# Fast Table-Driven Algorithms for Interval Elementary Functions

Douglas M. Priest
Sun Microsystems, Inc.
Mountain View, California
douglas.priest@eng.sun.com

## Abstract

*We present table-driven algorithms for computing interval bounds on several common elementary functions. Our algorithms use directed rounding to obtain sharp bounds—within 1.5 units in the last place of the exact range of the function over the argument interval—without the explicit use of extended precision. Moreover, by performing all floating point operations in the same rounding mode, our algorithms can exploit software pipelining to provide better performance than simply evaluating the corresponding point elementary function at each endpoint of the argument interval and rounding.*

## 1. Introduction

The IEEE 754-1985 standard[4] specifies two directed rounding modes, round-to-negative-infinity and round-to-positive-infinity, that are intended to support interval arithmetic. Indeed, the basic interval operations are relatively easy to implement using these modes (in the absence of certain exceptions), and since IEEE 754 arithmetic is now nearly ubiquitous, we might expect to see interval arithmetic supported more widely. Such support must include an elementary function library capable of delivering correct and preferably sharp bounds on the ranges of various standard functions over real intervals.

There are several viable approaches to computing interval elementary functions. One could simply reimplement a point elementary function algorithm using interval arithmetic with appropriate error terms to accommodate the approximation errors, but that would typically produce much wider results than the best possible. One alternative, used by Kearfott, *et al*, in the portable version of their INTLIB library[6], uses interval arithmetic to compute interval bounds on the point function values at the ends of the argument interval. This approach generally produces tighter bounds than evaluation over the entire argument interval, but the bounds are still appreciably wider than the best possible—on the order of a hundred units in the last place for a degenerate argument—and the computation of two-sided bounds on the function values at the endpoints wastes some computing effort.

Knüppel's BIAS library[7] uses the point function itself to compute function values at the ends of the argument interval; it then perturbs the computed values to account for the error in the point function. As a result, the BIAS elementary functions are reasonably fast, usually costing only moderately more than twice the cost of the corresponding point function, and they can deliver results that are only a few units in the last place wider than the best possible. By using the underlying platform's point functions, however, the BIAS functions depend for their correctness on knowing the accuracy of those point functions, and that accuracy can be difficult to ascertain when the source codes for the point functions are not available.

Another approach is reflected in the algorithms of Braune[1] and Krämer[8]. They start with point function algorithms and perform a careful error analysis to estimate the worst case error as a function of the floating point precision used. They then implement their algorithms computing intermediate results in an extended precision wider than the desired precision of the result: they compute the function value in this extended precision, perturb it by the worst case error bound, and then round the perturbed value outward to working precision. They show that if the extended precision format carries a modest number of extra bits beyond working precision, they can almost always deliver the sharpest bounds possible in working precision. (How much extra precision is needed to guarantee that the final results are *always* the best possible is an open question, although Muller and Tisserand[11] have made progress in this area.)

We have chosen to use an approach similar to that of Braune and Krämer, but we trade a little accuracy for efficiency. Our algorithms use only IEEE 754 double precision arithmetic, and we make sparing use of well-known techniques for simulating additional precision implicitly. Nevertheless, because we use carefully designed, table-driven point function algorithms as a basis, and because we rely on

1063-6889/97 $10.00 © 1997 IEEE

directed rounding modes to aid in controlling the signs of roundoff errors, our algorithms still achieve worst case errors smaller than 1.5 units in the last place. Our algorithms are also quite fast: besides avoiding extended precision, we manipulate the signs of the intermediate results so that we can perform all computations in the same rounding mode. In this way, we need only save, set, and restore the rounding direction once for each function evaluation. Also, this technique allows us to "pipeline" the computation of the upper and lower bounds for typical arguments, so we can take advantage of the pipelined floating point architectures of modern CPUs. As a result, our algorithms compute sharp interval bounds at a cost that is often no more than twice that of the corresponding point function.

In section 2, we review some tools and techniques we have found useful in the error analysis of the interval functions, which is similar to but in some ways more rigorous than the corresponding analysis for point functions. Section 3 presents algorithms for the exponential, logarithm, and arctangent functions, while section 4 describes the methods we use to perform argument reduction for the trigonometric functions and gives algorithms for sine, cosine, and tangent. Throughout, we omit the treatment of special cases such as infinite intervals and tiny arguments; for the most part, these cases are trivial to analyze and only slightly more difficult to implement. Finally, section 5 tabulates the worst observed errors for our algorithms and compares the performance of the interval functions with that of their point counterparts.

