Towards an API for the Real Numbers

Hans-J. Boehm
Google
USA
hboehm@google.com

Abstract

The real numbers are pervasive, both in daily life, and in mathematics. Students spend much time studying their properties. Yet computers and programming languages generally provide only an approximation geared towards performance, at the expense of many of the nice properties we were taught in high school.

Although this is entirely appropriate for many applications, particularly those that are sensitive to arithmetic performance in the usual sense, we argue that there are others where it is a poor choice. If arithmetic computations and result are directly exposed to human users who are not floating point experts, floating point approximations tend to be viewed as bugs. For applications such as calculators, spreadsheets, and various verification tasks, the cost of precision sacrifices is high, and the performance benefit is often not critical. We argue that previous attempts to provide accurate and understandable results for such applications using the recursive reals were great steps in the right direction, but they do not suffice. Comparing recursive reals diverges if they are equal. In many cases, comparison of numbers, including equal ones, is both important, particularly in simple cases, and intractable in the general case.

We propose an API for a real number type that explicitly provides decidable equality in the easy common cases, in which it is often unnatural not to. We describe a surprisingly compact and simple implementation in detail. The approach relies heavily on classical number theory results. We demonstrate the utility of such a facility in two applications: testing floating point functions, and to implement arithmetic in Google's Android calculator application.

CCS Concepts: • **Mathematics of computing → Arbitrary-precision arithmetic**.

Keywords: real numbers, API design, comparison, decidability

PLDI '20, June 15–20, 2020, London, UK
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-7613-6/20/06.
https://doi.org/10.1145/3385412.3386037

ACM Reference Format:
Hans-J. Boehm. 2020. Towards an API for the Real Numbers. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3385412.3386037

1 Introduction

The mathematical real numbers clearly play play a central role, both in everyday life and in academic mathematics. But little of what we learn in school applies fully and directly to conventional computer floating point arithmetic. This is particularly problematic, since the quirks of computer floating point arithmetic often get exposed to end users, who do not understand them, and often, arguably quite reasonably, consider them as bugs (e.g. [3, 4, 33]).[1]

Here we explore an interface and associated implementation that behaves much more like the real numbers we learned about in school. To our knowledge, this is the first exploration of a practical general purpose real number type that both reflects the mathematical laws of the real numbers, and also supports exact comparisons in situations in which that's normally expected. Classical mathematical results provide surprisingly effective tools in making this practical and useful.

We demonstrate the utility of such a computational type with two applications, and suggest others.

Most mathematical software represents real numbers using the machine's floating point representation, commonly using IEEE standard floating point.[24]. The resulting computation usually provides sufficiently accurate results, especially for physically motivated problems where the computation is performed on inherently inexact data in any case. This provides spectacular performance, and well-defined computational properties, but with important sacrifices of accuracy and mathematical properties that we discuss below.

A number of applications, for example the calculator application we previously discussed in [12], do not require the spectacular performance afforded by machine floating point, but do suffer significantly from issues related to accuracy. Our goal here is to explore alternative implementations of the real numbers that, at the expense of computation time and space, guarantee accuracy and obey the laws of real numbers. We do not advocate this as a general replacement

[1] Based on the author's experience, this also applies to some extent to practicing software engineers.

for IEEE floating point; the approaches explored here are impractical for most applications that perform billions floating point operations, especially if those operations are deeply nested.

There are many examples of mathematical computations for which accuracy beyond floating point is either essential or provides much more informative answers. The fact that $e^{\pi\sqrt{163}}$ is 2.625374126E17, may not be interesting; The fact that it is 262,537,412,640,768.9999999999992507... is much more so. If I want to find out how close $\cos(10^{-8})$ is to $\cos(0)$, and enter $\cos(10^{-8}) - 1$, I'm likely to be much less happy with the usual answer of 0, than something in the correct neighborhood of -4.99999999999E-17 (for radians). [12] gives some additional examples, including more for which standard calculators produce no correct digits.

Prior work on recursive real arithmetic, discussed in more detail below, shows how to represent real numbers in a form that allows them to be manipulated exactly, and then be evaluated to any requested precision, with a guaranteed error, even for large composite expressions, of < 1 in the last displayed digit, thus providing accurate answers for expressions like $\cos(10^{-100}) - 1$.

Recursive real arithmetic can be quite useful, and provides much cleaner mathematical properties than machine floating point arithmetic. Numerical accuracy is guaranteed, without complicated analysis by the user/programmer, at the expense of performance.

Unfortunately, for such representations, equality of two numbers is not practically decidable. Thus, we cannot in general tell whether a number has a finite decimal representation, or whether taking the reciprocal of a number is finite. We argue that for many applications, the ability to decide equality, at least in "easy cases", is expected and essential. Arguably this is analogous to our expectation that even young children can determine that 1 + 1 is exactly 2, while we have no expectation that determining general equality of real numbers is easy. (E.g. determining whether $e^{\pi\sqrt{163}}$ is exactly equal to 262,537,412,640,769 could be quite challenging without the right tools.)

2 Contributions

We show how to implement arithmetic, including basic transcendental functions, on the real numbers in a way that is correct, even by the standards of naive users. An answer that is exactly representable on the screen is always displayed exactly correctly. Noise, in the form of extra digits, is minimized. We are not aware of other detailed published descriptions of such a data type.

Our earlier work in [12] focused on motivating such an implementation in the context of a calculator application. Our implementation at the time was too ad hoc to warrant a detailed description. Here we present much more elegant and interesting refinements of those algorithms. We show that classical number theory results lead to a surprisingly comprehensive set of equality decision procedures for commonly occurring real numbers computed via limited applications of transcendental functions. Our decision procedures include the "easy cases" in which decidable comparison is commonly expected. For example, where the calculator from [12] would evaluate $(\sqrt{17})^2$ to a scrollable display of `17.000000`, followed by an infinite sequence of zeroes, we now determine that the result is exactly 17, and thus should be displayed as such. We do this in a way that does not add significant computation cost.

We show how to cleanly expose this kind of "best effort" decidability as an API.

We show that, in addition to applications that directly make arithmetic visible to end users, this flavor of real arithmetic is useful for testing applications involving floating point arithmetic.

Although performance of our real arithmetic is usually orders of magnitude slower than floating point, we show that empirically it is more than adequate to be useful for our intended applications.

3 Motivating Applications

We target applications that are much less concerned with the speed of computation than the quality of the result. Here are some examples:

3.1 A Calculator Application

Our primary example, as in [12], is a calculator application on a phone or laptop. As described there, we would like calculator results to satisfy the following constraints, where the first one is a hard constraint, and the others are highly desirable:

1. Displayed results should never expose an error equal to or greater than one in the value of the last displayed digit. Displayed approximations of 2/3 must always have a last digit of 6 or 7. Numbers with terminating representations must always be exact once the last nonzero digit is displayed.
   This requirement applies to all expressions including those that are numerically unstable. This may require extreme precision for intermediate results. Correctly evaluating the first digit of $\tan(\arctan(10^{100}))$ requires the arctan result to be accurate to about 100 digits.
   Since the preceding constraint essentially requires (sub)expressions to be evaluable to arbitrary precision, satisfying it also allows a non-terminating decimal result to be displayed in a scrollable window, providing arbitrary precision to the user.
2. Results with finite decimal expansions should be displayed as such, whenever possible. Certainly, if the user adds \$7.23 and \$4.13, the answer should be \$11.36, rather than \$11.360000000, possibly with an infinite

sequence of scrollable trailing zeroes. Similarly, we would like to see $(\sqrt{11})^2$ evaluate to 11, rather than 11.00000000000.

3. Whenever possible, we would like results to show the correctly rounded or truncated value, with no error. In particular, if we display scrollable results, we would like results to be correctly truncated, notably to avoid situations in which we initially display a result ending in a sequence of 0's when the true result instead contains a sequence of 9's. E.g., we don't want to start out displaying 17.0000000 when the correct result is 16.99999999999999999999. Doing so would require that the leading digits change as we're scrolling, so as not to violate our first constraint, possibly confusing the user. (Note that such a bait-and-switch display is consistent with the accuracy requirement in the first constraint, while switching from 9's to 0's is not.)
4. We would like expressions involving domain errors to terminate with an error, instead of resulting in a nonterminating computation. For example $1 \div 0$ should result in an immediate divide-by-zero error, rather than a delayed timeout or the like.

