Exact real arithmetic for Tcl

Kevin B. Kenny

1 Introduction

Once in a great while, the programmer has need for floating point arithmetic in greater precision than the hardware provides. This capability is usually not performance critical, since most performance critical code deals with measured quantities in the physical world, where it is hard to imagine a measurement precision that exceeds the sixteen or so decimal digits provided by standard IEEE-754 double precision arithmetic. Instead, the high-precision arithmetic is used to guard against catastrophic loss of significance, where a difference is computed between floating point numbers of nearly the same magnitude, or against accumulation of roundoff errors. In the course of developing numeric software, the analyst will usually rectify these problems as a matter of course, but the need arises to verify that nothing has been missed, and to compute intermediate constants (such as coefficients of power series) so that they are guaranteed accurate.

Unfortunately, it is often extremely difficult to determine in advance what precision will be required for these calculations. Using interval arithmetic can often give an indication of what precision results are known to, and then the analyst can determine the intermediate precision through trial and error, but this task is laborious. Moreover, the required precision may depend on the input data.

This paper presents a library that avoids these issues by representing numbers as algorithms, rather than as streams of digits. Each number's representation is a TclOO object that has methods to compute the number's value by successive approximation, keeping the number bounded by an ever-decreasing interval at each step. The library provides algorithms for arithmetic, for real powers and roots, and for the elementary functions.

The chief drawback to this approach is that not all numbers are computable! (The set of reals is not countable, while the set of computable numbers is.) Moreover, equality of two numbers in this scheme is undecidable. Deciding when two numbers, both specified by algorithms, are equal is equivalent to the Halting Problem. Nevertheless, despite these limitations, arithmetic on the computable numbers is useful for the sort of applications (software testing and high-precision development of constants) envisioned here.

2 Motivation

Floating-point arithmetic has acquired a certain, mostly deserved reputation for being fraught with peril. Even fairly simple calculations sometimes fail quite badly if its vagaries are not considered. For instance, consider the high-school solution to the quadratic formula $ax^2+bx+c=0$:

$$x=\frac{-b\pm\sqrt{b^2-4ac}}{2a} .$$

Let's attempt to do a naïve implementation of the quadratic formula in floating point, and examine the result.

```
proc quad1 {a b c} {
    set d [expr {sqrt($b*$b - 4.*$a*$c)}]
    set r0 [expr {(-$b - $d) / (2. * $a)}]
    set r1 [expr {(-$b + $d) / (2. * $a)}]
    return [list $r0 $r1]
}
```

Choose the coefficients a, b, c to have the two roots $x=-200, x=7.5\cdot 10^{-15}$, performing the calculations in floating point. So far, all calculations are close to machine accuracy:

$$a=1, b=200.0, c=-1.5\cdot 10^{-12}$$

But what happens when we run the high-school quadratic formula on this set of parameters? The large root $x=-200$ comes out to machine precision, but the small root is precise to zero significant figures. Even the leading digit is wrong.

```
% puts [quad1 1. 200. -1.5e-12]
-200.0 1.4210854715202004e-14
```

What's gone wrong here? If we look at the intermediate result $d=\sqrt{b^2-4ac}$, we'll see that it prints out as

```
200.00000000000003
```

This value is so close to 200.0 that subtracting the two gives a value that is off by a large factor. Instead of the correct value of $1.5\cdot 10^{-14}$, what prints is:

```
2.8421709430404007e-14
```

nearly a factor of two too high. Subtracting two nearly equal quantities has caused a catastrophic loss of significance.

An experienced numerical analyst will, possibly with some effort, be able to rework the procedure to avoid ever losing significance in the numerator of the result, by observing that

$$\frac{-b+\sqrt{(b^2-4ac)}}{2a} = \frac{2c}{-b-\sqrt{b^2-4ac}},$$

and rewriting the procedure:

```
proc quad2 {a b c} {
    set d [expr {sqrt($b*$b - 4.*$a*$c)}]
    if {$b < 0} {
     set s [expr {-$b + $d}]
    } else {
     set s [expr {-$b - $d}]
    }
    set r0 [expr {$s / (2. * $a)}]
    set r1 [expr {(2. * $c) / $s}]
    return [list $r0 $r1]
}
```

The new procedure is immune to this particular pathology:

```
% puts [quad2 1. 200. -1.5e-12]
-200.0 7.5e-15
```

But even the experienced analyst will have a greater challenge when asked how to address the loss of significance when $b^2 \approx 4ac$. Fortunately, this case causes “only” the loss of about half the significant digits of the result. For instance, consider:

```
% puts [quad1 94906265.625 \
            -189812534. 94906268.375]
1.0000000144879793 1.0000000144879793
% puts [quad2 94906265.625 \
             -189812534. 94906268.375]
1.0000000144879793 1.000000014487979
```

(The correct answers, to IEEE precision, are 1.0 and 1.0000000289759583.) [Kahan 2004]

At this point, the analyst will mutter wisely about things like “internal quad precision for intermediate results,” and “Kahan's summation algorithm,” and offer to start a project to assess sensitivity of the results to the initial parameters, and you realize that you are going to be out a lot of money letting him pursue this – and all for what started out a four-line procedure.

