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, 11, 13, 18]. Higher precision is indispensable for computer algebra and is becoming increasingly important in scientific applications [1]. Many libraries have been developed for arbitrary-precision arithmetic. The de facto standard is arguably MPFR [12], 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 [26, 19, 14]. However, these implementations generally do not provide rigorous error bounds (a promising approach to remedy this situation is [18]), 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 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. First, 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 [23]), we want to do as little multiplying as possible. Secondly, since multiprecision arithmetic currently

*INRIA Bordeaux
†fredrik.johansson@gmail.com

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 algebraic transformations. 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 evaluation of Taylor series. To keep overhead at a minimum, all arithmetic uses the low-level mpn layer of the GMP library [8], as outlined in section 2.

We use lookup tables in arguably the simplest possible way, storing values of the function itself on a regularly spaced grid. At high precision, a good space-time tradeoff is achieved by using bipartite tables. Several authors have studied the problem of constructing optimal designs for elementary functions in resource-constrained settings, where it is important to minimize not only the size of the tables but also the numerical error and the complexity of circuitry to implement the arithmetic operations [10], [21], [24]. We ignore such design parameters since guard bits and code size are cheap in our setting.

While implementations in double precision often use minimax or Chebyshev polynomial approximations, which require somewhat fewer terms than Taylor series for equivalent accuracy, Taylor series are superior at high precision since the evaluation can be done faster. Smith's rectangular splitting algorithm [22] allows evaluating a degree-N truncated Taylor series of suitable type using $O(\sqrt{N})$ $(n \times n)$-word multiplications whereas evaluating a degree-N minimax polynomial using Horner's rule requires $O(N)$ such multiplications.

The main contribution of the paper, described in section 4, is an improved version of Smith's rectangular splitting algorithm for evaluating Taylor series, in which we use fixed-point arithmetic efficiently and avoid most divisions. Section 5 describes the global algorithm including error analysis.

Our implementation of the elementary functions is part of version 2.4.0 of the open source arbitrary-precision interval software Arb [16]. The source code can be retrieved from [15].

Since the goal is to do interval arithmetic, we compute a rigorous bound for the numerical error. Unlike MPFR, our code does not output a correctly rounded floating-point value. This more of a difference in the interface than an inherent limitation of the algorithm, and only accounts for a small difference in performance (as explained in Section 5).

Our benchmark results in section 6 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 [25], including Smith's algorithm in some cases. The large 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 the QD library [14] which only targets 106 or 212 bits. QD uses a combination of lookup tables, argument reduction, Taylor series, and Newton iteration for inverse functions.

2 Fixed-point arithmetic

We base our multiprecision arithmetic on the GMP library [8] (or 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, a multiprecision integer is 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 a 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 $\text{ulp} = 2^{-Bn}$.

An advantage of fixed-point over floating-point arithmetic is that we can add numbers without any rounding or shift adjustments. The most important GMP functions 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.

Table 1: Fixed-point operations using GMP.

`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`	$X \leftarrow Y \times Z$
`mpn_sqr`	$X \leftarrow Y \times Y$
`mpn_divrem_1`	$X \leftarrow Y/c$

The first five GMP functions in Table 1 are usually implemented in assembly code, 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. We can often avoid explicitly copying the high limbs by simply moving the pointer into the array.

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

We compute bounds for all errors when doing fixed-point arithmetic. For example, if X and Y are fixed-point numbers with respective errors ε_1, ε_2, then their sum has error bounded by $|\varepsilon_1| + |\varepsilon_2|$, and their product, rounded to a fixed-point number using a single truncation, has error bounded by

$$|Y||\varepsilon_1| + |X||\varepsilon_2| + |\varepsilon_1\varepsilon_2| + (1 \text{ ulp}).$$

If c is an exact integer, then the product $X \times c$ has error bounded by $|\varepsilon_1||c|$, and the quotient X/c, rounded to a fixed-point number using a single truncation, has error bounded by $|\varepsilon_1|/|c| + (1 \text{ ulp})$. Similar bounds are used for other operations that arise in the implementation.

In parts of the code, we use a single-limb variable to track a running error bound measured in ulps, instead of determining a formula that bounds the cumulative error in advance. This is convenient, and cheap compared to the actual work done in the multiprecision arithmetic operations.

3 Argument reduction

The standard method to evaluate elementary functions begins with one or several argument reductions to restrict the input to a small standard domain. The function is then computed on the standard domain, typically using a polynomial approximation such as a truncated Taylor series, and the argument reduction steps are inverted to recover the function value [4], [20].

