39.42 Minimal function kernels

In unpublished notes [9], IEEE 754 chief architect William Kahan lists a small set of elementary functions from which all of the other standard ones can be computed stably, with minimal additional computation, and largely preserving the important property of monotonicity. Several of his procedures carry out argument adjustments that must be *exact*: he implicitly assumes a base $\beta = 2$. His methods still work for other floating-point bases, but the results are then expected to be slightly less accurate because of argument perturbations. We investigate that later in **Section 39.42.9** on page 16.

The needed kernel functions are:

- $\exp(x)$, $\mathrm{expm1}(x)$, $\log(x)$, $\mathrm{log1p}(x)$, and $\mathrm{sqrt}(x)$ for their entire defined argument ranges;
- $\mathrm{atan}(x)$ for the sharply limited range x in $[-(\sqrt{2}-1), +(\sqrt{2}-1)] \approx [-0.415, 0.415]$;
- $\sin(x)$ for the range x in $[-\pi/4, +\pi/4] \approx [-0.785, +0.785]$ (optional, but desirable);
- $\tan(x)$ for the sharply limited range x in $[-\pi/8, +\pi/8] \approx [-0.393, +0.393]$; and
- the utility functions $\mathrm{copysign}(x, y)$ and $\mathrm{fabs}(x)$.

The range restrictions on the sine, and the tangent and its inverse, are chosen to facilitate their implementation by a straightforward Taylor series summation with a modest number of terms, and early loop exit on convergence, or with a Horner form polynomial applied uniformly over the argument range, or with optimized fits of those functions using rational polynomials or Chebyshev sums (see **Section 39.42.6** on page 11 and **Section 39.56.1** on page 221).

Some less-common programming languages may lack some of the standard mathematical functions available in major languages, or have lower quality implementations, so it is an interesting project to use Kahan's recipes to implement the hyperbolic and trigonometric functions, and their inverses. The new implementations, if done in a language with an adequate function repertoire in multiple precisions, can then be tested against the highest precision functions to measure the average and worst-case errors of Kahan's versions. We show graphical and numerical measurements in the figures and tables in following subsections.

In addition, if the Kahan functions are implemented only at the highest available precision, with wrappers for lower-precision companions, then it may be that those additional functions are almost always correctly rounded.

Another important use for Kahan's recipes is the testing of function-specific implementations against those from his independent computational routes, which is always a healthy situation for code developers.

Kahan ignores the difficult problem of accurate range reduction for the trigonometric functions, and in this section, we do the same. Other sections of this book discuss how that reduction can be done correctly. He also does not address special

handling of Infinity, NaN, and signed zero, but we augment his recipes to treat those values.

39.42.1 Kernel function conventions

To fit his recipes on just over one compact page of formulas, Kahan introduces a short helper function, and four variables, that are assumed to be available in the recipes for the remaining functions in this section:

$$
\begin{aligned}
T(x) &= 2\tan(\tfrac{1}{2}x), && \textit{range reduced tangent via series summation,}\\
q &= (T(y))^2 && \textit{evaluated after range reduction from } x \textit{ to } y,\\
w &= \mathrm{fabs}(x), && w = |x|,\\
s &= \mathrm{copysign}(1,x), && \textit{sign of } x \textit{ attached to } 1,\\
\texttt{tiny} &= \beta^{-n}, && \textit{choose smallest } n \textit{ for which } \mathrm{fl}(1-\texttt{tiny}^2) = 1.
\end{aligned}
$$

In the last of those, $n = \mathrm{ceil}(p/2)$ for p-digit significands. The value `tiny` is then usually a cutoff for which Taylor series that have only even, or only odd, powers of x can be reduced to their first term, assuming the IEEE 754 default of *round to nearest, with ties to even*. More careful handling of rounding modes requires inclusion of the *first two series terms* for tiny x.

Parentheses in Kahan's expressions must be strictly obeyed, and completely evaluated before surrounding material.

In all of the recipes, it is important to prevent humans, and compilers, from collapsing expressions into mathematically, but not computationally, equivalent versions. That might require subterfuges, like use of the C `volatile` type qualifier, or if that is not available, an external *separately compiled* function like this one:

```
fp_t
store(fp_t * px)
{
    return (*px);
}
```

Both the type qualifier and that function ensure that the compiler cannot assume that x can be elided into other expressions: its value could be changed in ways unknown to the compiler, and thus must be retrieved from its storage location when needed for arithmetic operations. That also prevents a value of higher intermediate precision being used for x, as is common with compilers on systems with the IEEE 754 binary80 format.

In the C and C++ language families, handle out-of-range arguments by setting the global `errno` variable to `EDOM`, where both are defined in the standard system header file `errno.h`, and return a run-time quiet NaN to set the IEEE 754 `FE_INVALID` sticky exception flag.

Section 39.42.7 on page 13 shows a complete implementation of one of Kahan's functions.

39.42.2 Minimal hyperbolic functions

As long as the exponential function does not overflow, Kahan defines these routes to the standard hyperbolic functions, which are defined for all x in $[-\infty, +\infty]$.

$$
\begin{aligned}
e &= \exp(w).\\
h &= \tfrac{1}{2}e.\\
E &= \mathrm{expm1}(w).\\
F &= \mathrm{expm1}(-2w).\\
\cosh(x) &= h + 1/(4h),\\
\sinh(x) &= \begin{cases} x, & \text{if } x \text{ is zero,}\\ x\,[+x^3/6], & \text{if } w < \texttt{tiny},\\ \frac{1}{2}s(E + E/(1+E)), & \text{otherwise.}\end{cases}\\
\tanh(x) &= \begin{cases} x, & \text{if } x \text{ is zero,}\\ x\,[-x^3/3], & \text{if } w < \texttt{tiny},\\ -sF/(2+F), & \text{otherwise.}\end{cases}
\end{aligned}
$$

The square bracketed terms in the case of tiny arguments are not needed in the default IEEE 754 rounding mode, but are essential for other rounding modes. We include them in all of Kahan's formulas.

Figure 1 shows the errors of those recipes in binary64 computation, measured against native library functions in the binary80 size, with 11 more significand bits. Plots are similar when the recipes are computed in binary32, and are therefore omitted.

The sampling in that figure, and four following ones, is of two types: (i) uniformly spaced in x over the plotted interval, *including* both end points, and (ii) logarithmically distributed over the interval, *excluding* the end points.

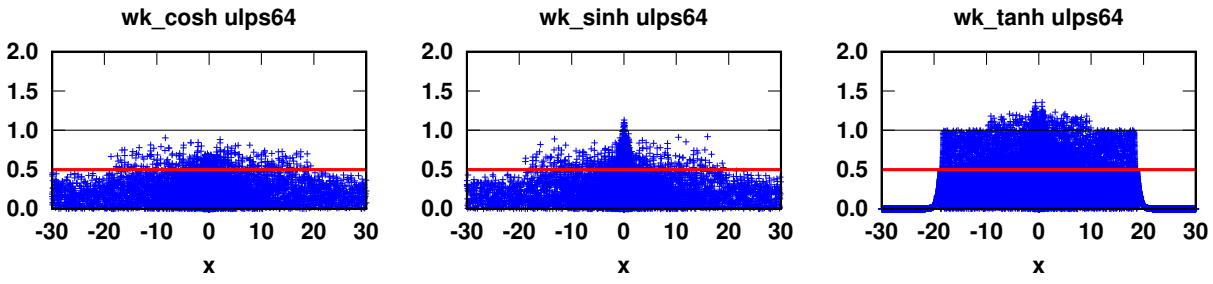


Figure 1: **Kahan hyperbolic function errors**. Results are correctly rounded if the errors do not exceed 0.5 ulps. The maximum errors found are 0.901 (cosh), 1.156 (sinh), and 1.359 (tanh) ulps. For display purposes, the plots show about 65,000 points. Increasing the sample counts to 10^8 finds new maxima of 0.999, 1.479, and 1.435 ulps respectively.