Of course, most floating-point calculations never trip over troubles quite this bad – these results are somewhat contrived. Nevertheless, we sometimes want to know that we have the exact answer, without worrying about intermediate precision. We may be doing software testing, and attempting to verify that our floating calculations have not gone far astray in test cases. Or we may be developing a numerical library, in which we need some constant like $\ln(\sqrt{2\pi})$ and want it to full machine precision so that we can promise results to some level of accuracy. In these cases, performance is generally not an issue. We don't care that the machine might take a very long time to get the answer. We merely care that the answer is correct.

3 Possible approaches

3.1 Extended precision

The first approach to doing calculations that suggests itself is simply to extend the precision of results beyond the machine's native precision. This is what was historically chosen by the Unix calculator `bc` [Cherry & Morris 1996], and by the `mpexpr` extension to Tcl [Poindexter] . We can gain a good deal of confidence by repeating our calculations at several levels of precision, and seeing where the results appear to stabilize. Unfortunately, while this approach certainly improves on the naïve one, it provides no guarantees. We can never be sure that some catastrophe is not lurking, to be revealed by the next increase in precision that we will try. Full certainty can be provided only by the same sort of laborious numerical analysis that we are trying to avoid.

3.2 Streams of digits

If an extended, but fixed, precision cannot achieve what we are after, the next possibility that suggests itself is to have an indeterminate precision. We could represent a number by a procedure, possibly a coroutine, that produces a stream of decimal or binary digits. Expressions could be represented by procedures that take their arguments, and combine the streams using grade-school arithmetic to produce sums, differences, products, quotients, and so on.

We do not need to go very far to see where this approach fails us. Let's assume that we use streams of decimal digits, and attempt to compute the simple expression $3\cdot(1/3)$. The right-hand factor is, of course, 0.333…, and the multplying by 3 yields 0.999…. When the product procedure tries to decide what digit to emit to the left of the decimal point, it goes into an infinite loop. It has no knowledge that the stream representing 1/3 will be an endless stream of 3's: perhaps it will be 0.3333….4. For this reason, it must simply loop, consuming more and more 3's, trying to determine whether the 0.999… will ever carry and yield a number greater than 1. In fact, this problem is fundamental to a computational structure for the real numbers, and is the problem that Alan Turing was attempting to address in his most famous paper, the one in which the Turing Machine was first introduced. (In order to attack it, he had to show that the problem is fundamentally undecidable. If the stream of 3's is produced by an arbitrary program, the problem of deciding whether something other than a 3 will ever be emitted is equivalent to deciding whether the program will ever halt.

3.3 Continued fractions

The next method that suggested itself was to represent real numbers as continued fractions, representing a real number x as a stream of integers $a, b, c, \ldots\ (b, c, \ldots \geq 1)$ such that

$$x = a + \cfrac{1}{b + \cfrac{1}{c + \cfrac{1}{\ddots}}}.$$

This representation solves the problem of $0.333\ldots \cdot 3$, (and indeed any other problem in rational arithmetic), because the continued fraction representations of all rational numbers terminate. R. William Gosper expounded on continued fractions as an approach to perfect arthmetic as part of the eclectic but seminal paper HAKMEM [Beeler,Gosper & Schroppel 1972] , and developed the theory further in an unpublished (but widely circulated) paper that again urged developers to consider the approach. He presented a simple algorithm for the four arithmetic operations and for square roots.

Jean Vuillemin expanded on Gosper's work to show how fundamental constants such as e and π , general powers and roots, and the elementary functions could all be computed with continued fractions [Vuillemin 1988] .

The author of the current paper went as far as to implement a fair fraction of a Tcl library for continued-fraction calculations, before stumbling over the same problem that occurs with digit streams: arithmetic over continued fractions is not decidable. The problem appears as soon as one tries to multiply the square root of 2 by itself.

$$\sqrt{2} = 1 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{\ddots}}}$$

A program trying to output the integer part of $(\sqrt{2})^2$ will consume successive approximations to the continued fraction:

“ $\sqrt{2} = 1 + 1/z$ is between 1 and 2, so its square is between 1 and 4. I don't know the integer part, so I need a better approximation.”

“ $\sqrt{2} = 1 + 1/(2 + 1/z)$ is between 4/3 and 3/2, so its square is between 16/9=1.777… and 9/4=2.25. I don't know the integer part, so I need a better approximation.”

