12 Hyperbolic functions

OUT, HYPERBOLICAL FIEND! HOW VEXEST THOU THIS MAN!

— SHAKESPEARE'S *Twelfth Night* (1602).

PERFORMANCES ARE LARGELY SATISFACTORY, EVEN IF THE LEAD ACTRESS TENDS TO GO HYPERBOLIC AT TIMES.

— FILM REVIEW FOR *Khoey Ho Tum Kahan* (2001).

The hyperbolic functions are often treated with the trigonometric functions in mathematical texts and handbooks [AS64, OLBC10, Chapter 4], because there are striking similarities in their series expansions, and the relations between family members, even though their graphs are quite different.

IBM 709 Fortran provided the hyperbolic tangent function in 1958, and that remained the only family member through the first Fortran Standard in 1966. The 1977 Fortran Standard [ANSI78] adds the hyperbolic cosine and sine, but even the 2004 Standard [FTN04a] fails to mention the inverse hyperbolic functions, although a few vendor libraries do provide them.

Fortran versions of the hyperbolic functions and their inverses are available in the PORT library [FHS78b]. The FNLIB library [Ful81b, Ful81a] provides only the inverse hyperbolic functions. Both libraries were developed in the 1970s by researchers at AT&T Bell Laboratories. We discuss their accuracy in the chapter summary on page 352.

Neither Java nor Pascal has any hyperbolic functions in its mathematical classes or libraries, but C# supplies the hyperbolic cosine, sine, and tangent.

Common Lisp requires the hyperbolic functions and their inverses, but the language manual notes that direct programming of mathematical formulas may be inadequate for their computation [Ste90, page 331]. Tests of those functions on two popular implementations of the language suggest that advice was not heeded.

The C89 Standard requires the hyperbolic cosine, sine, and tangent, and the C99 Standard adds their inverse functions. In this chapter, we show how those functions are implemented in the mathcw library.

12.1 Hyperbolic functions

The hyperbolic companions of the standard trigonometric functions are defined by these relations:

$$\begin{aligned}\cosh(x) &= (\exp(x) + \exp(-x))/2,\\ \sinh(x) &= (\exp(x) - \exp(-x))/2,\\ \tanh(x) &= \sinh(x)/\cosh(x)\\ &= (\exp(x) - \exp(-x))/(\exp(x) + \exp(-x)).\end{aligned}$$

They are connected by these equations:

$$\begin{aligned}(\cosh(x))^2 - (\sinh(x))^2 &= 1,\\ \tanh(x) &= \sinh(x)/\cosh(x).\end{aligned}$$

Those functions exist for all real arguments, and vary smoothly over the ranges $[1, \infty)$ for $\cosh(x)$, $(-\infty, +\infty)$ for $\sinh(x)$, and $[-1, +1]$ for $\tanh(x)$. **Figure 12.1** on the following page shows plots of the functions.

The hyperbolic functions have simple reflection rules:

$$\begin{aligned}\cosh(-|x|) &= +\cosh(|x|),\\ \sinh(-|x|) &= -\sinh(|x|),\\ \tanh(-|x|) &= -\tanh(|x|).\end{aligned}$$

© Springer International Publishing AG 2017
N.H.F. Beebe, *The Mathematical-Function Computation Handbook*, DOI 10.1007/978-3-319-64110-2_12

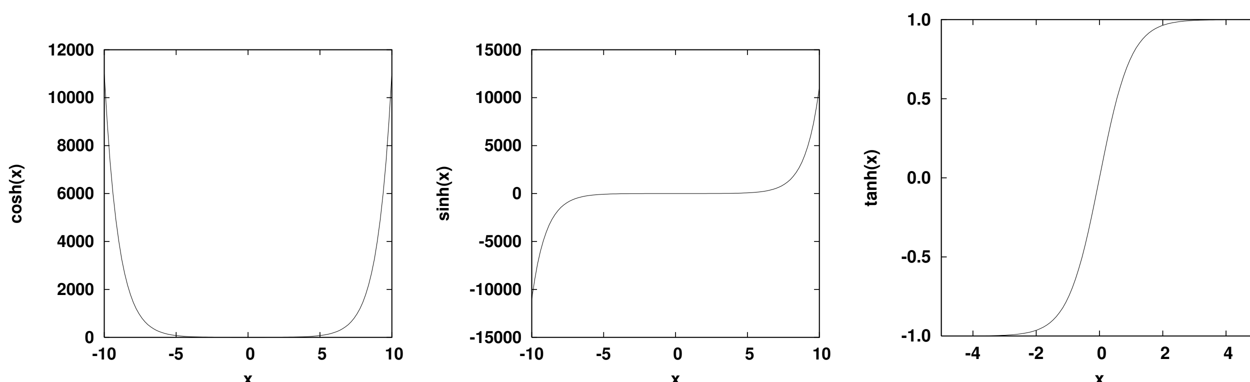

Figure 12.1: The hyperbolic functions.

We can guarantee those symmetries by computing the functions only for the absolute value of the argument, and then, for $\sinh(x)$ and $\tanh(x)$, if the argument is negative, inverting the sign of the computed result. That practice is common in mathematical software, but is incorrect for rounding modes other than the default of *round to nearest*. Preservation of mathematical identities is usually more important than how inexact results are rounded, but that would not be true for computation with interval arithmetic.