## 2. Error Analysis Tools

The primary goal of a point elementary function algorithm is to compute an approximation whose error is small but may be either positive or negative; thus, the error analysis for these algorithms centers on bounding the absolute value of the error. In contrast, the goal of an interval elementary function algorithm is to compute true bounds for the range of the function over the argument interval, so the error analysis focuses on proving that the errors in the computed interval endpoints have the correct signs. Following the typical style of analysis for point functions[13], we consider two distinct sources of error: the error in the approximation of a transcendental function by a polynomial, and the error due to roundoff in the computation.

For interval elementary function algorithms, we typically need to find a polynomial approximation $p(x)$ to a transcendental function $f(x)$ such that $p(x) \geq f(x)$, or vice versa, for all $x$ in a given range; ideally, we would also like $p$ to be a good approximation to $f$ over this range. Fortunately, this task is equivalent to the well-studied problem of finding a best uniform approximating polynomial to $f$: in any linear subspace that contains constant functions, the best uniform one-sided approximations to $f(x)$ are simply $g(x) \pm \epsilon$ where $g$ is the best overall uniform approximation to $f$ and $\epsilon$ is its worst case error. Thus we can apply the Remes algorithm to find the best uniform approximating polynomial and simply add or subtract an upper bound on the worst case error.

An estimate of the worst case error $\sup |f(x) - g(x)|$ over the range of the approximation arises as a by-product of the Remes algorithm. This value is generally only an estimate, however, for several reasons: the Remes algorithm is iterative, the values of $f$ used in the algorithm are themselves approximate (though necessarily much more accurate than the approximation being sought), and depending on the implementation, the algorithm may fail to account for the error due to rounding the coefficients of $g$ to working precision. For interval functions, we need a rigorous bound on the approximation error, so rather than rely on the value produced by the Remes algorithm, we compute a bound directly: given double precision coefficients $a_i$ of a polynomial $p(x) = \sum_{i=0}^{n} a_i x^i$, rational coefficients $b_i$ of the $m$-th degree Taylor polynomial $q(x) = \sum_{i=0}^{m} b_i x^i$ of $f(x)$ (which is always analytic in the cases of interest), a bound $c$ on the remaining coefficients of the Taylor series for $f$, and an interval $[l, r]$, we estimate the approximation error

$$\sup_{[l,r]} |f(x) - p(x)| \leq \sup_{[l,r]} |f(x) - q(x)| + \sup_{[l,r]} |q(x) - p(x)|$$

$$\leq \frac{ct^{m+1}}{1-t} + \max_{z \in Z \cup \{l,r\}} |q(z) - p(z)|$$

where $t = \max(|l|, |r|)$ and $Z$ is the set of roots of $q'(x) - p'(x)$ in the interval $[l, r]$. We use a secant algorithm employing quadruple precision rational interval arithmetic to find double precision intervals containing these roots and compute an upper bound on the second error term over each interval. Combining the largest such error with the truncation error represented by the first term, we obtain an upper bound on the total approximation error.

To ensure that the errors in the computed interval endpoints have the correct signs in the presence of roundoff, we rely on several common techniques. In many cases, we use an *a priori* analysis to estimate the contribution due to roundoff errors and simply add a term that bounds these errors, much the same way that we handle approximation error. For some steps in the computation, the error analysis is easy to carry out by inspection, but for the evaluation of polynomial approximations, we rely on a variation of a well-known technique[5] called "running" roundoff error analysis. As above, letting $a_0, \ldots, a_n$ denote the coefficients of $p$, we compute an upper bound on the roundoff error in the evaluation of $p(x)$ for any $x \in [l, r]$ by an augmented version of Horner's rule:

$t := \max(|l|, |r|)$
$p := |a_n|, e := 0$

for $i = n-1, \ldots, 0$ do
$$p := p \times t, \quad e := e \times t + \text{ulp}(p)$$
$$p := p + |a_i|, \quad e := e + \text{ulp}(p)$$

(Here ulp($x$) denotes a unit in the last place of $x$ and can be computed easily using the nextafter function recommended by IEEE 754.) Provided this computation is carried out rounding up, the final value of $e$ is a rigorous upper bound on the roundoff error.

The directed rounding modes of IEEE 754 arithmetic provide another way to control the sign of roundoff errors. We use directed rounding in two ways corresponding to two different interpretations of the effects of roundoff, namely backward error and forward error. In some cases, we perform argument reductions carefully to ensure that the computed value of the reduced argument is the exact value that would be obtained in the absence of roundoff from an original argument no less than (or greater than, depending on context) the actual original argument. By contrast, the last reconstruction steps of all of our algorithms rely on directed rounding to ensure that the final computed result is no less than (or greater than) the value that would be delivered in the absence of roundoff. To use directed rounding in this way, we must keep track of the signs of the intermediate quantities computed by our algorithms much more carefully than would be necessary in a point function; we also manipulate those signs so that we can perform all computations in the same directed rounding mode. Fortunately, as our performance results show, the cost of manipulating signs is more than offset by the savings afforded by using a single rounding mode for all computations.