“ $\sqrt{2} = 1 + 1/(2 + 1/(2 + 1/z))$ is between 7/5 and 10/7, so its square is between 49/25=1.96 and 100/49 = 2.0408…. I don't know the integer part, so I need a better approximation.”

We have once again hit the Halting Problem, just as before. As soon as the algorithm that is producing $a, b, c, \ldots$ outputs an endless string of 2's, the integer part of the product remains undetermined. As soon as it outputs anything else, the integer part is known.

Gosper was clearly aware of the problem even in 1972, and suggested solving it by allowing non-positive values for $b, c, d, \ldots$. This change essentially would allow a stream to retract an earlier result and replace it with a different one. He never developed an effective computational procedure based on this scheme. Eventually, David Lester did reduce the theory to an effective procedure, but his algorithms were fairly complex, requiring that up to seven terms of a continued fraction be consumed before one result term could be produced, and requiring fairly extensive auxiliary tables [Lester 2001] .

3.4 Möbius transformations

Before attempting to impement Lester's algorithms in a Tcl continued fraction library, the author stumbled upon another representation for the real numbers: sequences of Möbius transformations (also called linear fractional transformations) [Potts 1998]

$$y = \frac{ax + b}{cx + d}.$$

These transformations have properties that make them useful computationally.

First, if $x \geq 0$, and $b/d \leq a/c$, then y lies in the closed interval $[b/d\,..\,a/c]$. If instead $b/d \geq a/c$, then y lies outside the open interval $(a/c\,..\,b/d)$. Every Möbius transformation therefore represents an interval of rational numbers. If $a=c$ and $b=d$, then the matrix represents a single rational number, and the value of x is immaterial.

Second, Möbius transformations are composable. If $x=(ay+b)/(cy+d)$ and $y=(ez+f)/(gz+h)$,

then $x=\dfrac{(ae+bg)z+(af+bh)}{(ce+dg)z+(cf+dh)}$.

This formula is itself a Möbius transformation. The mathematical reader will recognize that it is simply a matrix product. It will therefore be convenient to represent Möbius transformations as matrices, and write:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{bmatrix}.$$

It is also useful to consider matrices that are scalar multiples of one another as being equivalent, since

$$\frac{pax+pb}{pcx+pd} = \frac{ax+b}{cx+d}$$

for any $p \neq 0$.

Third, continued fractions map gracefully into Möbius transformations. The number represented by

$$x = a + \cfrac{1}{b + \cfrac{1}{c + \ddots}}$$

is the same number as that represented by the Möbius transformation product

$$\begin{bmatrix} a & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} b & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} c & 1 \\ 1 & 0 \end{bmatrix} \cdots.$$

This correspondence immediately implies that all of Gosper's and Vuillemin's algorithms for continued fractions can be brought forward into the algebra of Möbius transformations.

Finally, there is a special form of Möbius transformation that offers an effective calculation procedure. It works around the problem of decidability by using overlapping intervals to represent numbers. Any number near the edge of an interval has an alternative representation that starts with another interval, so that it only finite information is ever required to emit the next transformation. A transformation, once emitted, is never retracted, and the interval represented by the product of the transformations decreases monotonically in width. In this way, every step makes progress.

4 Representing real numbers

4.1 Fundamental entities

The implementation of exact real arithmetic begins with three fundamental objects: 2-vectors, 2×2 matrices, and $2 \times 2 \times 2$ third-order tensors, where all components are integers. A vector is simply a list of integers, a matrix is a list of its two columns, and a tensor is a list of its two matrices.

A vector $\{a \;\; b\}$ represents the rational number a/b. We assume that a and b are not both zero. ($a \neq 0, b=0$ is permissible, and represents an infinite quantity, of unknown sign.) By convention, we represent fractions in lowest terms, and cast out the greatest common divisor when necessary.)

A matrix

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

represents the Möbius transformation $x=(ay+b)/(cy+d)$. It also, as we have seen, represents the interval $[b/d\,..\,a/c]$. Once again, multiplication by a scalar does not change what a matrix represents, and we cast out any prime factor common to all four elements. The two elements in a column cannot both be zero, and the matrix cannot be singular. (Equivalently, the second column cannot be a scalar multiple of the first, or the endpoints of an interval cannot both be the same point.)

A tensor,

$$\left[\begin{array}{cc|cc} a & b & c & d \\ e & f & g & h \end{array}\right]$$

represents a bilinear fractional transformation of two variables:

$$z = \frac{axy+bx+cy+d}{exy+fx+gy+h}.$$

Like vectors and matrices, tensors do not change what they represent when multiplied by a scalar, and are conventionally represented in lowest terms.

