Efficient implementation of elementary functions in the medium-precision range

Fredrik Johansson

Efficient implementation of elementary functions in the medium-precision range

Fredrik Johansson *†

Abstract

We describe a new implementation of the elementary transcendental functions exp, sin, cos, log and atan for variable precision up to approximately 4096 bits. Compared to the MPFR library, we achieve a maximum speedup ranging from a factor 3 for cos to 30 for atan. Our implementation uses table-based argument reduction together with rectangular splitting to evaluate Taylor series. We collect denominators to reduce the number of divisions in the Taylor series, and avoid overhead by doing all multiprecision arithmetic using the mpn layer of the GMP library. Our implementation provides rigorous error bounds.

1 Introduction

Considerable effort has been made to optimize computation of the elementary transcendental functions in IEEE 754 double precision arithmetic (53 bits) subject to various constraints [6, 5, 10, 12, 16]. Precision beyond 53 bits is indispensable for computer algebra and is becoming increasingly important in scientific applications [1]. A number of libraries have been developed for arbitrary-precision arithmetic. The de facto standard is arguably MPFR [11], which guarantees correct rounding to any requested number of bits.

Unfortunately, there is a large performance gap between double precision and arbitrary-precision libraries. Some authors have helped bridge this gap by developing fast implementations targeting a fixed precision, such as 106, 113 or 212 bits [21, 17, 13]. However, these implementations generally do not provide rigorous error bounds (a promising approach to remedy this situation is [16]), and performance optimization in the range of several hundred bits still appears to be lacking.

The asymptotic difficulty of computing elementary functions is well understood. From several thousand bits and up, the bit-burst algorithm or the arithmetic-geometric mean algorithm coupled with Newton iteration effectively reduce the problem to integer multiplication, which has quasilinear complexity [3, 4]. Although such high precision has uses, most applications that go beyond double precision only require modest extra precision, say a few hundred bits or rarely a few thousand bits.

In this "medium-precision" range beyond double precision and up to a few thousand bits, i.e. up to perhaps a hundred words on a 32-bit or 64-bit computer, there are two principal hurdles in the way of efficiency. Firstly, the cost of $(n \times n)$-word multiplication or division grows quadratically with n, or almost quadratically if Karatsuba multiplication is used, so rather than "reducing everything to multiplication" (in the words of [19]), we want

*INRIA Bordeaux / IMB, fredrik.johansson@gmail.com

†This research was partially funded by ERC Starting Grant ANTICS 278537.

to do as little multiplying as possible. Secondly, since multiprecision arithmetic currently has to be done in software, every arithmetic operation potentially involves overhead for function calls, temporary memory allocation, and case distinctions based on signs and sizes of inputs; we want to avoid as much of this bookkeeping as possible.

In this work, we consider the five elementary functions exp, sin, cos, log, atan of a real variable, to which all other real and complex elementary functions can be delegated via simple algebraic transformations. Our implementation of these functions has been included in the open source arbitrary-precision software [14].

Our algorithm for all five functions follows the well-known strategy of argument reduction based on functional equations and lookup tables as described in section 3, followed by Taylor series evaluation. To keep overhead at a minimum, we perform all arithmetic using the low-level mpn layer of the GMP library [8], as outlined in section 2. To optimize the Taylor series evaluation, we use an improved version of Smith's rectangular splitting algorithm [18] in which we avoid most divisions. This improved algorithm, described in section 4, is the main contribution of the paper.

Our benchmark results in section 5 show a significant speedup compared to the current version (3.1.2) of MPFR. MPFR uses several different algorithms depending on the precision and function [20], including Smith's algorithm in some cases. Our improvement is in part due to our use of lookup tables (which MPFR does not use) and in part due to the optimized Taylor series evaluation and elimination of general overhead. Our different elementary functions also have similar performance to each other. Indeed, the algorithm is nearly the same for all functions, which simplifies the software design and aids proving correctness.

While our implementation allows variable precision up to a few thousand bits, it is competitive in the low end of the range with, for example, the QD library [13] which only targets 106 or 212 bits.

