The constructive reals as a Java Library

Hans-J. Boehm
Internet Systems and Storage Laboratory
HP Laboratories Palo Alto
HPL-2004-70
April 19, 2004*

E-mail: Hans.Boehm@hp.com

constructive reals, exact arithmetic, inverse function, calculator

We describe an implementation of the computable (or constructive) real numbers as a pure Java library. To the user, the library interface appears very similar to that of some other numeric types provided by the standard Java library. The primary goal of the implementation is simplicity, so that the implementation could be easily understood, and to allow simple informal correctness arguments. We hope to demonstrate that even such a basic implementation of constructive real arithmetic can be useful in a number of contexts, including in a desk calculator utility distributed with the package. A secondary goal was to demonstrate that some second-order functions on the reals, such as restricted inverse and derivative operations, can be implemented with sufficient performance to be useful.

* Internal Accession Date Only Approved for External Publication

The constructive reals as a Java library

Hans-J. Boehm

HP Laboratories[1]

Abstract

We describe an implementation of the computable (or constructive) real numbers as a pure Java library. To the user, the library interface appears very similar to that of some other numeric types provided by the standard Java library. The primary goal of the implementation is simplicity, so that the implementation could be easily understood, and to allow simple informal correctness arguments. We hope to demonstrate that even such a basic implementation of constructive real arithmetic can be useful in a number of contexts, including in a desk calculator utility distributed with the package. A secondary goal was to demonstrate that some second-order functions on the reals, such as restricted inverse and derivative operations, can be implemented with sufficient performance to be useful.

Key words: Constructive reals, exact arithmetic, Java, inverse function, calculator

1 Introduction

The computable or recursive (cf. [12]) real numbers make it possible to implement computation on the real numbers which is exact in the sense that:

(1) Computations produce a representation of the exact result.
(2) Representations of real numbers generated in this way can be used to produce approximations guaranteed to be accurate to any error tolerance.

Effectively, the representation of a real number x is a program or function for approximating that number to a tolerance. The tolerance is specified on the final answer. The implementation guarantees that any rounding errors in intermediate computations are sufficiently bounded to preserve the guaranteed accuracy of the final result.

Email address: Hans.Boehm@hp.com (Hans-J. Boehm).

[1] Much of the underlying work was done while the author was at SGI.

The computable or recursive reals correspond to one interpretation of the constructive reals[4], and we use all three terms interchangeably.

For the programmer, constructive real arithmetic is easier to use than floating point arithmetic, since it largely removes the need for error analysis.

2 Our Model

We represent a real number x as a computable function f_x mapping an integer precision specification n to a scaled integer approximation $f_x(n)$ such that $|f_x(n) - x/2^n| < 1$. Informally, $f_x(n)$ produces an approximation to x, which is accurate to within an error of strictly $< 2^n$. The approximation is scaled, so that it can be represented as a (potentially very large) integer.

Basic arithmetic operations by themselves perform no numeric evaluation; they simply return an object representing the function. This function can then be invoked when we desire a numerical approximation to the generated result, *e.g.* to print it.

The equality test on computable reals is undecidable. We instead provide a comparison operation which may erroneously identify two values as equal if they are within a specified tolerance, as well as one that produces correct answers for unequal numbers, but diverges when they are equal. Like output operations, these force immediate evaluation of approximations, as do some transcendental functions that require prescaling.

Using our representation, addition can be implemented simply as

$$f_{x+y}(n) = round(f_x(n-2) + f_y(n-2))/4)$$

Informally, in order to get k bits of precision in the result, we evaluate each argument to $k+2$ bits. This is necessary since each argument approximation may then contribute an error of strictly less than $1/4$ in the scaled integer final result, while rounding may contribute an error of up to $1/2$. The division by 4 is necessary to adjust for the implicit scaling.

Unfortunately, in many other cases we cannot determine the precision required for argument evaluation *a priori*; it must instead be based on an approximation of the arguments themselves. In the case of multiplication, the larger one argument, the more precision we need in the other in order to guarantee a specified error bound on the result. A reciprocal computation requires less precision in the argument if the argument is large.

This approach leads to a relatively straightforward implementation. The algorithms are fairly simple, and they rely only on large integer ("bignum") arithmetic. There is no need for an underlying interval or rational arithmetic package, either of which can itself become complex.

Since we must sometimes approximate an argument in order to determine the real evaluation precision, the same argument may be evaluated more than once. In the case of deeply nested expressions, some subexpressions may be reevaluated many times.