There are tensors that represent all four of the arithmetic operations:

$$T_+ = \left[\begin{array}{cc|cc} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right],$$
$$T_- = \left[\begin{array}{cc|cc} 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right],$$
$$T_\times = \left[\begin{array}{cc|cc} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right],$$
$$T_\div = \left[\begin{array}{cc|cc} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{array}\right].$$

If $\boldsymbol{v}$ is a vector, $\boldsymbol{M}$ and $\boldsymbol{N}$ are matrices, and Ψ is a tensor, then the products $M \cdot v$ and $M \cdot N$ are defined in the conventional way. The product of a matrix and a tensor is

$$\begin{bmatrix} p & q \\ r & s \end{bmatrix} \cdot \left[\begin{array}{cc|cc} a & b & c & d \\ e & f & g & h \end{array}\right]$$
$$= \left[\begin{array}{cc|cc} pa+qe & pb+qf & pc+qg & pd+qh \\ ra+se & rb+sf & rc+sg & rd+sh \end{array}\right],$$

that is, it is the result of matrix multiplying the matrix independently with the two matrices that make up the tensor.

Tensors may be multiplied on the right by vectors in two different ways, $\Psi \cdot_L \boldsymbol{v}$ and $\Psi \cdot_R \boldsymbol{V}$. These correspond to replacing x or y, respectively, with p/q in the formula

$$z = \frac{axy+bx+cy+d}{exy+fx+gy+h}.$$

$$\left[\begin{array}{cc|cc} a & b & c & d \\ e & f & g & h \end{array}\right] \cdot_L \begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} ap+cq & bp+dq \\ ep+gq & fp+hq \end{bmatrix}$$
$$\left[\begin{array}{cc|cc} a & b & c & d \\ e & f & g & h \end{array}\right] \cdot_R \begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} ap+bq & cp+dq \\ ep+fq & gp+hq \end{bmatrix}.$$

Left and right multiplication of a tensor by a matrix is defined so as to satisfy the associative laws

$$(\Psi \cdot_L \boldsymbol{M}) \cdot_L \boldsymbol{v} = \Psi \cdot_L (\boldsymbol{M} \cdot \boldsymbol{v}), \text{and}$$
$$(\Psi \cdot_R \boldsymbol{M}) \cdot_R \boldsymbol{v} = \Psi \cdot_R (\boldsymbol{M} \cdot \boldsymbol{v}).$$

Matrix transposition is defined in the usual way. Tensor transposition is defined as swapping columns.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{\mathrm{T}} = \begin{bmatrix} a & c \\ b & d \end{bmatrix}.$$
$$\left[\begin{array}{cc|cc} a & b & c & d \\ e & f & g & h \end{array}\right]^{\mathrm{T}} = \left[\begin{array}{cc|cc} a & c & b & d \\ e & g & f & h \end{array}\right].$$

Matrix inversion is somewhat unusual. Since a scalar multiple $a\boldsymbol{M}$ of a matrix $\boldsymbol{M}$ has the same meaning as the matrix itself, there is no step of dividing by the determinant, which in turn means that the determinant may be zero, and inversion is still safe. We simply have the pseudo-inverse

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} \simeq \begin{bmatrix} d & -c \\ -b & a \end{bmatrix}.$$

All of these operations are simply defined as functions that operate on Tcl lists.

4.2 Expression trees

The next layer of the software wraps vectors, tensors and matrices up in expression trees that define how mathematical expresssions are to be evaluated. The fundamental base class, `Expression`, has three subclasses, `V`, `M`, and `T`, which encapsulate vectors, matrices, and tensors respectively.

The `V` class encapsulates a vector of two integers, *n* and *d*, and represents the rational number n/d.

The `M` class encapsulates a matrix `M`, and represents the result of applying that matrix to the result of another expression. The other expression may be known at compile time (the `Mstrict` subclass handles this case), or may be constructed lazily on demand. If it is constructed lazily, the matrix is cached once constructed, so that lazy evaluation happens only once.

The `T` class encapsulates a tensor Ψ, and represents the result of applying Ψ to two other expressions: $(\Psi \cdot_L e_1) \cdot_R e_2$. Once again, the expressions may be known at compile time (`Tstrict`), or may be constructed lazily on demand. Lazily-constructed expressions are cached.

Since the lazy evaluation, and the actual numeric calculation, results in a great number of ephemeral objects being passed around, the Expression base class is reference counted. The caller claims a reference by calling `[$a ref]` on some expression `$a`, and releases the reference with `[$a unref]`. Objects are constructed with a zero reference count, and deleted when an unref operation results in a zero reference count again.

4.3 Expressions

The command at the center of the exact arithmetic system is `math::exact::exactexpr`. This command is similar to the Tcl `expr` command, and accepts a single argument that is the expression to be evaluated. The result, instead of being the value, is an `Expression` object that represents the value. It is returned with a reference count of 1.

