Delft University of Technology

Tableless Calculation of Circular Functions on Dyadic Rationals

Kourzanov, Peter

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Tableless Calculation of Circular Functions on Dyadic Rationals

Peter Kourzanov

TU Delft

`kourzanov@acm.org`

I would like to tell a story. A story about a beautiful mathematical relationship that elucidates the computational view on the classic subject of trigonometry. All stories need a language, and for this particular story an *algorithmic* language ought to do well. What makes a language algorithmic? From our perspective as the functional programming community, an algorithmic language provides means to express computation in terms of *functions*, with no implementation-imposed limitations. We develop a new algorithm for the computation of trigonometric functions on dyadic rationals, together with the language used to express it, in Scheme. We provide a mechanically-derived algorithm for the computation of the inverses of our target functions. We address efficiency and accuracy concerns that pertain to the implementation of the proposed algorithm either in hardware or software.

1 Introduction

The use of square roots and circular (trigonometric) functions is pervasive in computer science and engineering. In computational geometry and physics, for example, both square roots and trigonometric functions are used extensively. In many engineering domains, the evaluation of both trigonometric functions and their inverses is required for e.g., pre-computation of the Fast Fourier Transform (FFT) tables, angle calculations and rotations of complex numbers. For digital signal processing in general and in wireless communications systems in particular, efficient and accurate implementation of such functions is fundamental (e.g., for frequency error estimation and correction).

In this paper we argue that square roots and their infinite continuation (hereafter: nested radicals) deserve to be quite a bit more pervasive in computing, just the more so because of the pervasiveness of dyadic rationals. These are the numbers of the form $x = \frac{n}{m}, m = 2^k, n \in \mathbb{Z}, k \in \mathbb{N}$. Both embedded software and hardware are generally designed using a fixed-point representation of (complex) values. In such a representation of a number, a fixed number of bits (W) are allocated for both the *integer part* ($I \geq 0$, which may include the sign bit), leaving the rest of the bits for the *fractional* part ($F = W - I$ bits). This naturally corresponds to dyadics $\frac{n}{2^F}, n \in [0, 2^W - 1]$ (or $[-2^{W-1}, 2^{W-1} - 1]$ for two's complement representation of negative numbers). Such fixed-point types are very frequently used in embedded systems as they possess predictable precision and resource efficiency characteristics. Conversion to a dyadic rational is a simple division by 2^F followed by rational simplification.

And yet, maintaining explicit tables containing fixed-point values for e.g., FFT or a Coordinate Rotation Digital Computer (CORDIC) [19] can be too expensive for resource-constrained implementations. For the software, the efficiency is limited by the memory hierarchy bottleneck while for hardware excessive (de)multiplexing that is associated to large (frequently $> 4Kbit$) tables can outweigh the advantages of avoiding explicit multipliers - the primary reason for tabling as opposed to Taylor/Maclaurin series. There is a trade-off between multiplier-based approach with series and multiplexer-based approach with tabling that a resource-constrained application may not want to make.

Maybe we can get rid of explicit tables of values inside implementations of trigonometric functions at the expense of restricting the domain to dyadic rationals? We set our goal therefore to provide a computational proof of this conjecture, by presenting an algorithm for the evaluation of trigonometric functions and their inverses. We shall be eliding tables and/or Taylor/Maclaurin series, using only the very basic arithmetic operations that can map well to hardware and embedded software architectures. These are: addition/subtraction, multiplication/division by a small power-of-2 (bit-shifts), squaring, square root and reciprocals. Note that all these operations except the last three map trivially to modern digital logic. Squaring may be performed without full multipliers using one of the existing methods [17] and [4], while the square root may be computed using a fixed-bound iteration described below.

This paper is structured as follows. First, in section 2, we will focus on the development of our *reversible* algorithm starting from the basics of trigonometry, as well as the basics of Revised[5] Report on the Algorithmic Language Scheme (R5RS). We use the *Algorithmic* Language Scheme [2] primarily because of its flexible syntax and conceptual simplicity. Scheme, to our knowledge, is the only language besides Algol and Refal (both of which we shall avoid here) specifically targeting "algorithms". We will, however, abstain from the impure subset of Scheme, making the code directly translatable to Haskell, Meta Language (ML) or indeed any other functional language. We shall incidentally use the simple algorithm for the square root to highlight our choice for Scheme as an implementation language.

Gradually, we will introduce a number of extensions to both the algorithm and the language used to implement it. Scheme-related sections are tagged as "tangential" to highlight their supplementary nature. Our final result is evaluated and compared to prior art in section 3. We conclude and cast the light on future work in section 4.

2   Development of the algorithm

In this section we present observations to show that $\cos \frac{x}{2^k}$ can indeed be computed iteratively using nested radicals, provided that x is an even multiple of $\frac{\pi}{4}$ (i.e., a multiple of $\frac{\pi}{2}$). For the rest of the paper, we will be restricting the domain of the argument $\frac{x}{2^k} \in [0, \frac{\pi}{2}]$ as well as factoring it by $\frac{\pi}{2}$, thus keeping only integers $n \in [0, 2^k], n = \frac{2x}{\pi}$ such that the argument is always of the form $\frac{n\pi}{2m}, m = 2^k$. This effectively partitions the range into an exponential number of equally-spaced bins forming a *lattice*, where each bin is represented by a dyadic rational $\frac{n}{m}$. For each bin, our focus is to find an efficient and accurate method to compute $\cos \frac{n\pi}{2m}, m = 2^k, n \in \mathbb{Z}, k \in \mathbb{N}$.

It is not difficult to see why nested radicals appear naturally in evaluation of circular functions (also called trigonometric functions) such as cos & sin on modern hardware. One may remember from the trigonometry lessons that squares of the (results of) these two functions add up to 1. A complex number using polar representation (comprising magnitude and angle) naturally corresponds to the conjoined (results of) trigonometric functions acting on the angle, scaled by the magnitude. I.e., projections of a complex number onto a rectangular, or Cartesian coordinate system follows the rules of trigonometry, and therefore simple geometric observations on a two dimensional plane (such as the Pythagoras's theorem) can also serve as a good illustration of the trigonometric facts that are important in this paper:

1. There is an angle, $\frac{\pi}{4}$ for which both projections are equal: the *Pythagorean identity* implies that $\sin \frac{\pi}{4} = \cos \frac{\pi}{4} = \frac{1}{\sqrt{2}} = \sqrt{\frac{1}{2}}$.

2. For any two angles, the addition of angles can be performed by a multiplication of the corresponding complex numbers having magnitude 1. This can be used to show how to compute trigonometric functions of the argument which is a double (or indeed, any multiple) of an angle, given (results

of) trigonometric functions of that angle. Similarly, the *half-angle* formulæ tell us how to reduce the computation of e.g., $\cos \frac{x}{2}$ to $(-1)^{\lfloor \frac{x+\pi}{2\pi} \rfloor} \sqrt{\frac{1+\cos x}{2}}$.

3. Combining above two observations, we get the following sequence:

$$\cos \frac{x}{2^k} = s_0^k(x)\sqrt{\frac{1+\cos \frac{x}{2^{k-1}}}{2}} = s_0^k(x)\sqrt{\frac{1}{2} + s_1^k(x)\sqrt{\frac{1+\cos \frac{x}{2^{k-2}}}{2^2*2}}} = s_0^k(x)\sqrt{\frac{1}{2^0*2} + s_1^k(x)\sqrt{\frac{1}{2^2*2} + s_2^k(x)\sqrt{\cdots}}}$$

$$= s_0^k(x)\sqrt{\frac{2}{2^{2^1}} + s_1^k(x)\sqrt{\frac{2}{2^{2^2}} + s_2^k(x)\sqrt{\frac{2}{2^{2^3}} + \cdots}}} \text{ where } s_i^k(x) = (-1)^{\lfloor \frac{x}{\pi 2^{k-i}} + \frac{1}{2} \rfloor}.$$

One natural question to ask is whether the above nested, or *continued* radical is actually well-founded. We see that the repeated doubling of the argument with subsequent reversal of the sign $s_i^k(x)$ whenever $\frac{x}{2^k} > \frac{\pi}{2}$ (where $\cos x$ becomes negative) leads to an oscillation of the value of our target function cos around 0, with smaller and smaller radical terms. Restricting precision of the term to match desired accuracy, we will inevitably hit $\cos \frac{\pi}{2} = 0$, terminating the recursion. A number of signs of $s_i^4(n)$ for $n \in [1,15]$ and $i \in [1,3]$ (when defined, depending on n as explained in section 2.2) is given in Figure 1.

So we can indeed compute $\cos \frac{n\pi}{2m}$ using iterated `sqrt` computation. We can compute the sign sequence $s_i^k(n)$ which is uniquely determined by the index of n as the dyadic rational in the lattice induced by 2^{-k}, and we know that the process stops as soon as the next positive radical term can no longer be represented. But how do we compute the `sqrt` in the first place?