3 Other Approaches

Existing implementations of the constructive reals differ in several dimensions. Many implementations, like ours, explicitly build up a data structure representing the function corresponding to a real number (cf. [3,7,20,10,16,8,9]). Building the representation explicitly may consume substantial amounts of space. Although the functions representing reals are normally treated completely extensionally, their internal representation essentially consists of the expression which was used to generate the number. However, this approach makes it far easier to implement the constructive reals purely as a library, which can be used inside arbitrary programs without special precautions.

The function may be represented either as a function in the programming language[6], as a class with an "apply" operator (our present approach), or as a lazily evaluated data structure[7,20]. Approximations may be rational numbers[3], prefixes of redundant decimal expansions[7], prefixes of continued-fraction expansions[10,20], or scaled integers as in [6,8,9] or our current approach. Experience to date suggests that the scaled integer approach leads to the best performance in this category of applications.

Other implementations [1,2,15,14,17] instead use reexecution of the original program with increased precision, and hence the function representing a particular real number remains implicit. These implementations initially execute the program with a fixed, somewhat arbitrary precision, and maintain explicit error bounds on results. So long as error bounds are sufficient to resolve conditionals, avoid singularities for built-in functions, and satisfy constraints on output statements, execution proceeds normally. When precision is insufficient, execution is restarted with a higher precision.

Reexecution-based approaches require a mechanism such as the "multi-value cache" in [17] or the "decision histories" in [15] to ensure that repeated reexecutions appear to follow the same control path through the program. It is also necessary to hide any side-effects from the repeated executions. Explicit

storage of functions avoids these issues at the cost of space consumption.

Implementations that explicitly store functions representing real numbers may determine calculation precision in one of too different ways:

(1) They may use a *top-down* error propagation approach, in which every evaluation of a subexpression guarantees a requested error bound. The error bound is a parameter to the evaluation. This approach appears to be the simplest. Its primary disadvantage is that a given subexpression is often reevaluated a large number of times during a single computation.
(2) It is also possible to use a *bottom-up* error propagation approach in which the calculation precision, not the tolerance in the result, is the parameter to the evaluation. This is essentially the same approach as used with reexecution. Implementations are based on variable precision interval arithmetic, and are thus arguably a bit more complex. Unlike the top-down approach, calculation precisions can be increased in geometric progression; hence the cost of reevaluations tends to be far less.

Approaches based on bottom-up error propagation appear to typically outperform the top-down approach, especially for deeply nested computations. However, this approach may suffer from the fact that all computations are often carried out to the same precision. For example, if we are evaluating $10^{-1000}\pi + 1$ to 1000 digits, the top down approach would notice that there is no need for any substantial evaluation of π, where the bottom up approach would normally evaluate it to about 1000 digits.

For another more detailed discussion of comparable arithmetic packages, see [11].

4 Implementation Strategy

The details of our implementation are affected both by the implementation language, and by our desire for simplicity with acceptable performance rather than optimal performance.

The Java language does not support dynamically constructed "functions" (usually called "closures") *per se*. However, Java objects and inheritance can be used to get essentially the same effect.

A constructive real number is a member of a subclass of **CR**, the class of constructive real numbers. Each such number provides an *approximate* method, corresponding to the approximation function in our original model. This is the only essential element of the representation.

As a concession to performance, each constructive real number representation also includes it's best known approximation, if any, as in [6]. This is represented as the smallest precision argument ever passed to the *approximate* method, together with the result it yielded. [2] All references to argument approximations call the *get_appr* method. instead of calling *approximate* directly. The difference is that *get_appr* both consults and updates the cache. As a result, a constructive real is never reevaluated with a less demanding precision request than for a prior evaluation.

To give the flavor of the representation, we present the complete implementation of addition. The class **CR** contains an *add* method implemented as follows:

```
public CR add(CR x) {
    return new add_CR(this, x);
}
```

Addition simply returns a new object, logically a function, representing the sum of two constructive reals. The new object is a member of the **add_CR** class, which is a subclass of **CR**. Hence every **add_CR** can be used as a **CR**.

An element of the **add_CR** class contains (in addition to the members of **CR**), two fields corresponding to the two arguments, and an *approximate* method which computes approximations exactly as we described above. Its entire implementation is:

```
class add_CR extends CR {
    CR op1;
    CR op2;
    add_CR(CR x, CR y) {
        op1 = x;
        op2 = y;
    }
    protected BigInteger approximate(int p) {
        return scale(op1.get_appr(p-2).add(op2.get_appr(p-2)),
                     -2);
    }
}
```

Here $scale(..., -2)$ shifts its first argument right by two bits, rounding the result. It corresponds to the rounded division by 4 above. Functions such a *get_appr* and *scale* are inherited from **CR**.