The `exactexpr` command is built from a conventional expression grammar using an Aycock-Early-Horspool parser generator (the `grammar::aycock` module in Tcllib). The expressions may include:

- integers
- the fundamental constants `pi` and `e`
- references to Tcl variables, which are expected to contain `Expression` object instances
- mathematical functions such as `sqrt` and `sin`
- parentheses
- the four arithmetic operations `+`, `-`, `*`, `/`
- the exponentiation operation `**`

Most of these are translated in a straightforward fashion. (There are compile-time optimizations that apply, which are not discussed here, mostly having to do with constant folding.) An integer *x* is represented by the vector $\begin{bmatrix} x \\ 1 \end{bmatrix}$.

Unary negation is represented by the matrix $\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$ applied to its argument. The four arithmetic operators, as we have seen above, correspond to tensors, which are applied to their left and right operands.

4.4 Infinite expression trees

Exponentiation and the mathematical functions are rather more complicated, and generally involve lazy evaluation. The mathematical algorithms that construct the lazy objects are somewhat complex, and the interested reader is referred to the source code (which, in turn, references the relevant papers) for details. As a simple example, though, let's work through calculating a square root.

If a program requests `[exactexpr {sqrt(2)}]`, that is translated into an instance of the `SqrtWorker` class (which inherits from `T`). This class's tensor is always the constant:

$$\Psi_{\sqrt{}} = \left[\begin{array}{cc|cc} 1 & 2 & 1 & 0 \\ 0 & 1 & 2 & 1 \end{array}\right],$$

representing the function:

$$\frac{xy+2x+y}{x+2y+1}.$$

The left operand, *x*, is the quantity whose square root is to be found, and the right operand, *y*, is a copy of the `Sqrtworker` object. The expression (which is lazily constructed) is an infinite sequence

$$\Psi_{\sqrt{}} \cdot_L x \cdot_R \Psi_{\sqrt{}} \cdot_L x \cdot_R \Psi_{\sqrt{}} \cdot_L x \cdot_R \cdots$$

Let's fold $x=2$ into the tensor, since we are computing `sqrt(2)`. The function that will be iterated becomes

$$\frac{3y+4}{2y+3}, \text{ or the matrix} \begin{bmatrix} 3 & 4 \\ 2 & 3 \end{bmatrix}.$$

The expression becomes the infinite product:

$$\begin{bmatrix} 3 & 4 \\ 2 & 3 \end{bmatrix} \cdot \begin{bmatrix} 3 & 4 \\ 2 & 3 \end{bmatrix} \cdot \begin{bmatrix} 3 & 4 \\ 2 & 3 \end{bmatrix} \cdot \cdots.$$

Let's verify for ourselves that this is doing what we want. After the first iteration, the product is the matrix itself, asserting that $4/3 \le \sqrt{2} \le 3/2$. (This is indeed the case.)

If we need more information than this, we lazily request another layer of the expression tree, and compose the two matrices, giving us the matrix

$$\begin{bmatrix} 17 & 24 \\ 12 & 17 \end{bmatrix}.$$

This asserts that $24/17 \le \sqrt{2} \le 17/12$. Once again, this is indeed the case, and it has refined the estimate so that we know that the leading decimal digits are 1.41. If we need more digits than this, we turn the crank one more time, obtaining the matrix

$$\begin{bmatrix} 99 & 140 \\ 70 & 99 \end{bmatrix}.$$

and another decimal digit, making the leading digits 1.414. Further iterations refine it to 1.41421, 1.414213, 1.41421356, adding a digit or two of precision with each level of expression evaluated.

5 Evaluating real numbers

With the representation of real numbers as (possibly infinite) trees of tensor and matrix products, with vectors at the leaves, we can now begin to consider how to go about evaluating real numbers and producing results in a usable form, such as decimal fractions or rational numbers together with statements of precision. The library does this by converting numbers to streams of simple matrices, with only a few constant matrices allowed in the streams. These are similar to streams of digits in conventional representation, except that digits can take on negative values as well as positive ones. The result is a redundant representation, in which any number can be represented in multiple ways. The redundancy means that at any given digit position, a decision can be made without needing an infinite amount of precision. Problems such as the infinite loop computing $\sqrt{2}\times\sqrt{2}$ no longer appear.

5.1 Sign-and-magnitude representation

The assertion that $a/c \geq (ax+b)/(cx+d) \geq b/d$, which we have used in asserting that matrices represent intervals, fails if x is negative. The library handles this by exporting up a 'sign matrix' as the first element in the stream. The sign matrix is a redundant division of the number line into quadrants:

$$\begin{array}{llll} +: & x\geq 0 & 0: & |x|\leq 1 \\ -: & x\leq 0 & \infty: & |x|\geq 1. \end{array}$$