2.1 The (square) root of all good

Of all known methods to compute square roots [14] the following suits our needs the best. We start with an estimation of the $error^2 = 2^{2i}$ (the highest power of 4 that is smaller than the argument), and an estimation of the $result = 0$. Then we gradually re-balance the equation $number = (result + error)^2 = result^2 + 2 \times result \times error + error^2$ by adding bits to the $result$ (focusing on the second term) so that the $error$ is minimised. A popular C implementation [6] cites [21] as the source of this method.

```
(define (sqrt-loop n start)
   (let loop ([number n]
              [result 0]
              [error start])
      (cond
         ([zerofx? error] result)
         ([>=fx number (+fx result error)]
          (loop (-fx number (+fx result error))
             (+fx error (bit-rsh result 1))
             (bit-rsh error 2)))
         (else
          (loop number
             (bit-rsh result 1)
             (bit-rsh error 2)))
)))
```

All this can be accomplished *tail-recursively* in Scheme with efficient right bit-shifts (that correspond to repeated halving) and additions on fixnums (which in our dialect of Scheme (Bigloo) are invoked using "fx" suffix). The `loop` binding captures the body of the iteration, which is parameterised by the "variables" involved in the re-balancing. Initial values are given at the first invocation (inside `let`), and the `cond` clauses specify how these should be "updated" in each case, when the `loop` is invoked

(arguments are positional, following the order given by the `let`). Because in each iteration exactly 2 bits are right-shifted out from the *error*, the iteration limit linearly depends on the precision of used fixnums. This value will eventually reach 0, at that time we can return the current approximation of the *result*.

```
(def-syntax *maxbit* 60)
(define (sqrt-start n)
 (do ([bit (bit-lsh 1 (-fx *maxbit* 2)) (bit-rsh bit 2)])
    ([<=fx bit n] bit)
  ))
```

Because we may already know highest power of 4 before the `sqrt` loop is run, we pass a parameter (`start`) to be computed separately beforehand, if needed. E.g., by the `sqrt-start` function above. Here, we assume a maximal precision for the fixnums, specified as a syntactic "constant" `*maxbit*`. We search for the largest power of 4 that is smaller than `n` using Scheme's iterator construct, that macro-expands to a tail-recursion similar to the one we used in `sqrt-loop`. As we have seen above, the calculation of trigonometric functions also involves even powers of 2, so adding this argument might avoid extraneous invocation of `sqrt-start` if the value of (`n`) is bounded by a known integer.

$s_i^k(n)$	$\frac{n}{m}$	$\cos\frac{n\pi}{2m}$	sig	sig'	**tree**
(+,+,+)	$\frac{1}{16}$	$\sqrt{2^{-1}+\sqrt{2^{-3}+\sqrt{2^{-7}+\sqrt{2^{-15}}}}}$	0	16	
(+,+)	$\frac{1}{8}$	$\sqrt{2^{-1}+\sqrt{2^{-3}+\sqrt{2^{-7}}}}$	0	8	
(+,+,-)	$\frac{3}{16}$	$\sqrt{2^{-1}+\sqrt{2^{-3}+\sqrt{2^{-7}-\sqrt{2^{-15}}}}}$	1	17	
(+)	$\frac{1}{4}$	$\sqrt{2^{-1}+\sqrt{2^{-3}}}$	0	4	
(+,-,-)	$\frac{5}{16}$	$\sqrt{2^{-1}+\sqrt{2^{-3}-\sqrt{2^{-7}-\sqrt{2^{-15}}}}}$	3	19	
(+,-)	$\frac{3}{8}$	$\sqrt{2^{-1}+\sqrt{2^{-3}-\sqrt{2^{-7}}}}$	1	9	
(+,-,+)	$\frac{7}{16}$	$\sqrt{2^{-1}+\sqrt{2^{-3}-\sqrt{2^{-7}+\sqrt{2^{-15}}}}}$	2	18	
()	$\frac{1}{2}$	$\sqrt{2^{-1}}$	0	2	
(-,-,+)	$\frac{9}{16}$	$\sqrt{2^{-1}-\sqrt{2^{-3}-\sqrt{2^{-7}+\sqrt{2^{-15}}}}}$	6	22	
(-,-)	$\frac{5}{8}$	$\sqrt{2^{-1}-\sqrt{2^{-3}-\sqrt{2^{-7}}}}$	3	11	
(-,-,-)	$\frac{11}{16}$	$\sqrt{2^{-1}-\sqrt{2^{-3}-\sqrt{2^{-7}-\sqrt{2^{-15}}}}}$	7	23	
(-)	$\frac{3}{4}$	$\sqrt{2^{-1}-\sqrt{2^{-3}}}$	1	5	
(-,+,-)	$\frac{13}{16}$	$\sqrt{2^{-1}-\sqrt{2^{-3}+\sqrt{2^{-7}-\sqrt{2^{-15}}}}}$	5	21	
(-,+)	$\frac{7}{8}$	$\sqrt{2^{-1}-\sqrt{2^{-3}+\sqrt{2^{-7}}}}$	2	10	
(-,+,+)	$\frac{15}{16}$	$\sqrt{2^{-1}-\sqrt{2^{-3}+\sqrt{2^{-7}+\sqrt{2^{-15}}}}}$	4	20	

Figure 1: Values from the hexadecaic interval

We could stop here - the algorithm is simple enough. The purists would of course object and complain about the ugliness of the code, the abundance of fixed-point modifiers, parentheses, etc. Therefore we will *not* stop, and proceed to actually extend Scheme with *anatypes* (short for *anaphoric* types), described in section A.1 which is included as supplementary material since this largely concerns only syntactic improvements. All such syntactic improvements (implemented as macros) will be used for code snippets in the followup. Additionally, from now on, most binding forms will be capitalised to highlight the fact that the language they use is Scheme extended with anatype annotations as in e.g., Figure 2.

2.2 First target: Cosine

Returning to our goal of implementing all trigonometric functions, let us first focus on the simplest one: the cosine. One may remember from elementary trigonometry that in order to compute $\cos x$ it is only needed to find a value for $x \in [0, \frac{\pi}{2}]$, the rest can be found by symmetry (please see section 3 for a common way to resolve this). Also, we only need to implement just one of these functions really, since standard trigonometric identities can be used to derive all the other ones.

So in our quest let us focus on just the values of $\cos \frac{n\pi}{2m}$ for dyadic rationals $\frac{n}{m}$ within this interval. Remember that we are re-scaling the inputs to our function by $\frac{\pi}{2}$, keeping dyadics $n*2^{-k}, 0 \leq n \leq 2^k$. Leaving out $\frac{\pi}{2}$ as an implicit domain/range factor will also simplify reverse-mode trigonometric functions, making them more precise at the same time (see section 2.5). A first observation is that $\cos 0 = 1$ or $\cos \frac{2^k \pi}{2*2^k} = 0$ so the exact value is already known. A second observation is that this function is strictly *monotonically decreasing*. This means that when the argument $n*2^{-k} \to 0$ the result should approach 1 from below. So for example for all rationals $\frac{1}{2}, \frac{1}{4} ... \frac{1}{2^k}$ the signs are necessarily positive: $s_i^k(n) = 1$ (as was already shown by [22]), since the sqrt function itself is strictly monotonically *increasing*.

But how do we compute correct signs for other dyadic rationals in the wanted domain? Immediate use of the definition of $s_i^k(n)$ seems prohibitive from the efficiency perspective. Looking at a few values in Figure 1 we can observe the pattern: as the numerator increases, the signs change from positive, then to zero (indicated by the absence of a sign in the left column) and then to negative (in the first half). If there is an odd number of negative signs to the left inside the nested radical (in the sequence $s_j^k(n), j < i$), then the order reverses to first negative, then zero and then positive. One can also observe from Figure 1 that for the bottom half of the domain, the order of signs is reversed w.r.t. the top half. So we can consider the evaluation of such functions as finding a "signature", or a position, of each dyadic rational in a *binary search* "tree" (depicted in the rightmost column of Figure 1), determined by the linear partitioning of the sector into an exponential number of equally-spaced sub-sectors (minus one, since we are excluding 0 and 1). Because of the monotonicity of the target function ($\cos x$) and the function used to implement it ($\sqrt{x}$), the signature must exhibit a memory-like behaviour, recording the number of negative signs in the nested radical expansion. Once a single (or an odd number of) negative sign(s) is reached, all further signs are inverted. An involution (i.e., a multiple of two of negative turns) makes the sign positive again.

To implement the signature we can use a simple bit-mask sig (column 4 in Figure 1) that contains the bit 1 in the position of each negative turn. Since a *continued* radical is best defined in a continued form, an obvious approach in functional programming tradition for implementing an approximation to the continued radical above is to use the venerable *continuation passing style*. In other words, on every expansion (i.e., at each level of the binary search) of the half-angle formula, we take the following steps:

1. compute the term $\frac{2}{2^{2^i}}, i \geq 1$ as (* 2 f)

    (a) starting from a rational $f = \frac{1}{4}$ or flonum (exp2 -2)
    (b) for each level (i), repeatedly square it

2. add the continuant to the term

3. take the square root, in either fixnums or flonums

4. compute the sign using bit-mask `sig` (masking the last turn, bit 0)

5. deliver the signed result to the continuation

```
(Define (dyadic-cos1 n :fix m :fix sig :fix f :flo cont :proc)
 (cond ([= n 0] (cont 1))                                          ;; exact
       ([= n m] (cont 0))                                          ;; results
 (else (Let* (fix:[m (rsh m 1)]
                  [sign (if [= (bit-and sig 1) 1] -1 1)])
  (cond ([= n m]
         (cont (* sign (sqrt (* 2 f)))))
        ([< n m]
         (dyadic-cos1 n m
            (bit-or (bit-lsh sig 1)
               (if (even? (bit-count sig))
                  0 1))
            (sqr f)
            (λ (x) (cont (* sign (sqrt (+ (* 2 f) x)))))
            ))
        ([> n m]
         (dyadic-cos1 (- n m) m
            (bit-or (bit-lsh sig 1)
               (if (even? (bit-count sig))
                  1 0))
            (sqr f)
            (λ (x) (cont (* sign (sqrt (+ (* 2 f) x)))))
            )))))))
```