## 3. Exponential and Logarithm Functions

### 3.1. Exponential

To compute $\exp(x)$, we set $j = [x/n]$ where $n = \log(2)/256$ to working precision and the square brackets denote the nearest integer; we then write $j = 256k + i$, where $k$ and $i$ are integers and $0 \le i < 256$, so that $\exp(x) = 2^k 2^{i/256} \exp(x - j\log(2)/256)$. From a table, we obtain $e_h$ and $e_l$ such that $|2^{i/256} - (e_h + e_l)| < 2^{-105}$ and $|e_l| < 2^{-52}$.

We compute $r = (x - jn_h) - jn_l$ where $n_h = \log(2)/256$ rounded to 32 significant bits and $n_l = \log(2)/256 - n_h$ rounded to working precision. Note that if $|x|$ is small enough that $\exp(x)$ does not trivially underflow or overflow, $|j| < 2^{20}$, so $jn_h$ is exact, and the total roundoff error in computing $r$ is smaller than $2^{-60}$. As $|r|$ is less than about $\log(2)/512$, the corresponding error in $\exp(r)$ is smaller than $2^{-59}$.

We approximate $\exp(r)$ by a polynomial $1 + E(r)$ where $E(r) := r(1 + r(E_1 + r(E_2 + rE_3)))$. (Table 1 gives the coefficients for all of the polynomial approximations used in this paper.) The approximation error satisfies $|\exp(r) - 1 - E(r)| < 2^{-57}$, and the roundoff error in computing $E(r)$ is smaller than $2^{-60}$. Note also that $|e_l E(r)| < 2^{-62}$, so computing $2^k(e_h + (e_h E(r) + (e_l + 2^{-56})))$ rounding up yields an upper bound on $\exp(x)$, while computing $2^k(-e_h + (-e_h E(r) + (2^{-56} - e_l)))$ rounding up yields the negative of a lower bound.

| Coeff. | Hexadecimal value |
|---|---|
| $E_1$ | 3fdfffff fffffff6 |
| $E_2$ | 3fc55555 721a1d14 |
| $E_3$ | 3fa55555 6e0896af |
| $l_1$ | 3fd55555 55555e83 |
| $l_2$ | bfcfffff ffffb840 |
| $l_3$ | 3fc99999 98215569 |
| $l_4$ | bfc55555 48cee8eb |
| $l_5$ | 3fc2492d 1b233010 |
| $l_6$ | bfc0009d 1058b3ef |
| $l_7$ | 3fbc588d 3a223fb0 |
| $l_8$ | bfb669ae 7e3daa5c |
| $L_1$ | 3fe55555 5550a044 |
| $L_2$ | 3fd999b4 d293087c |
| $A_1$ | bfd55555 555554ee |
| $A_2$ | 3fc99999 997a1559 |
| $A_3$ | bfc24923 158dfe02 |
| $A_4$ | 3fbc639d 0ed1347b |
| $s_1$ | bfc55555 555554d0 |
| $s_2$ | 3f811111 1108c703 |
| $s_3$ | bf2a019f 75ee4be1 |
| $s_4$ | 3ec718e3 a6972785 |
| $S_1$ | bfc55555 5555240f |
| $S_2$ | 3f81110e b1933012 |
| $C_1$ | bfdfffff ffff6328 |
| $C_2$ | 3fa55551 5f7acf0c |
| $c_1$ | bfdfffff ffffff4a |
| $c_2$ | 3fa55555 554d5306 |
| $c_3$ | bf56c16b a66c5bb7 |
| $c_4$ | 3ef9fcc6 2ae464e7 |
| $T_1$ | 3fd55555 55555526 |
| $T_2$ | 3fc11111 111239fa |
| $T_3$ | 3faba1ba 16d5e0be |
| $T_4$ | 3f9664f9 16ac0a28 |
| $T_5$ | 3f8224b3 cfd6cedc |
| $T_6$ | 3f6e745d 4523fa96 |

**Table 1. Coefficients for polynomial approximations. The values are shown as hexadecimal representations of IEEE 754 double precision numbers.**

### 3.2. Logarithm

For $31/32 \le x < 17/16$, we let $v = x - 1$ and approximate $\log(x)$ by a polynomial $v + v^2(-1/2 + l(v))$ where $l(v) := v(l_1 + v(l_2 + \ldots + vl_8))$. The approximation error satisfies $|\log(x) - v - v^2(-1/2 + l(v))| < 2^{-56}|v|$, and the roundoff error in evaluating $l(v)$ is less than $2^{-56}$. Thus computing $v + |v|(2^{-56} + |v|(-1/2 + (2^{-56} + l(v))))$ rounding up yields an upper bound on $\log(x)$ while computing $-v + |v|(2^{-56} + |v|(1/2 + (2^{-56} - l(v))))$ rounding up yields the negative of a lower bound.