The Taylor series of the hyperbolic functions for $x \approx 0$ are:

$$\begin{aligned}
\cosh(x) &= 1 + (1/2!)x^2 + (1/4!)x^4 + \cdots + (1/(2n)!)x^{2n} + \cdots, \\
\sinh(x) &= x + (1/3!)x^3 + (1/5!)x^5 + \cdots + (1/(2n+1)!)x^{2n+1} + \cdots, \\
\tanh(x) &= x - (1/3)x^3 + (2/15)x^5 - (17/315)x^7 + (62/2835)x^9 + \cdots \\
&= \sum_{k=1}^{\infty} \frac{4^k(4^k-1)B_{2k}}{(2k)!} x^{2k-1}.
\end{aligned}$$

The first two converge quickly, and have simple coefficients, but the series for the hyperbolic tangent converges more slowly, and involves the *Bernoulli numbers*, B_{2k}, that we first met in the series expansion of the tangent (see **Section 11.2** on page 302), and that we discuss further in **Section 18.5** on page 568.

Like trigonometric functions of sums of two angles, hyperbolic functions with argument sums can be related to functions of each of the arguments:

$$\begin{aligned}
\cosh(x+y) &= \cosh(x)\cosh(y) + \sinh(x)\sinh(y), \\
\sinh(x+y) &= \sinh(x)\cosh(y) + \cosh(x)\sinh(y), \\
\tanh(x+y) &= \frac{\tanh(x) + \tanh(y)}{1 + \tanh(x)\tanh(y)}.
\end{aligned}$$

Those relations are used in the ELEFUNT tests of the hyperbolic functions.

The exponential function of a real argument is always positive, and satisfies the reflection rule $\exp(-|x|) = 1/\exp(|x|)$, so it would appear that the hyperbolic functions could be computed from a single exponential and at most four additional elementary operations.

There are three serious computational problems, however:

- The argument at which overflow happens in $\cosh(x)$ and $\sinh(x)$ is larger than that for $\exp(x)$. Thus, it is impossible to compute the two hyperbolic functions correctly by direct application of their definitions for arguments in the region where the hyperbolic functions remain finite, but $\exp(x)$ overflows.
- The subtractions in the definitions of $\sinh(x)$ and $\tanh(x)$ produce serious significance loss for $x \approx 0$.
- Hexadecimal arithmetic requires special consideration to combat the accuracy loss from wobbling precision.

We address those problems by dividing the argument interval $[0, \infty)$ into five consecutive regions:

$[\mathbf{0}, x_{\mathrm{TS}}]$: use two terms of the Taylor series;

$(x_{\mathrm{TS}}, x_{\mathrm{loss}}]$: use rational polynomial approximation where there is bit loss from subtraction in $\sinh(x)$ and $\tanh(x)$, and otherwise, use the mathematical definition of $\cosh(x)$;

$(x_{\mathrm{loss}}, x_c]$: use the mathematical definitions;

$(x_c, x_{\mathrm{ovfl}}]$: special handling near function overflow limits;

$(x_{\mathrm{ovfl}}, \infty)$: $\cosh(x) = \sinh(x) = \infty$, and $\tanh(x) = 1$.

Cody and Waite observe that premature overflow can be avoided by suitable translation of the argument, using the relation $\exp(x)/2 = \exp(x) \times \exp(-\log(2)) = \exp(x - \log(2))$. Unfortunately, because $\log(2)$ is a transcendental number, we have to approximate it, thereby introducing an argument error. Because the error-magnification factor of $\exp(x)$, and thus also of $\cosh(x)$ and $\sinh(x)$, is proportional to x (see **Table 4.1** on page 62), and x is large, that translation introduces large errors in the function, and is unsatisfactory.

Instead, we write $\exp(x)/2 = \exp(x)/2 \times (v/v) = v \exp(x - \log(v))/2$, and choose $\log(v)$ to be a constant slightly larger than $\log(2)$, but having the property that the number of integer digits in x plus the number of digits in $\log(v)$ does not exceed the significand size. The subtraction is then *exact*, and the argument translation introduces no additional error. Of course, we then have an additional error from the multiplication by $v/2$, but that should be at most one ulp, and independent of the size of x.

For bases that are powers of two, the value $45427/65536$ is a suitable choice for the constant $\log(v)$. That value needs only 16 bits, and the overflow limit for the exponential in single-precision arithmetic requires at most 8 integer bits for current and historical architectures, so their sum can be represented in the 24 significand bits available. We need these stored constants:

$$\begin{aligned} \log(v) &= 45427/65536, \\ v/2 - 1 &= 1.383\,027\,787\,960\,190\,263\,751\,679\,773\,082\,023\,374\,\ldots \times 10^{-5}, \\ 1/v^2 &= 0.249\,993\,085\,004\,514\,993\,356\,328\,792\,112\,262\,007\,\ldots. \end{aligned}$$

For base 10, assuming at least six significand digits, we use these values:

$$\begin{aligned} \log(v) &= 0.6932, \\ v/2 - 1 &= 5.282\,083\,502\,587\,485\,246\,917\,563\,012\,448\,599\,540\,\ldots \times 10^{-5}, \\ 1/v^2 &= 0.249\,973\,591\,674\,870\,159\,651\,646\,864\,161\,934\,786\,\ldots. \end{aligned}$$

It is imperative to use the value $v/2 - 1$ directly, instead of $v/2$, because the latter loses about five decimal digits.