In their book, *Software Manual for the Elementary Functions* [2], Cody and Waite observe that because the spacing between consecutive floating-point numbers grows with their magnitude, when the interval is large, such as $[0, 10^{308}]$, uniform sampling would select all but the first value near the larger endpoint. Instead, they use random samples from a logarithmic distribution selected by these two functions:

```
fp_t
randl(fp_t x)
{   /* random value log distributed over (1, exp(x)) */
    return (EXP(x * unirand())); /* unirand() random on [0,1] */
}

fp_t
randlab(fp_t a, fp_t b)
{   /* random value log distributed over (a, b) */
    return (a * randl(LOG(b / a)));
}
```

The interval (a, b) excludes its endpoints, which must both be nonzero, and of the same sign. Code optimization in `randlab()` can minimize the number of logarithm calculations by computing them only when the interval changes. If the interval spans the entire floating-point range, then the logarithm must be computed as `LOG(FABS(b)) - LOG(FABS(a))` to avoid overflow in b/a.

39.42.3 Minimal inverse hyperbolic functions

Kahan computes the inverses of the standard hyperbolic functions with the following algorithms, where underflows and overflows are ignored.

Valid x range for acosh(x): $[+1, +\infty]$:

$$\operatorname{acosh}(x) = \begin{cases} +\infty, & \text{if } 2x \text{ overflows,} \\ \operatorname{log1p}(\sqrt{x-1}(\sqrt{x-1}+\sqrt{x+1})), & \text{otherwise.} \end{cases}$$

This alternative may provide a slight improvement:

$$\operatorname{acosh}(x) = \begin{cases} +\infty, & \text{if } 2x \text{ overflows,} \\ \log(x)+\log(2), & \text{if } x > 1/\texttt{tiny}, \\ \log(2x-1/(x+\sqrt{x^2-1})), & \text{if } x \text{ in } (5/4, 1/\texttt{tiny}], \\ \operatorname{log1p}((x-1)+\sqrt{(2(x-1))+((x-1)^2)}), & \text{otherwise.} \end{cases}$$

If a fused multiply-add (FMA) is available, replace x^2-1 by $\operatorname{fma}(x, x, -1)$, and $2(x-1)+(x-1)^2$ by $\operatorname{fma}(x-1, x-1, 2(x-1))$.

Valid x range for asinh(x): $[-\infty, +\infty]$:

$$\operatorname{asinh}(x) = \begin{cases} x, & \text{if } x \text{ is zero,} \\ x\,[-x^3/6], & \text{if } w < \texttt{tiny}, \\ s\operatorname{log1p}(w + w/(1/w+\sqrt{1+w^{-2}})), & \text{if } w \le 4/3, \end{cases}$$

Optionally, define $r = 1/w$, and replace $1 + w^{-2}$ by $\mathrm{fma}(r, r, 1)$.

This variant may offer better accuracy:

$$\mathrm{asinh}(x) = \begin{cases} x, & \text{if } x \text{ is zero,} \\ x\,[-x^3/6], & \text{if } w < \texttt{tiny}, \\ s\infty, & \text{if } 2w \text{ overflows,} \\ s\,\mathrm{log1p}(w + w/(1/w + \sqrt{1 + w^{-2}})), & \text{if } w \le 4/3, \\ s\log(2w + 1/(+\sqrt{1 + w^2})), & \text{if } w < 1/\texttt{tiny}, \\ s(\log(w) + \log(2)), & \text{otherwise.} \end{cases}$$

Optionally, define $r = 1/w$, and replace $1 + w^{-2}$ by $\mathrm{fma}(r, r, 1)$, and replace $1 + w^2$ by $\mathrm{fma}(w, w, 1)$.

Valid x range for atanh(x): $[-1, +1]$:

$$\mathrm{atanh}(x) = \begin{cases} x, & \text{if } x \text{ is zero,} \\ x\,[+x^3/3], & \text{if } w < \texttt{tiny}, \\ \frac{1}{2}s\,\mathrm{log1p}(2w/(1 - w)), & \text{otherwise.} \end{cases}$$

Figure 2 on the following page shows the errors of those recipes in binary64, measured against native library functions in the binary80 size.

39.42.4 Trigonometric argument reduction

At least one case in each of the three trigonometric functions discussed in following sections requires accurate argument reduction with a *precision-dependent* function like this **hoc** version:

```
if (P == 113) \
{
    PI_HALF_HI := 40780204165075941004168715406870680 / \
                  25961484292674138142652481646100480;
    PI_HALF_LO := 4.335858469968755291048747229615390820e-35
} \
else \
{
  # splits of PI/2 for P = 64, 53, and 24 are elided here
}

func PI_HALF_minus_w(w) \
{
    global PI_HALF_HI, PI_HALF_LO

    return ((PI_HALF_HI - w) + PI_HALF_LO)
}
```

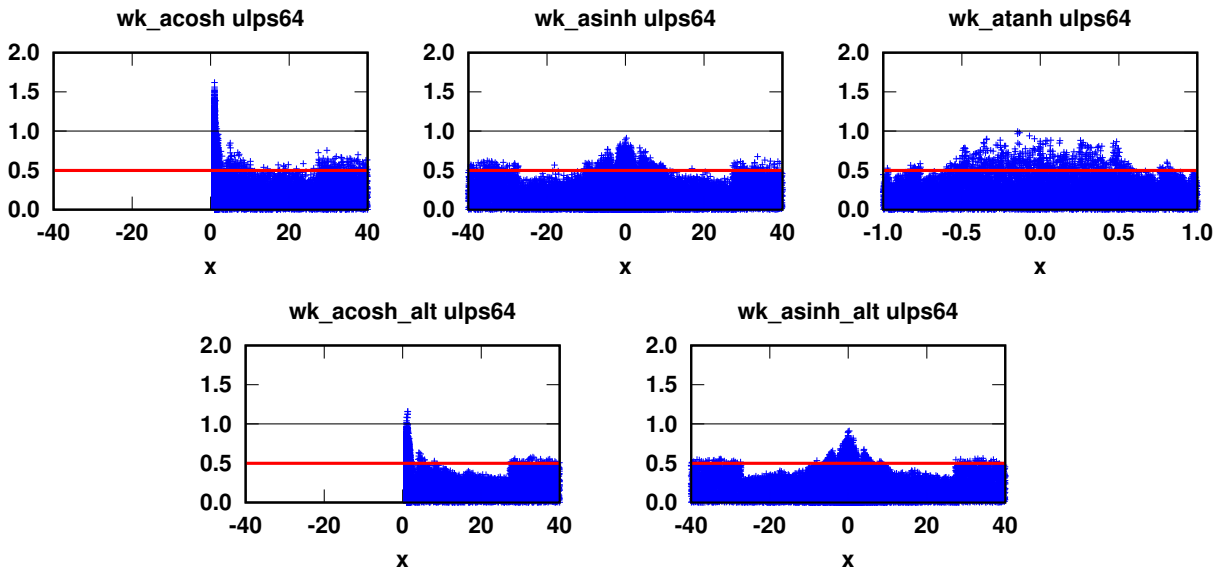


Figure 2: **Kahan inverse hyperbolic function errors**. Results are correctly rounded if the errors do not exceed 0.5 ulps. The maximum errors found are 1.620 (acosh), 0.913 (asinh), and 0.997 (atanh) ulps in the top row, and 1.159 (acosh) and 0.923 (asinh) ulps for the alternate algorithms in the bottom row.

Measurements against higher precision results show that for w in $[0, \frac{1}{2}]$, errors increase from 0.5 ulps to 0.776 ulps. In the remaining region, w in $(\frac{1}{2}, \pi/2]$, the reduction is exact.