For general $x$, we write $x = 2^n u$ where $91/128 \le u < 91/64$, choose $v$ such that $|u - v| \le 2^{-8}\min(|u|, |v|)$, and use the formula $\log(x) = n\log(2) + \log(v) + \log(u/v)$.

Let $b_h = \log(2)$ chopped to 38 bits and $b_l = \log(2) - b_h$ to double precision, so $|\log(2) - b_h - b_l| < 2^{-92}$. As $|n| \le 1074$, the product $nb_h$ is computed exactly, and the total of the approximation error in $b_l$ and the roundoff error in evaluating $nb_l$ is less than $2^{-80}$.

From a table, we obtain $l_h$ and $l_l$ where $l_h = \log(v)$ rounded to 26 bits past the binary point and $l_l = \log(v) - l_h$ to double precision, so $|\log(v) - l_h - l_l| < 2^{-79}$. Note that $h$ has enough trailing zeroes to ensure that $l_h + nb_h$ is also computed exactly.

Finally, we apply the identity $\log(u/v) = 2\tanh^{-1}(s)$ where $s = (u - v)/(u + v)$. We have $|s| < 2^{-9}$, and we approximate $2\tanh^{-1}(s)$ by an odd polynomial $L(s) := s(2 + s^2(L_1 + s^2 L_2))$. The approximation error satisfies $|\log(u/v) - L(s)| < 2^{-63}$. By choosing $v$ to be less than, respectively greater than $u$, we ensure that $s$ is positive, respectively negative, so if we compute $s$ so that roundoff errors cause it to exceed the exact value $(u - v)/(u + v)$ in magnitude, then the roundoff errors in computing $L(s)$, respectively $L(-s)$, rounding up will all be positive. Thus, computing $(l_h + nb_h) + (2^{-62} + l_l + nb_l + L(s))$ rounding up yields an upper bound on $\log(x)$ while computing $-(l_h + nb_h) + (2^{-62} - l_l - nb_l + L(-s))$ rounding up yields the negative of a lower bound.

### 3.3. Arctangent

As the arctangent function is odd and globally monotonic, we can obtain a lower bound on $\arctan(x)$ from an upper bound on $\arctan(-x)$, so it suffices to show how to compute the latter for any $x$.

If $|x| < 1/32$, we approximate $\arctan(x)$ by an odd polynomial $x + x^2 A(x)$ where $A(x) := x(A_1 + x^2(A_2 + x^2(A_3 + x^2 A_4)))$. The approximation error satisfies $|\arctan(x) - x - x^2 A(x)| < 2^{-62}|x|$, and the roundoff error in computing $|x|A(x)$ is less than $2^{-62}$, so computing $x + |x|(2^{-60} + |x|A(x))$ rounding up yields an upper bound on $\arctan(x)$.

If $1/32 \le |x| < 32$, we find a value $y$ satisfying $0 \le y - x \le 2^{-4}|x|$, so $\arctan(x) = \arctan(y) + \arctan((x - y)/(1 + xy))$. From a table, we obtain $a_h$ and $a_l$ such that $a_h = \arctan(y)$ to double precision and $0 \le a_h + a_l - \arctan(y) < 2^{-105}$. As $y \ge x$, evaluating $s = (x - y)/(1 + xy)$ rounding up ensures that the computed value is greater than the exact value. Now $|s| < 1/32$, so computing $a_h + (s + (a_l + |s|(2^{-60} + |s|A(s))))$ rounding up yields an upper bound on $\arctan(x)$.

If $|x| \ge 32$, we compute $\arctan(x)$ as $\mathrm{sgn}(x)\pi/2 + \arctan(-1/x)$, where the latter term is computed as above.