Figure 2: Computing dyadic cosine using continuations

In Figure 2 we see that as soon as bisection of the interval hits the rational $\frac{n}{m}$ corresponding to the argument of $\cos\frac{n\pi}{2m}$, the continuant can be assumed to be 0 and the value can finally be delivered to the chain of continuations that we have built by this tail-recursion. On every level, the interval is halved (m and n steadily approach each other, the former by halving and the latter by subtraction whenever $n > m$). The signature is maintained using bit-shift and bit-counting according to the scheme described above. Please see Figure 1 for successive values of `sig`, which appear to behave quite erratically. However, it can be shown that `sig`, in fact, is an example of an infinite *fractal* sequence (containing itself as a sub-sequence) when one concatenates sequences for each successive lattice $2^{-k}, k = [1,2,3...]$: [0] [0 0 1] [0 0 1 0 3 1 2] [0 0 1 0 3 1 2 0 6 3 7 1 5 2 4].... We have attempted to derive an explicit formula for computing `sig`, by priming it with one extra bit in front (and reversing bit-counting logic in the algorithm of course, see section 2.4). Resulting values of `sig'` (column 5 in Figure 1) look regular, however, the pattern depends on both n and m and exhibits fractal mirroring, so further simplifications remain elusive (fractal sequences from the OEIS A101279 for binary trees, A131987 for dyadic rationals and A025480 for the Grundy values do appear to be related to our sequence for `sig'`).

Running (dyadic-cos1 n m #b0 (/ 1 4) (λ (x) x)) is an effective way to compute $\cos\frac{n\pi}{2m}$ and costs $O(\log m)$ evaluations of `sqrt` at each level (all logarithms in this paper are base-2) requiring

constant time each. No tables are in sight and the result is available at varying levels of accuracy. The fixed-point operator modifiers are replaced by anatype annotations for each binding: the inputs integers use `fix:` while the output floats use `flo:`, in either case, appropriate type-specialised arithmetic operators are automatically substituted in place of their generic versions. Below, we shall also elide prefixes from bitwise operators e.g., *bit-and* $\mapsto$ *and*, typesetting logical operators such as **and** to distinguish.

2.3 Following Reynolds ...

Let us proceed in improving our algorithm for $\cos\frac{n\pi}{2m}$. The code in Figure 2 contains many nested expressions for computing various quantities. It would be nice if we could *thread* the temporary values through such expressions without having to deal with deep nesting that is inevitable with a language such as Scheme. In section A.2 we introduce our macros for threading which help to alleviate the problem. The `>->` form used in code below takes a flat list of expressions (which can be read inside-out, from last to first) and substitutes in each expression all occurrences of _ (i.e., *anaphroric* references) by the syntax of the preceding expression. Again, this is another straightforward syntactic transformation that will prove helpful in later code snippets in this paper.

```
(Define (dyadic-cos2 fix: n m sig f lf)
(cond ([= n 0] 1)                              ;; exact
      ([= n m] 0)                              ;; results
(else (Let* ([m :fix (rsh m 1)])
 (cond ([= n m] (apply2 sig f lf))
       ([< n m] (dyadic-cos2 n m
                    (>-> sig (lsh _ 1)
                       (or _ (if (even? (bit-count sig))
                              0 1)))
                    (lsh f lf)
                    (lsh lf 1)
                    ))
       ([> n m] (dyadic-cos2 (- n m) m
                    (>-> sig (lsh _ 1)
                       (or _ (if (even? (bit-count sig))
                              1 0)))
                    (lsh f lf)
                    (lsh lf 1)
                    )))))))
```

Figure 3: Defunctionalized dyadic cosine

Introduction of a bit-mask `sig` tracking the number of negative signs as the algorithm dichotomises the interval $[0,\frac{\pi}{2}]$ has a benevolent side-effect: it provides enough information to actually recreate the path to the wanted value in a corresponding binary search "tree". This can be captured by *defunctionalized* [15] version of the algorithm found in Figure 3. With defunctionalization of a higher-order function such as `dyadic-cos1`, we are constructing an object (i.e., a data-structure according to tradition in defunctionalization literature) instead of a chain of continuations built with λs. This object is interpreted in a separate function that re-creates all the needed steps. In this particular case, the object is just the integer `sig`, which is unwound by right bit-shifts found in Figure 4.

Our new algorithm also uses bit-shifts to avoid squaring, using a new fixnum `f` and its $\log f = lf$ (initial $f_0 = 4, lf_0 = 2$). Each time, `f` is "squared" by multiplying it with 2^{lf}, while `lf` is multiplied by 2.

The bit-mask `sig` is used in a separate function, `apply2` (see Figure 4) where it is replayed to recreate all the turns of the binary search. Because now factor `f` is a fixnum value, a conversion between `fix:` and `flo:` anatypes is needed (expanding to standard `exact->inexact` library call). Note that now we can also play with other defunctionalizations. For example, we can replace `(/ 2 (toflo f)` by `(exp2 (- 1 lf))`, keeping only `lf` in the loop (elided here).

```
(Define (apply2 fix: sig f lf)
 (Let loop (fix:[s sig]
               [f f]
               [lf (rsh lf 1)]
               [a :flo 0])
    (if (zero? lf)
        a
        (loop (rsh s 1)
           (rsh f lf)
           (rsh lf 1)
           (* (if [= (and s 1) 1] -1 1)
              (sqrt (+ (/ 2 (toflo f))
                       a)))))))))
```

Figure 4: Apply for defunctionalized dyadic cosine

2.4   Control and representation optimisations

The fact that using our recursion approach, vanishingly small radical terms need to be evaluated (e.g., `(sqr f)`, `(/ 2 (toflo f))` or `(exp2 (- 1 lf))`) means that the accuracy of our result (evaluation of a trigonometric function) is limited by the precision of the minimal representable positive term. Using fixnums for the latter, or flonums for the former limits the number of useful, positive radical terms. This implies that accuracy below `1e-10` (for `dyadic-cos1`) or below `1e-5` (for `dyadic-cos2`) is unattainable. Like in tail-recursion, which ensures that no memory is lost during recursion, we would like to be able to iterate continued radical as much as we need, with no loss of the accuracy. Of course, the final accuracy will always be limited by the precision of the output flonum (and indeed by the accuracy of the `sqrt` implementation). Our experiments with the trigonometric Taylor series and the CORDIC indicate that these best-in-class algorithms do reach expected accuracy for IEEE *double-precision* flonums (also called `binary64`) of about $2^{-52}/2 \approx$ `1e-16`, so we need to do better.

```
(Define (apply3 fix: sig d)                   (Define (apply4 fix: sig')
 (Let loop (fix:[s sig] [i 0] [a :flo 0])      (Let loop (fix:[s sig'] [a :flo 0])
   (if (>= i d)                                  (if (<= s 3)
     (+ 2 a)                                       (+ 2 a)
     (loop (rsh s 1) (+ i 1)                       (loop (rsh s 1)
       (* (>-> s (and _ 1) (lsh _ 1)               (* (>-> s (and _ 1) (lsh _ 1)
                          (- 1 _))                                    (- 1 _))
         (sqrt (+ 2 a)))                             (sqrt (+ 2 a)))
       ))))                                        ))))
```

Figure 5: Improved depth-limited `apply3` and final `apply4`

Going back to our very first equation (Pythagorean identity), however, we see that $\cos\frac{\pi}{4} = \frac{1}{2}\sqrt{2}$. We observe that halving the radical can simplify (possibly vanishing) term(s) that are inside the radical. To see how this works out let us rewrite the whole expression as $s_0^k\sqrt{f_1 + s_1^k\sqrt{f_2 + s_2^k\sqrt{\ldots}}}$ where $f_0 = 2^{-0} = 1, f_1 = 2^{-1}, f_2 = 2^{-3}, f_i = \frac{f_{i-1}^2}{2} = 2^{-2^i+1}$ (for $i \geq 0$). Let us also drop the omnipresent argument to $s_i^k(n)$ for simplicity of the formulas. Transforming our last equation from the beginning of section 2, let us therefore halve the whole radical, bringing inside the radical a factor of $2^2 = 4$ to compensate:

$$\frac{s_0^k}{2}\sqrt{\frac{2^2}{2} + 2^2 s_1^k\sqrt{\frac{1}{2*2^2} + s_2^k\sqrt{\frac{1}{2*(2*2^2)^2}} + \ldots}} = \frac{s_0^k}{2}\sqrt{2 + s_1^k\sqrt{\frac{4^2}{2*2^2} + 4^2 s_2^k\sqrt{\frac{1}{2*(2*2^2)^2}} + \ldots}}$$

We see that in order to bring the re-balancing factor inside each nested radical we need to square the last factor, starting from 2.

$$\frac{s_0^k}{2}\sqrt{2 + s_1^k\sqrt{2 + s_2^k\sqrt{\frac{4^4}{2*(2*4)^2} + 4^4 s_3^k\sqrt{\ldots}}}} = \frac{s_0^k}{2}\sqrt{2 + s_1^k\sqrt{2 + s_2^k\sqrt{2 + s_3^k\sqrt{2 + \ldots}}}}$$

Thus the iterated re-balancing of the factors $f_i, \forall i > 0$ through multiplication by $g_0 = 2, g_1 = 2^2, g_i = 2^{2^i}$ (for $i \geq 0$) giving $g_i * f_i = 2^{2^i - 2^i + 1} = 2$ successfully brings the halving that causes terms to become smaller and smaller completely outside the radical. This leaves us with much simpler `apply4` that does not even need to perform squaring and uses a single constant factor $f_i = 2, \forall i > 0$, see Figure 5.