The PORT library authors recognize the problem of premature overflow, and compute the hyperbolic cosine with simple Fortran code like this:

```
t = exp(abs(x / 2.0e0))
cosh = t * (0.5e0 * t) + (0.5e0 / t) / t
```

The argument scaling is only error free in binary arithmetic, and the factorization introduces for all arguments two additional rounding errors that could easily be avoided. The PORT library hyperbolic sine routines use a similar expression with a subtraction instead of an addition, but do so only for $|x| \geq 1/8$. Otherwise, the PORT library code sums the Taylor series to convergence, starting with the largest term, thereby losing another opportunity to reduce rounding error by adding the largest term last.

In order to preserve full precision of the constants, and avoid problems from wobbling precision in hexadecimal

arithmetic, for arguments near the function overflow limit, we compute the hyperbolic cosine and sine like this:

$$\begin{aligned}
u &= \exp(x - \log(v)),\\
y &= u + (1/v^2)(1/u),\\
z &= u - (1/v^2)(1/u),\\
\cosh(x) &= (v/2)\big(u + (1/v^2)(1/u)\big)\\
&= y + (v/2 - 1)y,\\
\sinh(x) &= (v/2)\big(u - (1/v^2)(1/u)\big)\\
&= z + (v/2 - 1)z.
\end{aligned}$$

For hexadecimal arithmetic, those formulas should be used even for moderate $|x|$ values, because if $\exp(x)$ has leading zero bits from hexadecimal normalization, $\exp(x - \log(v))$ is about half that size, has no leading zero bits, and retains full accuracy. When $\cosh(x)$ or $\sinh(x)$ have leading zero bits, they are computed from an exponential with a similar number of leading zero bits, so accuracy of the two hyperbolic functions is no worse. That special handling is easily done by reducing the limit x_{ovfl} for $\beta = 16$.

For large-magnitude arguments, $\tanh(x)$ quickly approaches its limits of ± 1. We could simply return those values as soon as $\exp(x) \pm \exp(-x)$ is the same as $\exp(x)$ to machine precision. However, to produce correct rounding, and setting of the *inexact* flag, at the limits, the return value must be computed as $\pm(1 - \tau)$ (Greek letter *tau*) for a suitable tiny number τ, such as the machine underflow limit, and that computation must be done at run time, rather than at compile time.

To find the cutoff for the limits in $\tanh(x)$, multiply the numerator and denominator by $\exp(-x)$, and then add and subtract $2\exp(-2x)$ in the numerator to find:

$$\begin{aligned}
\tanh(x) &= (1 - \exp(-2x))/(1 + \exp(-2x))\\
&= (1 + \exp(-2x) - 2\exp(-2x))/(1 + \exp(-2x))\\
&= 1 - 2/(\exp(2x) + 1).
\end{aligned}$$

The right-hand side is 1 to machine precision when the second term falls below half the negative machine epsilon, ϵ/β, where $\epsilon = \beta^{1-t}$ for t-digit arithmetic:

$$\begin{aligned}
\tfrac{1}{2}\epsilon/\beta &= \tfrac{1}{2}(\beta^{1-t}/\beta)\\
&= \tfrac{1}{2}\beta^{-t}\\
&= 2/(\exp(2x_c) + 1)\\
&\approx 2/\exp(2x_c),\\
\exp(2x_c) &= 4\beta^t,\\
x_c &= \tfrac{1}{2}(\log(4) + t\log(\beta)).
\end{aligned}$$

Similar considerations for $\cosh(x)$ and $\sinh(x)$ show that the $\exp(-x)$ terms can be ignored for x values larger than these:

$$x_c = \begin{cases} \tfrac{1}{2}(\log(2) + (t-1)\log(\beta)), & \text{for } \cosh(x),\\ \tfrac{1}{2}(\log(2) + t\log(\beta)), & \text{for } \sinh(x). \end{cases}$$

The cutoff values, x_c, can be easily computed from stored constants. Our result for $\tanh(x)$ differs slightly from the result $\frac{1}{2}(\log(2) + (t+1)\log(\beta))$ given by Cody and Waite [CW80, page 239], but the two are identical when $\beta = 2$. For other bases, their cutoff is slightly larger than needed.

For small-magnitude arguments, $\cosh(x)$ can be computed stably like this:

$$\begin{aligned}
w &= \begin{cases} \exp(|x|), & \text{usually},\\ \exp(-|x|), & \text{sometimes when } \beta = 16\text{: see text}, \end{cases}\\
\cosh(x) &= \tfrac{1}{2}(w + 1/w).
\end{aligned}$$

We have $\exp(|x|) \geq 1$, so the reciprocal of that value is smaller, and the rounding error of the division is minimized. However, when $\beta = 16$, we need to be concerned about wobbling precision. Examination of the numerical values of $\exp(|x|)$ show that it loses leading bits unless $|x|$ lies in intervals of the form $[\log(8 \times 16^k), \log(16 \times 16^k)]$, or roughly $[2.08, 2.77]$, $[4.85, 5.55]$, $[7.62, 8.32]$, ... Outside those intervals, the computed value of $\exp(-|x|)$ is likely to be more accurate than that of $\exp(|x|)$. However, the growth of the exponential soon makes the first term much larger than the second, so in our code, we use a block that looks similar to this:

```
#if B == 16
    if ( (FP(2.079442) < xabs) && (xabs < FP(2.772589)) )
        z = EXP(xabs);
    else
        z = EXP(-xabs);
#else
    z = EXP(xabs);
#endif

    result = HALF * (z + ONE / z);
```

For tiny arguments, the three functions can be computed from their Taylor series. From the first two terms of the series, and the formulas for the machine epsilons, we can readily find these cutoffs:

$$x_{\mathrm{TS}} = \begin{cases} \sqrt{6}\beta^{-t/2} & \text{for } \sinh(x) \text{ and } \tanh(x), \\ \sqrt{2\beta}\beta^{-t/2} & \text{for } \cosh(x). \end{cases}$$

For simplicity, Cody and Waite use a smaller value, $\beta^{-t/2}$, for each of those cutoffs, and they keep only the first term of each of the series. However, in the mathcw library implementation, we sum two terms to set the *inexact* flag and obey the rounding mode.