There are two ways to achieve almost-always exact argument reduction. Both require use of error compensation to recover the rounding errors in the sum and difference. One way applies that compensation to the return value in our **hoc** function, and the other way eliminates precision dependence in the two constants by representing $\frac{1}{2}\pi$ as a sum of terms, each of which is exactly representable in *all* floating-point precisions from binary32 to binary128.

Error compensation in sums was treated in independent papers in 1965 by Kahan [8] and Møller [10], and extended in 1971 by Dekker [3] in portable algorithms to recover errors in add, subtract, multiply, divide, and square root. This author's book [1, Chapter 13] treats that work in greater detail, mentions several historical architectures that cannot be handled by the algorithms in those papers, and emphasizes that their methods fail near the floating-point underflow and overflow limits. Fortunately, for our needs here, we can assume IEEE 754 floating-point arithmetic, and we avoid the problematic limiting regions.

Kahan's formula is now commonly called the *2sum* algorithm, and it has recently been extended [4, 5, 6] with *3sum* and *4sum* variants for more addends.

The simplest form of the *2sum* algorithm requires its arguments to be ordered in

nonincreasing magnitude, and in that form, is called *fast2sum*. It needs arithmetic in base 2 that rounds to nearest, and does not apply if the intermediate operations underflow without subnormals, or if they overflow. General, and longer, forms that are branch free and avoid external function calls, and that are valid in other rounding modes, are known [12, 13, §4.3], but for simplicity, we use here the shorter form with internal argument reordering when needed. The implementation in C looks like this:

```
fp_t
twosum(fp_t a, fp_t b, fp_t * err)
{   /* result + err = exact sum of a + b */
    volatile fp_t s, v;

    if (FABS(b) > FABS(a))
    {   /* ensure |a| >= |b| */
        s = b;
        b = a;
        a = s;
    }

    s = a + b;
    v = s - a;
    *err = b - v;

    return (s);
}
```

The `volatile` qualifier is essential to prevent optimizing compilers from incorrectly replacing `v` by `b`, and `*err` by zero.

The argument reduction then looks like this:

```
fp_t
PI_HALF_minus_w(fp_t w)
{   /* return accurate value of PI/2 - w */
    fp_t err1, err2, result, sum1, sum2;
    volatile fp_t err3;

    sum1 = twosum(-w,   PI_HALF_LO, &err1);
    sum2 = twosum(PI_HALF_HI, sum1, &err2);
    err3 = err1 + err2;
    result = sum2 + err3;

    return (result);
}
```

Our second method with a multiterm representation of $\frac{1}{2}\pi$ is supplied by this function, whose leading comments show how the array of constants is computed:

```
fp_t
```

```
PI_HALF_minus_w_alt(fp_t w)
{   /* return accurate value of PI/2 - w */
    /*
    ** Compute PI/2 - w from a multiple-precision representation
    ** of PI/2 as a sum of EXACT 20- or 21-bit terms, summing
    ** from smallest to largest.  The hexadecimal values of the
    ** terms are computed like this:
    **
    ** % maple
    ** > Digits := 75:
    ** > c := evalf(Pi/2):
    ** >
    ** > scale := 2**20:
    ** >
    ** > for k from 0 to 6 do
    ** >     h := trunc(scale * c):
    ** >     printf("FP(0x%06xp%+d),\n", h, -(k + 1) *20):
    ** >     c := scale * c - h:
    ** > end do:
    */

    fp_t err, sum, t;
    int k;
    static const fp_t pi_half[] =
    {
        FP(0x1921fbp-20),
        FP(0x054442p-40),
        FP(0x0d1846p-60),
        FP(0x09898cp-80),
        FP(0x0c5170p-100),
        FP(0x01b839p-120),
        FP(0x0a2520p-140)
    };

    sum = -w;
    err = ZERO;

    for (k = sizeof(pi_half) / sizeof(pi_half[0]) - 1;
         k >= 0; --k)
    {
        sum = twosum(sum, pi_half[k], &t);
        err += t;
    }

    return (sum + err);
}
```

This alternate function is slower, because it has seven calls to `twosum()`, instead of two, but it works for all IEEE 754 binary and decimal formats, and also for several important historical floating-point systems, such as the 36-bit DEC PDP-10 and Univac 1100 systems, the 16-bit DEC PDP-11 family, and the 32-bit DEC VAX and IBM System/360 systems, *without* code changes. Each of those would require different splits of $\frac{1}{2}\pi$ into exact high, and approximate low, parts. The argument ordering in the `twosum()` calls reduces the number of internal argument swaps.

Extensive testing of our two improved reduction algorithms with more than 10^9 random samples show that errors never exceed 0.488 ulps, and thus, their results are *exact*.

39.42.5 Minimal trigonometric functions

For the standard trigonometric functions, Kahan uses these formulas, applicable *only* for x in $[-\pi/2, +\pi/2]$:[1]

$$
\begin{aligned}
y &= \tfrac{1}{2}\pi - w, && \text{NB: need argument reduction with split } \tfrac{1}{2}\pi.\\
q &= (T(y))^2, && \text{intermediate variable.}
\end{aligned}
$$

$$
\cos(x) = \begin{cases}
1[-x^2/2], & \text{if } w < \texttt{tiny},\\
\sin(y), & \text{if } w \text{ in } [\tfrac{1}{4}\pi, \tfrac{1}{2}\pi],\\
1 - 2/(1 + 4/q), & \text{if } q < \tfrac{4}{15},\\
\tfrac{3}{4} + ((1 - 2q) + \tfrac{1}{4}q)/(4 + q), & \text{otherwise.}
\end{cases}
$$

$$
\sin(x) = \begin{cases}
x, & \text{if } x \text{ is zero,}\\
x\,[-x^3/6], & \text{if } w < \texttt{tiny},\\
\cos(y), & \text{if } w \text{ in } [\tfrac{1}{4}\pi, \tfrac{1}{2}\pi],\\
\text{series sum}, & \text{otherwise, } w \text{ in } [\texttt{tiny}, \tfrac{1}{4}\pi),\\
T(x)/(1 + \tfrac{1}{4}(T(x))^2) & \text{alternate form via tangent of half angle.}
\end{cases}
$$

$$
u = T(-(\tfrac{1}{2}\pi - 2w)), \qquad \text{intermediate variable,}
$$

$$
\tan(x) = \begin{cases}
x, & \text{if } x \text{ is zero,}\\
x\,[+x^3/3], & \text{if } w < \texttt{tiny},\\
\tfrac{1}{2}T(2x), & \text{if } w < \tfrac{1}{8}\pi,\\
2s/T(2(\pi/2 - w)), & \text{if } w > \tfrac{3}{8}\pi,\\
s(2 + u)/(2 - u), & \text{otherwise, } w \text{ in } [\tfrac{1}{8}\pi, \tfrac{3}{8}\pi].
\end{cases}
$$

Kahan's formula for the small argument case of the sine function is wrong, and omitted here. We offer two alternatives: a Taylor series sum, and a relation to $T(x)$.

Numerical experiments show that series summation for small arguments in the sine function is more accurate than the alternative formula involving $T(x)$, and that choice also affects the cosine. We therefore default to series summation, but our code

[1] The formula in [9] for the intermediate argument case of $\sin(x)$ is *wrong*; we replace it by a Taylor series, or an alternative with $T(x)$.

uses the alternative formula if the symbol `USE_SIN_T_FORM` is defined at compile time.

Figure 3 shows the errors of those recipes in binary64, measured against native library functions in the binary80 format.