Note that both the depth of binary search as well as the signature are now represented by `sig'` where the initial value is 2 (`#b10` in binary). This initial value is our termination condition when we unwind the bit-mask by right-shifting. In the code above, however, we check for $s \leq 3$ (the reason for this shall become apparent very soon). Before, the depth needed to be represented separately (by `lf`) since the mapping between $s_i^k(n)$ and `sig` is not *bijective*. In fact, `sig` has a value of 0 for all $\frac{n}{m} = 2^{-k}, k \geq 1$. Although this is sufficient for identifying the signs given a value (i.e., the mapping is *surjective*), we really need *injectivity* in order to keep track of the tree depth and stop the tail recursion appropriately.

Let us use `sig'` in Figure 6 then, which is a primed version of `sig` we discussed above. It implies that the bit-counting logic for the main loop needs to be reversed (since there is now an extra bit in the most significant position). Next to this, we can now simplify calculation of $s_i^k(n)$ even more by making use of $1 - 2(s \bmod 2)$ to represent the mapping $\{1 \mapsto -1, 0 \mapsto 1\}$. Similarly in `dyadic-loop` function, we can rewrite the "update" of `sig` for the next iteration accordingly, making use of the representation $\{1 \mapsto 0, 0 \mapsto 1\} \equiv 1 - s \bmod 2$:

```
(>-> s (lsh _ 1)
     (or _ (if (even? (bit-count s))
             1 0)))
```

$-\{\text{strength-reduce}\} \rightarrow$

```
(>-> s                 ;; [Nth 5]
     (lsh _ 1)         ;; [Nth 4]
     (bit-count s)     ;; [Nth 3]
     (and _ 1)         ;; [Nth 2]
     (- 1 _)           ;; [Nth 1]
     (or _ [Nth 4]))   ;; refer to the above sub-expressions 1 and 4
```

```
(Define (dyadic-loop x :rat sig' :fix)
(cond ([= x 0] (* 4 (- 1 (and sig' 1))))       ;; exact
      ([= x 1] (* 4 (and sig' 1)))             ;; results
(else (Let loop (fix:[n (numerator x)]
                     [m (denominator x)]
                     [s sig'])
 (Let* ([m :fix (rsh m 1)])
  (cond ([= n m] (apply4 s))
        ([< n m] (loop n m
                       (>-> s (lsh _ 1)
                            (bit-count s)
                            (and _ 1)          ;; bit-counting
                            (- 1 _)            ;; logic reversed
                            (or _ [Nth 4])
                       )))
        ([> n m] (loop (- n m) m
                       (>-> s (lsh _ 1)
                            (bit-count s)
                            (and _ 1)          ;; bit-counting
                            (or _ [Nth 3])     ;; logic reversed
                       )))))))))
```

Figure 6: Improved and generalised loop for dyadic rationals

Combining all these improvements together we arrive at our final parsimonious iterative, mostly control-flow free driver for our target function, which now inputs a `rat:` annotated dyadic rational `x` and the initial mask `sig'`. Even though rational arithmetic is specified in R5RS, not all implementations include them by default. With macros, however, it is not difficult to extend Bigloo to handle rational numbers seamlessly (see section A.3 where we show how `sqrt` can be overloaded for rational inputs).

A keen reader will have noticed that `apply4` now stops earlier and does not apply the final `sqrt` (and halving). For the same reason, in the `dyadic-loop`, the exact result is multiplied by 4. The answer as to why was to be expected: we want to be able to calculate functions other than just $\cos\frac{n\pi}{2m}$ of course!

2.4.1   A surprising generalisation: Sine

Keeping track of the signs in the sequence $s_i^k, i \in [0,1,2...]$ and interpreting bit-mask `sig` as such a sequence (using 1 to represent -1 and 0 to represent 1) has another benevolent side effect: we can actually represent the sin function by exact same algorithm as for cos, just using initial value of $s_0^k = -1$.

This corresponds to the initial value of `sig'=#b11=3` instead of `#b10=2`, incidentally explaining why we used ≤ 3 to check for termination in `apply4` from Figure 5. This is because in general, $\sin x = \sqrt{1 - \cos^2 x} = \sqrt{1 - \frac{1}{2^2}(2 \pm \sqrt{...})} = \frac{1}{2}\sqrt{4 - \frac{4}{2^2}(2 \pm \sqrt{...})} = \frac{1}{2}\sqrt{2 \mp \sqrt{...}}$, so simply inverting the sign under the first radical will do the trick. Equipped with the higher-order machinery from section A.4 which introduces a syntactic composition operator ($\circ$) and a currying form (κ), we can now attack other trigonometric functions by playing with initial values of `sig'` and values returned from `dyadic-loop/apply4`. For example, squared results are simply obtained by *not* performing final `sqrt`. Expressions for tan and cot are well-known from elementary trigonometry.

```
(Define (dyadic-cos x) ((○ half sqrt) (dyadic-loop x #b10)))
(Define (dyadic-sin x) ((○ half sqrt) (dyadic-loop x #b11)))
(Define (dyadic-tan x) ((○ sqrt sub1 reciprocal [κ / _ 4]) (dyadic-loop x #b10)))
(Define (dyadic-cot x) ((○ sqrt sub1 reciprocal [κ / _ 4]) (dyadic-loop x #b11)))
(Define (dyadic-cos² x) ([κ / _ 4] (dyadic-loop x #b10)))
(Define (dyadic-sin² x) ([κ / _ 4] (dyadic-loop x #b11)))
(Define (dyadic-tan² x) ((○ sub1 reciprocal [κ / _ 4]) (dyadic-loop x #b10)))
(Define (dyadic-cot² x) ((○ sub1 reciprocal [κ / _ 4]) (dyadic-loop x #b11)))
```

2.5 Inverse trigonometric functions

```
(Define (dyadic-cos-1 x :flo ss :fix *eps* :flo)
(cond ([= x 0.] (- 1 ss))                                   ;; exact
      ([= x 1.] ss)                                         ;; results
      (else
(Let loop ([y :rat #r:1/2]
           [s :fix ss]
           [d :fix 0])
 (Let ([m :fix (>-> (denominator y) :fix (lsh _ 1))]
       [xe :flo ((○ half sqrt) (apply3 s d))]
       [sss (>-> ss (lsh _ 1) (- 1 _) (toflo _))])
  (cond ([< (abs (- x xe)) *eps*] y)
        ([< (* sss x) (* sss xe)]
         (loop (+ y #r:1/m)
            (>-> s (lsh _ 1)
               (bit-count s)
               (and _ 1)
               (- 1 _)
               (or _ [Nth 4]))
            (+ d 1)
            ))
        ([> (* sss x) (* sss xe)]
         (loop (- y #r:1/m)
            (>-> s (lsh _ 1)
               (bit-count s)
               (and _ 1)
               (or _ [Nth 3]))
            (+ d 1)
            ))))))))
```

Figure 7: First attempt at inverse dyadic cosine

There are many approaches for implementing a reverse-mode algorithm provided an implementation of a forward-mode bijective function. Most straightforward is to search for the value in the domain provided a value from the range. Since both $\sin x$ and $\cos x$ are monotonic in the interval $x \in [0, \frac{\pi}{2}]$, we could simply apply a search procedure like the one depicted in Figure 7. On first sight, it seems to be efficient, using binary search that dichotomises the input domain $[0, 1]$ according to the values returned by the forward function given the corresponding search key (i.e, our dyadic rational #r:n/m representing the argument), which gets successively approximated according to the deviation from the target value returned by the forward function evaluation.