As an example, consider the exponential function $\exp(x)$. Setting $m = \lfloor x/\log(2) \rfloor$ and $t = x - m\log(2)$, we reduce the problem to computing $\exp(x) = \exp(t)2^m$ where t lies in the standard domain $[0, \log(2))$. Writing $\exp(t) = [\exp(t/2^r)]^{2^r}$, we can further reduce the argument to the range $[0, 2^{-r})$ at the expense of r squarings, thereby improving the rate of convergence of the Taylor series. Analogously, we can reduce to the intervals $[0, \pi/4)$ for sin and cos, $[0, 1)$ for atan, and $[1, 2)$ for log, and follow up with r further transformations to reduce the argument to an interval of width 2^{-r}.

This strategy does not require precomputations (except perhaps for the constants π and $\log(2)$), and is commonly used in arbitrary-precision libraries such as MPFR [25].

The argument reduction steps can be accelerated using lookup tables. 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. 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. We use the following formulas, in which $x \in [0, 1)$, $q = 2^r$, $i = \lfloor 2^r x \rfloor$, $t = i/q$, $w = x - i/q$, $w_1 = (qx - i)/(i + q)$, and $w_2 = (qx - i)/(ix + q)$:

$$\begin{aligned}
\exp(x) &= \exp(t)\exp(w) \\
\sin(x) &= \sin(t)\cos(w) + \cos(t)\sin(w) \\
\cos(x) &= \cos(t)\cos(w) - \sin(t)\sin(w) \\
\log(1 + x) &= \log(1 + t) + \log(1 + w_1) \\
\operatorname{atan}(x) &= \operatorname{atan}(t) + \operatorname{atan}(w_2)
\end{aligned}$$

The sine and cosine are best computed simultaneously. 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 greater for log and atan than for exp, sin and cos, since the "argument-halving" 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 $p/(rm)$. 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.

Our implementation uses the table parameters shown in Table 2. For each function, we use a fast table up to 512 bits and a more economical table 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 parameters in Table 2 were chosen based on experiment to give good performance at all precisions while keeping the total size (less than 256 KiB) insignificant compared to the overall space requirements of most applications and small enough to fit in a typical L2

Table 2: Size of lookup tables.

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

cache. For simplicity, our code uses static precomputed tables, which are tested against MPFR to verify that all entries are correctly rounded.

The restriction to 4096-bit and lower precision is done since lookup tables give diminishing returns at higher precision compared to asymptotically fast algorithms that avoid precomputations entirely. In a software implementation, there is no practical upper limit to the size of lookup tables that can be used. One could gain efficiency by using auxiliary code to dynamically generate tables that are optimal for a given application.

4 Taylor series evaluation

After argument reduction, we need to evaluate a truncated Taylor series, where we are given a fixed-point argument $0 \leq X \ll 1$ and the number of terms N to add. In this section, we present an algorithm that solves the problem efficiently, with a bound for the rounding error. The initial argument reduction restricts the possible range of N, which simplifies the analysis. Indeed, for an internal precision of $p \leq 4608$ bits and the parameters of Table 2, $N < 300$ always suffices.

We use a version of Smith's algorithm to avoid expensive multiplications [22]. The method is best explained by an example. To evaluate

$$\operatorname{atan}(x) \approx x \sum_{k=0}^{N-1} \frac{(-1)^k t^k}{2k+1}, \quad t = x^2$$

with $N = 16$, we pick the splitting parameter $m = \sqrt{N} = 4$ and write $\operatorname{atan}(x)/x \approx$

$$\begin{aligned} & [1 - \tfrac{1}{3}t + \tfrac{1}{5}t^2 - \tfrac{1}{7}t^3] \\ + & [\tfrac{1}{9} - \tfrac{1}{11}t + \tfrac{1}{13}t^2 - \tfrac{1}{15}t^3] \quad t^4 \\ + & [\tfrac{1}{17} - \tfrac{1}{19}t + \tfrac{1}{21}t^2 - \tfrac{1}{23}t^3] \quad t^8 \\ + & [\tfrac{1}{25} - \tfrac{1}{27}t + \tfrac{1}{29}t^2 - \tfrac{1}{31}t^3] \quad t^{12}. \end{aligned}$$

Since the powers $t^2, \ldots, t^m$ can be recycled 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]).

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 [17], 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 [17] 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, namely to simply collect several consecutive denominators in a single word, replacing most $(n \times 1)$-word divisions by cheaper $(n \times 1)$-word multiplications.