For arguments of intermediate size, because of significance loss in the subtraction, $\sinh(x)$ and $\tanh(x)$ require rational polynomial approximations. In binary arithmetic, bit loss sets in as soon as $\exp(-x)$ exceeds $\frac{1}{2}\exp(x)$, which we can rearrange and solve to find

$$\begin{aligned} x_{\mathrm{loss}} &= \tfrac{1}{2}\log(2) \\ &\approx 0.347. \end{aligned}$$

However, Cody and Waite base their rational approximations on the wider interval $[0, 1]$, so we do as well. The polynomials provide as much accuracy as the exponential function, and are faster to compute, although they could be slightly faster if they were fitted on the smaller interval $[0, 0.347]$, reducing the total polynomial degree by about one.

If the macro `USE_FP_T_KERNEL` is not defined when the code is compiled, the next higher precision is normally used for $\frac{1}{2}(\exp(x) \pm \exp(-x))$ in order to almost completely eliminate rounding errors in the computed results, which for x in $[1, 10]$ otherwise can approach one ulp with *round-to-nearest* arithmetic. Experiments with Chebyshev and minimax polynomial fits over that interval demonstrate that polynomials of large degree defined on many subintervals would be necessary to achieve comparable accuracy, at least until we can identify a better approximating function that can be easily computed to effective higher precision using only working-precision arithmetic. A variant of the exponential function that returns a rounded result, along with an accurate estimate of the rounding error, would be helpful for improving the accuracy of the hyperbolic functions.

We omit the code for the hyperbolic functions, but **Figure 12.2** on the following page shows measured errors in the values returned by our implementations. The largest errors are 0.95 ulps (`cosh()`), 1.00 ulps (`sinhf()`), and 1.21 ulps (`tanh()`), but the results from the decimal functions `tanhdf()` and `tanhd()` are almost always correctly rounded.

12.2 Improving the hyperbolic functions

Plots of the errors in the hyperbolic cosine and sine functions show that the difficult region is roughly $[1, 10]$, where the roundings from the reciprocation and addition or subtraction in $\frac{1}{2}(\exp(x) \pm 1/\exp(x))$ have their largest effect.