2 Fixed-point mpn arithmetic

We base our multiprecision arithmetic on the GMP library [8] (interchangeably the fork MPIR [9]), which is widely available and optimized for common CPU architectures. We use the mpn layer of GMP, since the mpz layer has unnecessary overhead. On the mpn level, GMP represents a multiprecision unsigned integer as an array of limbs (words). We assume that a limb is either $B = 32$ or $B = 64$ bits, holding a value between 0 and $2^B - 1$. We represent an approximate real number in fixed-point format with Bn-bit precision using n fractional limbs and zero or more integral limbs. An n-limb array thus represents a value in the range $[0, 1 - \text{ulp}]$, and an $(n+1)$-limb array represents a value in the range $[0, 2^B - \text{ulp}]$ where $\text{ulp} = 2^{-Bn}$.

The most important GMP functions for fixed-point arithmetic are shown in Table 1, where X, Y, Z denote fixed-point numbers with the same number of limbs and c denotes a single-limb unsigned integer. Since the first five functions return carry-out or borrow, we can also use them when X has one more limb than Y.

Our rationale for using fixed-point arithmetic instead of floating-point arithmetic is that it allows us to add and subtract numbers without any shifts or other adjustments, and without rounding error. Indeed, the first five GMP functions in Table 1 are implemented entirely in assembly code on common architectures, and we therefore try to push the work onto those primitives. Note that multiplying two n-limb fixed-point numbers involves computing the full $2n$-limb product and throwing away the n least significant limbs. In many cases, we can avoid explicitly copying the high limbs by simply moving the pointer

`mpn_add_n`	$X \leftarrow X + Y$ (or $X \leftarrow Y + Z$)
`mpn_sub_n`	$X \leftarrow X - Y$ (or $X \leftarrow Y - Z$)
`mpn_mul_1`	$X \leftarrow Y \times c$
`mpn_addmul_1`	$X \leftarrow X + Y \times c$
`mpn_submul_1`	$X \leftarrow X - Y \times c$
`mpn_mul_n + mpn_copyi`	$X \leftarrow Y \times Z$
`mpn_sqr + mpn_copyi`	$X \leftarrow Y \times Y$
`mpn_divrem_1`	$X \leftarrow Y/c$

Table 1: Fixed-point operations using GMP.

into the array.

The mpn representation does not admit negative numbers. However, we can store negative numbers implicitly using twos complement representation as long as we only add and subtract fixed-point numbers with the same number of limbs. We must then take care that the value is positive before multiplying or dividing.

3 Argument reduction

To illustrate argument reduction, consider the exponential function $\exp(x)$, where we may assume that $x \in [0, \log(2))$ after removing a multiple of $\log(2)$. Writing $\exp(x) = [\exp(x/2^r)]^{2^r}$, we reduce the argument to the range $[0, 2^{-r})$ at the expense of r multiplications, thereby improving the rate of convergence of the Taylor series. However, if we precompute $\exp(i/2^r)$ for $i = 0 \ldots 2^r - 1$, we can write $\exp(x) = \exp(x - i/2^r)\exp(i/2^r)$ where $i = \lfloor 2^r x \rfloor$. This achieves r halvings worth of argument reduction for the cost of just a single multiplication, with the drawback of having to store a lookup table. To save space, we can use a bipartite (or multipartite) table, e.g. writing $\exp(x) = \exp(x - i/2^r - j/2^{2r})\exp(i/2^r)\exp(j/2^{2r})$.

This recipe works for all elementary functions. In particular, we use the following formulas, in which $x \in [0, 1)$, $q = 2^r$, and $i = \lfloor 2^r x \rfloor$:

$$\begin{aligned}
\exp(x) &= \exp(i/q)\exp(w), & w &= x - i/q \\
\sin(x) &= \sin(i/q)\cos(w) + \cos(i/q)\sin(w), & w &= x - i/q \\
\cos(x) &= \cos(i/q)\cos(w) - \sin(i/q)\sin(w), & w &= x - i/q \\
\log(1+x) &= \log(i/q) + \log(1+w), & w &= (qx - i)/(i + q) \\
\operatorname{atan}(x) &= \operatorname{atan}(i/q) + \operatorname{atan}(w), & w &= (qx - i)/(ix + q)
\end{aligned}$$

Note that the sine and cosine are best computed simultaneously, and the argument reduction formula for the logarithm is cheaper than for the other functions, since it requires $(n \times 1)$-word operations and no $(n \times n)$-word multiplications or divisions. The advantage of using lookup tables is even greater for log and atan than for exp, sin and cos, since the "argument-halving" reduction formulas for log and atan involve square roots.