However, a proper search procedure should not insist on extra $O(\log \frac{1}{p})$ evaluations of a continued radical (where $p = \frac{1}{m} = 2^{-k}$ is the precision of the input value). Even though with each new level we can stop evaluation at p (corresponding to the depth of $\log \frac{1}{p} = k$), the total cost still amounts to $1 + 2 + 3 + \ldots + k = O(k^2)$ which as we know grows very quickly as $p \to 0$. Besides, we are seeking a general method for all trigonometric inverses, not just one for $\cos^{-1} x$ (a value returned by `apply3` needs to be translated to actual forward function target value, using (○ half sqrt)). We would really like to fuse the binary search together with the evaluation of each corresponding target value used to drive the decision process, and get rid of the overhead that depends on the argument.

```
(Define (dyadic-find5 x :flo ss :fix *eps* :flo)
(cond ([= x 0.] (- 1 ss))                                    ;; exact
      ([= x 4.] ss)                                          ;; results
(else (Let loop ([y :rat #r:1/2]
                 [x :flo x]
                 [s :fix ss])
 (Let ([m :fix (>-> (denominator y) :fix (lsh _ 1))])
  (cond ([< (abs (- x 2)) *eps*] y)
        ([< x 2.]
         (loop (+ y (* (>-> s (bit-count _)
                              (and _ 1)
                              (lsh _ 1)
                              (- 1 _))
                       #r:1/m))
               (>-> x (- _ 2) (sqr _))
               (>-> s (lsh _ 1) (or _ 1))
               ))
        ([> x 2.]
         (loop (- y (* (>-> s (bit-count _)
                              (and _ 1)
                              (lsh _ 1)
                              (- 1 _))
                       #r:1/m))
               (>-> x (- _ 2) (sqr _))
               (>-> s (lsh _ 1) (or _ 0))
               ))))))))))
```

Figure 8: Fused and generalised inverse on dyadic rationals

Looking again at our radical from section 2.4, we see that repeated squaring can undo the steps performed by the forward-mode trigonometric algorithm. Since the argument is uniquely determined by the signature s_i^k (refer to Figure 1), this sequence of signs is exactly what we seek when computing an inverse. It is natural to accomplish this by performing the steps of the forward algorithm in reverse:

1. initial remainder is set to the input value, doubled and squared: $(○ \text{ half sqrt})^{-1} = (○ \text{ sqr double})$

2. at every un-nesting, determine:
    (a) if the remainder is approximately equal to 2, terminate the search
    (b) if less than 2, the sign $s_i^k = -1$
    (c) if greater than 2, the sign is $s_i^k = 1$
    (d) In either case, 2 is subtracted and the remainder is squared

The argument is adjusted according to the bit-counting logic, taking binary search history into account. We can do all of this precisely because $\sqrt{x} \geq 0, \forall x \geq 0$ always, allowing the conditional to uniquely determine the sign sequence (see Figure 8). And indeed, a repeated application of `sqr` instead of repeated application of `sqrt`, is essentially "unwinding" the nested radical and therefore correctly computing the $\frac{n}{m} = \frac{2\cos^{-1}x}{\pi}$ with provably no superfluous computational steps performed.

```
(def-syntax *eps* 1e-10)
(Define [dyadic-acos x] (dyadic-find5 ((∘ sqr double) x) 0 *eps*))
(Define [dyadic-asin x] (dyadic-find5 ((∘ sqr double) x) 1 *eps*))
(Define [dyadic-atan x] (dyadic-find5 ((∘ sqr double sqrt reciprocal add1 sqr) x) 0 *eps*))
(Define [dyadic-acot x] (dyadic-find5 ((∘ sqr double sqrt reciprocal add1 sqr) x) 1 *eps*))
```

Concluding, we notice in passing that (∘ sqr double sqrt reciprocal add1 sqr)≡(∘ [κ / 4 _] add1 sqr) so our final inverse functions can all be rewritten to avoid using the `sqrt`. This completes the exposition of the complementary relationship between trigonometric functions that we expect from the mathematical point of view: forward-mode functions use the `sqrt` while the reverse-mode functions use the `sqr`. It would be not surprising if the implementation of either side of this dualism would be derived mechanically from the other side, automatically proving correctness of the approach.

3   Evaluation

Because trigonometric functions are symmetric, a simple approach to extend the domain for our forward-mode to $[0, 2\pi]$ is to record the *quadrature* of the argument as a pair of signs (1st quadrant encoded as +/+, 2nd quadrant as -/+, 3rd quadrant as -/- and 4th as +/-). The first (`inv`) identifies the sign of cos x as x circles the quadrants in sequence. The second sign (`mir`) acts similarly for the sin function.

```
(Define (quadrature x :rat inv :fix mir :fix)
    (cond ((or [> x #r:1/4] [< x #r:-1/4])
           (quadrature (- x (* (sign x) #r:1/2)) (neg inv) (neg mir)))
          ([< x 0] (quadrature (neg x) inv (neg mir)))
          (else (values x inv mir))))
(Define (Cos x :rat)
    (let-values ([(y inv mir) (quadrature x 1 1)])
      (* inv (dyadic-cos (* y 4)))))
(Define (Sin x :rat)
    (let-values ([(y inv mir) (quadrature x 1 1)])
      (* mir (dyadic-sin (* y 4)))))
```

These functions use the *multiple-value* facility, specified in Scheme Request for Implementation (SRFI) #11 [7], which allows the caller to simultaneously receive more than one return value from the callee. The latter constructs such a multiple-value using the `values` form. Similarly, the signs of the input domain can be used to expand the range of the result for reverse-mode trigonometric functions to their appropriate values ($[\frac{-\pi}{2}, \frac{\pi}{2}]$ for arcsin and $[0, \pi]$ for arccos, $(\frac{-\pi}{2}, \frac{\pi}{2})$ for arctan and $(0, \pi)$ for *acot*).

In order to properly evaluate accuracy and performance of our algorithms and ensure fair comparison to baseline C implementations, we have re-implemented them the improved version of the above algorithms using C, taking the above Scheme versions as a specification (and as a proof of correctness). The use of multi-cycle `sqrt` in forward-mode trigonometry prevents efficient leverage of the approach. For inverse functions, however, we require *squaring*, which is much less complex than `sqrt` or a a full multiplier hardware-wise (see [4], [17]). All tests used GNU Compiler Collection (GCC) 7.2.0 running on `x86_64-linux-gnu` architecture (Ubuntu 17.10 4-way Intel(R) Core(TM)

i7-4600U CPU @ 2.10GHz). Used compiler options were: `-static -Ofast -mabm -ffast-math -fschedule-insns2`. We used `1e7` random IEEE `double` values as the basis for comparison.

3.1   Accuracy

We mentioned above that IEEE double-precision flonums support accuracies of down to `1e-16`. Our implementation of basic trigonometric functions reaches this accuracy (going as low as `8e-17` for cos and sin). Their inverses arccos and arcsin return exact (rational) results. To reach similar accuracy of `5e-17` (even on the interval $x \in [0, \frac{\pi}{2}]$), a Taylor series for $\cos x$ would need to be expanded to the 10th term (requiring calculation of x^{20}, as well as a division by 20!). Unfortunately, even when applying the Horner's rule, a series of 10 square operations, accompanied by 10 full multipliers and significantly complicated dividers would be needed to match the accuracy of the presented dyadic cosine.

Evaluation at a dyadic rational with depth equal to its precision still gives accurate results, while for regular iterative algorithms the performance has characteristics that follow the law of diminishing returns. One sees from Figure 9 that arctan and arccot do reach the error floor at $\approx$ `4e-24` while arccos and arcsin can yet be ≈ 2 orders of magnitude more accurate (as they don't use reciprocals).
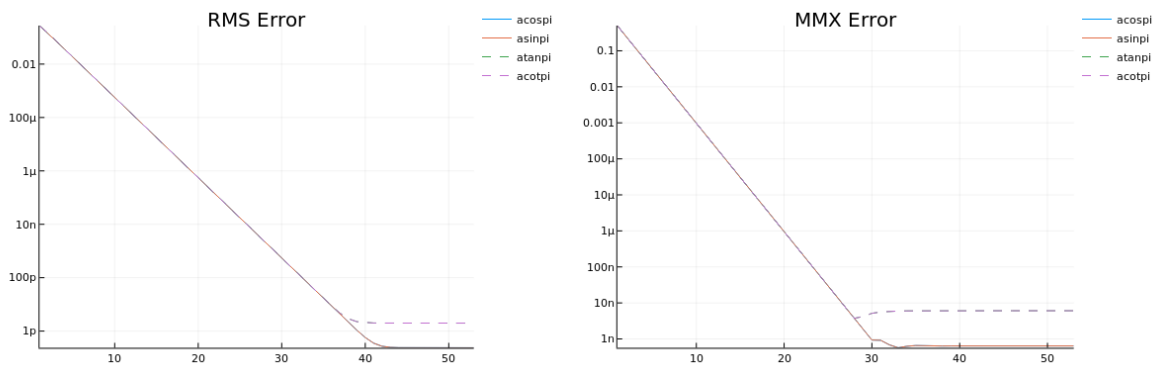
Figure 9: RMS and MXM errors (for reverse-mode trigonometric functions)

Similarly, for the CORDIC, at least 45 (for reverse-mode trigonometry) or 52 (for forward-mode trigonometry) iterations would be required to reach comparable accuracy of `1e-16`, since on each iteration, the CORDIC produces a single bit of the result. In our experiments we have observed that our algorithms for tan and cot turn out to be order-3 less accurate than the other two trigonometric functions. We attribute this to the use of floating-point `reciprocal` in their implementations. Figure 9 shows that Root Mean Square (RMS) error (defined as $\sqrt{(target - expect)^2}$) exhibits linear scaling up to ≈ 40 iterations while Max Magnitude (MXM) error (defined as $\max |target - expect|$) exhibits linear scaling up to ≈ 30 iterations, which is often acceptable in embedded applications.

3.2   Performance

Both forward and reverse trigonometric functions appear to require a logarithmic $O(\log m) = O(k)$ number of iterations (thanks to the binary search). The use of only basic arithmetic operations, and exclusive use of tail-recursion implies that presented algorithms map well to embedded hardware and software

implementations. Presented method of performing the `sqrt` and existing approaches for `sqr` together imply that the combined performance of our dyadic trigonometry is expected to be reasonably good.

