith Briggs

BT, Martlesham Heath, IP5 3RE, UK

Abstract

I discuss the design and performance issues arising in the efficient implementation of the scaled-integer exact real arithmetic model introduced by Boehm and others. This system represents a real number with a automatically controlled level of precision by a rational with implicit denominator. I describe three practical codes, in python, C++ and C. These allow the convenient use of this computational paradigm in commonly used imperative languages.

Keywords: Exact real arithmetic

1. Introduction

The set of real numbers equipped with the operations of addition and multiplication $\mathbb{R}$ is defined axiomatically as the unique (up to isomorphism) complete ordered field. That is, $\mathbb{R}$ is a field possessing an order relation $<$, in which every subset with a upper bound has a least upper bound. This definition, however, is noticeably non-constructive: we are given no explicit representation of any real except 0 and 1; we are given no algorithms for the field operations $+$ and $*$, and no algorithm to test the order relation $<$. This state of affairs represents a considerable challenge to computer scientists: how do we represent this uncountable set on a machine with finite storage resources, and how do we implement the field operations? The conventional answer is a *floating point* representation: some finite set of rationals $\mathbb{F}$ together with round-off rules for substituting a nearby member of $\mathbb{F}$ whenever the result of an operation is not in $\mathbb{F}$. But other answers are possible: for example, iRRAM [17], in which an automatic back-tracking method is used to control precision.

These heuristics are certainly sufficient for most scientific computation, but it should be noted that the emphasis is usually on accuracy of the field operations, and this comes at a price: the correctness of the order relation is not guaranteed. In some areas of application, the latter may be critical: computational number theory, computational geometry and computer-assisted theorem-proving in analysis are some examples of such areas. Thus, I suggest that an appropriate definition of an *exact real arithmetic* system is one in which the truth value of the order relation is always correctly computed. In fact, it is sufficient to impose the requirement that for any computed result x, the test $x > 0$, whenever it completes in finite time, returns 'true' or 'false' correctly. (We will see later in typical exact real systems, that if in fact $x = 0$, this test will never return.) We may thus view such a system as a method of proving suspected inequalities between computed quantities.

E-mail address: Keith.Briggs@bt.com.

0304-3975/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2005.09.058

One would like exact real arithmetic to be implemented in a way that is completely transparent to the programmer. Because of the backtracking requirement, a functional language is usually considered necessary, and implementations of varieties of exact real arithmetic in HOL [11], Miranda [18], Haskell [12], OCAML [8], and Mathematica [1] have been described. Though these are all theoretically very elegant, they suffer from two practical drawbacks: potentially poor efficiency (though this is a minor issue in the latest languages), and difficulty of embedding in existing software. Avoiding these disadvantages imposes severe, but not insurmountable, demands on the software designer. Thus, I have chosen to produce an implementation in better-known languages: python, and C++ using an add-on functional library. I will show that exact real arithmetic is also possible (and is indeed most efficient) in C. The user thus has a choice:

- python: easiest to use, good for algorithm development, but slow,
- C++: intermediate ease-of-use and speed, slow compilation time,
- C: requires function calls for each arithmetic operation (though this can be hidden with a thin C++ wrapper), best performance.

In this paper I will survey some existing theory, before focussing on the scaled-integer representation and describing the practical implementations. The ultimate aim is a fast package for exact real arithmetic, which non-experts will find easy to use. The codes are documented and distributed in [4,5].

2. Previous work

We consider throughout that we are working on a computing system that provides integer (and hence rational) arithmetic for arbitrarily large integers, and that the underlying representation is binary. The problem is to represent, and operate on, the fractional part of numbers, which may be irrational and therefore will not have a finite representation.

In a pioneering hakmem [9], Gosper considered continued fractions and defined algorithms for stream arithmetic on such representations; these have been further developed in [21,16,13]. The consensus is that although such a system can be made to work, it is not efficient for the following reasons. Firstly, the 'digits' x_i (partial quotients) are generically 1 with frequency about 48%, but can be arbitrarily large. Thus, large-integer arithmetic must be provided for, which is not used most of the time. Secondly, the outputs of the algorithms are not normalized, containing a large amount of redundant information, in the form of zero partial quotients. A normalization algorithm can be provided, but introduces extra inefficiencies. A method closely related to continued fractions is that of Möbius (linear fractional) maps [18].