All of the constraints, other than the first, require that we be able to perform exact comparisons on values. For the second one, it suffices to determine whether a result is exactly equal to the result without the trailing zeroes. For the third one, a similar comparison would tell us whether the actual result is less than the displayed value. For the last one, we need to determine whether e.g. a divisor is zero.

3.2 Testing Floating Point Arithmetic

The Java Language[20] makes very specific guarantees about the accuracy of floating point operations. Basic arithmetic operations like addition and multiplication are required to be correctly rounded, implying that they differ by no more than half an ulp (unit in the last place, i.e. the value of the last represented bit) from the correct answer.[2] Mathematical functions in `java.lang.Math` have various other precision specifications. For example, the exp, log, and trigonometric functions are required to be within 1 ulp of the true result.

An implementation of the reals, of the kind we describe, can be used to simply compare floating point results with the true results, to check that the floating point results satisfy the desired constraints. We implemented such a checker for basic Java arithmetic, exponential, and trigonometric functions. We compute the difference in ulps between a true value t and a floating point value f as follows, where all arithmetic is done using our implementation of the reals, and we've elided cases that handle infinite floating point values:

```
int errorInUlps(t, f) {
  if (f == t) return CORRECTLY_ROUNDED;
  if (f < t) return errorInUlps(-t, -f);
  // We know f > t.
  p = nextSmallerFpValue(f);
  if (p > t) {
    pp = nextSmallerFpValue(p);
    return pp > t ?
      INCORRECT // > 2 ulp error
      : TWO_ULP_ERROR;
  } else {
    return f - t <= t - p ?
      CORRECTLY_ROUNDED;
      : ONE_ULP_ERROR;
  }
}
```

Note that if we were to check whether floating point arithmetic produced the correct answer for $2 + 2$, in the first line we would compare the 4 against 4. If we used conventional recursive real arithmetic, this would already diverge.

If we test this using some form of approximate comparison, e.g. by resorting to a higher precision floating point arithmetic, it becomes very hard to ensure correctness. A tiny error in the final comparison could cause us to decide incorrectly between CORRECTLY_ROUNDED and ONE_ULP_ERROR.

We provide an API that makes such tests easy to write, by abstracting away the tricky mathematics that would otherwise be required.

Our testing code, including the reals package, is available as open source.[9]

3.3 Other Applications

Although we focus on the preceding two applications here, there are other promising candidates. We conjecture that the vast majority of spreadsheets could be easily evaluated using our arithmetic, without objectionable slowdown.

As already mentioned in [10], there are also numerical problems for which the precision of the final answer depends on the precision of a small sub-computation. The technique we describe here would also be usable for such applications, though the decidability issues we focus on are typically less important.

Other possible candidates are solid modeling applications[34], and systems that require an accurate "ground truth" for analyzing floating point accuracy, such as [30] or [32]. In the latter case, decidable comparisons may be less important.

4 The Computer Arithmetic Landscape

Many types of computer arithmetic have been explored, some with the express goal of getting us close to mathematical real numbers. These include:

[2] We allow the midpoint between two representable floating point values to be represented by either value. The `java.lang.Math` spec is not completely clear on this point.

4.1 Floating Point

Most computer arithmetic is performed using machine floating point[18, 28], most commonly following the IEEE standard[24]. A real number is essentially represented by a rational approximation of the form $m \times 2^e$, where m is a signed binary fraction containing a limited number (e.g. 53 + sign) of bits, and the exponent is also represented as a small (e.g. 11 bit) signed binary integer.[3] As we already pointed out, this is generally sufficient for most calculations involving physical quantities, and modern CPU cores can often perform billions of calculations per second. However it sacrifices:

- **Expected mathematical properties.** Addition is no longer associative, $x + 1 - 1$ may be wildly different from x, a loop repeatedly adding 0.1 to 0.0 until the answer becomes 10.0 may not terminate, etc.
- **Expected behavior.** When results are made visible to users, they expect to see familiar results, similar to what they might produce with decimal arithmetic on pencil and paper. When adding 0.7 and 0.1, users expect to see 0.8, not 0.7999999999999999, as is produced by IEEE double precision with Java output formatting rules.
- **Ease of solution for some problems.** As we point out in [12], we cannot easily approximate rate of change, or derivatives by computing very small finite differences. Precision limitations will completely hide the small differences and invalidate the results unless the approach is carefully tailored to the limitations of machine floating point. This usually applies to mathematical experiments, not physical problems. But, especially in an educational context, those may be important applications.
- **Simplicity.** Although answers are usually "close enough", the analysis required to convincingly demonstrate that is typically complex, and requires expertise that is rare among programmers, and essentially nonexistent among calculator or spreadsheet users.

IEEE floating point arithmetic guarantees correctly rounded results for individual basic arithmetic operations. Implementations perhaps should, though commonly do not, also provide correctly rounded results for trigonometric functions and the like. But the same is not true for composite expressions, and it may be quite difficult to obtain any reasonable error bound for those.

We already saw a few examples in which even small composite floating point expressions produce no useful digits at all. As another example, observe that $\arctan(x)$ approaches $\frac{\pi}{2}$ as x gets large. So we may be interested in understanding how quickly it does so, and thus want to compute e.g. $\frac{\pi}{2} - \arctan(10^{15})$. The Java double precision (IEEE floating point) answer is $8.881784197001252 \times 10^{-16}$, while the true answer is almost exactly 10^{-15} (to within 30 digits). If we instead compute the difference for $\arctan(10^{20})$, the floating point answer becomes zero, while the true answer is almost exactly 10^{-20}. The pattern exposed by the true answers is completely hidden by their machine floating point counterparts.

This kind of error is an unavoidable consequence of the representation: $\frac{\pi}{2}$ differs from $\arctan(10^{20})$ only in the 60th bit, but an IEEE double precision floating point value is accurate to at most 53, both values map to the same floating point representation.

As a result of this behavior, it is often undesirable to directly expose users to floating point arithmetic results, though clearly machine floating point will continue to rule in performance critical applications that need more than simple integer arithmetic.

4.2 Rational Arithmetic

It is possible to avoid all these problems for rational numbers, including all values exactly representable using floating point, by simply representing the number as the numerator and denominator of a fraction, each of which is in turn represented by an unbounded integer.[4]

However this is clearly an incomplete solution; for our example from the last section, neither of the values we subtracted were rational. To attack this problem, we would again have to approximate. For any fixed approximation precision we choose, there will be an n such that $\frac{\pi}{2} - \arctan(10^n)$ will produce a completely incorrect answer.

Clearly, we sacrifice large amounts of performance by moving from machine floating point arithmetic to exact rational arithmetic, a disadvantage shared to some extent with our proposed solution. However, in the case of rational numbers, this is compounded by the fact that if we naively add $\frac{a}{b}$ to $\frac{c}{d}$, we get $\frac{ad+bc}{bd}$, which may or many not be reducible, at significant additional computational expense. If it is not, or we choose not to, and each of the input numerators and denominators contains n bits, then the output number is represented as roughly $4n$ bits, i.e. as big as the total size of the inputs. As expressions become deeply nested, this can compound quickly. And indeed, our experience has been that results rapidly grow in size as we compose expressions.

4.3 Algebraic Arithmetic

It is also possible to perform exact arithmetic on algebraic numbers, that is numbers that are the roots of polynomials equations over the rationals (cf. [16]). This is also quite computationally expensive. It appears to be of interest primarily for computational geometry, which naturally deals with algebraic numbers. For our purposes, it remains problematic, for

[3] We gloss over details like gradual overflow and the implied one bit that are not relevant to this discussion.

[4] Other representations, such as continued fractions, are also possible. For present purposes, they are equivalent.

the same reason that rational arithmetic is: computational expense, and the restriction to a subset of the operations we're interested in.

4.4 Recursive Real Arithmetic

This serves as the starting point for our work. It uses a fundamentally different approach: A real number x is represented as a computable function f_x mapping an error tolerance e to a rational number $f_x(e)$, such that the difference between x and $f_x(e)$ is less than the error tolerance e. This can be viewed as an implementation of the real numbers from constructive real analysis[6, 7]. The numbers that can be represented in this way are generally referred to as the *constructive reals*, *computable reals*, or *recursive reals*. We use the latter term.

From a practical perspective, this differs from other approaches to multi-precision arithmetic in that when the user specifies only the final tolerance, i.e. the argument to f_x. It is the implementations responsibility to determine calculation precision for intermediate results sufficient for the final result to obey the error tolerance.