We precompute tables of integers $u_k, v_k < 2^B$ such that $1/(2k+1) = u_k/v_k$ and v_k is the least common multiple of $2i - 1$ for several consecutive i near k. To generate the table, we iterate upwards from $k = 0$, picking the longest possible sequence of terms on a common denominator without overflowing a limb, starting a new subsequence from each point where overflow occurs. This does not necessarily give the least possible number of distinct denominators, but it is close to optimal (on average, v_k is 28 bits wide on a 32-bit system and 61 bits wide on a 64-bit system for $k < 300$). The k such that $v_k \neq v_{k+1}$ are

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

and

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

In the supported range, we need at most one division every three terms (32-bit) or every seven terms (64-bit), and 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 two's 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.

Algorithm 1 can be shown to be correct by a short exhaustive computation. We execute the algorithm symbolically for all allowed values of N. In each step, 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 17 and 21 since we use two's 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 12, 19, 22). The computation proves that the error is bounded by 2 ulp at the end.

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, $2 < N < 300$

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

$m \leftarrow 2\lceil \sqrt{N}/2 \rceil$

$T_1 \leftarrow X \times X + \varepsilon$ ▷ 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 n limbs

Code to generate coefficients and prove correctness of Algorithm 1 (and its variants for the other functions) is included in the source repository [15] in the form of a Python script `verify_taylor.py`.

Making small changes to Algorithm 1 allows us to compute log, exp, sin and cos. For log, we write $\log(1+x) = 2\,\mathrm{atanh}(x/(x+2))$, since the Taylor series for atanh has half as many nonzero terms. To sum $S = \sum_{k=0}^{N-1} X^{2k+1}/(2k+1)$, we simply replace the subtractions with additions in Algorithm 1 and skip lines 11 and 15.

For the exp series $S = \sum_{k=0}^{N-1} X^k/k!$, we use different tables u and v. For $k! < 2^B - 1$, $u_k/v_k = 1/k!$ and 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. The k such that $v_k \neq v_{k+1}$ are

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

and

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

Algorithm 1 is modified by skipping line 12 (in the next line, the division has one less limb). The remaining changes are that line 23 is removed, line 2 becomes $T_1 \leftarrow X$, and the output has $n+1$ limbs instead of n limbs.

For the sine and cosine $S_1 = \sum_{k=0}^{N-1}(-1)^k X^{2k+1}/(2k+1)!$ and $S_2 = \sum_{k=0}^{N-1}(-1)^k X^{2k}/(2k)!$, we use the same u_k, v_k as for exp, and skip line 12. 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 15 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 only saves time at high 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 Top-level algorithm and error bounds

Our input to an elementary function f is an arbitrary-precision floating-point number x and a precision $p \geq 2$. We output a pair of floating-point numbers (y, z) such that $f(x) \in [y - z, y + z]$. The intermediate calculations use fixed-point arithmetic. Naturally, floating-point manipulations are used for extremely large or small input or output. For example, the evaluation of $\exp(x) = \exp(t)2^m$, where m is chosen so that $t = x - m\log(2) \in [0, \log(2))$, uses fixed-point arithmetic to approximate $\exp(t) \in [1, 2)$. The final output is scaled by 2^m after converting it to floating-point form.

Algorithm 2 gives pseudocode for $\mathrm{atan}(x)$, with minor simplifications compared to the actual implementation. In reality, the quantities (y, z) are not returned exactly as printed; upon returning, y is rounded to a p-bit floating-point number and the rounding error of this operation is added to z which itself is rounded up to a low-precision floating-point number.

The variables X, Y are fixed-point numbers and Z is an error bound measured in ulps. We write $+\varepsilon$ to indicate that a result is truncated to an n-limb fixed-point number, adding at most 1 ulp $= 2^{-Bn}$ error where $B = 32$ or 64.

After taking care of special cases, $|x|$ or $1/|x|$ is rounded to a fixed-point number $0 \le X < 1$. Up to two argument transformations are then applied to X. The first ensures $0 \le X < 2^{-r_1}$ and the second ensures $0 \le X < 2^{-r_1-r_2}$. After line 21, we have (if $|x| < 1$)

$$|\operatorname{atan}(x)| = \operatorname{atan}\Big(\frac{p_1}{2^{r_1}}\Big) + \operatorname{atan}\Big(\frac{p_2}{2^{r_1+r_2}}\Big) + \operatorname{atan}(X) + \delta$$

or (if $|x| > 1$)

$$|\operatorname{atan}(x)| = \frac{\pi}{2} - \operatorname{atan}\Big(\frac{p_1}{2^{r_1}}\Big) - \operatorname{atan}\Big(\frac{p_2}{2^{r_1+r_2}}\Big) - \operatorname{atan}(X) + \delta$$

for some $|\delta| \le Z$. The bound on δ is easily proved by repeated application of the fact that $|\operatorname{atan}(t+\varepsilon) - \operatorname{atan}(t)| \le |\varepsilon|$ for all $t, \varepsilon \in \mathbb{R}$.