Some other theoretically interesting representations which have been considered but seem to suffer one or more practical limitations include: radix representations with negative digits, non-integral or irrational bases such as $\frac{2}{3}$ or the golden ratio, and nested sequences of rational intervals. Details of these may be found in [7]. In [1], Andersson has used a Cauchy sequence representation with rational terms and an explicit modulus of convergence function. This approach is likely to suffer the common problem of any system using rational numbers, namely the typical explosive growth of the numerators and denominators.

3. The present implementations

In [2], Boehm et al. introduced another representation, which is not a radix representation, but a *scaled-integer* representation. Boehm et al. suggested representing a real $\hat{x} \in \mathbb{R}$ by a function $x : \mathbb{Z}^+ \to \mathbb{Z}$ satisfying

$$|B^n \hat{x} - x(n)| < 1 \quad \forall n \in \mathbb{Z}^+, \tag{1}$$

where $B \geqslant 2$ is some fixed integer. In other words, $\hat{x}$ is sandwiched as

$$\frac{x(n) - 1}{B^n} < \hat{x} < \frac{x(n) + 1}{B^n}. \tag{2}$$

Note that the $\hat{}$ is on the number, not the function, hinting that I will eventually regard the function as more fundamental. Thus, $x(n)$ is an integer close to a scaled multiple of our real number. The system is non-incremental, meaning that each improved approximant $x(n+1)$ contains all the information in $x(n)$, as well as additional information. B represents the *granularity* of our system—larger values of B cause information to grow at a greater rate as n increases, at the possible cost of spending time getting more information than actually needed. An important advantage of this representation is that less precise values may be efficiently computed from more precise ones by simple bit-shifting (if B is a power of two). This makes caching (memoizing) of the most precise known value of all intermediate quantities worthwhile; some requests are then satisfied from the cache. Comparison is also easy: if we can find a value of n such that $x(n)$ and $y(n)$ differ by more than one, then x and y must be unequal.

Boehm et al. gave algorithms for the basic arithmetic operations, and demonstrated that a lazy stream implementation in Lisp is possible. Note that a language supporting anonymous functions constructed at run-time is required since the algorithms typically take two functions as input, and construct and return a new function representing the output.

In [16], Ménissier-Morain significantly extended the theory of scaled-integer representations, giving proofs of correctness for all the basic algorithms, and adding new algorithms for transcendental functions. This representation looks potentially very efficient, since all operations reduce to large-integer arithmetic, for which fast libraries are available. I now summarize the results of Ménissier-Morain which I use in my implementation. A rational $\hat{q}$ is correctly represented by $q(n) = \lfloor B^n \hat{q} \rfloor$. This allows us to initialize exact reals from exact integer or rational data. We consider that exact reals should never be initialized from inexact data such as IEEE doubles; this is explicitly prevented by the absence of such a constructor.

Addition is defined by $[x+y](n) = \lfloor x(n+1) + y(n+1)/B \rceil$ if $B \geqslant 4$, and by $\lfloor x(n+2) + y(n+2)/B^2 \rceil$ if $2 \leqslant B < 4$. To give a flavor of the proofs, I show the correctness of the addition algorithm for the case $B = 2$, which makes it clear that '+2' expresses the additional precision required to satisfy the definition:

$$\begin{aligned}
|[x+y](n) - 2^n(\widehat{x}+\widehat{y})| &= |\lfloor (x(n+2) + y(n+2) + 2)/4 \rfloor - 2^n(\widehat{x}+\widehat{y})| \\
&\leqslant 1/2 + |(x(n+2) + y(n+2))/4 - 2^n(\widehat{x}+\widehat{y})| \\
&= 1/2 + |x(n+2) + y(n+2) - 2^{n+2}(\widehat{x}+\widehat{y})|/4 \\
&\leqslant 1/2 + |x(n+2) - 2^{n+2}\,\widehat{x}|/4 \\
&\qquad + |y(n+2) - 2^{n+2}\,\widehat{y}|/4 \\
&< 1.
\end{aligned}$$

Note that only one large-integer addition is needed here, and if B is chosen as a power of 2, then the truncated division may be done very efficiently by right-shifting.