We use our open source implementation from [11],[5] which makes the approach more tractable by choosing a particularly convenient specification of the error tolerance e, and restricting the values $f_x(e)$ to a particular kind of binary representation. It is easily shown that this does not change the set of representable numbers.

In particular, the error tolerance e is specified as a (commonly negative) integer n, which specifies that the rational approximation should be accurate to 2^n. The resulting rational approximation is specified as an unbounded integer m, such that $|m2^n - x| < 2^n$.

In this formulation, addition becomes

$$f_{x+y}(n) = \lfloor \frac{f_x(n-2) + f_y(n-2)}{4} \rceil$$

In effect, to compute the result to $-n$ bits to the right of the binary point, we evaluate each of the arguments to $(-n) + 2$ bits to the right of the binary point, add those approximations, and adjust for the scale factors. Each argument evaluation contributes an error of strictly less than $\frac{1}{4}^{\text{th}}$ of 2^n, and the final rounding ($\lfloor \rceil$) can add another error of $\frac{1}{2}$ of 2^n. Thus the final error is strictly less than 2^n.

As another example, to compute the multiplicative inverse $\frac{1}{x}$, for nonzero x, we first evaluate x to increasing precision until we get a result that bounds x away from zero. We can do this by calling f_x with decreasing arguments until we get $f_x(n) >= 2$ for some n. From that it is straightforward to calculate the precision required for x to guarantee sufficient precision in $\frac{1}{x}$.

The multiplication algorithm is described in e.g. [10]. For transcendental functions, we generally pre-scale the argument so it is in a range where the Taylor series expansion converges rapidly. To produce approximations, we determine how much argument precision, extra calculation precision, and how many terms we need to ensure an approximation that differs from the true result by less than $\frac{1}{2}$ the value of the last requested bit. We evaluate the Taylor series accordingly, and finally round to the requested precision, again ensuring a final answer differs from the true answer by strictly less than one in the last requested bit.

Pre-scaling of trig function arguments requires a similar exact representation of π. We use the Gauss-Legendre algorithm for generating the approximations.

In this particular implementation, functions are represented by Java objects, with an `approximate()` function that corresponds to applying the function.

Note that the number of bits we compute still increases as expressions nest but, at least for pure sums, more slowly than for rational arithmetic.

In general, many different implementation approaches for recursive real arithmetic have been explored.[8, 13, 14, 21, 26, 27, 29, 36–38] The iterated variable precision interval arithmetic of [1, 2] is also closely related. Any of these could be used here, with different performance consequences.

4.5 (Practical) Decidability

Although we can evaluate a recursive real number to arbitrary precision, there is no immediate way to tell whether, for example, a particular x, represented by a function f_x represents the recursive real number one. We can repeatedly evaluate f_x demanding higher and higher precision. If $x \neq 1$, we will eventually discover that. But if they are equal, this process will continue forever. For the pure recursive real implementation we outlined above, even testing whether 4 is equal to 4 would fail to terminate. Similarly computing $1 \div 0$ would compute forever using the above representation, as we try to evaluate 0 to forever increasing precision to bound it away from zero.

In fact, it is straightforward to show that it is undecidable whether two arbitrary recursive real numbers are equal. If it were possible to decide equality, we could easily decide whether an arbitrary program P halts. Simply define a recursive real number x between 0 and 1, whose i^{th} bit to the right of the binary point is 0, if P halts after no more than i steps, and 1 if not. Each bit is easily computable, and it is easy to define f_x in terms of P. To decide whether P halts we simply have to determine whether $x = 1$.

However, this does not fully resolve the issue for us. There is no way to request that a conventional scientific calculator compute the number x above; it can clearly not be described using the operations normally provided by a calculator.

We really want to ask an alternate, less well-studied, question: Can we decide equality for recursive reals computed

[5] Our updated version can be found at https://android.googlesource.com/platform/external/crcalc/+/master/src/com/hp/creals.

from integer constants by a combination of the following operations, which we will refer to as "calculator operations":

1. The four basic arithmetic operations, and square roots.
2. The sin, cos, and tan trigonometric functions and their inverses.
3. Exponential and (natural) logarithm functions.

Although this does not appear to be widely known, and is not immediately apparent from the paper, the core of this question was in fact largely resolved in [31]. They give an algorithm to determine whether a constant described by a system of equations over the complex numbers is equal to zero. This algorithm eventually terminates unless the problem contains a counterexample to Shanuel's conjecture.

Here the equations have one of the following two forms:

1. A (multivariate) polynomial is equal to zero.
2. e^x = y

All of our "calculator operations" can be encoded in such a system. For example, $y = \sin(x)$ can be written as $y = \frac{e^{ix}-e^{-ix}}{2i}$, which can easily be written in the required equation form by introducing new variables for ix, $-ix$, etc. Thus constants can be described in the expected form.

Although [31] more or less gives a decision procedure, we are not aware of any implementations, or any arguments about practicality.[6]

Half of the decision procedure consists of determining inequality by repeatedly numerically evaluating to higher precision, which basically corresponds to evaluating the recursive reals involved in a comparison to increasing precision. We do not believe that this can yield a practical solution for us: The the difference between recursive reals may be tiny but nonzero, even for recursive reals that are trivially computable using standard calculator functions. Consider $1 - e^{-e^{1000}}$, which can be distinguished from 1 only by computing far more digits than there are atoms in the universe.

Although this is certainly not a proof that there exists no practical decision procedure, we will proceed on that assumption. Even if there were a practical decision procedure for this subset of the recursive reals, it would be unsatisfying to use it, since it depends on the particular choice of supported functions. All of the arguments we've discussed here would be invalidated by adding, for example, the gamma function.

We observe that while floating point does provide (very efficient!) decidability of comparisons, it does not really have an advantage here. In the cases of equality, in which recursive real arithmetic might diverge, floating point would give us a quick, but generally meaningless answer; it would only tell us whether the two computed approximations happened to result in exactly the same bit pattern. Approximate comparisons of recursive reals are also computable and more informative, in that the error margin can be easily specified. But we want to exactly compare the true values, a much more demanding requirement.

4.6 Why Decidability Matters

As we pointed out above, producing results from which we can extract arbitrarily precise approximations is usually helpful, but commonly doesn't solve the whole problem.

For floating point arithmetic testing, we want to compare errors in floating point computations against the exactly evaluated specifications. Having this comparison diverge because the two numbers happened to be exactly equal is very undesirable, and in fact was an issue with earlier, less refined versions of our approach. It is possible to work around this by allowing an error tolerance in the comparison, but that means we are checking against an approximation of the specification, not the actual specification.

In order to have the calculator display $11.36 when the user adds $7.23 and $4.13, it is desirable for the calculator to determine that $11.36 is exactly the correct answer, and there is no point in giving the user the option of scrolling through trailing zeroes to get more precision. In fact, in easy cases like this, it is our experience that users do not tolerate the trailing zeroes.

To determine whether we want to display a potentially scrollable result as 17.0000000 or 16.99999999, we would like to know whether the true answer is $\geq$ 17. If we just want our calculator to round the result of a complex expression correctly, we have a similar issue.

In the absence of decidable comparisons, there is no straightforward way to write conditionals in our code. Only continuous functions on the recursive reals are computable; anything else will diverge at the discontinuity. And something as simple as an absolute value function cannot be written in terms of a simple conditional; we instead need to retest the condition every time we re-evaluate the final result to a higher precision.

5 An API Proposal

Given the tension between the need to guarantee absolute (as defined here) accuracy in the result, the need to support even transcendental numbers, and the desire for a decision procedure wherever possible, we converged on an API that provides the following functionality:

- The expected arithmetic, trigonometric, exponential and logarithmic functions and square root. We attempt to detect domain errors, such as division by zero, and throw exceptions. Under unusual circumstances, domain errors may still cause the computation to diverge. Under very rare conditions a function whose domain is a (partially) closed interval, like square root, or the inverse trigonometric functions, may initially produce an answer, but will then produce an error if too much

[6] Although this paper describes constants via systems of equations involving polynomials and complex exponentials, this problem is different from questions about whether systems of equations involving transcendentals have a solution. The latter problem is generally undecidable (cf. [25]).

precision is requested. This happens for example, for $\sqrt{-10^{-10000}}$. Since we cannot always check for a non-negative argument, initial evaluations of the argument will yield zero, but we will detect the error if we ask for a sufficiently accurate square root. In the absence of programmer-induced domain errors, these behave exactly as expected.