For each of these quadrants, there is an associated matrix:

$$\boldsymbol{S}_+=\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} \quad \boldsymbol{S}_0=\begin{bmatrix}1 & -1\\ 1 & 1\end{bmatrix}$$
$$\boldsymbol{S}_-=\begin{bmatrix}0 & 1\\ -1 & 0\end{bmatrix} \quad \boldsymbol{S}_\infty=\begin{bmatrix}1 & 1\\ -1 & 1\end{bmatrix}$$

so that if $\boldsymbol{x}=\boldsymbol{S}\boldsymbol{y}$ is in a given quadrant, then y is positive (when vectors are given their interpretation as rational numbers). Each of the subclasses of `Expression` supports a method, `getSignAnd-Magnitude`, that returns a pair consisting of an appropriate sign matrix $\boldsymbol{S}$ and an expression $\boldsymbol{y}$ such that $\boldsymbol{x}=\boldsymbol{S}\boldsymbol{y}$ and $\boldsymbol{y}$ is nonnegative. The method absorbs information from subexpressions as necessary to determine which sign matrix to use.

5.2 Digit streams

After the sign and magnitude extraction, the nonnegative numbers that remain are broken down into products of 'digit matrices'. There are three of these, which divide the non-negative half of the number line into overlapping pieces:

$$\begin{array}{rl} -1: & 0\leq x\leq 1 \\ 0: & 1/3\leq x\leq 3 \\ 1: & 1\leq x. \end{array}$$

The corresponding matrices are:

$$D_{-1} = \begin{bmatrix}1 & 0\\ 1 & 2\end{bmatrix} \quad D_0 = \begin{bmatrix}3 & 1\\ 1 & 3\end{bmatrix} \quad D_1 = \begin{bmatrix}2 & 1\\ 0 & 1\end{bmatrix}$$

Numbers usually begin with a series of D_{-1} or D_1 matrices that fix the order of magnitude. A number that begins with D_1 is at least 1; one that begins with $D_1 D_1$ is at least 3; and in general if a number begins with k D_1 matrices, its value is at least 2^k-1. Similarly, a number beginning with k D_{-1} matrices has a value that is at most $1/(2^k-1)$.

After the leading string of repeated D_{-1} or D_1 matrices, the rest of the number can be thought of as a significand. Each digit cuts the size of the interval in which the number can appear roughly in half. The number of digit matrices needed to represent a quantity x to some desired degree of precision can therefore be estimated as $|\log_2 x|+b$, where b is the number of bits of significance that are desired.

Just as each of the subclasses of `Expression` supports `getSignAndMagnitude`, each subclass supports a method, `getLeadingDigitAndRest`, that extracts the leading digit matrix $\boldsymbol{D}$, returning a pair consisting of $\boldsymbol{D}$ and an expression $\boldsymbol{x}=\boldsymbol{D}\boldsymbol{y}$, with $\boldsymbol{y}$ still nonnegative. It absorbs information from subexpressions as necessary to compute the digit.

5.3 Digit exchange

Both of the preceding two sections have spoken of 'absorbing information from subexpressions', but have glossed over what absorption of the information means. In both cases, what 'absorbing information' means is that a digit matrix will be extracted from a subexpression and composed into the current matrix or tensor. (This, of course, may in turn cause the subexpression to need to absorb information from its subexpressions, and so on.) Denoting the value of the current expression by $\boldsymbol{x}$, the process of exchanging digits with subexpressions continues until for some digit matrix $\boldsymbol{D}$, we can prove that $\boldsymbol{D}^{-1}\boldsymbol{x}$ represents a non-negative number for any value of the subexpressions. At that point, it is safe to return the digit $\boldsymbol{D}$ and the result of left-multiplying $\boldsymbol{D}^{-1}$ with the current matrix or tensor. Since the size of the interval represented by a subexpression roughly halves with each digit emitted, and since the ranges of intervals represented by the digit matrices overlap, this is an effective procedure: it never requires an unbounded amount of work to extract the next digit matrix.

5.4 Formatting for printing

The `asPrint` method is responsible for formatting an exact real number for display. This method accepts the number of digit matrices to process. It composes those matrices, resulting in a representation of the number as a rational interval. It then formats the number in floating point “E format” (significand and power of 10), terminating the significand at the point where the interval becomes wider than 1 unit in the least significant place. The true value of the real number as printed is always within ± 1 unit in the least significant digit.

6 Examples

Let's run through a few examples to see how the library works in a few practical cases. First, let's try to solve the quadratic formula over exact real numbers. We won't worry about the roundoff errors from the first section, and simply let the machinery request enough precision from the intermediate results to give the requested accuracy of the final answer.