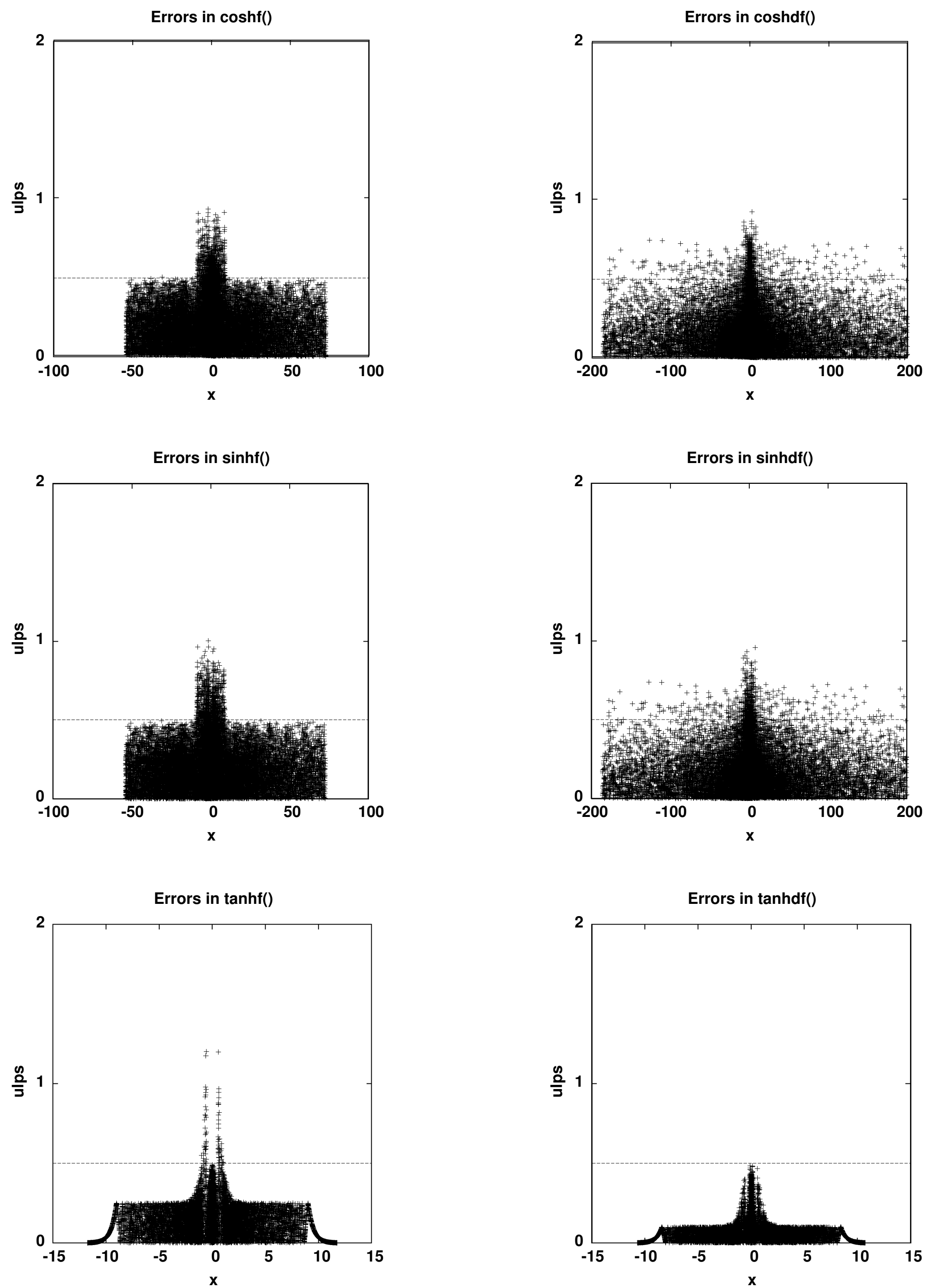

Figure 12.2: Errors in the single-precision hyperbolic functions for binary and decimal arithmetic *without* use of intermediate higher precision. Plots for the double-precision functions are similar, and thus, not shown.
The mathcw library code for $\cosh(x)$ and $\sinh(x)$ normally uses the next higher precision for the exponentials. When that is possible, for randomly chosen arguments, fewer than 1 in 10^7 function results are incorrectly rounded.

When higher precision is available, those errors are negligible. However, some platforms do not supply a usable `long double` type in C, and for all platforms, we would like to improve the accuracy of the functions at the highest available precision. Doing so requires computing the exponential function in even higher precision.

Because the Taylor series of the exponential function begins $1 + x + x^2/2! + \ldots$, when x is small, we can achieve that higher precision by delaying addition of the leading term. Fortunately, we do not need to reimplement the exponential function, because we already have `expm1(x)` to compute the value of $\exp(x) - 1$ accurately. For larger x values, we can obtain small arguments as differences from tabulated exact arguments like this:

$$\begin{aligned}
e^x &= e^{x-c_k} e^{c_k} \\
&= (1 + \mathrm{expm1}(x - c_k)) \times (\exp(c_k)_{\mathrm{hi}} + \exp(c_k)_{\mathrm{lo}}) \\
&= \exp(c_k)_{\mathrm{hi}} + (\mathrm{expm1}(x - c_k) \times (\exp(c_k)_{\mathrm{hi}} + \exp(c_k)_{\mathrm{lo}}) + \exp(c_k)_{\mathrm{lo}}) \\
&= \exp(c_k)_{\mathrm{hi}} + \mathrm{fma}(\mathrm{expm1}(x - c_k), \exp(c_k)_{\mathrm{hi}}, \mathrm{fma}(\mathrm{expm1}(x - c_k), \exp(c_k)_{\mathrm{lo}}, \exp(c_k)_{\mathrm{lo}})) \\
&= \exp(x)_{\mathrm{hi}} + \exp(x)_{\mathrm{lo}}.
\end{aligned}$$

We need to choose enough c_k values on $[1, 10]$ to ensure that $1 + \mathrm{expm1}(x - c_k)$ represents $\exp(x - c_k)$ to a few extra digits, and so that k and c_k can be found quickly. Evenly spacing the c_k values $1/8$ unit apart means that $|x - c_k| < 1/16$, for which $\mathrm{expm1}(x - c_k)$ lies in $[-0.061, +0.065]$, giving us roughly two extra decimal digits at the expense of storing a table of 144 high and low parts of $\exp(c_k)$. For decimal arithmetic, we use a more natural spacing of $1/10$. The fused multiply-add operations allow accurate reconstruction of high and low parts of $\exp(x)$, such that their sum gives us about two more digits. That technique does not scale well: to get one more digit, we need a spacing of $1/64$, and a table of 1152 parts.

Once we have the higher-precision exponential, we can find the hyperbolic functions by careful evaluation of these expressions:

$$\begin{aligned}
H &= \exp(x)_{\mathrm{hi}}, \\
L &= \exp(x)_{\mathrm{lo}}, \\
\cosh(x) &= \tfrac{1}{2}((H + L) + 1/(H + L)), \\
\sinh(x) &= \tfrac{1}{2}((H + L) - 1/(H + L)).
\end{aligned}$$

We can recover the error, e, in the division like this:

$$\begin{aligned}
D &= 1/(H + L), && \textit{exact}, \\
d &= \mathrm{fl}(D), && \textit{approximate}, \\
D &= d + e, && \textit{exact}, \\
e &= D - d, && \textit{exact}, \\
&= (D \times (H + L) - d \times (H + L))/(H + L), && \textit{exact}, \\
&= D \times (1 - d \times (H + L)), && \textit{exact}, \\
&\approx d \times (1 - d \times (H + L)), && \textit{approximate}, \\
&\approx d \times (\mathrm{fma}(-d, H, 1) - d \times L).
\end{aligned}$$

Finally, we reconstruct the hyperbolic functions

$$\begin{aligned}
\cosh(x) &= \tfrac{1}{2}(H + (d + (L + e))), \\
\sinh(x) &= \tfrac{1}{2}(H + (-d + (L - e))).
\end{aligned}$$

using the accurate summation function family, `VSUM()`, to sum the four terms in order of increasing magnitudes.

Although that technique does not require a higher-precision data type, it gains only about two extra decimal digits in the exponential. We therefore use it only for the hyperbolic functions of highest available precision.