- Like the underlying recursive reals package, we provide approximate comparison operations that always terminate, but may erroneously consider two numbers equal if they are within the specified tolerance.
- An exact comparison function that may diverge in case of equality.
- A binary `isComparable()` function that determines whether two numbers may be safely compared without risking divergence. If this function returns `true`, then the two numbers may be compared exactly without risk, and "approximate" comparisons return accurate results. `isComparable()` should return `true` at least in "easy" cases where humans would have no trouble comparing values.
- Unary functions `definitelyRational()`, `definitelyIrrational()`, `definitelyAlgebraic()`, and `definitelyTranscendental()`. It is not yet clear how useful it is to make these user-visible. But they are used heavily by `isComparable()`.
- A `toStringTruncated()` function that, given a real number and a desired number of digits, produces approximations, accurate to the specified number of digits beyond the decimal point, for output. An `exactlyTruncatable()` function tells the user whether the result is correctly rounded towards zero, or may, on rare occasion, have a last digit that is one too high. (It is undecidable whether a general recursive real number truncates to a specific value. The same argument we used above for equality to one applies.) A `digitsRequired()` helper function determines if the result has a finite decimal representation, and the number of digits required to represent it exactly.
- Conversions to and from standard numeric types, including exact conversions from floating point types, and rounding conversions to floating point types, etc.

The existence of the `isComparable()` function makes it possible, in many cases, to avoid any risk of divergence in comparisons, which then makes it possible to check for domain errors up front. We found that, with a surprisingly small amount of effort, we could ensure that `isComparable()` returns true in most common situations.

As in the recursive reals case, all arithmetic operations logically produce exact answers. Thus standard mathematical properties such as associativity, hold. So long as only exact comparison is used to test for equality, and comparison and `toStringTruncated` are only used when `isComparable()` and `exactlyTruncatable()` yield true, everything behaves according to normal mathematical rules.[7]

Otherwise there remain cases in which approximations are visible. When `exactlyTruncatable(x)` or `exactlyTruncatable(x)` is false, `toStringTruncated(x, nbits)` may differ from `toStringTruncated(y, nbits)`, even if mathematically `x = y`. Similarly, again with `x` and `y` mathematically equal, if `isComparable(x, z)` is false, an approximate comparison of `x` and `z` to 100 bits might return true, while the same comparison of `x` and `z` returns false.

6 Implementation

An open source Java implementation of our reals package can be found in [9]. We outline the approach here.[8]

After some experimentation with more restrictive approaches, we now use the following representation for a real number. A real number is represented by three observationally immutable[9] fields:

```
BoundedRational ratFactor;
RecursiveReal rrFactor;
Property rrProperty;
```

The `rrFactor` field is a recursive real number as previously described, and is always present. The `rrProperty` field may be `null`. If it is present, it provides symbolic information about `rrFactor`. Often this symbolic information is sufficient to allow `rrFactor` to be reconstructed from `rrProperty`. We redundantly store both in this case.

A `BoundedRational` is a rational number represented as two Java `BigIntegers`: a numerator and denominator. Most operations ensure that either the denominator is one, or the total size of the representation is bounded by a constant (currently 10,000 bits). The limit ensures that we avoid catastrophic size blowups that could result from a purely rational representation.

The represented real number is the *the arithmetic product* of `ratFactor` and `rrFactor`.

Whenever the bound on rationals would naturally be violated, we instead use a `ratFactor` of one, and use `rrFactor` to directly represent the number.

Note that the size limit on the rational component is relevant even for the calculator application: $1 + 10^{-10000}$ has a large rational representation.

A `Property` contains two fields, a `tag` field, logically of enumeration type, and a `BoundedRational` function argument `arg`. In most cases, this indicates that `rrFactor` is $f(arg)$, where f is determined by `tag`. There are currently

[7] We ignore resource exhaustion issues which, in our implementation, result in exceptions.

[8] For clarity, identifiers here often do not match those in the code. The reals implementation described here is called `UnifiedReal` in the code. The recursive reals implementation from [11] is called `CR`.

[9] Our recursive real numbers internally cache the best known approximation to avoid redundant re-computation. This cached value is protected by a lock and mutated as the number is reevaluated.

```
Real add(Real x, Real y) {
  if (x.rrProperty == y.rrProperty
      && neither is null or IRRATIONAL tag) {
    rat = x.ratFactor + y.ratFactor;
    if (rat != null /* didn't overflow */) {
      return (rat, x.rrFactor, x.rrProperty);
   }
  }
 Handle sum of logs as described in caption.
  Property p =  null or IRRATIONAL, as described below;
  RecursiveReal xVal = x.ratFactor * x.rrFactor;
  RecursiveReal yVal = y.ratFactor * y.rrFactor;
  return (1, xVal + yVal, p);
}
```

Figure 1. Addition of reals. Triple notation is used for construction of reals from the three field values. When adding $a\ln(b) + c\ln(d)$, the result has property $\ln(b^a \times d^c)$.

tag values to indicate that `rrFactor` is 1, π, $\sqrt{\texttt{arg}}$, $e^{\texttt{arg}}$, $\ln(\texttt{arg})$, $\log_{10}(\texttt{arg})$, $\sin(\pi\texttt{arg})$, $\tan(\pi\texttt{arg})$, $\arcsin(\texttt{arg})$, or $\arctan(\texttt{arg})$.

We also use one additional tag value to indicate that `rrFactor` is irrational, without specifying the exact value.

All arithmetic operations try to attach a property to the result whenever possible. For example, when adding two numbers whose recursive real factors are the same, we simply add the rational factors and preserve the recursive real factor together with its property, subject to correct handling of rational overflow. Figure 1 gives pseudo-code for addition. Negation simply negates `ratFactor`.

Multiplying a number by a known-rational number (i.e. one that has an `rrFactor` and `rrProperty` of one) preserves the original `rrFactor` and `rrProperty`. If both operands have properties identifying them as square roots, or both are identified as applications of the exponential function, we can again associate a similar property with the result.

As another example, Figure 2 gives pseudo-code for our `sin` and `cos` implementations. The `reduceArg()` function reduces the rational multiplier of a multiple-of-π argument to the range $[-\frac{1}{2}, \frac{1}{2}]$ using $\sin(2\pi+x) = \sin(x)$ and $\sin(\pi-x) = \sin(x)$. (For simplicity, we've omitted the further reduction to $[0, \frac{1}{2}]$ using $\sin(-x) = -\sin(x)$.)

The implementation of the `asin()` function treats an argument with property $\sin(\pi x)$ appropriately. Division understands that dividing $y\ln(x)$ by $\ln(10)$ results in $y\log_{10}(x)$.

We also try to simplify when possible, and normalize the rational arguments in properties to as small a range as possible. These simplifications never remove a property; an application of any of the functions described by a property to a sufficiently small rational number produces a result with a non-null property. Specifically, we constrain property arguments as follows:

```
Real sin(Real x) {
  if (x.rrProperty is $\pi$) {
    if (x.ratFactor is a multiple n of $\frac{1}{12}$) {
      // Handle rational results etc.
      return lookupSpecialSin(n % 24);
    }
    Property p = (SIN_PI, reduceArg(x.ratFactor));
    return (1, sin(x.rrFactor), p);
  }
  if (x.rrProperty is ASIN(y))
    return (y, 1, ONE);
  Property p =
    x.definitelyAlgebraic() ? IRRATIONAL : null;
  return (1, sin(x), p);
}

Real cos(Real x) {
  Real result = sin(x + $\frac{\pi}{2}$);
  if (result.rrProperty == null
      && x.definitelyAlgebraic()) {
    return (result.ratFactor, result.rrFactor,
            IRRATIONAL);
  } else {
      return result;
  }
}
```

Figure 2. Sine and cosine

$\sqrt{x}$ Argument > 0, and not a perfect square. A perfect square is always simplified to a rational number times the recursive real number one.[10] We use our recursive reals implementation to identify perfect squares with large numerators and denominators.

e^x Argument is not zero. In the zero case, we just represent the exponential as 1.

$\ln(x)$ Argument is > 1. Otherwise we convert to minus the log of the reciprocal.