We'll write a procedure analogous to `quad1` that operates on exact reals:

```
proc exactquad {a b c} {
    set d [[exactexpr {
                sqrt($b*$b – 4*$a*$c)
            }] ref]
    set r0 [[exactexpr {
                 (-$b - $d) / (2 * $a)
             }] ref]
    set r1 [[exactexpr {
                 (-$b + $d) / (2 * $a)
             }] ref]
    $d unref
    return [list $r0 $r1]
}
```

The structure is identical to `quad1`, except for reference count management. Calling it also requires only changes to notation and to reference count management.

```
set a [[exactexpr 1] ref]
set b [[exactexpr 200] ref]
set c [[exactexpr {
           (-3/2) * 10**-12
       }] ref]
lassign [exactquad $a $b $c] r0 r1
$a unref; $b unref; $c unref
puts [list [$r0 asFloat 70] \
           [$r1 asFloat 110]]
$r0 unref; $r1 unref
```

As we might hope, the results are to full precision, with no worrying about the near-cancellation of significance in subtraction:

```
-2.000000000000000075e2
7.499999999999999719e-15
```

Similarly, when we attempt the arguments that gave near-cancellation under the square root sign, there is no problem. The near-multiple root is recovered to IEEE-754 double precision perfectly well:

```
set a [[exactexpr 94906265625/1000] ref]
set b [[exactexpr -189812534] ref]
set c [[exactexpr 94906268375/1000] ref]
# … other code as before …
1.0000000000000000e0 1.0000000289759583e0
```

And we can do other high-precision calculations just as nicely. For instance, we can easily disprove the old canard that $e^{\pi\sqrt{163}}$ is an integer:

```
% set r [[exactexpr {exp(pi()*sqrt(163))}]
ref]
% puts [$r asPrint 162]
2.62537412640768743999999999999925e17
% $r unref
```

Or we can solve a problem due to W.Kahan: [Kahan 2005] Let $x_0 = 4,$

$$x_1 = 4.25,\ \ x_k + 1 = 108 - \frac{815 - 1500/x_{k-1}}{x_{k-2}}.$$

What is x_{100}? When calculated in floating point, to any reasonable precision, straightforward coding of this question gives the answer 100.0. But the correct answer is very close to 5.

The results of the first 25 iterations are shown in Table 1. The first eight or nine iterations are fairly close between standard IEEE-754 double precision and exact arithmetic. By the eleventh, however, the floating point arithmetic is clearly trending in the wrong direction, and in the next couple of iterations, it falls apart altogether.

Table 1. *Kahan's problem: IEEE-754 vs. exact arithmetic*

Iteration	IEEE-754	Exact
1	4.47059	4.47059
2	4.64474	4.64474
3	4.77054	4.77054
4	4.85570	4.85570
5	4.91085	4.91085
6	4.94554	4.94554
7	4.96696	4.96696
8	4.98004	4.98004
9	4.98791	4.98798
10	4.99136	4.99277
11	4.96746	4.99566
12	4.42969	4.99739
13	-7.81724	4.99843
14	168.93917	4.99906
15	102.03996	4.99944
16	100.09995	4.99966
17	100.00499	4.99980
18	100.00025	4.99988
19	100.00001	4.99993
20	100.00000	4.99996
21	100.00000	4.99997
22	100.00000	4.99998
23	100.00000	4.99999
24	100.00000	4.99999
25	100.00000	5.00000

7 Limitations

Chief among the obvious limitations of the `math::exact` package is performance: it is atrociously slow and consumes a great deal of memory. Performance is rather beside the point: it is built to give exact results always, and not highly optimized. There are other limitations that are more subtle but likely to be more important in practice.

7.1 Floating point

The reader may have noticed that the expression grammar does not support floating point notation. The problem with floating point notation is that, in a world of exact arithmetic, it is ambiguous. Consider the simple constant 0.3333333333333333. What does this constant mean? There are three rational numbers that make roughly equal sense, but are not equal.

- The rational number $\frac{1}{3}$. This constant is the rational number with the smallest denominator that will yield the given constant in IEEE-754 floating point. (To a human eye, this is probably the most reasonable answer.)
- The rational number $\frac{3333333333333333}{10000000000000000}$. This constant is the obvious interpretation of the floating point number as printed.
- The rational number $\frac{6004799503160661}{18014398509481984}$. This is the exact value of the IEEE-754 floating point constant that prints out as 0.3333333333333333.

It seems better to let the programmer write one of these explicitly, rather than trying to divine which interpretation is intended in any given case.

7.2 Reference counting

Using reference counts at script level is always unweildy, and invites both premature object destruction and memory leaks. Nevertheless, exact reals are heavyweight objects that we do not want to have to copy when we copy one variable to another, pass a value to a procedure, or have a procedure return a value. Passing objects by reference (i.e., by command name) seems like an obvious approach.

A possible workaround is to accept “fragile references.” The reference to the object that represents a number would be carried as the internal representation of a Tcl_Obj. Copying a variable holding one of these objects would result in incrementing the reference count in the Tcl_Obj. When the Tcl_Obj is deleted, so is the underlying exact value.