## 4. Trigonometric Functions

### 4.1. Argument reduction

To compute a trigonometric function of an argument given in radians, we first reduce the argument to a suitable range, typically $[-\pi/4, \pi/4]$. Specifically, we want to compute $x - n\pi/2$ where $n$ is the integer nearest $2x/\pi$. (For some functions, we may restrict $n$ to be even or odd, leading to a larger reduced range of $[-\pi/2, \pi/2]$.) Of course, in order to preserve the accuracy of the computed function values for large arguments, we must compute $x - n\pi/2$ to high relative accuracy for all arguments $x$. Our implementation uses two different methods depending on the magnitude of the argument. Note that in both cases, the argument reduction is carried out rounding to nearest.

If $|x| < 2^{19}\pi$, we first compute $n = [2x/p]$, where $p$ is a double precision approximation to $\pi$ and the square brackets denote the nearest integer. Note that the errors in computing $n$ are not significant: at worst, the reduced argument will simply lie slightly outside the desired range. We now compute $x - nP$ to twice double precision (by well-known techniques), where $P$ is a multi-word, 157-bit approximation to $\pi/2$. As $|n| \le 2^{20}$, the total errors due to approximating $\pi/2$ by $P$ and to rounding the tail of the product $nP$ are each smaller than $2^{-136}$. Using McDonald's Nearpi program[10], we verified that all reduced arguments for $x$ in this range are at least $2^{-62}$ in magnitude, so the corresponding relative error in the reduced argument is smaller than $2^{-73}$. Roundoff in the accumulation of the reduced argument contributes a relative error no larger than $2^{-105}$, so the final result is a pair of double precision numbers $y_1$ and $y_2$ satisfying $|x - n\pi - y_1 - y_2| < 2^{-72}|x - n\pi|$ and $|y_2| \le 2^{-53}|y_1|$.

For $|x| \ge 2^{19}\pi$, we rely on a reduction method due to Payne and Hanek[12]. (The particular implementation we use is available in the fdlibm library[2].) Without directly computing $n$, the method simultaneously produces $n$ mod 8 and, in our case, double precision numbers $y_1$ and $y_2$ satisfying $|x - n\pi - y_1 - y_2| < 2^{-75}|x - n\pi|$ and $|y_2| \le 2^{-52}|y_1|$. (Because we reduce by an integer multiple of $\pi/2$, for a point function, we would only need to consider

$n$ mod 4 to determine the quadrant in which the original argument lies. For an interval function, however, we need to know how many quadrants are spanned by the original argument interval, and this information is most easily found by retaining an additional bit in $n$[14].)

In the descriptions that follow, we omit references to the lower part ($y_2$) of the reduced argument. Our implementation incorporates this part, typically by adding it to a trailing term in the approximation, such as the higher order terms in a polynomial or the difference between the argument and a nearby point at which function values are tabulated. The error analyses given below remain valid with this modification.

### 4.2. Sine and cosine

For $\sin(x)$, we first reduce the argument to the range $[-\pi/2, \pi/2]$ by subtracting an even multiple of $\pi/2$. (Point algorithms often rely on the identity $\sin(x) = \cos(x - \pi/2)$ to reduce to the range $[-\pi/4, \pi/4]$ instead. In table-driven implementations, this saves some table space at the cost of doubling the number of cases that must be considered. For interval functions, this extra logic would be multiplied by the already nontrivial number of cases needed to handle the local extrema in the sine and cosine functions, so we have chosen to use the larger reduced range and larger tables.)

If $|x| \leq 21/128$, we approximate $\sin(x)$ by an odd polynomial $x + x^2 s(x)$ where $s(x) := x(s_1 + x^2(s_2 + x^2(s_3 + x^2 s_4)))$. The approximation error satisfies $|\sin(x) - x - x^2 s(x)| < 2^{-57}|x|$, and the roundoff error in computing $|x|s(x)$ is less than $2^{-58}$. Thus computing $x + |x|(2^{-56} + |x|s(x))$ rounding up yields an upper bound on $\sin(x)$, while computing $-x + |x|(2^{-56} - |x|s(x))$ rounding up yields the negative of a lower bound.