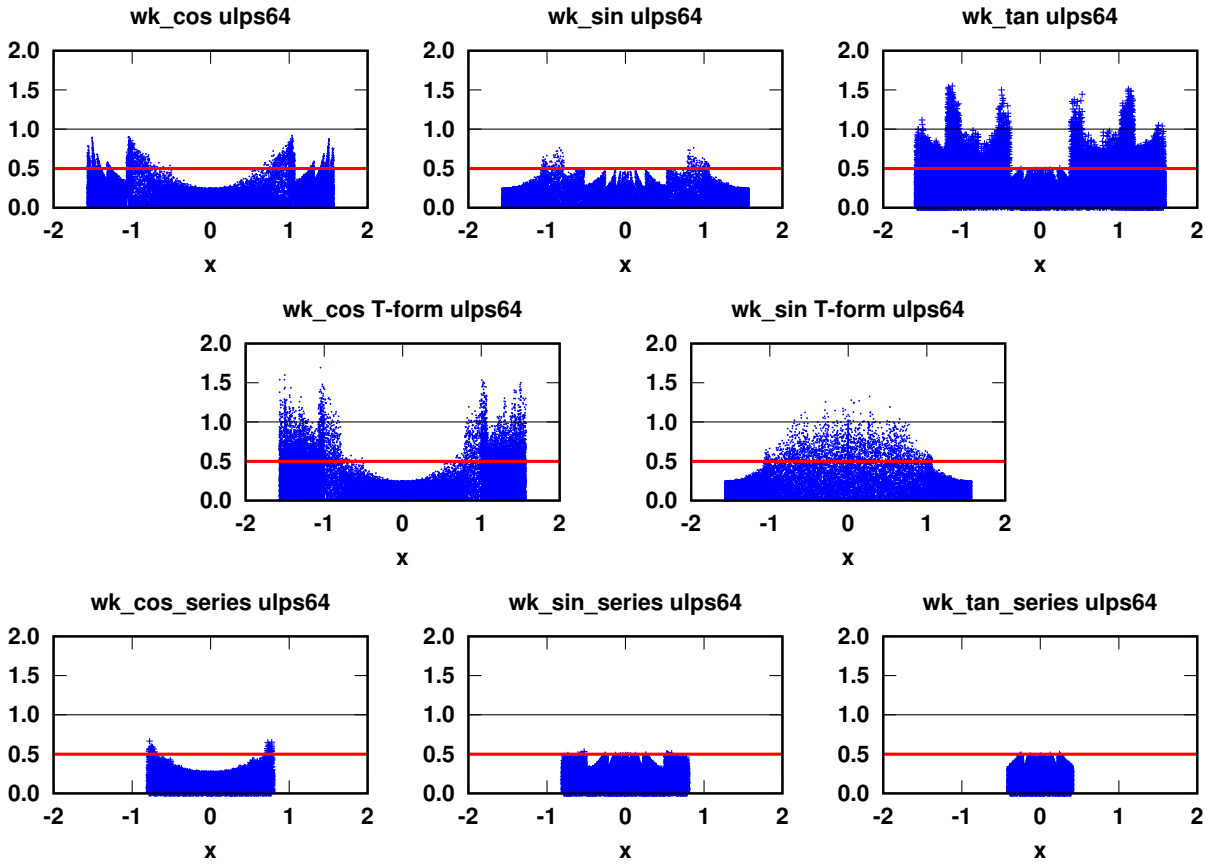


Figure 3: **Kahan trigonometric function errors**. The cosine and sine in the top row use the sine series for small arguments. Those in the middle row use the alternate form with $T(x)$. Those in the bottom row show direct series summation, but over restricted argument ranges that reduce term counts, and avoid subtraction loss. Results are correctly rounded if the errors do not exceed 0.5 ulps. The maximum errors found in the top row are 0.919 (cos), 0.771 (sin), and 1.523 (tan) ulps. The maximum errors in the middle row are 1.695 (cos) and 1.327 (sin) ulps, and in the bottom row, 0.667 (cos), 0.535 (sin), and 0.502 (tan) ulps.

39.42.6 Minimal inverse trigonometric functions

For the inverse of the standard trigonometric functions, Kahan recommends these computational routes.

Valid x range for acos(x): [−1, +1]:

$$\operatorname{acos}(x) = 2\operatorname{atan}(\sqrt{(1-x)/(1+x)}), \qquad \textit{ignore any divide-by-zero.}$$

Valid x range for asin(x): [−1, +1]:

$$\operatorname{asin}(x) = \begin{cases} x, & \textit{if } x \textit{ is zero,} \\ x\,[+x^3/6], & \textit{if } w < \texttt{tiny}, \\ \operatorname{atan}(x/\sqrt{1-w^2}), & \textit{if } w \textit{ in } [\texttt{tiny}, \tfrac{1}{2}], \\ \operatorname{atan}(x/\sqrt{2(1-w)-(1-w)^2}), & w \textit{ in } (\tfrac{1}{2}, 1]\textit{: ignore any divide-by-zero.} \end{cases}$$

Evaluate $1-w^2$ as fma$(-w, w, 1)$, if available.

Valid x range for atan(x): [−∞, +∞]:

$$\operatorname{atan}(x) = \begin{cases} x, & \textit{if } x \textit{ is zero,} \\ x\,[-x^3/3], & \textit{if } w < \texttt{tiny}, \\ s\tfrac{1}{2}\pi - \operatorname{atan}(1/x), & \textit{if } w > 1, \\ s(\tfrac{1}{4}\pi + \operatorname{atan}((x-s)/(x+s))), & \textit{if } w \textit{ in } (\sqrt{2}-1, \sqrt{2}+1), \\ \text{series sum}, & \textit{otherwise, } w \textit{ in } [\texttt{tiny}, 0.415). \end{cases}$$

In the last of those, as noted earlier, the inverse tangent can be computed from its Taylor series, or a polynomial economization thereof.

Figure 4 on the next page shows the errors of those recipes in binary64, measured against native library functions in the binary80 format.

In the series for the inverse tangent, we have $w < 0.415$. In the four IEEE 754 formats, the summation converges in at most these term counts: 8 (binary32), 19 (binary64), 23 (binary80), and 42 (binary128).

As usual, it is better to delay addition of the first two terms until the remaining sum has been accumulated, and then to add them in order of increasing magnitudes.

A plot of atan(x) in the range that we use it shows that it visually deviates from a straight line only near the interval endpoints, and then by at most 5.5%. A plot of tan(x) in its needed range is also nearly a straight line, with the same deviation. Both are therefore excellent candidates for a curve fit to a more rapidly computed function.

Here is an example of how such a fit can be done in a symbolic algebra system, producing a fit of the form $\operatorname{atan}(x) \approx x + x^2 P(x)/Q(x)$ with polynomials of degrees 5 and 6, where the polynomial quotient contributes at most a 5.5% correction to an exact value:

```
% maple
> with(numapprox):
```