Tests of the code that implements that algorithm shows that it lowers the incidence of incorrect rounding of the hyperbolic cosine and sine to about 10^{-4}, with a maximum error of about 0.53 ulps. By contrast, using the next higher precision lowers the rate of incorrect rounding to below 10^{-7}, with a maximum error of about 0.51 ulps.

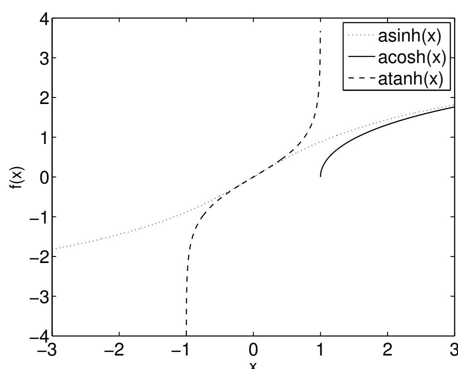

Figure 12.3: Inverse hyperbolic functions near the origin. An artifact of the MATLAB graphing software prevents display of the upper part of $\operatorname{atanh}(x)$.

12.3 Computing the hyperbolic functions together

Just as we found it efficient in **Section 11.8** on page 320 to compute the trigonometric cosine and sine together, we can do the same thing with their hyperbolic companions. The file `shchx.h` defines the function `sinhcosh(x, &sh, &ch)` and its companions with the usual suffixes for other data types.

The code in `shchx.h` is a straightforward interleaving of that from `coshx.h` and `sinhx.h`. They use the same algorithms, guaranteeing identical function results. When both the hyperbolic cosine and sine are needed, `sinhcosh()` is the recommended, and faster, way to compute them.

12.4 Inverse hyperbolic functions

The inverse hyperbolic functions, illustrated in **Figure 12.3** through **Figure 12.6** on page 350, all bear simple relations to the logarithm function that make them straightforward to compute:

$$\begin{aligned}
\operatorname{acosh}(x) &= \log(x + \sqrt{x^2 - 1}), && \text{for } x \geq 1, \\
\operatorname{asinh}(x) &= \operatorname{sign}(x) \log(|x| + \sqrt{x^2 + 1}), && \text{for } x \text{ in } (-\infty, +\infty), \\
\operatorname{atanh}(x) &= \tfrac{1}{2} \log(1 + 2x/(1 - x)), && \text{for } x \text{ in } [-1, +1].
\end{aligned}$$

The primary computational issues are premature overflow of x^2, and loss of significance from subtractions in the square roots and from evaluation of the logarithm for arguments near 1.0. For the latter, the $\operatorname{log1p}(x)$ function provides the required solution.

The inverse hyperbolic functions have these Taylor series:

$$\begin{aligned}
\operatorname{acosh}(1 + d) &= \sqrt{2d}\,(1 - (1/12)d + (3/160)d^2 - (5/896)d^3 + \\
&\qquad (35/18\,432)d^4 - (63/90\,112)d^5 + \cdots) \\
\operatorname{asinh}(x) &= x - (1/6)x^3 + (3/40)x^5 - (5/112)x^7 + (35/1152)x^9 - \\
&\qquad (63/2816)x^{11} + (231/13\,312)x^{13} - \cdots \\
\operatorname{atanh}(x) &= x + (1/3)x^3 + (1/5)x^5 + \cdots + (1/(2k+1))x^{2k+1} + \cdots
\end{aligned}$$

For a truncated series, we can reduce rounding error by introducing a common denominator to get exactly representable coefficients. For example, a five-term series for the inverse hyperbolic tangent can be evaluated with this code fragment from `atanhx.h`:

```
/* atanh(x) ~= (((945+(315+(189+(135+105 x^2)x^2)x^2)x^2)x^2)x)/945 */
```

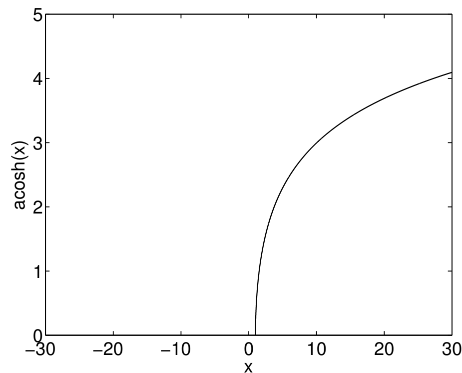

Figure 12.4: Inverse hyperbolic cosine. The function $\operatorname{acosh}(x)$ is defined only for x on the interval $[+1, \infty)$, and is slow to approach the limit $\lim_{x\to\infty} \operatorname{acosh}(x) \to \infty$. In the IEEE 754 32-bit format, the largest finite $\operatorname{acosh}(x)$ is less than 89.5, in the 64-bit format, about 710.5, and in the 80-bit and 128-bit formats, just under 11357.3.