If $|x| > 21/128$, we apply the identity $\sin(x) = \sin(a)\cos(x-a) + \cos(a)\sin(x-a)$ and use an "accurate" table method[3]: we find double precision numbers $a$, $s$, and $c$ such that $|a - x| < 0.00782$, $|\sin(a) - s| < 2^{-60}|s|$, and $|\cos(a) - c| < 2^{-60}|c|$. Setting $d = x - a$, we approximate $\sin(d)$ by an odd polynomial $d + d^2 S(d)$ where $S(d) := d(S_1 + d^2 S_2)$ and $\cos(d)$ by an even polynomial $1 + C(d)$ where $C(d) := d^2(C_1 + d^2 C_2)$. The approximation errors satisfy $|\sin(d) - d - d^2 S(d)| < 2^{-58}|d|$, and $|\cos(d) - 1 - C(d)| < 29 \cdot 2^{-61}$, while the roundoff error in computing $|d|S(d)$ is less than $2^{-67}$, and the roundoff error in computing $C(d)$ is less than $2^{-65}$. Thus computing $s + (s(\mathrm{sgn}(s)2^{-56} + C(d)) + c(d + |d|(2^{-56} + |d|S(d))))$ rounding up yields an upper bound on $\sin(x)$ while computing $-s + (-s(-\mathrm{sgn}(s)2^{-56} + C(d)) + c(-d + |d|(2^{-56} - |d|S(d))))$ rounding up yields the negative of a lower bound.

For $\cos(x)$, we reduce the argument to the range $[-\pi/2, \pi/2]$ by subtracting an odd multiple of $\pi/2$; we then compute bounds on the sine of the reduced argument as above. To obtain better performance for interval arguments with both endpoints already in the reduced range, however, we skip the reduction step and compute bounds on the cosine directly by the following method.

If $|x| \leq 21/128$, we approximate $\cos(x)$ by an even polynomial $1 + c(x^2)$ where $c(t) := t(c_1 + t(c_2 + t(c_3 + tc_4)))$. The approximation error satisfies $|\cos(x) - 1 - c(x^2)| < 21 \cdot 2^{-61}$, and the roundoff error in computing $c(t)$ is less than $11 \cdot 2^{-61}$. Computing $q = (-x)x$ and $p = x^2$ rounding up ensures that their computed values are smaller, respectively larger, in magnitude than the exact values. Thus computing $1 + (2^{-56} + c(-q))$ rounding up yields an upper bound on $\cos(x)$, while computing $-1 + (2^{-56} - c(p))$ rounding up yields the negative of a lower bound.

If $|x| > 21/128$, we again resort to the accurate table method, this time using the identity $\cos(x) = \cos(a)\cos(x-a) - \sin(a)\sin(x-a)$. We find $a$, $s$, and $c$ such that $|a - |x|| < 0.00782$, $|\sin(a) - s| < 2^{-60}s$, and $|\cos(a) - c| < 2^{-60}c$, and letting $d = |x| - a$, we approximate $\sin(d)$ and $\cos(d)$ by the same polynomials as above. Thus, computing $c + (c(2^{-56} + C(d)) + s(-d + |d|(2^{-56} - |d|S(d))))$ rounding up yields an upper bound on $\cos(x)$ while computing $(s(d + |d|(2^{-56} + |d|S(d))) + (-c)(C(d) - 2^{-56})) - c$ rounding up yields the negative of a lower bound.

### 4.3. Tangent

For the tangent function, we first reduce the argument to the range $[-\pi/4, \pi/4]$ by subtracting a multiple of $\pi/2$. Note that unlike the sine and cosine functions, we cannot use the larger range $[-\pi/2, \pi/2]$ since we must compute $x$ mod $\pi/2$ with high relative accuracy when $x$ is near any multiple of $\pi/2$ in order to approximate its tangent accurately. Therefore, the reduction takes the form $x = n\pi/2 + y$ where $n$ is an integer and $|y| < \pi/4$, and we must consider two cases: if $n$ is even, then $\tan(x) = \tan(y)$, and if $n$ is odd, $\tan(x) = -\cot(y)$.

If $|x| \leq 5/32$, we approximate $\tan(x)$ by an odd polynomial $x + x^2 T(x)$ where $T(x) := x(T_1 + x^2(T_2 + x^2(T_3 + x^2(T_4 + x^2(T_5 + x^2 T_6)))))$. The approximation error satisfies $|\tan(x) - x - x^2 T(x)| < 2^{-59}|x|$, and the roundoff error in computing $|x|T(x)$ is less than $2^{-57}$. Thus, computing $x + |x|(2^{-56} + |x|T(x))$ rounding up yields an upper bound on $\tan(x)$, while computing $-x + |x|(2^{-56} - |x|T(x))$ rounding up yields the negative of a lower bound.