[2] Unlike some of our earlier implementations[6], we do not include an eagerly evaluated fixed precision interval approximation. This would have improved performance at the cost of complexity.

We have compromised and used 32-bit machine integers to represent requested precision. This simplifies matters appreciably but can result in precision overflow. We check for those explicitly and raise a suitable exception if this should occur. These checks are confined to a few methods, one of which is *get_appr*.

The algorithms for other operations are also very similar to those in [6]. We use Newton iteration based on the previous approximation for the *sqrt* implementation, but reciprocal computations are performed directly, i.e. without reference to previous approximations. Transcendental functions use Taylor series, with suitable prescaling.

Unlike [17], we do not directly provide limit operations. We do however allow the library user to define new constructive real operations or constants by defining new subclasses of CR with appropriate *approximate* methods. Thus arbitrary constructive real numbers can be computed. Unlike more straightforward uses of the library, this does require understanding of the representation.

5 Higher Order Functions

A number of higher order functions (or functionals) on the constructive reals are computable (cf. [13,18]). In many cases, computation of such higher order functions is impractical[13]. We chose to implement three for which practical implementations do exist. In all cases we currently restrict ourselves to unary functions from the constructive reals to the constructive reals:

- Function composition.
- The inverse of a monotone (and well-defined, hence continuous) function on a closed interval. The arguments to this operation are the function to be inverted, and the two constructive real endpoints of the interval.
- The monotone derivative of a function on an open interval. We assume that the derivative is continuous and *monotone* in an open interval bounded by two specified constructive real endpoints.

The implementation of the first is trivial. We briefly outline the algorithms used for the last two.

5.1 Inverses

The inverse operation on a function f produces an object representing a function f^{-1} from constructive reals to constructive reals. Evaluation of a resulting inverse function at a particular real number produces a constructive real num-

ber r. The iterative evaluation algorithm we discuss here is contained in the *approximate* method of r.

In order to make the inverse operation practically useful, we have to address several aspects of performance:

(1) The number of evaluations of f. This is determined by the rate of convergence of the iterative algorithm.
(2) The precision to which f is evaluated, since that affects the cost of the evaluations.
(3) The precision to which the argument of f^{-1} is evaluated, since that affects the cost of nested evaluations involving inverse functions.

To keep the number of function evaluations small in typical (well-behaved) cases, but still guarantee convergence, we use a hybrid of binary search and linear interpolation (*regula falsi*, cf. [19]). At every point in an approximate computation of $f^{-1}(x)$, we have a (usually) rational interval $[l, h]$ [3] such that, assuming a monotone increasing f, $f(l) < x < f(h)$, and thus $l < f^{-1}(x) < h$.

Each iteration shrinks the interval $[l, h]$ as follows:

- We normally carry out an interpolation step. We compute a guess of the inverse by linear interpolation. If the guess is in the outer quarters of the interval, we adjust it by doubling its distance to the endpoint. This usually ensures that the actual inverse value is then in the smaller interval, thus avoiding the normal failure of regula falsi to shrink the interval under consideration. We then evaluate the function at our guess, and replace the appropriate endpoint with the guess, as if we were performing a binary search.

  It may happen that during the evaluation step the comparison between the function value at the guess and x is indeterminate (i.e. the two are equal to within our evaluation precision), making it impossible to determine which endpoint should be replaced. We resolve this issue by alternately increasing the precision to which x and $f(...)$ are evaluated, and perturbing the guess slightly, in case the two happen to be exactly equal. Repeatedly performing these operations must eventually lead to an unequal comparison.
- If interpolation repeatedly fails to shrink the interval substantially, we set the guess to the midpoint, and perform a traditional binary search step instead of using linear interpolation. This ensures linear convergence in the worst case.

Under normal circumstances, this converges quickly once the interval is small enough that the derivative is close to constant. An initial interpolation step

[3] L and h are of the from $2^k n$ unless they are set to the corresponding endpoint of fs domain, in which case they may not be rational.

ensures that one of the interval endpoints will be very close to $f^{-1}(x)$, though it may not reduce the size of the interval by much. In the next step, our guess will be very close to the endpoint, and the resulting adjustment will reduce the interval size to roughly twice the error in the first interpolation step.