The value of $\operatorname{atan}(X)$ is approximated using a Taylor series. By counting leading zero bits in X, we find the optimal integer r with $r_1 + r_2 \le r \le Bn$ such that $X < 2^{-r}$ (we could take $r = r_1 + r_2$, but choosing r optimally is better when x is tiny). The tail of the Taylor series satisfies

$$\left|\operatorname{atan}(X) - \sum_{k=0}^{N-1} \frac{(-1)^k}{2k+1} X^{2k+1}\right| \le X^{2N+1},$$

and we choose N such that $X^{2N+1} < 2^{-r(2N+1)} \le 2^{-w}$ where w is the working precision in bits.

Values of $\operatorname{atan}(p_1 2^{-r_1})$, $\operatorname{atan}(p_2 2^{-r_1-r_2})$ and $\pi/2$ are finally read from tables with at most 1 ulp error each, and all terms are added. The output error bound z is the sum of the Taylor series truncation error bound and the bounds for all fixed-point rounding errors. It is clear that $z \le 10 \times 2^{-w}$ where the w is the working precision in bits, and that the choice of w implies that y is accurate to p bits. The working precision has to be increased for small input, but the algorithm never slows down significantly since very small input results in only a few terms of the Taylor series being necessary.

The code for exp, log, sin and cos implements the respective argument reduction formulas analogously. We do not reproduce the calculations here due to space constraints. The reader may refer to the source code [15] for details.

Our software [16] chooses guard bits to achieve p-bit relative accuracy with at most 1-2 ulp error in general, but does not guarantee correct rounding, and allows the output to have less accuracy in special cases. In particular, sin and cos are computed to an absolute (not relative) tolerance of 2^{-p} for large input, and thus lose accuracy near the roots. These are reasonable compromises for variable-precision interval arithmetic, where we only require a correct enclosure of the result and have the option to restart with higher precision if the output is unsatisfactory.

Correct rounding (or any other strict precision policy) can be achieved with Ziv's strategy: if the output interval $[y - z, y + z]$ does not allow determining the correctly rounded p-bit floating-point approximation, the computation is restarted with more guard bits. Instead of starting with, say, 4 guard bits to compensate for internal rounding error in the algorithm, we might start with $4 + 10$ guard bits for a 2^{-10} probability of having to restart. On average, this only results in a slight increase in running time, although worst cases necessarily become much slower.

6 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. All measurements were obtained by evaluating the

Algorithm 2 Top-level algorithm for atan

Input: $x \notin \{0, \pm\infty, \text{NaN}\}$ with sign σ and exponent e such that $2^{e-1} \leq \sigma x < 2^e$, and a precision $p \geq 2$