If $x > 5/32$, we rely on the identity $\tan(x) = \tan(w) + \tan(x - w)(1 + \tan(w)^2)/(1 - \tan(x - w)\tan(w))$. We first find $w$, $t_h$, and $t_l$ such that $0 \leq x - w < 2^{-7}$, $t_h = \tan(w)$ chopped to double precision, and $t_l = \tan(w) - t_h$ rounded to double precision. Letting $d = x - w$, we approximate $\tan(d)$ by an odd polynomial $d + dt(d)$ where $t(d) := d^2(T_1 + d^2(T_2 + d^2 T_3))$. The approximation error satisfies $|\tan(d) - d - dt(d)| < 2^{-60}|d|$ and the roundoff

error in computing $t(d)$ is less than $2^{-66}$. Thus computing $u = d + d(2^{-56} + t(d))$ rounding up yields an upper bound on $\tan(d)$ while computing $v = -d + (-d)(-2^{-56} + t(d))$ rounding up yields an upper bound on $-\tan(d)$. Now computing $t_h + (t_l + (u + (t_h + 2^{-56})((t_h + 2^{-56})u))/-((t_h + 2^{-56})u - 1))$ rounding up yields an upper bound on $\tan(x)$ while computing $-t_h + (-t_l + (v + (t_h - 2^{-56})((t_h - 2^{-56})v))/((t_h - 2^{-56})v + 1))$ rounding up yields the negative of a lower bound. (If $x < -5/32$, we compute bounds on $-\tan(x) = \tan(-x)$ similarly.)

We compute $-\cot(x)$ as follows. As above, if $|x| < 5/32$, then computing $g = |x|(2^{-56} + |x|T(x))$ rounding up yields an upper bound on $\tan(x) - x$ while computing $h = |x|(2^{-56} - |x|T(x))$ rounding up yields the negative of a lower bound. To evaluate $-1/(x + g)$ accurately, we first compute $r = -1/(x + g)$ and set $r_h = r$ chopped to 21 significant bits (with an extra bit subtracted to ensure that $r_h$ is smaller than $r$), $x_h = (x + g)$ chopped to 21 significant bits, and $x_l = (x - x_h) + g$. Now computing $r_h + r((1 + r_h x_h) + \mathrm{sgn}(r)2^{-68} + r_h x_l)$ rounding up yields an upper bound on $-\cot(x)$. Likewise, to evaluate $-1/(-x + h)$, we compute $s = -1/(-x + h)$ and set $s_h = s$ chopped to 21 significant bits, $x_h = (-x + h)$ chopped to 21 significant bits, and $x_l = (-x - x_h) + h$, so computing $s_h + s((1 + s_h x_h) + \mathrm{sgn}(s)2^{-68} + s_h x_l)$ rounded up yields the negative of a lower bound.

If $x > 5/32$, we use the identity $-\cot(x) = -\cot(w) + \tan(x - w)(1 + \cot(w)^2)/(1 + \tan(x - w)\cot(w))$. We first find $w$, $c_h$, and $c_l$ such that $0 \le w - x < 2^{-7}$, $c_h = \cot(w)$ chopped, and $c_l = \cot(w) - c_h$ rounded. As above, if we set $d = x - w$, then computing $u = d + d(-2^{-56} + t(d))$ rounding up yields an upper bound on $\tan(d)$ while computing $v = -d + (-d)(2^{-56} + t(d))$ rounding up yields an upper bound on $-\tan(d)$. Now computing $-c_h + (-c_l + (u + (c_h - 2^{-56})((c_h - 2^{-56})u))/((c_h - 2^{-56})u + 1)$ yields an upper bound on $-\cot(x)$ while computing $c_h + (c_l + (v + (c_h + 2^{-56})((c_h + 2^{-56})v))/-((c_h + 2^{-56})v - 1))$ yields the negative of a lower bound. (If $x < -5/32$, we compute bounds on $\cot(x) = -\cot(-x)$ similarly.)

## 5. Accuracy and Performance Results

Table 2 shows the measured accuracy and performance of our interval elementary function algorithms. To measure the worst case error, we used Liu's Berkeley Elementary Function Test program[9] supplemented by custom test programs that use quadruple precision point functions to compute accurate function values. The second column in the table shows the worst error observed over the entire range of the function. The error is expressed in units in the last place (ulps) of the computed double precision function values; note that for interval functions, the theoretical best possible worst case error is essentially one ulp.

In addition to worst case errors, the test programs check monotonicity and sign symmetry. Because our algorithms compute all results in the same rounding mode, the approximations are slightly different for positive and for negative arguments, so in many cases we observed sign asymmetries; for example, the computed lower bound for $\sin(3.242851544064926372)$ differs by one ulp from the computed upper bound for $\sin(-3.242851544064926372)$. We did not observe any monotonicity failures, however. Moreover, our implementation takes care to preserve common inequalities such as $|\cos(x)| \le 1$ for all $x$ and cardinal vales such as $\log(1) = 0$ (e.g., the computed logarithm of the degenerate interval $[1, 1]$ is the degenerate interval $[-0, +0]$).