Usually convergence is quadratic and hence the number of function evaluations is logarithmic. We ensure that in almost all cases our guesses, the arguments of the evaluations, are freshly constructed from rationals. Hence function evaluations don't force evaluation of nontrivial arguments. We take care that the argument to the inverse function is evaluated only to slightly more precision than is absolutely necessary.

If a particular inverse function value has been approximated once, we use that approximation to obtain a small starting interval for later approximations to higher accuracy. If we need the answer to high precision, and no prior approximation is available, we first force evaluation to a much lower precision. This effectively ensure that only the last few iterations (typically four) are carried out to full precision.

5.2 Derivatives

Since we only compute monotone derivatives, the algorithm is straightforward. We compute finite approximations to the derivative from the left and right. Since these bound the derivative, we simply need to iteratively compute the difference approximations over smaller intervals to guarantee a sufficiently small error.

The initial interval width is chosen based on out experience with past evaluations. This allows us to minimize the number of iterations. The difference approximations themselves are computed using constructive real arithmetic; hence the choice of evaluation precision is essentially automatic.

6 The Implementation

Our implementation has been available on the web, with occasional improvements, since 1999, though it has not been well advertised. [4] . The implementation consists of both a Java library, and a calculator with a graphical user interface.

[4] It was originally available through SGIs web site. It has now been rehosted at **http://www.hpl.hp.com/personal/Hans_Boehm/new_crcalc**

Most uses of which we are aware involve the calculator. The canonical use appears to be to check the accuracy of fixed precision floating point implementations, though it also seems fine as a general desk calculator replacement.

6.1 The Library

The library contains two components. As we saw before, the elements of the class **CR** are constructive real numbers, and the class provides various methods, *e.g. add* and *sin* to operate on them. It inherits from the standard **java.lang.Number** class, and hence constructive reals can be used where a **java.lang.Number** is required. The interface suffers slightly from Java's inability to overload arithmetic operators.

The elements of the class **UnaryCRFunction** are unary functions over the constructive reals. It provides constants corresponding to various unary functions, mostly corresponding to methods of **CR**. It also provides the above-mentioned methods to compute the inverse of a function, to compute the derivate of a function (restricted as above) and to compose unary functions, yielding another **UnaryCRFunction**. **UnaryCRFunction** function objects provide an *execute* method [5] to evaluate the function at a particular constructive real point. New unary functions can be constructed explicitly by introducing a subclass of **UnaryCRFunction** with an explicit definition of the *execute method*. This is analogous to the introduction of new **CR** values by adding a subclass with an explicit *approximate* method.

6.2 The Calculator

The calculator provides a fairly conventional desk calculator user interface, with the addition of a scroll bar to allow display of additional digits of the result. [6] Movement of the scroll bar translates into reevaluation requests in the real numbers currently displayed.

The calculator is a Java applet, and hence can either be run directly from the web page, or downloaded and installed locally. The download size for the calculator .class files with supporting libraries is about 53KB.

Unlike a conventional calculator, certain calculator operations, *e.g.* division

[5] The name was chosen for consistency with some other Java interfaces which also represent functions as objects with an *execute* method.

[6] There is also a less natural mechanism to allow direct entry of the precision, *e.g.* when more than 1000 digits of the result are required.

by zero, potentially fail to terminate. This is addressed in several ways:

(1) The calculator keeps track of values that are known to be integers. Dividing by a known integer zero elicits an immediate error message.
(2) If an operation appears likely to diverge, *e.g.* a division by a number *very* close to zero, the user is warned and asked whether to continue.
(3) All constructive real operations are run in a thread distinct from the one supporting user interaction. The user interface provides a "stop" button to abort a very long running or infinite computation.

Since the leading decimal digits of an arbitrary real number are not computable (e.g. because it may be undecidable whether a number is slightly smaller than, or greater than one), it may happen that the leading digits change as we scroll through a number. We minimize the probability of this by always evaluating a number to 15 more digits than we display. Given that errors are always strictly less than one digit in the last place, this implies that such behavior is only possible if the exact representation of the number contains at least fifteen consecutive "9" digits. [7]

The calculator uses the generic inverse function operation to compute inverse trigonometric functions. The calculator code is also simplified by the ability to treat functions on constructive reals as objects. All unary (or binary) operators can be handled in a mostly uniform manner, and the logic associated with degree vs. radian displays can be easily isolated.

7 Performance

For reasonably small precisions (at most a few hundred digits) and moderately simple calculations, calculator evaluation is typically fast enough on a modern machine that the reevaluation time is not noticeable, and the scrollbar gives the desired visual effect of scrolling through an infinite number.