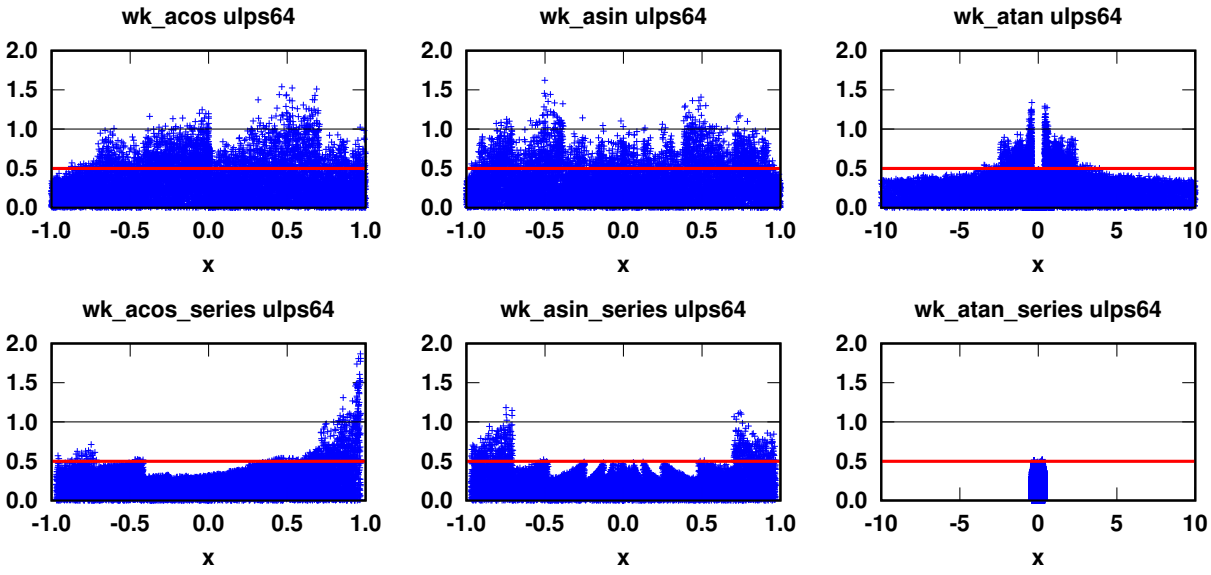


Figure 4: **Kahan inverse trigonometric function errors** in the first row, compared with direct series summation over limited ranges in the second row. Results are correctly rounded if the errors do not exceed 0.5 ulps. The maximum errors found in Kahan's formulas in the top row are 1.541 (acos), 1.621 (asin), and 1.339 (atan) ulps. The maximum errors in the direct series summation in the bottom row are 2.340 (acos), 1.185 (asin), and 0.524 (atan) ulps.

```
> Digits := 60:
> ulp_64 := 2**(-52):
> e := sqrt(2) - 1:
> r := minimax((arctan(x) - x) / x**2, x = 0 .. +e,
              [5, 6], 1, 'maxerror'):
> printf("%.3g\n", maxerror / ulp_64):
1.37
```

The error in the fit of the inverse tangent is at most $1.37(\sqrt{2}-1)^2/3 \approx 0.079$ ulps. The cost is about 12 multiply-add operations, compared to the 19 needed for the Taylor series polynomial. The disadvantage is that a new fit is needed for each floating-point precision, whereas the series summation is the same for any precision. In addition, the polynomial form requires the same work for every x, while the sum can be terminated early for smaller x values.

We can get a shorter fit with a variable transformation and a range extension:

```
r := minimax((arctan(sqrt(x)) - sqrt(x)) / (x * sqrt(x)),
             x = 0 .. +e**2, [3, 4], 1, 'maxerror'):
printf("%.3g\n", maxerror / ulp_64):
```

```
0.298
```

That corresponds to a relative error of $0.298(\sqrt{2}-1)^2/3 \approx 0.017$ ulps.

There is much more on accurate, and practical, polynomial fitting in [1, Chapter 3].

39.42.7 Sample code for the inverse sine

Here is a sample implementation in C of the inverse sine:

```
fp_t
wk_asin(fp_t x)
{
    fp_t result, w;

    w = FABS(x);

    if (ISNAN(x) || (w > ONE))
        result = range_error("0xdead0006");
    else if (x == ZERO)
        result = x;
    else if (w < __TINY__)
        result = x + x * x * x / SIX;
    else if ( (__TINY__ <= w) && (w <= HALF) )
        result = wk_atan(x / wk_sqrt(wk_fma(-w, w, ONE)));
    else
    {
        fp_t one_minus_w;

        one_minus_w = ONE - w;
        result = wk_atan(x /
                 wk_sqrt(wk_fma(-one_minus_w, one_minus_w,
                                TWO * one_minus_w)));
    }

    return (result);
}
```

The code has been parametrized for a generic floating-point type, `fp_t`, and numeric constants are named and defined earlier as wrapped values with a suitable length suffix. By default, the `wk_fma()` function first computes the product in a `volatile` variable, and returns the sum of that product and its last argument. However, if the macro `USE_FMA` is defined at compile time, that function instead invokes a library FMA function for the current working precision.

The argument range check returns a run-time quiet NaN with a unique hexadecimal payload: there are 27 such NaN returns in the complete code. CPU architectures preserve returned NaN payloads, so their appearance in output makes it easy to

identify where in the code that they were generated. Unfortunately, many current implementations of floating-point output functions in various programming languages fail to report NaN payloads. A workaround for the shorter formats is possible with wrapper functions like these:

```
const char *
float_to_string(float x)
{
    static char s[20];

    if (isnanf(x))
        (void)snprintf(s, sizeof(s), "QNaN(\"0x%06lx\")",
                       0x003fffffUL &
                       (long unsigned int)*(uint32_t *)&x);
    else
        (void)snprintf(s, sizeof(s), "%.9g", (double)x);

   return (s);
}

const char *
double_to_string(double x)
{
    static char s[30];

    if (isnan(x))
        (void)snprintf(s, sizeof(s), "QNaN(\"0x%013llx\")",
                       0x0007ffffffffffffULL &
                       (long long unsigned int)*(uint64_t *)&x);
    else
        (void)snprintf(s, sizeof(s), "%.17g", x);

    return (s);
}
```

Proper payload display for the binary80 and binary128 sizes requires additional code to deal with endian addressing issues, and with three storage sizes of binary80 on different platforms.

39.42.8 Rounding-mode effects on Kahan's algorithms

Once the main code for the implementation and testing of Kahan's algorithms is available, only about two dozen extra lines of code allow run-time selection of the floating-point rounding mode, along with recording those modes in the output files and their filenames.

Table 1 on the facing page summarizes the results of experiments made with the four IEEE 754 standard rounding modes.

Table 1: **Rounding modes and Kahan's algorithms**. Row values are the maximum errors in ulps for tests with the C type `double` (binary64 arithmetic).

Function	TONEAREST	DOWNWARD	TOWARDZERO	UPWARD
PI_HALF_minus_w	0.448	0.888	0.888	0.986
PI_HALF_minus_w_alt	0.448	0.888	0.888	0.986
wk_acos	1.723	2.896	2.057	2.423
wk_acos_series	2.316	5.129	2.784	7.445
wk_acosh	1.526	3.660	3.660	3.878
wk_acosh_alt	1.358	2.355	2.355	2.204
wk_asin	1.623	2.324	1.702	2.348
wk_asin_series	1.284	6.203	1.584	6.983
wk_asinh	0.925	1.727	1.761	1.816
wk_asinh_alt	0.925	1.727	1.907	1.816
wk_atan	1.388	2.171	1.019	1.567
wk_atan_series	0.523	1.009	0.999	1.010
wk_atanh	1.013	1.825	1.825	1.516
wk_cos	0.908	1.791	1.789	1.865
wk_cos_series	0.660	0.673	0.628	0.680
wk_cosh	0.929	1.962	1.962	1.995
wk_sin	0.833	1.018	1.000	1.009
wk_sin_series	0.534	1.030	1.000	1.052
wk_sinh	1.170	1.949	1.954	1.995
wk_tan	1.597	2.171	2.185	2.403
wk_tan_series	0.506	1.000	0.999	3.553
wk_tan_series_64	0.506	1.000	0.999	3.917
wk_tanh	1.369	3.000	2.079	2.457

Compared to values for the IEEE 754 default in the second column, the errors in the remaining columns are up to 5.7 ulps worse, with plots suggesting a typical increase of 1 to 3 ulps over the ranges used in earlier error graphs. Notice from the first two rows that trigonometric argument reduction, $\frac{1}{2}\pi - w$, using `twosum()` is *no longer exact* in nondefault rounding modes, as we reported earlier in **Section 39.42.4** on page 7.

The output files showed huge errors, excluded from the table, for nondefault modes with subnormal arguments, and also for these rounding modes and functions:

DOWNWARD: `asin()` for $x \to -1$, `acosh()` and `asinh()` for $|x| \to$ `MAXNORMAL`;

TOWARDZERO: `acosh()` and `asinh()` for $|x| \to$ `MAXNORMAL`;

UPWARD: `acosh()` and `asinh()` for $|x| \to$ `MAXNORMAL`.

Our experiments of this section show the importance of unbiased rounding for accuracy in computation, and why IEEE 754 arithmetic has been such an improvement over its predecessors.