Multiplication is defined by $[x*y](n) = \lfloor 1 + x(p)y(q)/B^{p+q-n} \rceil$, where p and q are given by

B	p	q
2	$\max(n - \mathrm{msd}(y) + 4, \lfloor (n+3)/2 \rfloor)$	$\max(n - \mathrm{msd}(x) + 4, \lfloor (n+3)/2 \rfloor)$
3	$\max(n - \mathrm{msd}(y) + 3, \lfloor (n+3)/2 \rfloor)$	$\max(n - \mathrm{msd}(x) + 3, \lfloor (n+3)/2 \rfloor)$
$\geqslant 4$	$\max(n - \mathrm{msd}(y) + 2, \lfloor (n+2)/2 \rfloor)$	$\max(n - \mathrm{msd}(x) + 2, \lfloor (n+2)/2 \rfloor)$

and the *most significant digit*, $\mathrm{msd}(x)$ is defined as $\mathrm{msd}(x) = \min_{n \in \mathbb{Z}}(|x(n)| > 1)$. Again, only one large-integer operation is needed. Similar algorithms exist for reciprocal, division, nth roots, sign and absolute value, and have been described and proved correct by Ménissier-Morain in [16]. She also gives algorithms for some transcendental functions such as exp, log, arctan. The only remaining operations required for practical purposes are output conversion, and construction of a general algebraic number.

3.1. Algebraic number construction

Not tackled in the existing literature for any of the varieties of exact real arithmetic is the construction of a general algebraic number, that is, a root of polynomial p with integer coefficients. The special case of nth roots is easy: we may

define $\hat{x}^{1/n}$ by the function y given by

$$k \mapsto \lfloor (x(kn))^{1/n} \rfloor,$$

and the floor z of the nth root of the integer $x(kn)$ may be very efficiently computed by the all-integer variant of Newton's iteration:

$$z \leftarrow \lfloor ((n-1)z + \lfloor x(kn)/z^{n-1} \rfloor)/n \rfloor.$$

I found the following method to be effective for the general case: given p and integers a, $B > 0$, $k > 0$, we can compute the sign of p at a/B^k with only integer operations by scaling the polynomial coefficients:

$$p(a/B^k) = \sum_{i=0}^{n} p_i (a/B^k)^i = B^{kn} \sum_{i=0}^{n} (B^{-k(i+n)} p_i) a^i.$$

Thus,

$$\operatorname{sign} p(a/B^k) = \operatorname{sign} \sum_{i=0}^{n} (B^{-k(i+n)} p_i) a^i.$$

I consider an algebraic number to be defined by p and a rationally bracketed root with the brackets having the same denominator:

$$\operatorname{sign} p(a/B^k) \operatorname{sign} p(b/B^k) < 0.$$

Given such we may refine it by bisection to the accuracy necessary to satisfy the bounds in Eq. (2). This is implemented in my software. An integer version of Newton's method may be possible and could be even better here.

3.2. General features of the functional implementations

I have built two separate implementations (generically called XR) in python [20] and C++. Both internally use only integer arithmetic, and are designed to be easily integrated with existing code. The C++ version is fully compiled, and at least an order of magnitude faster than the python version; the actual efficiency achieved will be compiler-dependent. Both versions define a class Q representing rational numbers.

Just one functional feature is required: *lambda*, an anonymous function constructor. This is supported directly in python, and in C++ via the FC++ library [15]. Implementation would also be possible in perl, but few languages other than these three have all the required features of object-orientation, first-class functions, lambdas and operator overloading.