$\log_{10}(x)$ Argument is > 1, and not a power of 10. If this is rational, then $n\ln(x) = m\ln(10)$, or $x^n = 10^m$, n and m integers. For rational x, this is only possible if x is a power of 10, which we excluded. Thus this is always irrational.

$\sin(\pi x)$, $\tan(\pi x)$ Argument is strictly between 0 and $\frac{1}{2}$. Argument is not $\frac{1}{6}$, $\frac{1}{4}$, or $\frac{1}{3}$ (corresponding to 30, 45, and 60 degrees). We reduce out-of-range arguments as discussed for `sin()` above. For the specific values we exclude, there are alternate ways to represent the result using at most square roots. Niven's Theorem tells us that this precludes all rational arguments for which the function result is rational; thus any recursive real described by such a property is again irrational.

[10] It might be desirable to normalize to $x > 1$, but we currently do not.

arcsin(x) Argument is strictly between 0 and 1, not equal to 0.5 (which simplifies to $\frac{\pi}{6}$). Since the sine of a rational number of radians is rational only at zero, this is irrational.

arctan(x) Argument is positive, and not equal to 1 (since arctan(1) simplifies to $\frac{\pi}{4}$)). Laczkovich's proof of Lambert's Theorem shows that this is again always irrational.

In the same spirit, we do not have a $\cos(\pi x)$ property, and instead compute $\cos(x)$ as in Figure 2.

For sufficiently small (in representation size) known-rational numbers, we always ensure that the recursive real factor is one and tagged as such.

The following observation is critical for providing accurate results from `isComparable()` in most practical situations:

Theorem If a recursive real number is described by any of the above properties then it is rational if and only if that property describes the constant 1.

Proof Clearly this holds for the constants 1 and π. For other properties except e^x and $ln(x)$, we already showed that our argument restrictions preclude describing a rational value.

For these two remaining cases, and for some observations below, we rely on the Lindemann-Weierstrass (or Hermite-Lindemann) Theorem, which can be stated as[5, 39]:

> If α_1, ..., α_n are distinct algebraic numbers, then the exponentials e^{α_1}, ..., e^{α_n} are linearly independent over the algebraic numbers.

It follows immediately that e^x, for non-zero x is irrational (in fact non-algebraic), since it must be linearly independent over the rational numbers (and more generally algebraic numbers) from $e^0 = 1$.

Similarly if we had $ln(x) = y$ where x and y are both rational, $x = e^y$, implying that either $x = 1$, which we disallowed, or contradicting the preceding observation. Thus a recursive real described by an $ln(x)$ property must also be irrational. •

The implementations of the `isComparable()` and comparison methods rely heavily on the property part of the number representation, as does the implementation of `exactlyTruncatable`. But we can produce arbitrarily precise approximations from a number without consulting the property at all.

6.1 Computing `isComparable()`

Our algorithm for computing `isComparable()` is outlined in Figure 3. We check in turn:

1. Do both numbers share the same (as determined by the property) known-non-zero (as determined by the property) recursive real factor?
2. Are both rational factors zero?
3. Can we prove (using the properties) that neither recursive real is a rational multiple of the other? We also explicitly check that at least one of the operands has an absolute value > 2^{-5000}. This ensures that we can find the nonzero digits in a reasonable amount of time.
4. Do they have the same rational factor, and the same property `tag`? Our properties are either constant or monotonic in the allowable `arg` values, so we can just compare those.
5. Do both numbers have properties of the form $\sqrt{x}$ (for different x). In that case they can easily be compared by comparing the (necessarily rational) squares.
6. Finally, can we prove they are different by just evaluating to a fixed precision?

```
Real isComparable(Real x, Real y) {
  return ((x.rrProperty == y.rrProperty &
           && neither is null)
          || x.ratFactor == y.ratFactor == 0
          || (definitelyIndependent(x, y)
              && either x or y > 2ˆ−5000)
          || (x.ratFactor == y.ratFactor
              && both have same rrProperty.tag
              && tag is not IRRATIONAL)
          || (x.rrProperty.tag == SQRT
              && y.rrProperty.tag == SQRT)
          || approximate comparison of x and y
             shows them definitely not equal);
}
```

Figure 3. isComparable implementation. The `definitelyIndependent` function returns true if we can determine, based on `rrProperty` values, that neither argument is a rational multiple of the other.

The third criterion is surprisingly powerful, largely due to the preceding theorem. We knew that we can compare two known rationals, i.e. numbers with a recursive reals factor of 1, just by comparing the rational factors. We now also know that it is safe to compare a known rational x against a number y with any other property, provided that its rational factor is also nonzero. The number y is guaranteed to be irrational since its property describes an irrational recursive real factor, and multiplying by a nonzero rational doesn't alter that fact. Thus x cannot be equal to y, and evaluating them to increasing precision is guaranteed to eventually determine which is larger.

In addition to relying on the above theorem, `isComparable()` can return true via the third criterion in a number of cases in which neither argument is rational, but properties are not null:

$\sqrt{x}$, $\sin(\pi x)$, $\tan(\pi x)$ Clearly square roots are algebraic numbers. The same is true for trig functions applied to rational multiples of π. Lindemann-Weierstrass actually shows that e^x, $ln(x)$, arcsin(x), and arctan(x) properties ensure that the described recursive real is transcendental. A transcendental number clearly can't be a rational multiple of an algebraic one or vice-versa.

Thus a number with a property in the former group can be compared against one with a property in the latter, by the same argument we used for comparing against rationals.

e^x It follows from Lindemann-Weierstrass that if $x \neq y$ then e^x cannot be a rational multiple of e^y. So long as one of the rational factors is nonzero, rational multiples of these can be safely compared.

$\ln(x)$, $\log_{10}(x)$ Two numbers with properties $ln(x)$ and $ln(y)$, can be rational multiples of each other only if there are integers m and n, such that $y^m = x^n$, which can be efficiently determined for integers with a gcd-like algorithm:
We observe that if $x = y^r$, where r is rational, then $(x/y) = b^{r-1}$. We divide the larger of x and y by the smaller. If the remainder is nonzero, there are no such exponents. Otherwise we repeat the process with the quotient and the divisor. The rational case can then be reduced to the integer case.
The same applies to base 10 logarithms.

The net effect of all of this is that whenever we can generate properties describing the recursive real components of numbers being compared, we can usually guarantee convergence of exact comparisons.

6.2 The "Irrational" Tag

So far, we have focused on properties that precisely describe the recursive real number. However, we would also like to know that e.g. $sin(x) + 1$ (x rational, nonzero) is safely comparable to a rational number. After all, we know that $sin(x)$ is irrational, and thus so is $sin(x) + 1$.

Since we currently do not explicitly track $sin(x)$ for rational (radian) arguments, we simply record the fact that it is irrational. The addition function preserves this tag when the properties tell us that neither argument is a rational multiple of the other. Thus $sin(x) + 1$ is tagged as irrational, allowing it to be compared to rationals.

This unfortunately needs to be done with some caution to preserve its practical utility. By blindly going ahead and comparing rational with irrational numbers, we are effectively assuming that they differ after some reasonable number of digits, and thus don't need ludicrous precision to do so. Generally that has proven to be a safe assumption. Many of us recognize Ramanujan's constant $e^{\pi\sqrt{163}}$ as very special because it differs from an 18 digit integer only in the thirteenth digit behind the decimal point. Thus it would be very surprising to randomly find an expression of reasonable size that looks like an integer to within thousands of digits, which it would take to cause us serious problems.

But we do know that it is easily possible to construct such expressions by adding very small values to much larger ones. Our previous example of $1 - e^{-e^{1000}}$ makes this point well.

To avoid this issue, we currently refuse to use the irrational tag on sums (and thus differences) if either operand is not known to be zero, but its most significant bit may be more than 3500 bits to the right of the binary point.[11]

6.3 Comparison

The actual comparison function is quite simple:

1. If both arguments are known zero, they're equal.
2. If both arguments have the same recursive real factor, either based on equivalent properties, or because the actual recursive real values are pointer-equivalent, the result is easily determined from the sign of the recursive real factor, and a comparison of the rational factors. Determining the sign of the recursive real factor may diverge if we have no property information. That's acceptable, since `isComparable` would have returned false in such cases.
3. Do both values have the same rational factors, and the same kind of property, i.e. did test (4) in the `isComparable()` tests above succeed? If so, compare the property arguments.
4. Check (using properties) if both values are square roots of rational numbers. If so, compare the squares.
5. If none of the preceding succeeded, multiply the rational and recursive real factors as recursive reals, and compare increasingly precise approximations until they differ. At this point we know that either `isComparable()` returned `false`, or either test (3) or (6) succeeded. If either (3) or (6) succeeded, this is guaranteed to converge. If not, divergence is acceptable.