If we want p-bit precision and chain together m lookup tables worth r halvings each, the total amount of space is $mp2^r$ bits, and the number of terms in the Taylor series that we have to sum is of the order $p/(rm)$. In practice, taking r between 4 and 10 and m between 1 and 3 gives a good space-time tradeoff. At lower precision, a smaller m is better.

In our implementation, we chose the lookup table parameters shown in Table 2. For each function, we use a fast table up to 512 bits and a slower but more economical table

Function	Precision	m	r	Entries	Size (KiB)
exp	≤ 512	1	8	178	11.125
exp	≤ 4608	2	5	23+32	30.9375
sin	≤ 512	1	8	203	12.6875
sin	≤ 4608	2	5	26+32	32.625
cos	≤ 512	1	8	203	12.6875
cos	≤ 4608	2	5	26+32	32.625
log	≤ 512	2	7	128+128	16
log	≤ 4608	2	5	32+32	36
atan	≤ 512	1	8	256	16
atan	≤ 4608	2	5	32+32	36
Total					236.6875

Table 2: Size of lookup tables.

from 513 to 4608 bits, supporting function evaluation at precisions just beyond 4096 bits plus guard bits. Some of the tables have less than 2^r entries since they end near $\log(2)$ or $\pi/4$. A few more kilobytes are used to store precomputed values of $\pi/4$, $\log(2)$, and coefficients of Taylor series. The total size is around 256 KiB, which is insignificant compared to the overall space requirements of most applications, and small enough to fit in a typical L2 cache (however, a transcendental function evaluation in multiprecision arithmetic is much slower than a main memory access, so this is a minor point).

Lookup tables can be effective at even higher precision than 4096 bits, but one eventually faces diminishing returns, and we decided to use static precomputed tables up to an arbitrarily fixed limit for simplicity. The tables are tested against MPFR to verify that all entries are correctly rounded. Tables could obviously also be generated at compile time or even at runtime in anticipation of a long sequence of function evaluations.

In our implementation, the input to the elementary functions is a floating-point number with arbitrary precison. We convert the floating-point number to a fixed-point number and perform the argument reduction steps using fixed-point arithmetic. As the working precision, we add a few bits to the target precision, and round up to a multiple of the word size B. The fixed-point operations introduce a few ulps of error, for which we output a rigorous bound. The error analysis is straightforward but tedious, and we spare the reader from the details in the present paper.

4 Taylor series evaluation

We use a version of Smith's algorithm to reduce the number of quadratic operations when evaluating Taylor series [18]. The method is best explained by an example. To evaluate $\operatorname{atan}(x) \approx x \sum_{k=0}^{N-1} (-1)^k t^k/(2k+1), t = x^2$ with $N = 16$, we pick the splitting parameter $m = \sqrt{N} = 4$ and write

$$\begin{aligned}\operatorname{atan}(x)/x \approx\ & [1 - (1/3)t + (1/5)t^2 - (1/7)t^3] \\ & + [1/9 - (1/11)t + (1/13)t^2 - (1/15)t^3]\, t^4 \\ & + [1/17 - (1/19)t + (1/21)t^2 - (1/23)t^3]\, t^8 \\ & + [1/25 - (1/27)t + (1/29)t^2 - (1/31)t^3]\, t^{12}.\end{aligned}$$

Reusing the powers $t^2, \ldots, t^m$ for each row, we only need $2\sqrt{N}$ full $(n \times n)$-limb multiplications, plus $O(N)$ "scalar" operations, i.e. additions and $(n \times 1)$-limb divisions. This

"rectangular" splitting arrangement of the terms is actually a transposition of Smith's "modular" algorithm, and appears to be superior since Horner's rule can be used for the outer polynomial evaluation with respect to t^m (see [4]).

In practice, a drawback of Smith's algorithm is that an $(n \times 1)$ division has high overhead compared to an $(n \times 1)$ multiplication, or even an $(n \times n)$ multiplication if n is very small. In [15], a different rectangular splitting algorithm was proposed that uses $(n \times O(\sqrt{N}))$-limb multiplications instead of scalar divisions, and also works in the more general setting of holonomic functions. Initial experiments done by the author suggest that the method of [15] can be more efficient at modest precision. However, we found that another variation turns out to be superior for the Taylor series of the elementary functions. In this variation, we simply collect several consecutive denominators in a single word, replacing most $(n \times 1)$-word divisions by cheaper $(n \times 1)$-word multiplications.