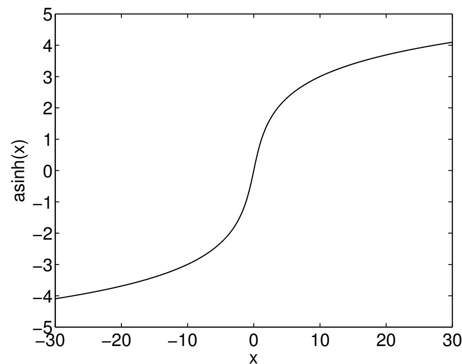

Figure 12.5: Inverse hyperbolic sine. The function $\operatorname{asinh}(x)$ is defined for x on the interval $(-\infty, +\infty)$, and is slow to approach the limit $\lim_{x\to\infty} \operatorname{asinh}(x) \to \infty$. In the IEEE 754 32-bit format, the largest finite $\operatorname{asinh}(x)$ is less than 89.5, in the 64-bit format, about 710.5, and in the 80-bit and 128-bit formats, just under 11357.3.

```
x_sq = x * x;
sum = (FP(105.0)       ) * x_sq;
sum = (FP(135.0) + sum) * x_sq;
sum = (FP(189.0) + sum) * x_sq;
sum = (FP(315.0) + sum) * x_sq;
sum *= xabs / FP(945.0);
result = xabs + sum;
```

Cody and Waite do not treat the inverse hyperbolic functions, so we base our code for those functions on the approach used in the Sun Microsystems' fdlibm library, but we also use the Taylor-series expansions to ensure correct behavior for small arguments. After the argument-range limits have been checked, the algorithms look like this for base β and a t-digit significand:

$$\operatorname{acosh}(x) = \begin{cases} \log(2) + \log(x) & \text{for } x > \sqrt{2\beta^t}, \\ \log(2x - 1/(\sqrt{x^2-1} + x)) & \text{for } x > 2, \\ \operatorname{log1p}(s + \sqrt{2s + s^2}) & \text{for } s = x - 1, \end{cases}$$

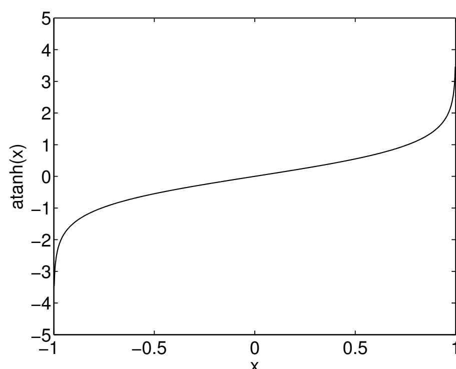

Figure 12.6: Inverse hyperbolic tangent. The function $\operatorname{atanh}(x)$ is defined only for x on the interval $[-1, +1]$, and has poles at $x = \pm 1$ that are only approximated in this plot.

$$\operatorname{asinh}(x) = \begin{cases} \operatorname{sign}(x)(\log(2) + \log(x)) & \textit{for } x > \sqrt{2\beta^{t-1}}, \\ \operatorname{sign}(x)\log(2|x| + 1/(|x| + \sqrt{x^2+1})) & \textit{for } |x| > 2, \\ x & \textit{if } \operatorname{fl}(x^2+1) = 1, \\ \operatorname{sign}(x)\operatorname{log1p}(|x| + x^2/(1+\sqrt{1+x^2})) & \textit{otherwise,} \end{cases}$$

$$\operatorname{atanh}(x) = \begin{cases} -\operatorname{atanh}(-x) & \textit{for } x < 0, \\ \frac{1}{2}\operatorname{log1p}(2x/(1-x)) & \textit{for } x \geq 1/2, \\ \frac{1}{2}\operatorname{log1p}(2x + 2x^2/(1-x)) & \textit{for } x \textit{ in } [0, 1/2). \end{cases}$$

Notice that for `acosh()` and `asinh()`, $\log(2x)$ is computed as $\log(2) + \log(x)$ to avoid premature overflow for large x. The cutoff $\sqrt{2\beta^t}$ is the value above which $\operatorname{fl}(x^2 - 1) = \operatorname{fl}(x^2)$ to machine precision, and $\sqrt{2\beta^{t-1}}$ is the corresponding cutoff that ensures $\operatorname{fl}(x^2 + 1) = \operatorname{fl}(x^2)$.

At most seven floating-point operations are needed beyond those for the logarithm and square-root functions, so as long as those two functions are accurate, the inverse hyperbolic functions are expected to be as well.

Figure 12.7 shows measured errors in the values returned by our implementations of the inverse hyperbolic functions when arithmetic is restricted to working precision. The largest errors found are 0.91 ulps (`acoshf()`), 0.88 ulps (`asinh()`), and 0.85 ulps (`atanhd()`). For the `ATANH()` family, computing just the argument of `LOG1P()` in the interval $[x_{TS}, \frac{1}{2})$ in higher precision, and then casting it to working precision, reduces the maximum error by about 0.10 ulps. Code to do so is present in the file `atanhx.h`, but is disabled because we can do better.

The errors can be almost completely eliminated by excursions to the next higher precision in selected argument ranges. When possible, we do so for acosh() and asinh() for argument magnitudes in $(2, 70]$ for a nondecimal base, and in $[x_{TS}, 2]$ for all bases. We do so as well for atanh() for argument magnitudes in $[x_{TS}, 1)$. The Taylor-series region covers 16% to 21% of the argument range for the single-precision functions `atanhf()` and `atanhdf()`. As with the trigonometric functions and the ordinary hyperbolic functions, compile-time definition of the `USE_FP_T_KERNEL` macro prevents the use of higher precision.

When higher precision is not available, the code in `atanhx.h` uses separate rational polynomial approximations of the form $\operatorname{atanh}(x) \approx x + x^3\mathcal{R}(x^2)$ on the intervals $[0, \frac{1}{2}]$ and $[\frac{1}{2}, \frac{3}{4}]$ to further reduce the errors. In binary arithmetic, that has better accuracy than the logarithm formulas. In decimal arithmetic, the results are almost always correctly rounded.