6.4 `toStringTruncated()`, `exactlyTruncatable()`, and `digitsRequired()`

In our implementation, the `exactlyTruncatable(`x`)` function returns true if x is known rational or known irrational. The `toStringTruncated(`x, n`)` function then simply generates an approximation including n digits to the right of the decimal point, with an error of < 1 in the last digit. [12] We then compare the resulting rational approximation to x, knowing from our previous arguments that this comparison cannot diverge. If the approximation is greater, we subtract one from the last digit of the approximation, otherwise we return the approximation directly. Based on the error bound, we know that this must be correct. `digitsRequired()` checks whether the result is rational, reduces it to lowest terms, and then effectively examines the factors of the denominator.

6.5 Alternatives

Numerous alternate representations could be combined with similar algorithms to implement our interface. Our notion of

[11] The value is chosen to allow adding 10^{-1000} without losing information, but otherwise arbitrary.

[12] More precisely, we generate an approximation of $x \times 10^n$, with an error of $< \pm 1$

"property" has the advantage that it supports sufficient symbolic manipulation to allow decidability comparisons in the most interesting cases. And it appears to mesh astonishingly well with classical number theory results.

We could choose to perform symbolic manipulation on more general expressions, like $\pi + e$. But knowing that a recursive real factor is $\pi + e$ is less directly helpful, since even in this simple case, it is not known whether the expression is rational and thus safe to compare to a rational number.

We chose to always store the recursive real factor, even if we have a property describing it exactly. That's not technically necessary, but it has the advantage that we don't have to repeatedly generate a recursive real representation from the same property, and we more effectively use the caches that are maintained when approximations of recursive reals are computed.

7 Evaluation and Experience

Our goal here is to evaluate practical usability in the context of the motivating applications we listed earlier. We primarily address the following questions:

1. Does the API we specified support the intended applications well?
2. Is the implementation we outlined sufficiently small, and does it perform sufficiently well to be usable in these contexts.

7.1 Limitations and Challenges

It is unclear what the correct performance metrics are. For the testing application, we perform a large number of separate computations and comparisons. For other applications expressions might be deeply nested, leading to increased precision demands in the deeper sections of the expression tree. For the calculator, runtime performance rarely matters, but when it does, it is generally because we're evaluating a huge result, scrolling through many digits of a result, or evaluating an extremely numerically unstable expression. In these cases, it matters much more that we can evaluate a single expression to often ridiculously high precision quickly, than that we can evaluate many expressions in a row. Thus we give measurements and anecdotal evidence covering several diverse use cases.

Actual timing is largely determined by performance of the Java BigInteger package, which underlies both our bounded rationals and recursive reals implementations. This varies widely, and may be optimized for a very different target, notably cryptography. The measurements we do include use OpenJDK 11, which generally performs quite well, but still uses 32-bit digits, which we expect to leave at least a factor of two in performance on the table.

Another important performance factor are the algorithms used to implement functions on the recursive reals. Our implementations are designed to keep the code simple, while avoiding anomalies that could lead to unacceptable performance "cliffs", i.e. inputs with dramatic unexpected performance degradation. For example, the recursive reals package we started with would compute e^x for negative x as $\frac{1}{e^{-x}}$. While mathematically correct, this means that something like $e^{-10^{100}}$ ends up falling off such a performance cliff. Overlooking such issues may hugely inflate running times, often to the point of nontermination.

7.2 Deployment Experience

We updated Google's Android calculator[12] with the reals package described here[13] The application is preinstalled on many devices, and Google Play now reports that we recently passed the 500 millionth installation.

We use our `digitsRequired()` function to determine when we have a finitely displayable result, and `toStringTruncated()` to display it, ensuring that we always display a correctly truncated result, when `exactlyTruncatable()` holds, i.e. for results that are known to be rational or known to be irrational. In these cases, leading digits cannot change from zeroes to nines as the user scrolls through the digits of a result.

We also use the partial decision procedure to more aggressively report domain errors, such as division by zero.

Our reals package is quite modest in size, making it suitable for a small mobile application. The underlying preexisting recursive reals package is fewer than 2500 lines of Java, significant pieces of which we don't use. Our bounded rationals package is about 800 lines of code, and the actual reals package, including the code to manipulate properties, is again a bit fewer than 2500 lines. (This includes comments, but excludes tests and the underlying Java-supplied BigInteger package.)

Experience with the calculator from[12] had already been very positive, and the application has always been highly rated. But earlier versions resulted in a few complaints about integral results displayed with infinite sequences of trailing zeroes. Our current implementation, though not completely immune from this, addresses those complaints using the properties scheme described here. For example, $\ln(e^2)$ is now displayed as 2, and $sin(65^0) - sin(65^0)$ is now displayed as 0, as is, as a result of our argument normalization, $sin(65^0) - sin(115^0)$.

It's easy to construct formulas (such as $\pi^2 - \pi^2$) for which this is not the case (because our properties representation is too constrained to represent π^2), but we believe we've covered the cases that are likely to arise, e.g. in a calculus class, surprisingly well. And the calculator design is such that it remains entirely usable if our decision procedure fails

[13] As of early 2000, we use a slightly older version, which lacks some minor code cleanups from the testing code used below. The only observable difference is that the new version handles properties of square roots slightly better, but not in a way that is likely to ever be noticed in the calculator context.

in complex cases. We do not fail in must-have cases, such as when adding up a bill.

We receive voluminous (public) user feedback.[19] We no longer receive bug reports about inaccurate results, as we occasionally did for the 2014 floating-point-based calculator.[14] Recently we have also not seen complaints about unnecessary zeroes, or the like. Nor do we receive complaints about the amount of time it takes to complete calculations. The latency to display a calculator result has not visibly changed from [12]. E.g. $(1+10^{-1000})^{10^{1000}}$ still displays `2.71828182845` instantaneously on a modern phone and $(1 + 10^{-10000})^{10^{10000}}$ completes on most devices with a lag of at most a couple of seconds or so.

We have recently started to expose the properties associated with recursive reals to provide a symbolic representation of the result. That required extending the API to provide this representation of the result when available. That was appreciated by some (most?) users, and disliked by others who viewed it as noise.

The one significant difficulty of using such a package in the calculator context is that we cannot absolutely guarantee that computations will finish in a reasonable amount of time, or at all. Thus we always need to run actual computations in a background thread, and arrange for it to time out if necessary. This has not been a problem with users. But, together with the fact that the calculator's history mechanism allows multiple parallel computations, this significantly complicates the code base.

It is probably desirable to extend the API to support an alternate usage model in which computations can be limited to always complete in a reasonable amount of time, at the expense of occasionally spuriously producing a domain error or "couldn't do that" result. This would be analogous to floating point's infinite and NaN results, but would still ensure that if a result is displayed, it is correct.

7.3 Precision Testing and its Performance Measurements

We used our reals package to test the OpenJDK and Android implementations of many mathematical functions provided by the `java.lang.Math` package, as well as Java language division of doubles. We use the approach described in Section 3.2. This application also allows us to give a more quantitative evaluation of the performance of our package.

We sample a double precision value for each function argument by generating 64 pseudo-random bits, interpreting them as a Java `double`, and discarding those representing infinite or NaN values. This generates `double` values with a very wide exponent range, initially pointing out, and forcing us to fix, several performance cliffs.[15]

The precision specifications we check here usually compare known rational numbers, such as floating point numbers, or compare a known rational number to a known irrational one. Thus they are safely comparable, and there is usually no risk of nontermination, no matter what floating point values we happen to sample.

For all but the `pow` calls, we explicitly precede each comparison on reals by an `isComparable()` call, which always succeeds, as expected. In the `pow` case, the result may not be rational if the exponent is not an integer, and we will typically not have a property tag to represent the recursive real part of the result. Though we commonly recognize `pow()` calls that have rational results, we do not guarantee to do so. And most of the remaining cases cannot be represented by one of our properties. In this case we are still forced to use approximate comparisons. (We use an absolute error tolerance of 2^{-2000}.)