In our implementation, we precompute two tables of integers $u[k]$ and $v[k]$ such that $1/(2k+1) = u[k]/v[k]$ and such that $v[k] < 2^B$ is the least common multiple of $2i-1$ for several consecutive i near k. To generate the table, we iterate upwards from $k = 0$, and pick the longest possible sequence of terms on a common denominator without overflowing a limb, starting a new subsequence from each point where an overflow occurs. This does not necessarily give the least possible number of distinct denominators, but it is close to optimal. The k such that $v[k] \neq v[k+1]$ are

$$12, 18, 24, 29, 34, \ldots, 226, 229, 232, \ldots \text{ (32-bit)}$$

$$23, 35, 46, 56, 67, \ldots, 225, 232, 239, \ldots \text{ (64-bit)}$$

In the supported range, we need at most one division every three terms (32-bit) or every seven terms (64-bit), and even less than this for very small N.

We compute the sum backwards. Suppose that the current partial sum is $S/v[k+1]$. To add $u[k]/v[k]$ when $v[k] \neq v[k+1]$, we first change denominators by computing $S \leftarrow (S \times v[k+1])/v[k]$. This requires one $((n+1) \times 1)$ multiplication and one $((n+2) \times 1)$ division. A complication arises if S is a twos complemented negative value when we change denominators, however in this case we can just "add and subtract 1", i.e. compute $((S + v[k+1]) \times v[k])/v[k+1] - v[k]$ which costs only two extra single-limb additions.

Pseudocode for our implementation of the atan Taylor series is shown in Algorithm 1. All uppercase variables denote fixed-point numbers, and all lowercase variables denote integers. We write $+\varepsilon$ to signify a fixed-point operation that adds up to 1 ulp of rounding error. All other operations are exact.

Our proof that Algorithm 1 is correct is based on a short exhaustive computation. This is simply a matter of executing the algorithm symbolically for all allowed values of N. In each step of the execution, we determine an upper bound for the possible value of each fixed-point variable as well as its error, proving that no overflow is possible (note that S may wraparound on lines 19 and 23 since we use twos complement arithmetic for negative values, and part of the proof is to verify that $0 \leq |S| \leq 2^B - \text{ulp}$ necessarily holds before executing lines 14, 21, 24). The computation proves that the error must be bounded by 2 ulp at the end of the algorithm.

It is not hard to see heuristically why the 2 ulp bound holds. Since the sum is kept multiplied by a denominator which is close to a full limb, we always have close to a full limb worth of guard bits. Moreover, each multiplication by a power of X removes most of the accumulated error since $X \ll 1$. At the same time, the numerators and denominators are never so close to $2^B - 1$ that overflow is possible. We stress that the proof depends on the particular content of the tables u and v.

Algorithm 1 Evaluation of the atan Taylor series

Input: $0 \le X \le 2^{-4}$ as an n-limb fixed-point number, $2 < N < 256$

Output: $S \approx \sum_{k=0}^{N-1} (-1)^k X^{2k+1}/(2k+1)$ as an n-limb fixed-point number with at most 2 ulp error

$m \leftarrow 2$
while $m^2 < N$ **do**
    $m \leftarrow m + 2$
$T[1] \leftarrow X \times X + \varepsilon$ ▷ Compute powers of X, n limbs each
$T[2] \leftarrow T[1] \times T[1] + \varepsilon$
for $(k = 4;\ k \le m;\ k \leftarrow k + 2)$ **do**
    $T[k-1] \leftarrow T[k/2] \times T[k/2-1] + \varepsilon$
    $T[k] \leftarrow T[k/2] \times T[k/2] + \varepsilon$
$S \leftarrow 0$ ▷ Fixed-point sum, with $n+1$ limbs
for $(k = N-1;\ k \ge 0;\ k \leftarrow k - 1)$ **do**
    if $v[k] \ne v[k+1]$ **and** $k < N-1$ **then** ▷ Change denominators
        if k is even **then**
            $S \leftarrow S + v[k+1]$ ▷ Single-limb addition
        $S \leftarrow S \times v[k]$ ▷ S temporarily has $n+2$ limbs
        $S \leftarrow S/v[k+1] + \varepsilon$ ▷ S has $n+1$ limbs again
        if k is even **then**
            $S \leftarrow S - v[k]$ ▷ Single-limb addition
    if $k \bmod m = 0$ **then**
        $S \leftarrow S + (-1)^k u[k]$ ▷ Single-limb addition
        if $k \ne 0$ **then**
            $S \leftarrow S \times T[m] + \varepsilon$ ▷ $((n+1) \times n)$-limb multiplication
    else
        $S \leftarrow S + (-1)^k u[k] \times T[k \bmod m]$ ▷ Fused addmul of n into $n+1$ limbs
$S \leftarrow S/v[0] + \varepsilon$
$S \leftarrow S \times X + \varepsilon$
return S ▷ Only the bottom n limbs