Based on some quick experiments using the calculator interface on some of the examples in [5], the calculator is typically somewhere between a factor of 5 and two orders of magnitude slower than iRRAM[17], when run from Internet Explorer with its built-in Java Virtual Machine. Given our general preference for simple algorithms over asymptotically fast ones, and the fact that these examples require 10,000 digits of precision, this is not surprising. We would

[7] The error must be within 10^{-15} of the value of the last displayed digit, since we computed 15 more digits. If it were slightly greater than the displayed number, we would have violated the error bound by initially displaying the smaller digit. Thus it must be slightly less than the displayed value, implying the sequence of "9"s.

expect relatively better performance for somewhat lower precisions, and worse performance for more deeply nested computations.

The use of the generic inverse operation clearly slows down the calculator somewhat, but not enough to be a serious problem in this application. On a 733 MHz Pentium III running the Internet Explorer JVM, a $sin(0.5)$ computation to the (atypically large) precision of 5000 decimal digits takes about 4 seconds, while $asin(0.5)$ takes about 20 seconds.

Had we been willing to sacrifice simplicity for performance, we expect that the most important performance improvements would be

(1) Bottom-up error propagation, with evaluation based on variable-precision interval arithmetic, in order to improve performance on deeply nested expressions.
(2) Asymptotically better algorithms for multiplication[8], reciprocals, and various transcendental functions. This could greatly improve performance of high precision evaluations.
(3) Special treatment of some "obviously rational" values.[9]

References

[1] O. Aberth. A precise numerical analysis program. *Communications of the ACM*, 17(9):509–513, September 1974.

[2] O. Aberth. *Precise Numerical Analysis*. Wm. C. Brown, 1988.

[3] P. Andersson. Exact real arithmetic with automatic error estimates in a computer algebra system. Technical Report 2001:P5, Upsala University Department of Mathematics, June 2001.

[4] E. Bishop and D. Bridges. *Constructive Analysis*. Springer-Verlag, 1985.

[5] J. Blanck. Exact real arithmetic systems: Results of competition. In *Proceedings of the Workshop on Computability and Complexity in Analysis, Springer LNCS 2064*, pages 389–393, 2000.

[6] H.-J. Boehm. Constructive real interpretation of numerical programs. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pages 214–221, 1987.

[8] A Java Virtual Machine could provide this as part of the java.math.BigInteger implementation. We suspect few do.

[9] Based on experience with earlier implementations, we don't believe that all numbers computed using only rational operations should be represented as rationals. But sometimes this is a large improvement.

[7] H.-J. Boehm, R. Cartwright, M. J. O'Donnell, and M. Riggle. Exact real arithmetic: A case study in higher order programming. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 162–173, 1986.

[8] K. Briggs. XR - exact real arithmetic. http://more.btexact.com/people/briggsk2/XR.html.

[9] J.-C. Filliatre. The Objective Caml Creal package. See http://www.lri.fr/~filliatr/software.en.html.

[10] W. Gosper. Continued fraction arithmetic. Technical Report HAKMEM Item 101B, Artificial Intelligence Memo 239, MIT, 1972.

[11] P. Gowland and D. Lester. A survey of exact arithmetic implementations. In *Proceedings of the Workshop on Computability and Complexity in Analysis, Springer LNCS 2064*, pages 30–47, 2000.

[12] J. Hartley Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, 1967.

[13] K.-I. Ko. *Computational Complexity of Real Functions*. Birkhauser Boston, 1991.

[14] V. Lee. *Optimizing Programs over the Constructive Reals*. PhD thesis, Rice University, 1991.

[15] V. Lee and H.-J. Boehm. Optimizing programs over the constructive reals. In *Proceedings of the SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 102–111, 1990.

[16] V. Ménissier-Morain. *Arithmétique exacte, conception, algorithmique et performances d'une implémentation informatique en précision arbitraire*. Thèse, Université Paris 7, December 1994.

[17] N. T. Mueller. The iRRAM: Exact real arithmetic in C++. In *Proceedings of the Workshop on Computability and Complexity in Analysis, Springer LNCS 2064*, pages 222–252, 2000.

[18] A. K. Simpson. Lazy functional algorithms for exact real functionals. In *Mathematical Foundations of Computer Science, Springer LNCS 1450*, 1998.

[19] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis*. Springer-Verlag, 1980.

[20] J. Vuillemin. Exact real arithmetic with continued fractions. *IEEE Transactions on Computers*, 39(8):1087–1105, 1990.