Comparing run-times of this implementation when executing the full scale of 2^{20} dyadic rationals considered in this paper (compiled to native code via C using Bigloo 4.1a), we see that it is only 2.74 times slower than standard implementations (originating in the C math library). For dyadics with precision 2^{-10} the slowdown is only 1.57 (for $k = 5$ it is just 1.2, and so on with the asymptotic limit of 1.06). We believe this overhead can be addressed by rewriting the algorithm in C, if needed (which is not a huge undertaking given the effort we spent to make it embeddable). The accuracy and performance is achieved at the cost of generality: our trigonometric functions only accept dyadic rationals and (in their presented form) may completely fail for other rationals. E.g., for $\frac{n}{m} = \frac{5}{2^k - 1}$ our `dyadic-loop` may even fail to terminate!
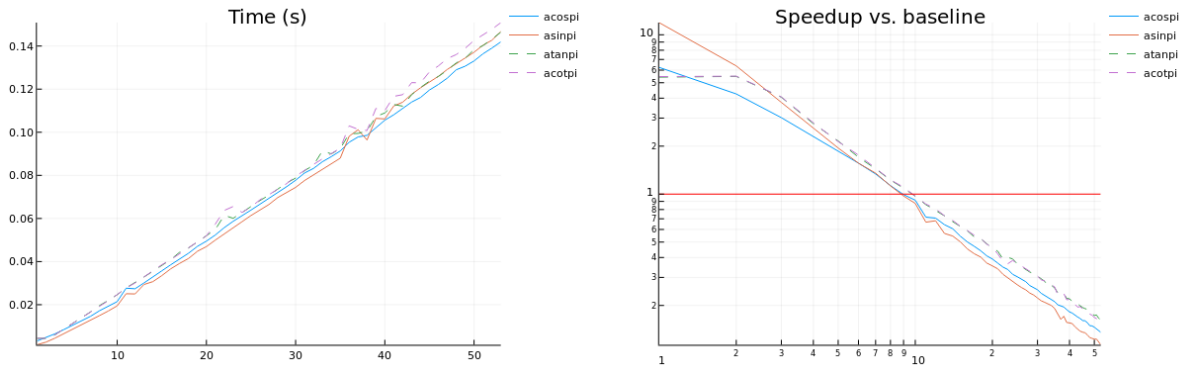

Figure 10: Performance and speedup wrt. the baseline (for reverse-mode trigonometric functions)

From Figure 10 one can observe that unlike baseline implementations (which require constant time for any argument), our algorithm can be scaled, depending on required precision. With our current C implementation, we observe that at around 9 iterations, one can generate a result which is still accurate to ≈ 3 decimal digits faster than the off-the-shelf implementation.

3.3   Related work

Trigonometry has been in widespread use since ancient times. It still forms the basis of many branches of science and engineering. Complex numbers, for example, naturally correspond to conjoined trigonometric functions. Many approaches to compute trigonometric functions have been tried. Tables [3] have been created in Babylon, during the classic period, in the Arabic world, in India and China. CORDIC methods [18] rely on tabling of the values x for which corresponding $\tan x = 2^{-k}, k \in \mathbb{N}$. Using Taylor/Maclaurin series is also popular, however, applying these beyond just a few terms can quickly become costly. Methods that combine tabling, series expansions or polynomial interpolation are also known [5].

Efficient methods to compute `sqrt` and derived functions have been extensively studied too [1]. Computer arithmetic in general and IEEE `binary64` format in particular seem to be suitable for efficient implementations of `sqrt` [13]. Although the basic idea to express basic trigonometric functions using radicals is not new and has appeared for example in [20] and in [16], the exact algorithm presented in this paper was not generally known (at least, to the knowledge available to us). Even more so, our approach to fuse binary search together with the evaluation of inverses of basic trigonometric functions on dyadic rationals is new, as far as we can judge.

4   Conclusions

We believe our approach is radically different in that it brings together two seemingly unrelated functions (`sqrt` and `cos`) and expresses the latter in terms of the former. This application of functional programming is beneficial in showing that much good can materialise when one thinks algorithmically, in addition to thinking mathematically. In particular, one can derive an algorithm for the inverse quite straightforwardly given the algebraic representation of the forward calculation.

Comparing Taylor/Maclaurin series of forward and reverse trigonometric functions one sees very few similarities. We believe that our approach to derive the algorithm (and prove correct) the implementation of trigonometric functions and their inverses deserves some attention.

Threading macros provide powerful syntax to automatically generate complex applicative forms, where bindings are propagated to an expression in a deeply nested program structure. This is more general than traditional (point-free) function composition in that it does not require concatenative style programming ala Forth, with `dup`, `curry`, `uncurry`, sorting of the argument lists into the appropriate order etc., for each of the sub-expressions.

Our algorithms exhibit decent performance mainly because of the use of type-specialised operators and library functions. Dynamic typing and dispatch may be fine to start with, but in order to match the performance of C, any functional language targeting numerical and scientific computing needs to address both of these aspects.

We use the application of dyadic rational trigonometry to show the ease with which Scheme's weak hygiene may be subverted to suit somewhat more advanced (albeit esoteric to some) desires: sectioning, overloading, threading, extensions to the number tower. All these, together with composition transformers are beneficial as core components of an algorithmic Domain-Specific Language (DSL). This improves the exposition of the algorithm as well as elucidates basic mathematical relationships between the algorithms for computation of trigonometric functions and their inverses. Although especially useful for this particular domain, tail-recursive style of Scheme in general is excellent in prototyping of embedded scientific and numerical algorithms.

4.1   Future work

Continued radicals are similar in spirit to continued fractions. The fact that one can compute `reciprocal` trivially with continued fractions, may indicate that by changing the representation again to a continued fraction, we may improve the accuracy of tan and cot significantly.

It might be interesting to extend relational arithmetic of [10], which works using unary arithmetic, to binary and augment it with an implementation of `sqrt` as a relation. This can then be used to define a single relation implementing both forward-mode and reverse-mode trigonometric functions.

As of now, presented functions do not handle non-dyadic rationals or flonums. An obvious extension is to generalise presented method to a low-order Taylor expansion centered around the "nearest" dyadic rational. Since trigonometric functions are *analytic* (viz., infinitely differentiable), and because we can simultaneously compute both cos as well as its derivative $(-\sin)$ by simple swapping of signs outside and inside the first radical, the Taylor theorem may be applied to yield a good approximation using only two or three terms.

I would like to thank the anonymous reviewers.

References

[1] Milton Abramowitz & Irene A Stegun (1964): *Handbook of mathematical functions: with formulas, graphs, and mathematical tables*. 55, Courier Corporation.

[2] N. I. Adams, IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, M. Wand & H. Abelson (1998): *Revised^5 report on the algorithmic language scheme*. SIGPLAN Not. 33(9), pp. 26–76, doi:10.1145/290229.290234.

[3] Carl B Boyer & Uta C Merzbach (2011): *A history of mathematics*. John Wiley & Sons.

[4] Aditya Deshpande & Jeff Draper (2010): *Squaring units and a comparison with multipliers*. In: *Circuits and Systems (MWSCAS), 2010 53rd IEEE International Midwest Symposium on*, IEEE, pp. 1266–1269, doi:10.1109/MWSCAS.2010.5548763.

[5] Shmuel Gal (1986): *Computing elementary functions: A new approach for achieving high accuracy and good performance*. In: *Accurate Scientific Computations*, Springer, pp. 1–16, doi:10.1007/3-540-16798-6_1.

[6] Martin Guy (1985): *sqrt.c*. Internet. Available at `http://medialab.freaknet.org/martin/src/sqrt/sqrt.c`.

[7] Lars T Hansen (2000): *SRFI 11: Syntax for receiving multiple values*. Internet. Available at `http://srfi.schemers.org/srfi-11`.

[8] Oleg Kiselyov (2002): *How to write seemingly unhygienic and referentially opaque macros with syntax-rules*. In: *Scheme Workshop*.

[9] Oleg Kiselyov (2002): *Macros That Compose: Systematic Macro Programming*. In: *GPCE*, pp. 202–217. Available at `http://dx.doi.org/10.1007/3-540-45821-2_13`.

[10] Oleg Kiselyov, William E Byrd, Daniel P Friedman & Chung-chieh Shan (2008): *Pure, declarative, and constructive arithmetic relations (declarative pearl)*. In: *International Symposium on Functional and Logic Programming*, Springer, pp. 64–80, doi:10.1007/978-3-540-78969-7_7.

[11] Peter Kourzanov (2015): *anaphora*. Internet. Available at `https://github.com/kourzanov/anaphora`.

[12] Peter Kourzanov (2015): *The art of an anaphoric macro*. In: *IFL*. Available at `https://drive.google.com/drive/folders/0Bymk4OgLRGHkZEZxRGwwd05Wd0E`.

[13] David Kushner (2002): *The wizardry of id [video games]*. IEEE Spectrum 39(8), pp. 42–47, doi:10.1109/MSPEC.2002.1021943.

[14] M Nikulin (2001): *Hazewinkel, Michiel, Encyclopaedia of mathematics: an updated and annotated translation of the Soviet Mathematical encyclopaedia*.

[15] John C. Reynolds (1972): *Definitional Interpreters for Higher-order Programming Languages*. In: *Proceedings of the ACM Annual Conference - Volume 2*, ACM '72, ACM, New York, NY, USA, pp. 717–740, doi:10.1145/800194.805852.

[16] Leslie D Servi (2003): *Nested square roots of 2*. The American mathematical monthly 110(4), pp. 326–330, doi:10.1080/00029890.2003.11919968.

[17] Kabiraj Sethi & Rutuparna Panda (2015): *Multiplier less high-speed squaring circuit for binary numbers*. International Journal of Electronics 102(3), pp. 433–443, doi:10.1080/00207217.2014.897381. arXiv:http://dx.doi.org/10.1080/00207217.2014.897381.