Most numerical software is written without explicit rounding control, and thus assumes default rounding. Guaranteeing its behavior in the presence of user-requested changes in the rounding mode would require software to save the mode at entry, switch to the default mode, and then restore the saved mode at exit. Our implementation does not do that, but the results of this section raise the issue for future code implementers.

39.42.9 Kahan's algorithms in other bases

We reported on page 1 that Kahan's implicit assumption of base-2 floating-point arithmetic could diminish accuracy from inexact argument reductions in other bases, notably 16 on the IBM System/360 family, and 10 in decimal arithmetic. The latter was specified in IEEE 854-1987, but never achieved support in international standards for programming languages, and is now required in IEEE 754-2008.

In base-16 arithmetic, significand fractions lie in $[\frac{1}{16}, 1)$, and special care is needed in algorithm design to avoid generating significands out of the range $[\frac{1}{2}, 1)$: smaller values lose 1 to 3 leading bits due to hexadecimal normalization. Thus, division by two generally loses 1 bit, division by eight may lose 3 bits, and those operations may cause rounding that affects low-order bits. In the worst case, the rounding changes *all* of the significand bits, and may reduce the exponent.

In base-10 arithmetic, division by two causes rounding in all full-width significands that end in an odd digit. In the decimal32 format, $0.9999999/2 \rightarrow 0.49999995 \rightarrow 0.5000000$. Shorter significands are reduced exactly: $0.999999/2 \rightarrow 0.4999995$.

This author's book [1] describes portable algorithms that are implemented in the MathCW library for 17 different floating-point formats, including several for bases 10 and 16.

The compilers and Unix-family operating systems available to this author on IBM s390x CPUs now support only the IEEE 754 formats, so testing of Kahan's algorithms with base-16 arithmetic has not been possible.

However, with a few hours of work, I converted the binary implementation of his algorithms, and an extensive test suite, to support the decimal32, decimal64, and decimal128 formats, which offer precisions of 7, 16, and 34 decimal digits. The needed compiler and run-time library support for several CPU architectures and operating systems is freely available at a Web site.[2]

The entire test suite has now been run successfully with all three decimal formats, and the results are perhaps surprising: on most tests, the errors in ulps are *smaller* than they are in binary arithmetic. To give a flavor of the improvements, compare **Figure 5** on the next page with **Figure 3** on page 10. Plots for the other trigonometric and hyperbolic functions show similar improvements in both decimal32 and decimal64 formats, but are omitted here.

It is not currently possible for this author to investigate other rounding modes in decimal floating-point arithmetic, because only the default of *round to nearest, with ties to even* is supported in the **gcc** compiler releases from 2006 to 2025.

[2] See https://www.math.utah.edu/pub/mathcw.

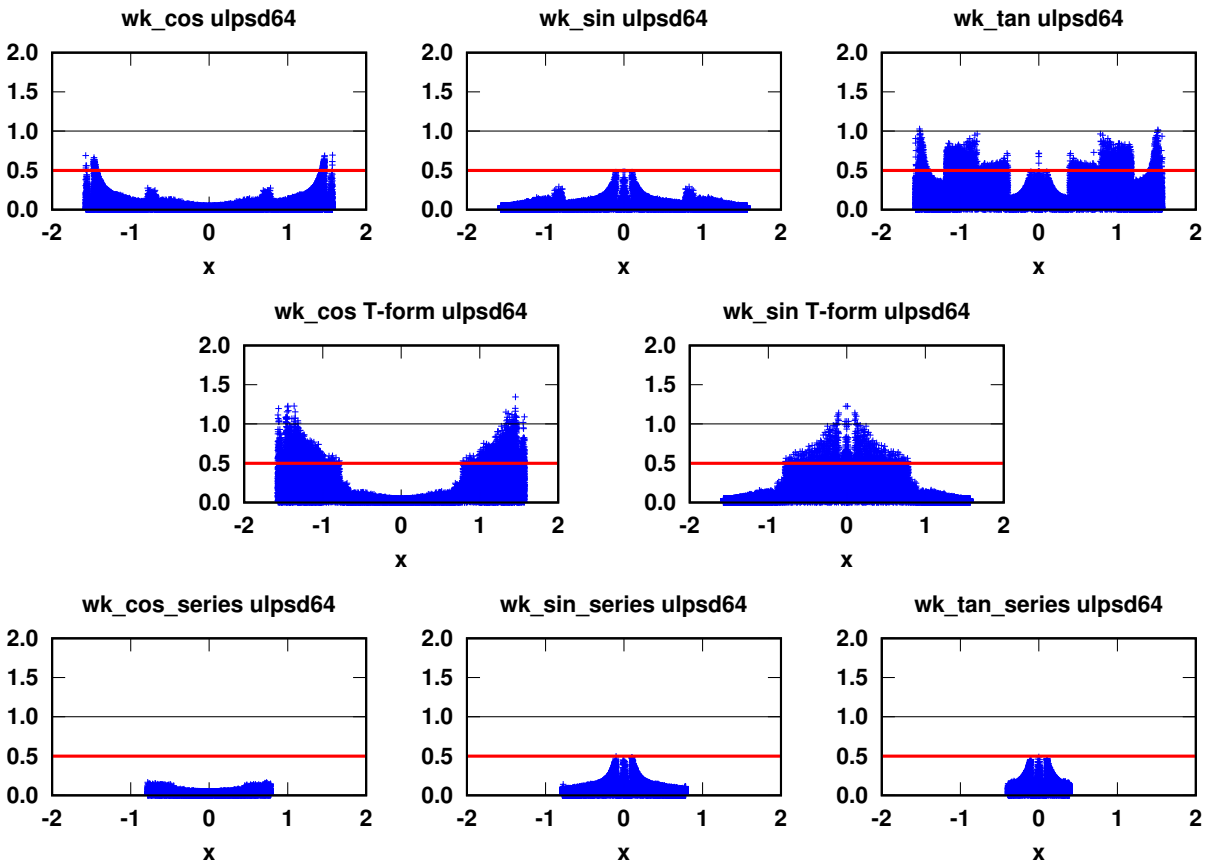


Figure 5: **Kahan decimal64 trigonometric function errors**. The `twosum()` correction, while not always exact in bases other than 2, generally improves accuracy of argument reduction in $\frac{1}{2}\pi - w$ computations: see **Table 2** on page 21. The cosine and sine in the top row use the sine series for small arguments. Those in the middle row use the alternate form with $T(x)$. Those in the bottom row show direct series summation, but over restricted argument ranges that reduce term counts, and avoid subtraction loss. Results are correctly rounded if the errors do not exceed 0.5 ulps. The maximum errors in ulps found are top: 0.696 (cos), 0.485 (sin), and 1.082 (tan); middle: 1.364 (cos) and 1.227 (sin); and bottom: 0.167 (cos), 0.497 (sin), and 0.488 (tan).

39.42.10 Correct rounding in Kahan's algorithms

In the IEEE 754 default rounding mode, the measured errors in Kahan's formulas never reach 2 ulps. That means that hardware implementations that can supply additional significand bits for intermediate computations, such as can be done with

ASIC and FPGA customized circuits, could provide correctly rounded results for all of the standard hyperbolic and trigonometric functions, and their inverses. The same holds for software multiple-precision arithmetic, if the internal precision can be temporarily increased.

Exhaustive testing of all possible inputs is not currently feasible for significands longer than about 30 bits. For the binary32 format, such testing *is* practical, and any exceptions to correct rounding could be tabulated as pairs (x, (correctly rounded) $f(x)$), and checked for at function entry. Otherwise, every ten additional significand bits decrease the probability of incorrect rounding by a factor of about 10^{-3}, so having fifty extra bits suggests a rounding error about once in 10^{15} inputs.