The algorithms for the Taylor series of log, exp, sin and cos follow the same outline as Algorithm 1. We do not evaluate log directly, but instead write $\log(1+x) = 2\,\mathrm{atanh}(x/(x+2))$, since the Taylor series for atanh only has half as many nonzero terms. To evaluate $S = \sum_{k=0}^{N-1} X^{2k+1}/(2k+1)$, we simply replace the subtractions with additions in Algorithm 1 and skip lines 13 and 17.

The algorithm for the exponential series $S = \sum_{k=0}^{N-1} X^k/k!$ is almost the same as the algorithm for atanh, but with different tables u and v. These tables give $u[k]/v[k] = 1/k!$ for $k! < 2^B - 1$. For larger k, $u[k]/v[k]$ equals $1/k!$ times the product of all $v[i]$ with $i < k$ and distinct from $v[k]$. Denominator changes occur at

$$12, 19, 26, 32, 38, \ldots, 264, 267, 270, \ldots \text{ (32-bit)}$$

$$20, 33, 45, 56, 66, \ldots, 266, 273, 280, \ldots \text{ (64-bit)}$$

For exp, Algorithm 1 is modified by skipping line 14 (in the following line, the division has one less limb). The only remaining changes are that line 25 is removed, line 4 becomes $T[1] \leftarrow X$, and the output has $n+1$ limbs instead of n limbs.

The algorithm for the sine and cosine series $S_1 = \sum_{k=0}^{N-1} (-1)^k X^{2k+1}/(2k+1)!$ and $S_2 = \sum_{k=0}^{N-1} (-1)^k X^{2k}/(2k)!$ is also very similar. We use the same table of numerators and denominators as for the exponential, and skip line 14. As in the atan series, the table of powers starts with the square of X, and we multiply the sine by X in the end. The alternating signs are handled the same way as for atan, except that line 17 becomes $S \leftarrow S - 1$. To compute sin and cos simultaneously, we execute the main loop of the algorithm twice: once for the sine (odd-index coefficients) and once for the cosine (even-index coefficients), recycling the table T.

When computing sin and cos above circa 300 bits and exp above circa 800 bits, we optimize by just evaluating the Taylor series for sin or sinh, after which we use $\cos(x) = \sqrt{1 - [\sin(x)]^2}$ or $\exp(x) = \sinh(x) + \sqrt{1 + [\sinh(x)]^2}$. This removes half of the Taylor series terms, but it is not worthwhile at lower precision due to the square root. The cosine is computed from the sine and not vice versa to avoid the ill-conditioning of the square root near 0.

5 Benchmarks

Table 3 shows benchmark results done on an Intel i7-2600S CPU running x86_64 Linux. Our code is built against MPIR 2.6.0. The input to each function is a floating-point number close to $\sqrt{2} + 1$. We include timings for the double-precision functions provided by the default libm installed on the same system (EGLIBC 2.15). Table 4 shows the speedup compared to MPFR 3.1.2 at each level of precision.

Table 5 provides a comparison at IEEE 754 quadruple (113-bit) precision against MPFR and the libquadmath included with GCC 4.6.4. We include timings for the comparable double-double ("dd", 106-bit) functions provided by version 2.3.15 of the QD library [13]. Table 5 also compares performance at quad-double ("qd", 212-bit) precision against MPFR and QD. The timings in Table 5 were obtained on a slower CPU than the timings in Table 3, which we used due to the GCC version installed on the faster system being too old to ship with libquadmath.