Output: A pair (y, z) such that $\text{atan}(x) \in [y - z, y + z]$

if $e < -p/2 - 2$ **then**
  return $(x, \pm 2^{3e})$ ▷ $\text{atan}(x) = x + O(x^3)$
if $e > p + 2$ **then**
  return $(\sigma\pi/2, 2^{1-e})$ ▷ $\text{atan}(x) = \pm\pi/2 + O(1/x)$
if $|x| = 1$ **then**
  return $(\sigma\pi/4, 0)$
$w \leftarrow p - \min(0, e) + 4$ ▷ Working precision in bits
if $w > 4608$ **then**
  return Enclosure for $\text{atan}(x)$ using fallback algorithm
$n \leftarrow \lceil w/B \rceil$ ▷ Working precision in limbs
if $|x < 1$ **then**
  $X \leftarrow |x| + \varepsilon$, $Z \leftarrow 1$
else
  $X \leftarrow 1/|x| + \varepsilon$, $Z \leftarrow 1$
If $w \leq 512$ **then** $(r_1, r_2) \leftarrow (8, 0)$ **else** $(r_1, r_2) \leftarrow (5, 5)$
$p_1 \leftarrow \lfloor 2^{r_1} X \rfloor$ ▷ First argument reduction
if $p_1 \neq 0$ **then**
  $X \leftarrow (2^{r_1} X - p_1)/(2^{r_1} + p_1 X) + \varepsilon, Z \leftarrow Z + 1$
$p_2 \leftarrow \lfloor 2^{r_2} X \rfloor$ ▷ Second argument reduction
if $p_2 \neq 0$ **then**
  $X \leftarrow (2^{r_1+r_2} X - p_2)/(2^{r_1+r_2} + p_2 X) + \varepsilon, Z \leftarrow Z + 1$
Compute $r_1 + r_2 \leq r \leq Bn$ such that $0 \leq X < 2^{-r}$
$N \leftarrow \lceil (w - r)/(2r) \rceil$
if $N \leq 2$ **then**
  $Y \leftarrow \sum_{k=0}^{N-1} \frac{(-1)^k}{2k+1} X^{2k+1} + 3\varepsilon$ ▷ Direct evaluation
  $Z \leftarrow Z + 3$
else
  $Y \leftarrow \sum_{k=0}^{N-1} \frac{(-1)^k}{2k+1} X^{2k+1} + 2\varepsilon$ ▷ Call Algorithm 1
  $Z \leftarrow Z + 2$
if $p_1 \neq 0$ **then** ▷ First table lookup
  $Y \leftarrow Y + (\text{atan}(p_1 2^{-r_1}) + \varepsilon),\ Z \leftarrow Z + 1$
if $p_2 \neq 0$ **then** ▷ Second table lookup
  $Y \leftarrow Y + (\text{atan}(p_2 2^{-r_1 - r_2}) + \varepsilon),\ Z \leftarrow Z + 1$
if $x > 1$ **then**
  $Y \leftarrow (\pi/2 + \varepsilon) - Y, Z \leftarrow Z + 1$
return $(\sigma Y,\ 2^{-r(2N+1)} + Z 2^{-Bn})$

function in a loop running for at least 0.1 s and taking the best average time out of three such runs.

The input to each function is a floating-point number close to $\sqrt{2}+1$, which is representative for our implementation since it involves the slowest argument reduction path in all functions for moderate input (for input larger than about 2^{64}, exp, sin and cos become marginally slower since higher precision has to be used for accurate division by log(2) or $\pi/4$).

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

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 4: Speedup vs MPFR 3.1.2.

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 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 [14]. 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).

Our implementation is consistently faster than MPFR. The smallest speedup is achieved

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.

	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

for the cos function, as the argument reduction without table lookup is relatively efficient and MPFR does not have to evaluate the Taylor series for both sin and cos. The speedup is largest for atan, since 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, making the idea of using larger lookup tables to cover even higher precision somewhat less attractive.

Differences in accuracy should be considered when benchmarking numerical software. The default libm, libquadmath, and QD do not provide error bounds. MPFR provides the strongest guarantees (correct rounding). Our implementation provides rigorous error bounds, but allows the output to be less precise than correctly rounded. The 20% worse speed at 64-bit precision compared to 53-bit precision gives an indication of the overhead that would be introduced by providing correct rounding (at higher precision, this factor would be smaller).

7 Future improvements

Our work helps reduce the performance gap between double 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 minimax polynomials with Horner's rule, as is generally 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 code [27], and implementations targeting GPUs [26]. 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 such a function).

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. We also test the code against itself for millions of random inputs by comparing the output at different levels of precisions or at different points connected by a functional equation. Random inputs are generated non-uniformly to increase the chance of hitting corner cases. The functions are also tested indirectly by being used internally in many higher transcendental functions.

Nevertheless, since testing cannot completely rule out 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].

Acknowledgments

This research was partially funded by ERC Starting Grant ANTICS 278537. The author thanks the anonymous referees for valuable feedback.

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] F. De Dinechin and A. Tisserand. Multipartite table methods. *IEEE Transactions on Computers*, 54(3):319–330, 2005.

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

[12] 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`.

[13] 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.

[14] 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/`.

[15] F. Johansson. Arb, version 2.4.0 or later (git repository). `https://github.com/fredrik-johansson/arb`.

[16] F. Johansson. Arb: A C library for ball arithmetic. *ACM Communications in Computer Algebra*, 47(3/4):166–169, January 2014.

[17] 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.

[18] 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.

[19] 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.

[20] J. M. Muller. *Elementary functions: algorithms and implementation*. Springer Science & Business Media, 2006.

[21] M. J. Schulte and J. E. Stine. Approximating elementary functions with symmetric bipartite tables. *IEEE Transactions on Computers*, 48(8):842–847, 1999.

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

[23] 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/`.

[24] J. E. Stine and M. J. Schulte. The symmetric table addition method for accurate function approximation. *Journal of VLSI signal processing systems for signal, image and video technology*, 21(2):167–177, 1999.

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

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

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