As a partial test of that statement, the inverse trigonometric functions were computed in binary80 with the recipes in **Section 39.42.6** on page 11, and with the native math library, for all possible binary32 arguments in the function domains, and then the result pairs were cast to binary32 and compared. The pairs were identical for the inverse sine and tangent, and only two exceptions were reported for the inverse cosine:

```
Test of wk_acos in long double, then rounded to float
x = 0x1.110b_46p-26   a = 0x1.921f_b4p+0   e = 0x1.921f_b5p+0
x = 0x1.04c4_44p-12   a = 0x1.920f_68p+0   e = 0x1.920f_69p+0
```

The `a` value is from Kahan's algorithms, and the `e` value is from the native library, which is identified as version *2.39-0ubuntu8.6* in library `libm.so.6` from package `libc6` in a recent Ubuntu distribution.

Computation with the native library in binary128 produces these results:

```
hoc128> x = 0x1.110b_46p-26
hoc128> hexfp(acos(x)); hexfp(double(acos(x))); \
        hexfp(single(acos(x)))
+0x1.921f_b500_0000_0469_898b_f603_e69dp+0
+0x1.921f_b5p+0
+0x1.921f_b6p+0

hoc128> x = 0x1.04c4_44p-12
hoc128> hexfp(acos(x)); hexfp(double(acos(x))); \
        hexfp(single(acos(x)))
+0x1.920f_6900_0000_00b7_fd1e_1f78_a97dp+0
+0x1.920f_69p+0
+0x1.920f_6ap+0
```

Notice the runs of 29 and 32 consecutive *zero* bits in the binary128 values. Evidently, reliable identification of exceptional binary32 x-values needs to be done in higher precision than binary64, perhaps with even more than the 113-bit significand of binary128.

The tests with hardware floating-point arithmetic each took less than four minutes to run on a recent x86_64 CPU. Comparison against higher-precision results from a multiple-precision package, such as the MPFR library [7], which is available for most

modern systems, is desirable. The test program was modified to do that for the three inverse trigonometric functions, setting the MPFR PREC variable to 200 bits, about eight times the number of bits in the binary32 significand. The runs took about one CPU hour for each function, and no exceptional x values were found for `wk_asin()` and `wk_atan()`. For `wk_acos()`, just two exceptions were identified, at the same arguments as found with binary80 computation:

```
x = 0x1.110b_46p-26   a = 0x1.921f_b4p+0   e = 0x1.921f_b6p+0
wk_acos(x)   = 0x1.921f_b4p+0
acosf(x)     = 0x1.921f_b4p+0
acos (x)     = 0x1.921f_b5000_0000p+0
acosl(x)     = 0x1.921f_b5000_0000_46ap+0
mpfr_acos(x) = 0x1.921f_b5000_0000_4698_98bf_603e_69d1_a8c5_\
                    979f_90fe_c47d_c6a2_ep+0

x = 0x1.04c4_44p-12   a = 0x1.920f_68p+0   e = 0x1.920f_6ap+0
wk_acos(x)   = 0x1.920f_68p+0
acosf(x)     = 0x1.920f_68p+0
acos (x)     = 0x1.920f_6900_0000_0p+0
acosl(x)     = 0x1.920f_6900_0000_00b8p+0
mpfr_acos(x) = 0x1.920f_6900_0000_00b7_fd1e_1f78_a97d_0ea2_\
                    4046_4334_c497_169c_18p+0
```

With some known exceptions, transcendental functions with floating-point arguments (and thus, rational numbers) normally produce transcendental results, and long runs of zero bits, or one bits, are improbable. Nevertheless, we have identified two cases where such runs are found, and, in addition, are positioned such that they cause incorrect rounding in two of the four IEEE 754 binary formats.

The test program was then extended to compute the high-precision reference value with MPFR, comparing the cast of that to binary32 with a similar cast of a result from Kahan's algorithm computed in binary64, for all of the hyperbolic and trigonometric functions and their inverses.

That effort recovered the two exceptional values reported earlier, plus two more, with minor changes in the output:

```
Exhaustive test of wk_atan() in double, then rounded to float

native name = atan
wk     name = wk_atan
mpfr   name = mpfr_atan
mpfr   PREC = 200 bits
MIN_X       = 0x1p-126 = 1.175_494_350_822_287_51e-38
MAX_X       = 0x1p+3 = 8.0000_0000_0000_0000_0e+00

x = 0x1.1ad6_46p-4   a = 0x1.1a63_84p-4   e = 0x1.1a63_86p-4
wk_atan(x)          = 0x1.1a63_8500_0000_0p-4
atanl(x)            = 0x8.d31c_2800_0000_10e0p-7
```

```
atan (x)            = 0x1.1a63_8500_0000_0p-4
atanf(x)            = 0x1.1a63_84p-4
(float)wk_atan(x)   = 0x1.1a63_84p-4
(float)atan (x)     = 0x1.1a63_84p-4
(float)atanl(x)     = 0x1.1a63_86p-4
mpfr_atan(x)        = 0x1.1a63_8500_0000_021b_8142_752b_7e2c_\
                          5ef7_0666_5447_344b_ee4c_eap-4

Done: 1_082_130_433 arguments tested


Exhaustive test of wk_sinh() in double, then rounded to float

native name = sinh
wk     name = wk_sinh
mpfr   name = mpfr_sinh
mpfr   PREC = 200 bits
MIN_X       = 0x1p-126 = 1.175_494_350_822_287_51e-38
MAX_X       = 0x1.633c_e8fb_9f87_dp+9 \
            = 7.104_758_600_739_438_63e+02

x = 0x1.250b_fep-11   a = 0x1.250c_00p-11   e = 0x1.250b_fep-11
wk_sinh(x)          = 0x1.250b_ff00_0000_0p-11
sinhl(x)            = 0x9.285f_f7ff_ffff_e1f0p-14
sinh (x)            = 0x1.250b_ff00_0000_0p-11
sinhf(x)            = 0x1.250b_fep-11
(float)wk_sinh(x)   = 0x1.250c_00p-11
(float)sinh (x)     = 0x1.250c_00p-11
(float)sinhl(x)     = 0x1.250b_fep-11
mpfr_sinh(x)        = 0x2.4a17_fdff_ffff_f87e_8d32_2786_ec89_\
                          bd9f_edf2_d5ab_6e1d_7648_4cp-12

Done: 1_135_713_909 arguments tested
```

For the exceptional argument for the inverse tangent, we find a run of 30 consecutive *zero* bits in the function value that affects the rounding in the binary32 and binary64 function values.

For the exceptional case in the hyperbolic sine, we have a run of 30 consecutive *one* bits that affects the rounding in the binary32 and binary64 function values.

Thus, if the native math library is at least as accurate as that with which these tests were run, we can use table lookup for just *four values* of x in three functions, and otherwise, compute all of Kahan's functions in type `double`, and then downcast to `float`, to guarantee correct rounding for all possible binary32 arguments.

Table 2 on the facing page shows the percentage of correctly rounded results in 64-bit binary and decimal arithmetic, treating binary80 values as 'exact'. Based on the binary32 timing with MPFR, an exhaustive test of one inverse trigonometric function

in binary64 arithmetic would take about $2^{53}/2^{24} \approx 5.37 \times 10^8$ hours, or more than 61 000 years.

Table 2: **Correct rounding in Kahan's algorithms** using the `twosum()` correction to ensure exact argument reduction in $\frac{1}{2}\pi - w$ computations. Values in columns 2 and 4 are the percentage of results where the error in ulps in the computed value does not exceed $\frac{1}{2}$ for 10^8 tests with the C type `double` (binary64 arithmetic), and 10^7 tests with type `decimal_double` (decimal64 arithmetic).