As the name suggests, these references are fragile. If the object name is interpolated into a string, used as a singleton list, used as a hash key, or otherwise used in a way that causes its internal representation to shimmer, the reference to the underlying exact number will be lost and it will be destroyed prematurely. This behaviour leads to subtle bugs that are often difficult to locate.

Nevertheless, users of Tcl interfaces to managed code systems such as Java, COM and .NET face the fragile reference problem whenever they deal with a managed object. It appears that they contrive to lead useful lives in spite of the fragility. For this reason, a future version of `math::exact` may offer fragile references with automatic object reclamation as an option.

7.3 Comparison

Similarly, the `math::exact` package does not offer comparison operators `==`, `!=`, `<`, `<=`, `>`, and `>=` in its expressions. The reason is that they are not decidable! In order to decide whether 0.999… < 1, the library might have to do an unbounded amount of work trying ever more precise values, without ever arriving at an answer.

The solution to this problem is fairly well understood: offer fuzzy comparison operators. For any rational number $\epsilon > 0$, we can define a comparison operator

$$a \ll_\epsilon b := \begin{cases} 1, & \text{if } a < b - \epsilon, \\ \text{either } 0 \text{ or } 1, & \text{if } b - \epsilon \le a \le b + \epsilon \\ 0, & \text{if } a > b + \epsilon. \end{cases}$$

and define similar operators $\gg_\epsilon$, $\approx_{epsilon}$, and $\neq_\epsilon$. These operators require only finite precision, because they are allowed to give any consistent answer for the given tolerance ϵ. The correct value for ϵ will depend on the application, so there must be a way to specify it in the expression. Implementing this functionality awaits only the specification of an appropriate syntax for it.

7.4 Safety

It is possible for user code to drive the `math::exact` package into an infinite loop or a stack overflow. Typically, this occurs when code requests something that is not decidable: For instance, the command:

```
% set r [exactexpr {sqrt(tan(pi()/4)-1)}]
```

overflows the stack. The reason is that the `sqrt` function is discontinuous at zero (it does not exist to the left of the origin), and the code goes into endless recursion trying to decide whether the value of the transcendental `tan` function is less than or greater than 1. Some anomalies like this one appear to be fundamentally unfixable, and the code would be improved if there were enforced limits on stack depth, run time, and size of intermediate results, possibly with optimistic assumptions made for function evaluations close to the edge of the functions' domains. These changes are a fairly sizable project, and will likely need to be attacked piecemeal.

8 Conclusions

The `math::exact` package, while clearly imperfect, offers a proof of concept for how exact arithmetic can be implemented for Tcl. It already has demonstrated utility for performing high-precision calculation of fundamental constants (such as coefficients for series or continued fraction approximations to functions) and for software testing (offering exact results so that code can be tested for numerical instability). The intention is that the package will appear in the next formal release of Tcllib.

References

Beeler, M., Gosper, R.W. and Schroppel, R.. 1972. HAKMEM. `http://dspace.mit.edu/bitstream/handle/1721.1/6086/AIM-239.pdf`.

Cherry, Lorinda and Morris, Robert. 1996. BC – An Arbitrary Precision Desk-Calculator Language. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.52.557&rep=rep1&type=pdf`.

Gosper, R.W.. 1972. Continued Fraction Arithmetic. `http://perl.plover.com/yak/cftalk/INFO/gosper.txt`.

Kahan, W.. 2005. How Futile are Mindless Assessments of Roundoff in Floating-Point Computation?. `https://www.cs.berkeley.edu/~wkahan/Qdrtcs.pdf`.

Kahan, W.. 2004. On the Cost of Floating-Point Computation Without Extra-Precise Arithmetic. `http://www.cs.berkeley.edu/~wkahan/Qdrtcs.pdf`.

Lester, David. 2001. Effective Continued Fractions. In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, 163--. Washington, DC, USA: IEEE Computer Society. `http://apt.cs.manchester.ac.uk/ftp/pub/apt/papers/drl_ieee01.pdf`

Poindexter, Tom. Mpexpr. `http://mpexpr.sourceforge.net/`.

Potts, Peter John. 1998. Exact real arithmetic using Mobius transformations. PhD diss., University of London. `http://peterpotts.com/pdf%20files/phd.pdf`

Turing, A. M.. 1937. On computable numbers, with an application to the Entscheidungsproblem. A correction. *Proceedings of the London Mathematical Society. Second Series* 43: 544--546. `https://www.wolframscience.com/prizes/tm23/images/Turing2.pdf`

Turing, A. M.. 1936. On Computable Numbers, with an Application to the Entscheidungsproblem. P*roceedings of the London Mathematical Society. Second Series* 42: 230--265. `http://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf`

Vuillemin, Jean. 1988. Exact Real Computer Arithmetic with Continued Fractions. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, 14--27. New York, NY, USA: ACM. `https://hal.inria.fr/inria-00075792/document`