3.3. The python implementation

The python implementation has the great virtue that most algorithms translate directly and transparently into python code. Python provides large-integer support as a built-in. I define a class XR, and overload all operators to operate on instances of this class. An outline of the class definition should make the basic idea clear:

```
class XR:
   B=2
   def __init__(s,x):
      if type(x) is IntType:
         s.data=Q(x)
      else
         s.data=x
   def __call__(s,n):
      return s.data(n)
```

Note that s refers to 'self', the object instance itself. The addition and square root algorithms appear as (for $B = 2$):

```
def __add__(x,y):
    return lambda n: (x(n+2)+y(n+2)+2)/4

def __sqrt__(x):
    return lambda n: sqrt(x(2* n))
```

and comparison is computed by

```
def cmp(x,y):
    n=0
    while 1:
        xn,yn=x(n),y(n)
        if xn<yn-1: return -1 # => x < y
        if xn>yn+1: return 1 # => x > y
    n+ =1
```

A general feature of all exact real systems is that equality is undecidable. Thus, for example, the statement `XR(1)+XR(1)< XR(2)` will loop forever. This is unavoidable since the < operator sees only function values, and is unaware of the full definition of the functions representing its left and right arguments. Interestingly, though, since the absolute value function is computable, computation of the minimum and maximum, and thus sorting is possible:

```
def minmax(x,y):
    s,d=x+y,abs(x-y)
    return (s-d)/2,(s+d)/2
```

3.4. The C++ implementation

The python version just described is useful, but slow, mainly due to the interpreted nature of that language. However, it forms a useful testbed for algorithm development, and having been verified in this way, an exact real algorithm may be easily translated into the faster C++ version to be described now.

C++ is not normally viewed as a functional language. However, it is possible to use overloading of the `()` operator (which means making the object callable, in the style of an ordinary function) to create a callable object and thus achieve the desired effect of emulating a `lambda` anonymous function constructor. To achieve this in a way that is convenient for the programmer to use is possible, but not simple. Of several attempts at this I have selected the FC++ library by McNamara and Smaragdakis [14]. I will briefly describe the features of this library which make it particularly appropriate for implementing exact real arithmetic.

FC++ is a library for functional programming in C++. McNamara and Smaragdakis call FC++ functions *functoids*. These are strongly typed, in the spirit of C++, but polymorphism is possible. FC++ provides higher-order polymorphic operators like `compose`, `map` and `filter`, which generally follow Haskell syntax, supports currying with `bind` operators, and has the Lisp-like list operators `head`, `tail` and `cons`. Using FC++ makes it possible to write a C++ class, which I call XR, which a programmer may use transparently in a traditional C++ style, without being aware of the functional concepts being used internally. This feature is of great advantage if a small calculation in exact real arithmetic needs to be embedded in a large, possibly pre-existing, C++ project.

The basic `lambda` type needed for exact real arithmetic may be defined in FC++ as a unary function from a four-byte hardware integer (**int**) to a large-integer type (`Z`), for which I have used NTL's `ZZ` type [19], which in turn is built on `gmp`'s `mpz_t` type [10]. The latter provides a very efficient low-level C implementation of large-integer arithmetic, from which I build a class `Q` representing rational numbers. Using FC++, the basic class definition for XR

with constructors initialized from several different types looks in outline like this:

```
typedef Fun1<int,Z> lambda;
typedef CFunType<int,Z> XRsig;
class XR: public XRsig {
  public:
    lambda x;
    XR(): x(makeFun1(Q(0))) {}
    XR(const int xx, const int yy=1): x(makeFun1(Q(to_Z(xx),to_Z(yy)))) {}
    XR(const Z& xx): x(makeFun1(Q(xx))) {}
    XR(const Q& xx): x(makeFun1(xx)) {}
    XR(const lambda xx): x(xx) {}
    Z operator() (const int n) const { return x(n); }
};
```

The required overloaded operators may be programmed quite simply, with the addition of helper functions (which are not used directly by normal users). For example, the addition operator becomes:

```
class AddHelper: public XRsig {
  XR f; XR g;
  public:
    AddHelper(const XR& ff, const XR& gg): f(ff), g(gg) {}
    Z operator()(const int n) const {
      return (f(n+2)+g(n+2)+2>>2;
    }
};
struct XRADD: public CFunType<XR, XR, AddHelper> {
  AddHelper operator() (const XR& f, const XR& g) const {
    return AddHelper(f,g);
  }
} XRadd;
XR operator+(const XR& x, const XR& y) {
  return memo(XRadd(x,y));
}
```

Note the `memo`, which provides caching of the computed value. Though this code might seem convoluted in comparison to the python version, all internal details are hidden in the file `XR.h` and thus need not concern the user.