Function	binary64		decimal64	
	exact	inexact	exact	inexact
PI_HALF_minus_w	100.00%	0.00%	100.00%	0.00%
PI_HALF_minus_w_alt	100.00%	0.00%	100.00%	0.00%
wk_acos	91.86%	8.14%	95.16%	4.84%
wk_acos_series	98.10%	1.90%	99.98%	0.02%
wk_acosh	87.04%	12.96%	93.73%	6.27%
wk_acosh_alt	94.75%	5.25%	98.75%	1.25%
wk_asin	92.83%	7.17%	97.75%	2.25%
wk_asin_series	98.25%	1.75%	99.95%	0.05%
wk_asinh	97.65%	2.35%	99.66%	0.34%
wk_asinh_alt	98.09%	1.91%	99.70%	0.30%
wk_atan	96.45%	3.55%	98.93%	1.07%
wk_atan_series	99.88%	0.12%	100.00%	0.00%
wk_atanh	97.51%	2.49%	99.90%	0.10%
wk_cos	96.33%	3.67%	99.79%	0.21%
wk_cos_series	99.87%	0.13%	100.00%	0.00%
wk_cosh	98.14%	1.86%	99.35%	0.65%
wk_sin	99.57%	0.43%	100.00%	0.00%
wk_sin_series	99.97%	0.03%	100.00%	0.00%
wk_sinh	97.36%	2.64%	98.30%	1.70%
wk_tan	89.53%	10.47%	95.82%	4.18%
wk_tan_series	100.00%	0.00%	100.00%	0.00%
wk_tan_series_64	100.00%	0.00%	100.00%	0.00%
wk_tanh	92.71%	7.29%	98.96%	1.04%

39.42.11 Retrospective on Kahan's algorithms

The tests of all of Kahan's formulas show little improvement from the FMA, but when it has low cost, its use is advisable, because it removes one unnecessary rounding error. Kahan's descriptions of his algorithms predate by five years the first appearance in hardware of the FMA, in the IBM POWER architecture in early 1990. It took several years after that before the importance, and wide applicability, of the FMA began to be appreciated, and profitably exploited, by numerical analysts. The FMA was first

standardized in IEEE 754-2008, joining the earlier five basic operations of arithmetic: all are required to be correctly rounded in any of four rounding modes.

Several of Kahan's procedures are not at all obvious, and are not found in standard tables of relationships between mathematical functions. Thus, it is likely that they present different approaches to computation that provide independent checks of more conventional algorithms.

We extend his procedures to correctly handle signed zero, Infinity, and NaN arguments, and for out-of-range inputs, we return a quiet NaN with a distinctive payload, and set `errno` to `EDOM`.

Our testing found holes in Kahan's notes, with an incorrect algorithm for the small-argument case of the sine function, a parenthesis error in his second formula for the inverse tangent, and failure to handle the small-argument region of that function. In this work, the holes have been filled, and the necessary algorithmic corrections have been made.

We have also demonstrated by high-precision computation and exhaustive testing of binary32 arguments with IEEE 754 default rounding that Kahan's algorithms computed in `double` , and then downcast to `float`, augmented with special handling of *four* specific function arguments, can guarantee correct rounding of *all* binary32 function results.

Bibliography

[1] Nelson H. F. Beebe. *The Mathematical-Function Computation Handbook: Programming Using the MathCW Portable Software Library*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 2017, xxxvi + 1114 pp. ISBN 3-319-64109-3 (hardcover), 3-319-64110-7 (e-book); ISBN-13 978-3-319-64109-6 (hardcover), 978-3-319-64110-2 (e-book). LCCN QA75.5-76.95. URL http://www.springer.com/us/book/9783319641096. doi: 10.1007/978-3-319-64110-2.

[2] William J. Cody, Jr. and William Waite. *Software Manual for the Elementary Functions*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1980, x + 269 pp. ISBN 0-13-822064-6; ISBN-13 978-0-13-822064-8. LCCN QA331 .C635 1980.

[3] Theodorus J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, June, 1971. CODEN NUMMA7. ISSN 0029-599X (print), 0945-3245 (electronic). ISSN-L 0029-599X. URL http://www-gdz.sub.uni-goettingen.de/cgi-bin/digbib.cgi?PPN362160546_0018. doi:10.1007/BF01397083.

[4] Stef Graillat and Jean-Michel Muller. Emulation of 3Sum, 4Sum, the FMA and the FD2 instructions in rounded-to-nearest floating-point arithmetic. Report hal-04624238, Sorbonne Université, CNRS, LIP6 and CNR, Laboratoire LIP, Université de Lyon, Paris, France and Lyon, France, 25 June, 2025, 21 pp. URL https://hal.science/hal-04624238v1.

[5] Stef Graillat and Jean-Michel Muller. Emulation of the FMA and the correctly-rounded sum of three numbers in rounding-to-nearest floating-point arithmetic. Report hal-04575249, Sorbonne Université, CNRS, LIP6 and CNR, Laboratoire LIP, Université de Lyon, Paris, France and Lyon, France, 17 May, 2025, 29 pp. URL https://hal.science/hal-04575249v2.

[6] Stef Graillat and Jean-Michel Muller. Emulation of the FMA and the correctly-rounded sum of three numbers in rounding-to-nearest floating-point arithmetic. *Numerische Mathematik*, 157(5):1615–1641, October, 2025. CODEN NUMMA7. ISSN 0029-599X (print), 0945-3245 (electronic). ISSN-L 0029-599X. URL https://link.springer.com/article/10.1007/s00211-025-01487-2. doi:10.1007/s00211-025-01487-2.

[7] Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, Paul Zimmermann, Sylvie Boldo, David Daney, Mathieu Dutour, Emmanuel Jeandel, Laurent Fousse, Fabrice Rouillier, and Kevin Ryde. The MPFR library. World-Wide Web software project archive., 2000. URL http://www.mpfr.org/.

[8] W. Kahan. Pracniques: Further remarks on reducing truncation errors. *Communications of the Association for Computing Machinery*, 8(1):40, January, 1965. CODEN CACMA2. ISSN 0001-0782 (print), 1557-7317 (electronic). ISSN-L 0001-0782. doi:10.1145/363707.363723.

[9] William Kahan. Elementary functions from kernels and elementary inequalities among elementary functions, 24 October, 1985, 5 pp. URL https://ieeemilestones.ethw.org/w/images/1/16/Wk_elems_from_kernels_oct85.pdf.

[10] Ole Møller. Note on quasi double-precision. *Nordisk tidskrift for informationsbehandling*, 5(4):251–255, December, 1965. CODEN BITTEL, NBITAB. ISSN 0006-3835 (print), 1572-9125 (electronic). ISSN-L 0006-3835. URL http://www.springerlink.com/openurl.asp?genre=article&issn=0006-3835&volume=5&issue=4&spage=251. See [11]. doi:10.1007/BF01937505.

[11] Ole Møller. Quasi double-precision in floating point addition. *Nordisk tidskrift for informationsbehandling*, 5(1):37–50, March, 1965. CODEN BITTEL, NBITAB. ISSN 0006-3835 (print), 1572-9125 (electronic). ISSN-L 0006-3835. URL http://www.springerlink.com/openurl.asp?genre=article&issn=0006-3835&volume=5&issue=1&spage=37. See also [10]. doi:10.1007/BF01975722.

[12] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston Inc., Cambridge, MA, USA, 2010, xxiii + 572 pp. ISBN 0-8176-4704-X (hardcover), 0-8176-4705-8 (e-book); ISBN-13 978-0-8176-4704-9 (hardcover), 978-0-8176-4705-6 (e-book). LCCN QA76.9.C62 H36 2010. US$90 (est.). doi:10.1007/978-0-8176-4704-9;https://doi.org/10.1007/978-0-8176-4705-6.

[13] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser, Cambridge, MA, USA; Berlin, Germany; Basel, Switzerland, second edition, 2018, xxv + 627 pp. ISBN 3-319-76525-6, 3-319-76526-4 (e-book); ISBN-13 978-3-319-76525-9, 978-3-319-76526-6 (e-book). LCCN QA76.9.C62. doi:10.1007/978-3-319-76526-6.