[18] J Volder (1956): *Binary computation algorithms for coordinate rotation and function generation*. Convair Report IAR-1 148, p. 17.

[19] Jack E Volder (2000): *The birth of CORDIC*. Journal of VLSI signal processing systems for signal, image and video technology 25(2), pp. 101–105, doi:10.1023/A:1008110704586.

[20] Stephen Wolfram (2017): *Trigonometric Identities*. Internet. Available at `http://mathworld.wolfram.com/topics/TrigonometricIdentities.html`.

[21] C.C Woo (1922): *The Fundamental Operations in Bead Arithmetic. How to Use the Chinese Abacus*.

[22] Seth Zimmerman & Chungwu Ho (2008): *On infinitely nested radicals*. Mathematics Magazine 81(1), pp. 3–15, doi:10.1080/0025570X.2008.11953522.

A   Supplementary material

A.1   Tangential: anatypes

Often enough, one wants to specialise Scheme code to the types at hand. Although the language itself is uni-typed (with run-time type information attached to every variable), the hygienic `syntax-rules` macro system actually provides the means to create an annotation for every binding and secretly pass around syntactic information that can be used to statically overload common arithmetic operators and library functions. Other functions dealing with character, collections (i.e., vectors, structs, pairs, lists, strings, arrays and matrices) are not used in the paper, but still are supported. Our `sqrt-loop` could shed some clutter indeed if we could drop all type-specialization suffixes from the code.

It seems natural to use a quoted data-like construct (viz., *keywords* in LISP) for the annotation itself, and introduce a new set of basic binding constructs that support such annotations. These "symbols" are matched literally by the macro system, simplifying the expression of our macros somewhat. For the rest of the paper we shall primarily use new forms of `Define`, parallel `Let`, sequential `Let*`. The supporting site [11] includes implementations of many more standard and some less standard binding forms, all of which can be derived from the core forms of `Lambda` (introducing an annotated λ) and `As` (merely annotating an existing, possibly top-level binding). In Figure 14 we give a definition of these two core forms, which makes use of `define-anaphora` machinery (implementation of which is somewhat involved but is described in Figures 12 and 13).

The principle of operation can be described as "infectious" spreading of syntactic information across binding scopes (with credits to Oleg Kiselyov [8] for the term). The same symbol may be annotated differently in different scopes. Each occurrence of this symbol will therefore cause overloaded operators or library functions to (syntactically) expand to their anatype-specialised versions.

The core of the anatype implementation for overloading therefore contains a modular, extensible engine for resolving operators and library function bindings to their ana-typed variants. Starting from a stub for an `unknown:` anatype, each module may add new "sugar" to the syntactic environment, automatically causing all symbols annotated as e.g., `fix:` or `flo:` to be treated as fixnums or flonums, respectively. Figure 16 specifies some mappings that can be specialised this way, using separate syntax transformers `ty-trf` and `op-trf` (see Figure 15 also for the implementation of `extend-ty-sugar`).

One may wonder how this is implemented in pure Scheme, as the definition of `+`, for example, seems to be subjected to non-lexical scoping rules. The answer is simple: weak hygiene. Already described by Oleg Kiselyov, the idea is convert macros to Continuation Passing Style (CPS) [9] and then subvert weak hygiene for locally introduced bindings using Petrofsky extraction. Subsequently the binding macros themselves are re-defined such that any nested re-definition of the binding that is subjected to such "hygiene breaking" is properly propagated (i.e., "infected") into deeper scopes. More background details of the technique can be found in [8].

All this allows us to express the `sqrt-loop`/`sqrt-start` and all our trigonometric functions in much more succinct way. These are, of course not a replacement for regular types: there is neither built-in type-checking nor inference. Anaphoric types serve as mere annotations, supporting overloading in a seamless and transparent way.

```
;; syntax-level λ abstraction
(def-syntax [Λ forms . body]
    (syntax-rules .. () ([_ . forms] . body)))
;; untangle a concatenated list into 2 lists
;; of elements (before and after the separator)
;; (split (fn [] [] .  rest) a b c ...  SEP x y z ...)
;; → (fn (a b c ...)  (x y z ...)  .  rest)
(def-syntax split (syntax-rules (SEP)
 ([_ (k fst [snd ...] . r) SEP  . rest]
  (k fst [snd ... . rest] . r))
 ([_ (k [fst ...] snd . r) x . rest]
  (split (k [fst ... x] snd . r) . rest))
 ))
;; Extract a single colored keyword.  source:  [Kis02]
(def-syntax [extract sym body _k]
    (letrec-syntax ([tr (syntax-rules .. (sym)
      ([_ x sym tail (k (syms ..) . args)]
       (k (syms .. x) . args))
      ([_ d (x . y) tail k]
       (tr x x (y . tail) k))
      ([_ d1 d2 () (k (syms ..) . args)]
       (k (syms .. sym) . args))
      ([_ d1 d2 (x . y) k] (tr x x y k))])
      (tr body body () _k)
    ))
;; Extract multiple colored keywords.  source:  [Kis02]
(def-syntax extract* (syntax-rules ()
    ([_ (sym) body k] (extract sym body k))
    ([_ _syms _body _k]
     (letrec-syntax ([ex-aux
       (syntax-rules ()
          ([_ syms () body (k () . args)]
           (k syms . args))
          ([_ syms (sym . sym*) body k]
           (extract sym body
              (ex-aux syms sym* body k)))
         )])
       (ex-aux () _syms _body _k)))
))
```

Figure 11: Helper macros

```
(def-syntax [define-anaphora (thy ...) (n rd rl) ...]
  (let-syntax-rule ([K (rb) ns ([name rdef rules] ..)]
    (letrec-syntax ([rdef rules] ..)
      (begin (def-syntax [name . fs]
              (extract* (thy ... SEP . ns) fs
                [(Λ (al body)
                   (split (rdef [] [] . body) . al))
                 [] fs]
              )) ..
            )))
    (extract* (recursive-bindings) (rl ...)
      [K [] (n ...) ([n rd rl] ...)])
  ))
```

Figure 12: Defining anaphoric macros

```
(def-syntax redefine-thyself* (syntax-rules ()
 ([_ "internal" ids (slf ...) (rec ...) . body]
  (letrec-syntax ([slf (Λ ts
   (extract* ids ts
      [(Λ (al terms)
          (split (rec [] [] . terms) . al))
       [] ts])
  )] ...) . body
 ))
 ([_ (ids ...) [] (slf ...) recs . body]
  (redefine-thyself* "internal" (ids ... SEP slf ...)
     (slf ...) recs . body))
))
```

Figure 13: Re-definition engine

```
(define-anaphora (type + - * / = < > <= >=          (Lambda rec4 (syntax-rules ()
   neg min max zero? even? odd? positive?           ([_ "int" os ss ct () . k] k)
   negative? and lsh not or rsh xor fix flo)         ([_ "int" os ss ct (n m . r) lt (b ...) . ts]
(As rec3 (syntax-rules ()                             (TYPES (Λ types (member?? n types
 ([_ "int" os ss ct () . k] (begin . k))              (rec4 "int" os ss n (m . r) lt (b ...) . ts)
 ([_ "int" os ss ct (n m . r) . ts]                   (member?? m types
  (TYPES (Λ types (member?? n types                      (rec4 "int" os ss ct r
   (rec3 "int" os ss n (m . r) . ts)                       lt (b ... n)
   (member?? m types                                       (ty-sugar m n os
       (rec3 "int" os ss ct r                                 (redefine-thyself* os [] ss
         (ty-sugar m n os                                        recursive-bindings . ts)))
            (redefine-thyself* os [] ss                  (rec4 "int" os ss ct (m . r)
               recursive-bindings . ts)))                   lt (b ... n)
       (rec3 "int" os ss ct (m . r)                          (ty-sugar ct n os
         (ty-sugar ct n os                                      (redefine-thyself* os [] ss
            (redefine-thyself* os [] ss                            recursive-bindings . ts)))))))))
               recursive-bindings . ts))))))))        ([_ "int" os ss ct (n . r) lt (b ...) . ts]
 ([_ "int" os ss ct (n . r) . ts]                     (TYPES (Λ types (member?? n types
  (TYPES (Λ types (member?? n types                    (rec4 "int" os ss n r lt (b ...) . ts)
   (rec3 "int" os ss n r . ts)                          (rec4 "int" os ss ct r
   (rec3 "int" os ss ct r                                  lt (b ... n)
     (ty-sugar ct n os                                      (ty-sugar ct n os
       (redefine-thyself* os [] ss                             (redefine-thyself* os [] ss
          recursive-bindings . ts)))))))                          recursive-bindings . ts)))))))
 ([_ "int" _ . r] (syntax-error As . r))             ([_ "int" _ . r] (syntax-error Lambda . r))
 ([_ os ss ns . body]                                ([_ os ss nvs . body]
  (rec3 "int" os ss :unknown ns . body))              (rec4 "int" os ss :unknown nvs λ () . body))
))                                                   )) ;; ... other anatype binding forms ...)
```