At low precision, a function evaluation with our implementation takes less than half a microsecond, and we come within an order of magnitude of the default libm at 53-bit precision. Our implementation holds up well around 100-200 bits of precision, even compared to a library specifically designed for this range (QD).

Bits	exp	sin	cos	log	atan
53	0.045	0.056	0.058	0.061	0.072
32	0.26	0.35	0.35	0.21	0.20
53	0.27	0.39	0.38	0.26	0.30
64	0.33	0.47	0.47	0.30	0.34
128	0.48	0.59	0.59	0.42	0.47
256	0.83	1.05	1.08	0.66	0.73
512	2.06	2.88	2.76	1.69	2.20
1024	6.79	7.92	7.84	5.84	6.97
2048	22.70	25.50	25.60	22.80	25.90
4096	82.90	97.00	98.00	99.00	104.00

Table 3: Timings of our implementation in microseconds. Top row: time of libm.

Bits	exp	sin	cos	log	atan
32	7.9	8.2	3.6	11.8	29.7
53	9.1	8.2	3.9	10.9	25.9
64	7.6	6.9	3.2	9.3	23.7
128	6.9	6.9	3.6	10.4	30.6
256	5.6	5.4	2.9	10.7	31.3
512	3.7	3.2	2.1	6.9	14.5
1024	2.7	2.2	1.8	3.6	8.8
2048	1.9	1.6	1.4	2.0	4.9
4096	1.7	1.5	1.3	1.3	3.1

Table 4: Speedup ratio of our implementation compared to MPFR 3.1.2.

	exp	sin	cos	log	atan
MPFR	5.76	7.29	3.42	8.01	21.30
libquadmath	4.51	4.71	4.57	5.39	4.32
QD (dd)	0.73	0.69	0.69	0.82	1.08
Our work	0.65	0.81	0.79	0.61	0.68
MPFR	7.87	9.23	5.06	12.60	33.00
QD (qd)	6.09	5.77	5.76	20.10	24.90
Our work	1.29	1.49	1.49	1.26	1.23

Table 5: Top rows: timings in microseconds for quadruple (113-bit) precision, except QD which gives 106-bit precision. Bottom rows: timings for quad-double (212-bit) precision. Measured on an Intel T4400 CPU.

Our implementation is consistently faster than MPFR. The speedup is smallest for cos, which is explained by the fact that argument reduction without a lookup table for cos is relatively cheap, and that MPFR does not have to evaluate the Taylor series for both sin and cos. The speedup is largest for atan, which is explained by the fact that MPFR only implements the bit-burst algorithm for this function, which is ideal only for very high precision. Beyond 4096 bits, the asymptotically fast algorithms implemented in MPFR start to become competitive for all functions, which makes the idea of using much larger lookup tables to cover even higher precision somewhat less attractive.

MPFR guarantees that the result is correctly rounded. Our implementation uses a few guard bits to compute an approximation which typically has around 1 ulp of error,

and outputs a separate floating-point number bounding the error. The default libm, libquadmath, and QD do not guarantee a bound on the error.

6 Future improvements

Our work helps reduce the performance gap between double precision and multiple precision. Nonetheless, our approach is not optimal at precisions as low as 1-2 limbs, where rectangular splitting has no advantage over evaluating precomputed minimax polynomials with Horner's rule. This is generally what is done in libraries targeting a fixed precision.

At very low precision, GMP functions are likely inferior to inlined double-double and quad-double arithmetic or similar, especially if the floating-point operations are vectorized. Interesting alternatives designed to exploit hardware parallelism include the carry-save library used for double-precision elementary functions with correct rounding in CR-LIBM [5, 7], the recent SIMD-based multiprecision implementation described in [22], and implementations targeting GPUs [21]. We encourage further comparison of these options.

Other improvements are possible at higher precision. We do not need to compute every term to a precision of n limbs in Algorithm 1 as the contribution of term k to the final sum is small when k is large. The precision should rather be changed progressively. Moreover, instead of computing an $(n \times n)$-limb fixed-point product by multiplying exactly and throwing away the low n limbs, we could compute an approximation of the high part in about half the time (unfortunately, GMP does not currently provide a function that does this).

Our implementation of the elementary functions outputs a guaranteed error bound whose proof of correctness depends on a complete error analysis done by hand, aided by some exhaustive computations. To rule out any superficial bugs, we have tested the code by comparing millions of random values against MPFR. Due to the danger of human error, a formally verified implementation would be desirable. We believe that such a proof is feasible. The square root function in GMP is implemented at a similar level of abstraction, and it has been proved correct formally using Coq [2].