Encouragingly, we only found one minor violation of the `hypot()` precision specification, now filed as https://bugs.openjdk.java.net/browse/JDK-8229259.[16]

7.4 Performance Measurement Methodology

Our primary goal is to demonstrate sufficient performance for usability. Nonetheless, and in spite of the limitations we pointed out in Section 7.1, we give some performance numbers. We use a Lenovo 920 running Linux and OpenJDK 11, and using two Intel Xeon Gold 6154 CPUs and 192GB or RAM. All our benchmarks are single-threaded, except for any background compilation, garbage collection, etc. performed by the JVM. For most of the benchmark runs we observed slightly more than one of the 72 hardware threads in use. RAM was similarly underutilized. We used default parameters for the "java" command. We ran benchmarks 6 times in a row, discarding the results from the first run, and reporting the average of the next 5.

In all cases, we use the recursive reals implementation from [11]. We fixed a few "performance cliffs" in this package, as well as making some general improvements: We switched to the Gauss-Legendre algorithm for π (mostly for the benefit of calculator users scrolling through ridiculous numbers of digits of π). We compute inverse trigonometric functions more directly, rather than by using a more elegant but expensive generic algorithm for computing the inverse of a monotonic function. We also made access to cached approximations thread-safe. Although we try to use asymptotically reasonable algorithms, we generally did not try to tune the implementation to improve the running time by constant factors.

[14] This excludes reports from one or two bugs that have now been fixed for many months. Unfortunately, we continue to receive complaints about incorrect results, mostly for two reasons. Users often do not understand the difference between degrees and radians. Second, there is no standard way to parse calculator expressions. $1 + 10\%$ is 0.11. 10% is 0.1. What's $10\% + 10\%$?

[15] We now run a version of this test as a regression test for our package.

[16] This was initially discovered using approximate comparisons for `hypot()`, before we succeeded in improving the square root code to allow exact comparisons in this case.

Table 1. Floating point testing

function	µsecs	function	µsecs
division	3.0	`sqrt()`	100.0
`acos()`	77.6	`cos()`	121.5
`asin()`	81.0	`sin()`	840.2
`atan()`	110.3	`tan()`	925.7
`exp()`	47.0	`log()`	574.2
`log10()`	902.8	`pow()`	429.5
`hypot()`	404.8		

This choice of recursive reals representation performs adequately for the testing application, and appears to be quite well-suited to the calculator application. Other approaches, like those based on iterated interval arithmetic[2, 26] would probably perform better for applications involving deep nesting.

7.5 Testing Performance

Table 1 reports the amount of time, in microseconds, required to check the precision of a single function application to a random argument or arguments, as described in the last section. We time 10,000 iterations of the following logic:

```
generate random arguments x...;
if (arguments in f's domain) {
  ++numberOfOperations;
  res = f(x...) evaluated exactly;
  fpRes = f(x...) evaluated using double;
  check errorInUlps(res, fpRes) <= spec;
}
```

We report total running time divided by `numberOfOperations`. For all tests, `numberOfOperations` exceeds 4850. Results among the 5 runs varied by at most 10%, and usually by much less. We confirmed that random number generation and loop overhead adds at most 5% for the division test and is negligible for everything else.

The detailed numbers are highly dependent on our choice of algorithms and argument sampling. The fact that `cos()` is much faster than `sin()` can be explained by the fact that the recursive reals package implements `cos()` with prescaling and the standard Taylor series, which converges rapidly around zero. Random arguments are likely to be quite close to zero. `sin(x)` is implemented as $cos(\frac{\pi}{2}-x)$. For small x, this results in a `cos()` argument that is commonly much larger, such that `cos(x)` is very small. Thus we need high evaluation precision to compare to the floating point result. If `sin()` were implemented directly, the results would presumably be much better.

While the results will look embarrassing to anyone expecting floating point performance, we can perform each test in under a millisecond, often much faster. This is clearly within the acceptable range for our purposes. At a millisecond per operation, it would take about 1,200 CPU-core-hours to exhaustively test a unary single-precision floating point function, which is clearly within the feasible range (though we have not actually done so).

Table 2. Harmonic series evaluation times (msecs)

n	real	rr	real(div)	rr(div)
1000	3	11	1	8
5000	123	s.o.	12	37
10000	s.o.	s.o.	28	74

7.6 Nested Computation Performance

Neither the calculator nor testing application require the evaluation of deeply nested expressions, such as summing many elements, one at a time. On the other hand, an application like a spreadsheet may require summing of thousands of elements. A naive implementation, using our recursive reals package, results in an expression tree with one addition at each level, which is thus thousands of levels deep. Our recursive reals implementation requires two extra bits of precision for each level in the tree.

To give a rough idea of the performance that could be expected from such applications, we measured the time required to sum the first n terms of the harmonic series ($\sum_{x=1}^{n} \frac{1}{x}$) and print the result to 1000 digits. Unlike the preceding problems, this one does not actually require decidable comparisons, so we can compare to the underlying recursive reals `rr` package.

Results are given in Table 2. Here "real" denotes the running time with our package, "rr" that with the underlying recursive reals package, and "(div)" indicates the use of a divide and conquer algorithm that recursively sums odd and even terms, and then adds them. The expression trees for the "div" runs are much shallower. A spreadsheet could easily use the "div" algorithms; for other applications that may not be an option.

Since we timed individual runs, we saw larger variation, deviating up to 37% (or 1 msec for real-1000) from the reported mean. An entry of "s.o." indicates a stack overflow.

The "real(div)" results seem entirely acceptable. Perhaps surprisingly, we significantly outperform the underlying recursive reals package on this example; here it's clearly less expensive to perform the computations nearer the bottom of the expression tree on rationals, and resort to recursive real arithmetic only when that overflows, and would otherwise become too expensive. Since our implementation tries to preserve an `rrfactor` of one for rationals, this happens implicitly. We would not expect this behavior if we were summing irrational numbers.

8 Related Work

This is an extension of our earlier work in [12], which used a much more primitive and ad hoc arithmetic implementation that was not described in detail. [12] focused mainly on one particular motivating application for this kind of arithmetic, rather than a specific API and implementation of that arithmetic.

Recent versions of the IEEE floating point standard require that individual operations (as opposed to the composite expressions we deal with) be correctly rounded. I.e. they require that the error in the result *of a single operation* never exceed $\frac{1}{2}$ unit in the last place. This includes transcendental functions.[17] The argument that this is feasible, i.e. that the implementation can decide whether the true result is smaller than or greater than the midpoint between two representable floating-point numbers, essentially relies on variations of the Lindemann-Weierstrass theorem. We rely on a similar approach, but apply it much more generally to cases where neither argument is rational, and augment it with other algorithms to ensure that numbers can be safely compared.

There has been much work, both by computer scientists and mathematicians, on the recursive reals. A number of these explore decidability of a different nature (Cf [17, 35]). They avoid the fundamental decidability limitations of the recursive reals by allowing a small error, nondeterminism, or by not insisting that functions be computable everywhere, thus avoiding our restriction to "calculator functions", but also fundamentally failing to solve our problem. When performing division, we need an exact, total comparison to zero.

The only work of which we are aware that specifically addresses decidability for anything like the "calculator operations" subset of the recursive reals is [31], which we already extensively discussed above.

Wolfram Alpha seems to use some related techniques to attack a somewhat different problem. It doesn't produce numerical results for all our examples, focusing instead on much more substantial symbolic computation. We could not find a published description of its algorithms. Since they are focused on general symbolic algebra, we expect their implementation is far larger than our relatively compact one.

Microsoft's windows calculator application appears to use a mixture of rational and approximate floating point-like computations.[15]

Hardware-architecture-level alternatives to conventional machine floating point arithmetic have occasionally been proposed (for example [22, 23]). These address problems orthogonal to our concerns.

9 Conclusions

We have argued that it is possible to implement a computationally useful type that shares the basic mathematical properties of the real numbers, along with the commonly expected property that, in most commonly occurring cases, we can determine whether a pair of such numbers are equal. That ability translates into practical advantages, such as calculators displaying integral results as integers, and the ability to use our real number implementation as a reference for testing, in may cases without the possibility of either error or nontermination.

One could argue that this computational type also roughly reflects the way we reason about real numbers in real life, or at least real mathematics. In some cases, it is indeed very difficult to tell whether two numbers are exactly equal. But we still routinely use equality in other situations, e.g. when deciding whether someone paid their bill correctly.