The third column in table 2 shows the relative performance of our interval elementary functions compared with the corresponding point functions for arguments in the primary range (shown in the fourth column). We measured the performance on a Sun Ultra/1 system; for the comparison, we used the point elementary functions optimized for that system that are provided with the Sun Workshop 4.2 compilers. The ratios show the average number of cycles for one evaluation of the interval function divided by the average number of cycles for one evaluation of the point function. (Decimal approximations of the ratios are shown in parentheses.) As these results show, the added cost of saving, changing, and restoring the rounding mode is offset by the efficiency of pipelining the evaluation of the lower and upper bounds, so that in many cases the total cost of the interval function is no more than twice the cost of the corresponding point function.

## 6. Acknowledgements

The interval elementary function algorithms described above are based on point elementary function routines developed by Dr. K.-C. Ng. His carefully crafted algorithms and well-documented source code provided valuable insight and inspiration for the interval functions.

## References

[1] Braune, K., Standard Functions for Real and Complex Point and Interval Arguments with Dynamic Accuracy, in U. Kulisch and H. J. Stetter, Eds., *Scientific Computation with Automatic Result Verification*, Springer-Verlag, New York, 1988.

[2] fdlibm: A Freely Distributable Standard Math Library, available from netlib.

[3] Gal, S., and B. Bachelis, An Accurate Mathematical Library for the IEEE Floating Point Standard, *ACM Trans. Math. Soft.* **17** (1991), 26–45.

| Function | Error (ulp) | Perf. Ratio | Primary Range |
|---|---|---|---|
| exp | 1.17 | 142/91 (1.6) | $2^{-10} < \|x\| < 700$ |
| log | 1.29 | 169/98 (1.7) | $x < 31/32$ or $17/16 < x$ |
| atan | 1.16 | 179/97 (1.8) | $1/32 < \|x\| < 32$ |
| sin | 1.28 | 161/70 (2.3) | $21/128 < \|x\| < \pi/2$ |
| cos | 1.28 | 136/77 (1.8) | $21/128 < \|x\| < \pi/2 - 21/128$ |
| tan | 1.32 | 188/94 (2.0) | $5/32 < \|x\| < \pi/4$ |

**Table 2. Measured accuracy and performance of interval functions. The second column shows the largest error observed over the entire range of the function. The third column shows the ratio of the cost of the interval function to that of the corresponding point function measured as the average number of clock cycles per function evaluation for arguments in the primary range shown in the fourth column.**


[4] IEEE 754-1985 Standard for Binary Floating-Point Arithmetic, Institute of Electrical and Electronics Engineers, New York, 1985.

[5] Kahan, W., and I. Farkas, Algorithm 168 and Algorithm 169, *Comm. ACM* **6** (1963), 165.

[6] Kearfott, R. B., M. Dawande, K. Du, and Ch. Hu, Algorithm 737: INTLIB: A Portable Fortran-77 Elementary Function Library, *ACM Trans. Math. Soft.* **20** (1994), 447–459.

[7] Knüppel, O., BIAS—Basic Interval Arithmetic Subroutines, Bericht 93.3, Technische Universität Hamburg-Harburg, 1993.

[8] Krämer, W., Inverse Standard Functions for Real and Complex Point and Interval Arguments with Dynamic Accuracy, in U. Kulisch and H. J. Stetter, Eds., *Scientific Computation with Automatic Result Verification*, Springer-Verlag, New York, 1988.

[9] Liu, Z., Berkeley Elementary Function Test, available from netlib (in the UCBTEST package).

[10] McDonald, S., Nearpi, a C Program to Exhibit Large Floating-Point Numbers Very Close to Integer Multiplies of $\pi/2$, available from http://www.cs.berkeley.edu/~wkahan/testpi/.

[11] Muller, J.-M., and A. Tisserand, Towards Exact Rounding of the Elementary Functions, in G. Alefeld, A. Frommer, and B. Lang, Eds., *Scientific Computing, Computer Arithmetic and Validated Numerics (SCAN-95)*, Akademie-Verlag, Berlin, 1996.

[12] Payne, M., and R. Hanek, Radian Reduction for Trigonometric Functions, *ACM SIGNUM Newsletter* **18** (1983), 19–24.

[13] Tang, P. T. P., Table-Lookup Algorithms for Elementary Functions and Their Error Analysis, in P. Kornerup and D. Matula, Eds., *Proc. 10th Symposium on Computer Arithmetic*, IEEE Computer Society Press, Los Alamitos, Calif., 1991.

[14] Walster, G. W., personal communication.