3.5. The C implementation

Since exact real arithmetic changes the semantics of the elementary arithmetic operations, an implementation in C might seem impossible. However, by providing functions for each operation, we may build a dependency graph (a directed acyclic graph, or DAG) in which each node contains pointers to its argument(s), a description of the operation, and a cache for the largest argument with which the function has been called and the corresponding return value. In my implementation, a node looks like this:

```
enum op {rat,abs,neg,sqrt,recip,iadd,isub,imul,subi,divi,add,sub,mul,sqr,div,root,exp,pi};

struct node {      /* internal representation of an exact real */
  node* x;         /* left operand */
  node* y;         /* right operand */
  enum op f;       /* operation */
  mpz_t cache;     /* cache */
  int maxn;        /* cache high-water mark */
};
```

The special node type `rat` indicates a terminal node of the DAG, that is, a rational number. `mpz_t` is `gmp`'s large-integer type. Functions `f` are provided in the library for each operation, which internally work in exactly the same way as has been described for the C++ implementation. `sqr` represents the squaring operation, and `exp` the exponential function, the only transcendental function at present implemented. However, the lower-level nature of C means that many internal optimizations can be carried out, and the resulting code is typically about 10 times faster then the C++ version, and at least 100 faster than the python version. Actual timings are highly problem-dependent. The complete code is documented and distributed in [5].

3.6. Some applications

A standard test example is a quadratic map of the interval [0, 1), for example $x_0 = 0.9$; $x_{k+1} = 3.999x_k(1 - x_k)$, $k = 0, 1, 2, \ldots$ In IEEE double floating point, the computed x_{53} is less than $\frac{1}{2}$, but the correct result is greater than $\frac{1}{2}$. The complete program (which prints '1', and where '`Q(p,q)`' constructs the rational p/q) for this example is:

```
from XR import *
a=XR(Q(3999,1000))
x=XR(Q(9,10))
for k in range(53): x=a*x*(1-x)
print 2*x>1
```

In [1], Andersson gives the example of the computation of 100 decimals of $(1-\cos x)/x^2$ where $x = 10^{-100}$, which fails in Mathematica 4.2 with the default floating point settings. In my python implementation all that is required is:

```
from XR import *
x=XR(Q(1,10**100))
print ((1-cos(x))/x**2).dec(100)
```

An interesting example to test the exp, π, and $\sqrt{}$ functions is to evaluate the fractional part of $\exp(\pi\sqrt{163})$, which is non-zero, though this cannot be determined with IEEE double floating point. The non-zero result obtained with XR is in fact a *proof* that this quantity is non-integral.

A computational geometry example is to test whether the point (x_0, y_0) is to the left or to the right of the line through (x_1, y_1), (x_2, y_2). This is determined by $\text{sign}((y_1 - y_0)(x_2 - x_1) - (x_1 - x_0)(y_2 - y_1))$, which floating point systems can compute with the wrong sign in bad cases. With exact real arithmetic, the sign is always determined correctly and with the minimal necessary computation.

A more significant application is to Diophantine approximation algorithms [3] and related procedures such as the LLL algorithm (see, for example, [6]). In these algorithms a critical step involves a branch decision made on the basis of comparing two almost equal irrational quantities. Thus, a typical subproblem is: given $x_1, x_2 \in \mathbb{R}$ and large integers $p_1, p_2, q \in \mathbb{Z}$, and defining $e_1 = |\, q\, x_1 - p_1\,|$, $e_2 = |\, q\, x_2 - p_2\,|$ is $e_1 > e_2$ or not? The algorithm will have already chosen p_1, p_2, q, to make e_1, e_2 as small as possible. In exact real arithmetic such a decision is always correctly made, in contrast to floating point of any fixed precision, in which the algorithm will eventually take the wrong branch when it 'runs out' of precision in x_1 and x_2.