Experience with calculator arithmetic suggests that one should avoid exposing non-expert users to results that visibly exhibit floating point rounding. This provides a principled way to indeed avoid that. Under extreme conditions, where floating point would have produced a meaningless result, it may fail to produce a result at all. But it will never lie.

It is also interesting to consider the impact of such a library type on library design in general. Recursive real numbers, with undecidable equality, cannot easily be put into conventional containers, like Java HashMap. Nor can we check simple preconditions, such as the fact that a `sqrt` argument is non-negative. Since we added "usually"[18] decidable equality, these now become "usually" possible, perhaps sufficiently so to be useful. One might be able to use e.g. the `floor()` function to compute a hash value, and use a fixed hash value when it's not.

Acknowledgments

I would like to thank Pavel Panchekha, who found [31] and participated in an enlightening discussion of the equality decision procedure. Robert Bradshaw suggested the gcd-like algorithm for arguments of logarithms during a code review, allowing it to replace the silly, much less precise, algorithm I previously used. My current and former colleagues Justin Klaassen, Annie Chin, Christine Franks, and Thomas Hall implemented most of the non-arithmetic parts of Google's Android Calculator, and dealt with the added complexity introduced by asynchronous arithmetic computation. Andreas Gampe helped to expose the `hypot()` bug. My shepherd, Michael Carbin, and the anonymous reviewers suggested many improvements and corrections.

References

[1] Oliver Aberth. 1974. A Precise Numerical Analysis Program. *Commun. ACM* 17, 9 (September 1974), 509–513.

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

[3] anynmous. 2014. Odd bug in calculator app. https://www.reddit.com/r/Android/comments/2ph1wk/odd_bug_in_calculator_app/

[17] It is unclear to us how commonly this advice is followed.

[18] With occurrence frequency determined by e.g. what expressions users enter into a calculator.

[4] ASKVG. 2012. Microsoft Windows Calculator Bug, Sqrt(4) − 2 != 0. https://www.askvg.com/microsoft-windows-calculator-bug/

[5] Alan Baker. 1990. *Transcendental number theory, Cambridge Mathematical Library (2nd ed.).* Cambridge University Press.

[6] Erret Bishop. 1967. *Foundations of Constructive Real Analysis.* McGraw-Hill.

[7] Erret Bishop and Douglas Bridges. 1985. *Constructive Analysis.* Springer-Verlag.

[8] Jens Blanck. 2000. Exact Real Arithmetic Systems: Results of Competition. In *Proceedings of the Workshop on Computability and Complexity in Analysis, Springer LNCS 2064.* 389–393.

[9] Hans-J. Boehm. 2019. Floating point precision test. https://android-review.googlesource.com/c/platform/art/+/1012109

[10] Hans-J. Boehm. 1987. Constructive Real Interpretation of Numerical Programs. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques.* 214–221.

[11] Hans-J. Boehm. 2005. The constructive reals as a Java library. *J. Logic and Algebraic Programming* 64 (2005), 3-–11. Issue 1.

[12] Hans-J. Boehm. 2017. Small-Data Computing: Correct Calculator Arithmetic. *Commun. ACM* 60, 8 (July 2017), 44–49. https://doi.org/10.1145/2911981

[13] Hans-J. Boehm, Robert Cartwright, Michael J. O'Donnell, and Mark Riggle. 1986. Exact Real Arithmetic: A Case Study in Higher Order Programming. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming.* 162–173.

[14] Keith Briggs. 2006. Implementing Exact Real Arithmetic in Python, C++ and C. *Theor. Comput. Sci.* 351, 1 (Feb. 2006), 74–81. https://doi.org/10.1016/j.tcs.2005.09.058

[15] Raymond Chen. 2016. Why does the Windows calculator generate tiny errors when calculating the square root of a perfect square? https://devblogs.microsoft.com/oldnewthing/?p=93765

[16] M Coste and M. F. Roy. 1988. Thom's Lemma, the Coding of Real Algebraic Numbers and the Computation of the Topology of Semi-algebraic Sets. *Journal of Symbolic Computation* 5 (1988), 121–129.

[17] Sicun Gao, Jeremy Avigad, and Edmund M. Clarke. 2012. Delta-Decidability over the Reals. In *Proceedings of the 27th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '12).* IEEE Computer Society Press, 305–314.

[18] David Goldberg. 1991. What Every Computer Scientist Should Know About Floating-point Arithmetic. *ACM Comput. Surv.* 23, 1 (March 1991), 5–48. https://doi.org/10.1145/103162.103163

[19] Google Play. 2020. Google Calculator Reviews. https://play.google.com/store/apps/details?id=com.google.android.calculator&showAllReviews=true

[20] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, and Daniel Smith. 2008. *The Java® Language Specification: Java SE 11 Edition.* Oracle.

[21] Paul Gowland and David Lester. 2000. A Survey of Exact Arithmetic Implementations. In *Proceedings of the Workshop on Computability and Complexity in Analysis, Springer LNCS 2064.* 30–47.

[22] John L. Gustafson and Isaac T. Yonemoto. 2017. Beating floating point at its own game: Posit arithmetic. *Supercomputing Frontiers and Innovations* 4, 2 (2017), 71–86.

[23] M. Haselman, M. Beauchamp, A. Wood, S. Hauck, K. Underwood, and K.S. Hemmert. 2005. A comparison of floating point and logarithmic number systems for FPGAs. In *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05).* 181–190.

[24] IEEE CS. 2008. 754-2008 - IEEE Standard for Floating-Point Arithmetic.

[25] M. Laczkovich. 2003. The removal of π from some undecidable problems involving elementary functions. *Proc. Amer. Math. Soc.* 131, 7 (2003), 2235–2240.

[26] Vernon Lee and Hans-J. Boehm. 1990. Optimizing Programs over the Constructive Reals. In *Proceedings of the SIGPLAN 1990 Conference on Programming Language Design and Implementation.* 102–111.

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

[28] David Monniaux. 2008. The Pitfalls of Verifying Floating-point Computations. *ACM Trans. Program. Lang. Syst.* 30, 3, Article 12 (May 2008), 41 pages. https://doi.org/10.1145/1353445.1353446

[29] Norbert Th. Mueller. 2000. The iRRAM: Exact Real Arithmetic in C++. In *Proceedings of the Workshop on Computability and Complexity in Analysis, Springer LNCS 2064.* 222–252.

[30] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically Improving Accuracy for Floating Point Expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15).* ACM, New York, NY, USA, 1–11. https://doi.org/10.1145/2737924.2737959

[31] Dan Richardson and John Fitch. 1994. The Idenity Problem for Elementary Functions and Constants. In *ISSAC '94 Proceedings of the international symposium on Symbolic and algebraic computation.* 285–290.

[32] Alex Sanchez-Stern, Pavel Panchekha, Sorin Lerner, and Zachary Tatlock. 2018. Finding Root Causes of Floating Point Error. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018).* ACM, New York, NY, USA, 256–269. https://doi.org/10.1145/3192366.3192411

[33] Stephen Shankland. 2008. Google's calculator muffs some math problems. https://www.cnet.com/news/googles-calculator-muffs-some-math-problems

[34] Benjamin Sherman, Jesse Michael, and Michael Carbin. 2019. Sound and Robust Solid Modeling via Exact Real Arithmetic and Continuity. *Proceedings, ACM Program. Lang., ICFP* 3, Article 99 (8 2019), 29 pages. https://doi.org/10.1145/3341703

[35] Benjamin Sherman, Luke Sciarappa, Adam Chlipala, and Michael Carbin. 2018. Computable Decision Making on the Reals and Other Spaces: Via Partiality and Nondeterminism. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) *(LICS '18).* ACM, New York, NY, USA, 859–868. https://doi.org/10.1145/3209108.3209193

[36] A. K. Simpson. 1998. Lazy functional algorithms for Exact Real Functionals. In *Mathematical Foundations of Computer Science, Springer LNCS 1450.* 456–464.

[37] Jean Vuillemin. 1990. Exact Real Arithmetic with Continued Fractions. *IEEE Trans. Comput.* 39, 8 (1990), 1087–1105.

[38] K. Weirauch and C. Kreitz. 1987. Representations of the Real Numbers and of the open subsets of the real numbers. *Annals of Pure and Applied Logic* 35, 3 (1987), 247–260.

[39] Wikipedia. 2019. Lindemann-Weierstrass theorem. https://en.wikipedia.org/wiki/Lindemann-Weierstrass_theorem