References

[1] D. H. Bailey, R. Barrio, and J. M. Borwein. High-precision computation: Mathematical physics and dynamics. *Applied Mathematics and Computation*, 218(20):10106–10121, 2012.

[2] Y. Bertot, N. Magaud, and P. Zimmermann. A proof of GMP square root. *Journal of Automated Reasoning*, 29(3-4):225–252, 2002.

[3] R. P. Brent. The complexity of multiple-precision arithmetic. *The Complexity of Computational Problem Solving*, pages 126–165, 1976.

[4] R. P. Brent and P. Zimmermann. *Modern Computer Arithmetic*. Cambridge University Press, 2011.

[5] C. Daramy, D. Defour, F. de Dinechin, and J. M. Muller. CR-LIBM: a correctly rounded elementary function library. In *Optical Science and Technology, SPIE's 48th Annual Meeting*, pages 458–464. International Society for Optics and Photonics, 2003.

[6] F. de Dinechin, D. Defour, and C. Lauter. Fast correct rounding of elementary functions in double precision using double-extended arithmetic. Research Report RR-5137, 2004.

[7] D. Defour and F. de Dinechin. Software carry-save: A case study for instruction-level parallelism. In V. E. Malyshkin, editor, *Parallel Computing Technologies*, volume 2763 of *Lecture Notes in Computer Science*, pages 207–214. Springer Berlin Heidelberg, 2003.

[8] The GMP development team. GMP: The GNU Multiple Precision Arithmetic Library. `http://gmplib.org`.

[9] The MPIR development team. MPIR: Multiple Precision Integers and Rationals. `http://www.mpir.org`.

[10] M. Dukhan and R. Vuduc. Methods for high-throughput computation of elementary functions. In *Parallel Processing and Applied Mathematics*, pages 86–95. Springer, 2014.

[11] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software*, 33(2):13:1–13:15, June 2007. `http://mpfr.org`.

[12] J. Harrison, T. Kubaska, S. Story, and P. T. P. Tang. The computation of transcendental functions on the IA-64 architecture. In *Intel Technology Journal*. Citeseer, 1999.

[13] Y. Hida, X. S. Li, and D. H. Bailey. Library for double-double and quad-double arithmetic. *NERSC Division, Lawrence Berkeley National Laboratory*, 2007. `http://crd-legacy.lbl.gov/~dhbailey/mpdist/`.

[14] F. Johansson. Arb: A C library for ball arithmetic. *ACM Communications in Computer Algebra*, 47(3/4):166–169, December 2013.

[15] F. Johansson. Evaluating parametric holonomic sequences using rectangular splitting. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, ISSAC '14, pages 256–263, New York, NY, USA, 2014. ACM.

[16] O. Kupriianova and C. Lauter. Metalibm: A mathematical functions code generator. In Hoon Hong and Chee Yap, editors, *Mathematical Software – ICMS 2014*, volume 8592 of *Lecture Notes in Computer Science*, pages 713–717. Springer Berlin Heidelberg, 2014.

[17] Y. Lei, Y. Dou, L. Shen, J. Zhou, and S. Guo. Special-purposed VLIW architecture for IEEE-754 quadruple precision elementary functions on FPGA. In *2011 IEEE 29th International Conference on Computer Design (ICCD)*, pages 219–225, Oct 2011.

[18] D. M. Smith. Efficient multiple-precision evaluation of elementary functions. *Mathematics of Computation*, 52:131–134, 1989.

[19] A. Steel. Reduce everything to multiplication. In *Computing by the Numbers: Algorithms, Precision, and Complexity*. 2006. `http://www.mathematik.hu-berlin.de/~gaggle/EVENTS/2006/BRENT60/`.

[20] The MPFR team. The MPFR library: algorithms and proofs. `http://www.mpfr.org/algo.html`. Retrieved 2013.

[21] A. Thall. Extended-precision floating-point numbers for GPU computation. In *ACM SIGGRAPH 2006 Research posters*, page 52. ACM, 2006.

[22] J. van der Hoeven, G. Lecerf, and G. Quintin. Modular SIMD arithmetic in Mathemagix. *arXiv preprint arXiv:1407.3383*, 2014.