Figure 14: Core anatype forms (abridged)

```
;; all supported anatypes
(def-syntax [TYPES k] (k :unknown :fix :flo))
;; base-case for type
(def-syntax [type n k] (k :unknown))
;; deliver the anatype to the continuation
(def-syntax [ty-trf type typ n] (syntax-rules (n)
     ([_ n k] (k type))
     ([_ . r] (typ . r))
))
;; overload an operator for x
(def-syntax [op-trf opo opd x]
  (letrec-syntax ([op (syntax-rules .. (x)
     ([_ "internal" (c ..) x . y] (opo c .. x . y))
     ([_ "internal" (c ..) y . z] (op "internal" (c .. y) . z))
     ([_ "internal" c] (opd . c))
     ([_ . rest] (op "internal" [] . rest)))])
    (syntax-rules (op)
     ([_ . args] (op . args)))
  ))
;; base-case for ty-sugar
(def-syntax [ty-sugar . args] (syntax-error . args))
;; redefine ty-sugar in a modular fashion
(def-syntax [extend-ty-sugar clause ...]
  (def-syntax ty-sugar (let-syntax ([tysugar ty-sugar])
    (syntax-rules () clause ...
      ([_ . args] (tysugar . args))
      ))
    ))
;; Comparing symbols for syntactic equality
(def-syntax [eq?? th t kt kf]
   ((syntax-rules (th)
       ([_ th] kt)
       ([_ tt] kf)
       ) t))
;; get the n'th form from the db according to spec
(def-syntax [nth db . spec]
  (letrec-syntax ([rec (syntax-rules .. ()
                        ([_ (y .. x)] x)
                        ([_ (y .. x) 1 . r] (rec (y ..) . r))
                        )])
    (db (rec ['#?] . spec))
    ))
```

Figure 15: Anatype support macros

```
(Define (sqrt-loop fix: n start)
  (Let loop (fix:[num n]
                 [res 0]
                 [bit start])
    (cond ([zero? bit] res)
          ([>= num (+ res bit)]
           (loop (- num (+ res bit))
                 (+ bit (rsh res 1))
                 (rsh bit 2)))
          (else
           (loop num (rsh res 1) (rsh bit 2)))
  )))
```

A.2   Tangential: threading

We could stop here - the algorithm is clear enough. The abundance of parentheses still hurts the eye, however, so let us deploy our next secret weapon. Threading introduces an explicit *anaphoric* keyword (_ below) that binds successive terms that are arranged in a sequence (i.e., a thread) in 3 possible ways: syntactically (with >->), using a λ (with >=>) aka A-Normal Form (ANF) or using a MAYBE monad (using Scheme's #f as Nothing). The latter two are not used in this paper, and thus are elided from Figure 17. These definitions are trivial to derive from the given definition of >->.

   Similar to anaphoric types, threading also touches upon weak hygiene: the anaphoric keyword is bound differently and implicitly for each sub-expression. For example, calculation of the next value of sig can be described in English in the following sentence. First, take the value of sig, shift it to the left, then "or" it with a new bit depending on the bit-count of sig (refer to the beginning of the sentence which is 2 levels before this one). Alternatively, the same sentence can be expressed using threading:

```
(bit-or (lsh s 1)                           ;; s has fix:  anatype
        (if (even? (bit-count s))
            1 0))
```
$-\{\text{introduce threading}\} \rightarrow$
```
(>-> s                                       ;; s is already annotated with fix:
    (lsh _ 1)                                ;; sub-expressions inherit this anatype
    (or _ (if (even? (bit-count [Nth 2]))
              1 0)))
```

The ability to refer to the previous sub-expression (or indeed to any n'th sub-expression preceding the current one in closest thread introduced by a >->) is realised using anaphoric macros too. The binding for _ needs to be shadowed in each scope introduced by a new sub-expression, and therefore the macro-redefinition machinery (see Figure 13) introduced above for anatypes can be leveraged here as well. Figure 17 lists a representative implementation of this anaphoric construct.

A.3   Tangential: rational arithmetic

Although the code seems to be working well, with no explicit squaring/multiplication operations involved, it still can be improved. One shortcoming is that dyadic rationals are not represented as such explicitly. Including exact rationals would improve presentation of the algorithm as well as support the use of exact arithmetic.

```
(extend-ty-sugar (:fix)  ;; fixnums      (extend-ty-sugar (:flo)  ;; flonums
([_ :fix n (ty o+ o- o* o/ = < > <= >= neg   ([_ :flo n (ty o+ o- o* o/ = < > <= >= neg
  min max zero? even? odd? positive?           min max zero? even? odd? positive?
  negative? and lsh not or rsh xor length ...  negative? tofix toflo) . rs]
  tofix toflo) . rs]                         (let-syntax ([ty (ty-trf :flo ty n)]
(let-syntax ([ty (ty-trf :fix ty n)]           [o+ (op-trf +fl o+ n)] [o- (op-trf -fl o- n)]
 [o+ (op-trf +fx o+ n)] [o- (op-trf -fx o- n)]  ... other overloads ...)
 ... other overloads ...) . rs)              . rs)
))                                           ))
```

Figure 16: Arithmetic anatypes (abridged)

```
(def-syntax decimal->unary (syntax-rules ()
 ([_ k 0] (k))                ;; syntactically convert decimal
 ([_ k 1] (k 1))              ;; to unary for a reasonable
 ([_ k 2] (k 1 1)) ... )) ;; number of n, e.g., 0..10
(define-anaphora (_ __)
  (>-> rec1 (syntax-rules ()
   ([_ _ x fa] fa)
   ([_ (thy pre) slfs fa . fb]
    (let-syntax ([thy fa])
      (let-syntax-rule ([pre (k db . as)]
                       (pre (k (thy . db) . as)))
        (redefine-thyself* (thy pre) [] slfs (rec1 nrec)
          (rec1 (thy pre) slfs . fb)
          ))))))
  (Nth nrec (syntax-rules ()
   ([_ (thy pre) slfs n]
    (decimal->unary (Λ nl (nth pre . nl)) n)))
  ))
```

Figure 17: define-anaphora for (syntactic) threading

We can not present a complete module implementing rationals here (it would be boring anyway), so let us for example, take $\sqrt{\frac{n}{m}}$. This function can be computed using our implementation from section 2.1 and is quite representative for our purposes (we assume / and other basic arithmetic operators have already been overloaded for rationals).

```
(def-syntax *maxval* 1152921504606846975)       ;; (- (exp2 *maxbit*) 1)
(def-syntax *maxstart* 288230376151711744)      ;; (exp2 (- *maxbit 2))
(Define sqrt/rat
  (fn-with (apply sqrt)                          ;; chain the number tower
  | (<rat> : n m) =>
    (Let* (fix:[x (max n m)]                        ;; determine the best
            [f (/ *maxval* x)]                       ;; available precision
            [n (* n f)]                              ;; rescale n/m for
            [m (* m f)])                             ;; that precision
      (/ (sqrt-loop n (sqrt-start n))
        (sqrt-loop m (sqrt-start m))
        ))
    ))
(def-syntax sqrt sqrt/rat)                        ;; overload sqrt for rationals
```

Since rational values that we denote in this paper using `#r:n/m` reader syntax are represented by Scheme records (with numerator n and denominator m) they can be matched using an extensible pattern-matcher [12]. In case of a match failure, a default implementation of `sqrt` (specified as the first form of `fn-with` pattern-matching λ) is applied on the argument. Otherwise, both parts are re-scaled and passed to `sqrt-loop`. This ensures that the resulting rational represents the "best" rational approximation to the square root of the argument rational given maximal precision of available fixnums.

An interesting observation is that trigonometric functions such as `dyadic-cos1` only use basic operators that are exact for rationals (besides the above `sqrt`, of course). Therefore, passing an initial factor `f=(<rat> :  1 4)` gives the best rational approximation of the result automatically, with no change to the algorithm.

A.4 Tangential: composition transformers

A very useful functional-programming technique that is not streamlined in Scheme as well as it is in e.g., Haskell is currying (aka sectioning) and composition of functions. Traditionally, both currying and composition in Scheme is performed at the value-level, leading to extraneous closures even when the argument was known. We chose to perform composition at the syntax-level, possibly translating to value-level when needed. For the purposes of this paper, the following macro-based higher-order transformations suffice:

```
(def-syntax [∘ . fs]                        ;; => Value-level
  (Λ args                                    ;; => λ args
    (letrec-syntax ([compose
      (syntax-rules ()
        ([_ g] (g . args))                   ;; => (apply g args)
        ([_ f . gs]
         (f (compose . gs)))
        )])
      (compose . fs)
      )))

(def-syntax [κ . exps]
  (extract* (_) exps [                       ;; extract keyword _ in any scope
      (Λ ((x) terms)                          ;; obtain colored _ as x and terms
       (λ x  terms))                          ;; introduce a λ
        [] exps]                              ;; initial values
))
```

So for example, point-free definitions of common operators can now be given very concisely:

(Define add1 [κ + 1 _]) **(Define sub1** [κ - _ 1])
(Define double [κ * _ 2]) **(Define half** [κ / _ 2])
(Define reciprocal [κ / 1 _]) **(Define sqr** [κ * _ _])

B   Acronyms

ANF   A-Normal Form

CORDIC   Coordinate Rotation Digital Computer

CPS   Continuation Passing Style

DSL   Domain-Specific Language

FFT   Fast Fourier Transform

GCC   GNU Compiler Collection (formerly GNU C Compiler)

ML   Meta Language

MXM   Max Magnitude

R5RS   Revised[5] Report on the Algorithmic Language Scheme

RMS   Root Mean Square

SRFI   Scheme Request for Implementation