4. The future

The codes described are already practical and efficient. However, there is much scope for future work in this field, both theoretical and practical. For example, what is the optimum value of B? Could the code choose this itself with some heuristics? By relaxing Eq. (1), can we extend the real numbers in some useful ways? For example, the Kronecker delta function $\delta(n)$ has some of the properties of an infinitesimal.

Finally, one might ponder the fact that mathematical software has until now been designed to mirror existing mathematical structures; why should this situation not be reversed? If existing structures (such as the axiomatic reals) are not computationally convenient, why should we not replace our structures with those that *are* convenient?

References

[1] P. Andersson, Exact real computer arithmetic with automatic estimates in a computer algebra system, Department of Mathematics, Uppsala University, U.U.D.M. Report 2001:P5, 2001, <http://www.math.uu.se/research/pub/FPaAndersson1.pdf>.

[2] H.-J. Boehm, R. Cartwright, M. Riggle, M. O'Donnell, Exact real arithmetic: a case study in higher order programming, in: ACM Symp. on Lisp and Functional Programming, 1986, pp. 162–173, <http://dev.acm.org/pubs/citations/proceedings/lfp/319838/p162-boehm/>.

[3] A.J. Brentjes, Multi-dimensional continued fraction algorithms, Mathematical Centre Tracts, Vol. 145, Mathematisch Centrum Amsterdam, 1981, MR 83b:10038.

[4] K.M. Briggs, XR homepage, <http://members.lycos.co.uk/keithmbriggs/XR.html>, 2002.

[5] K.M. Briggs, xrc homepage, <http://keithbriggs.info/xrc.html>, 2003.

[6] H. Cohen, A course in computational algebraic number theory, Graduate Texts in Mathematics, Vol. 138, Springer, Berlin, 1993.

[7] M. Escardó, Introduction to exact numerical computation, http://www.cs.bham.ac.uk/~mhe/issac/>, notes for a tutorial at ISSAC, 2000.

[8] J.-C. Filliâtre, CREAL, <http://www.lri.fr/~filliatr/software.en.html>, 2004.

[9] W. Gosper, Continued fractions, <http://www.inwap.com/pdp10/hbaker/hakmem/cf.html>, 1972.

[10] T. Granlund, The GNU MP homepage, <http://www.swox.com/gmp/>, 2002.

[11] J. Harrison, Theorem proving with the real numbers, Ph.D. Thesis, University of Cambridge, Computer Laboratory, 1998.

[12] D. Lester, `Era.hs`: a tolerably efficient and possibly correct implementation of the computable reals using Haskell 1.2, <http://www.cs.man.ac.uk/arch/dlester/exact.html>, 2001.

[13] P. Liardet, P. Stambul, Algebraic computations with continued fractions, J. Number Theory 73 (1998) 92–121.

[14] B. McNamara, Y. Smaragdakis, Functional programming in C++, ACM SIGPLAN Notices 35 (9) (2000) 118–129, <http://www.acm.org/pubs/citations/proceedings/fp/351240/p118-mcnamara/p118-mcnamara.pdf>.

[15] B. McNamara, Y. Smaragdakis, Fc++, <http://www.cc.gatech.edu/~yannis/fc++/>, 2003.

[16] V. Ménissier-Morain, Arithmétique exacte: conception, algorithmique et performances d'une implémentation informatique en précision arbitraire, Ph.D. Thesis, Université de Paris VII, <http://calfor.lip6.fr/~vmm/>, 1994.

[17] N. Müller, iRRAM—exact arithmetic in C++, <http://www.informatik.uni-trier.de/iRRAM/>, 2004.

[18] P. Potts, Exact real arithmetic using Möbius transformations, Ph.D. Thesis, Department of Computing, Imperial College of Science, Technology and Medicine, University of London, see also <http://www.doc.ic.ac.uk/~ae/papers.html>, 1999.

[19] V. Shoup, NTL: a library for doing number theory, version 5.3.1, <http://www.shoup.net/ntl/>, 2002.

[20] G. van Rossum, Python language website, <http://www.python.org>, 2002.

[21] J. Vuillemin, Exact real computer arithmetic with continued fractions, IEEE Trans. Comput. 39 (1990) 1087–1105.