12.5 Hyperbolic functions in hardware

Although the Intel IA-32 architecture provides limited hardware support for the square root, exponential, logarithm, and trigonometric functions, it has no specific instructions for the hyperbolic functions. However, they can be computed with hardware support from their relations to logarithms tabulated at the start of **Section 12.4** on page 348, as

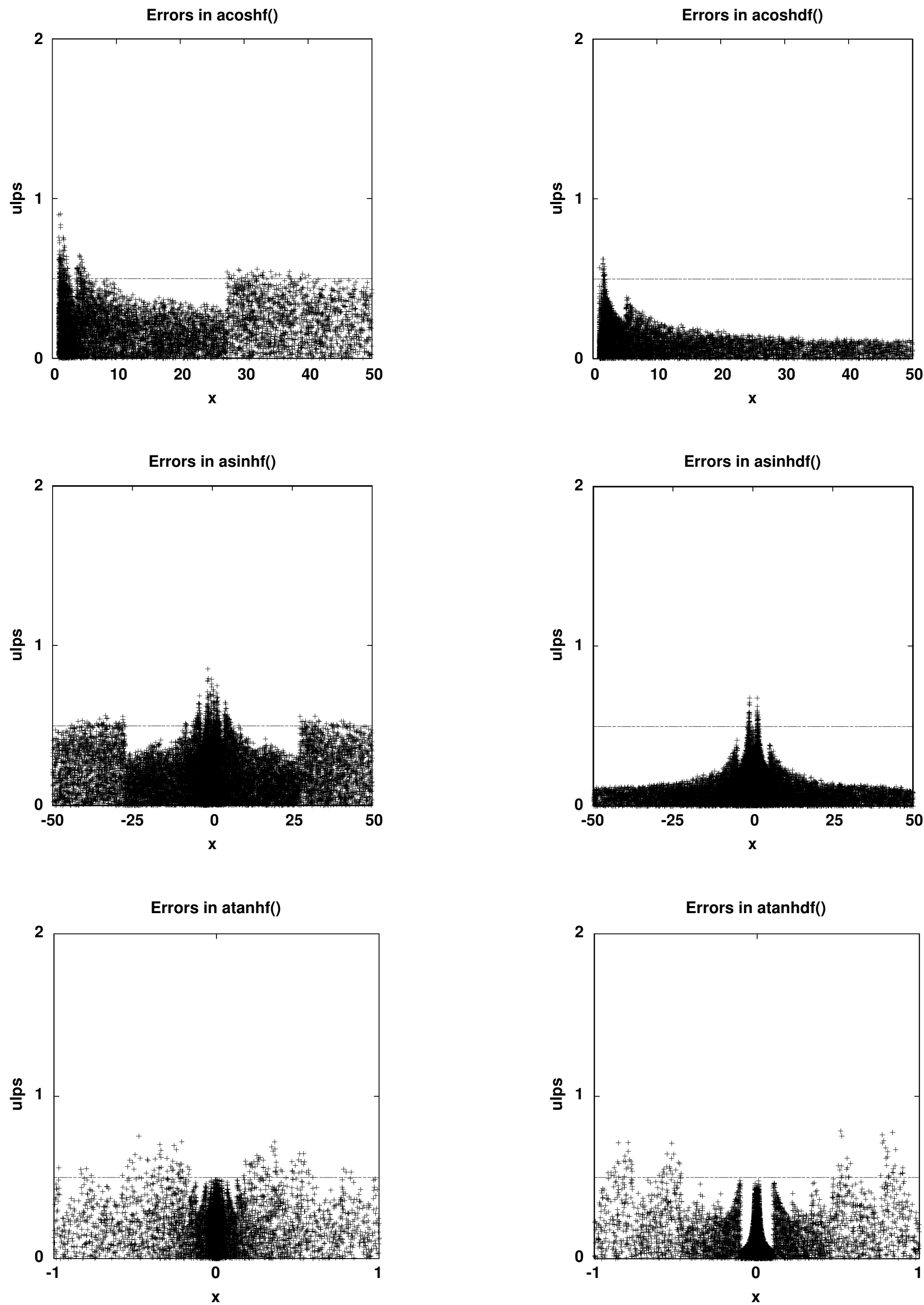

Figure 12.7: Errors in the single-precision hyperbolic functions for binary and decimal arithmetic *without* use of intermediate higher precision. Plots for the double-precision functions are similar, and thus, not shown.
The mathcw library code for those functions normally uses the next higher precision for the logarithms and square roots. When that is possible, for randomly chosen arguments, fewer than 1 in 10^7 function results are incorrectly rounded.

Table 12.1: Maximum errors in units in the last place (ulps), and percentages of results that are correctly rounded, for hyperbolic functions in the PORT library on AMD64.

Function	32-bit		64-bit	
	max err	OK	max err	OK
`acosh()`	512.4	95.9%	5619.8	89.7%
`asinh()`	4.7	94.3%	4.8	94.5%
`atanh()`	2.2	80.8%	3.8	76.5%
`cosh()`	1.9	72.1%	1.9	70.2%
`sinh()`	10.7	68.3%	9.3	66.9%
`tanh()`	3.5	82.6%	3.7	81.6%

Table 12.2: Maximum errors in units in the last place (ulps), and percentages of results that are correctly rounded, for inverse hyperbolic functions in the FNLIB library on AMD64.

Function	32-bit		64-bit	
	max err	OK	max err	OK
`acosh()`	10^9	0.0%	1790.0	94.8%
`asinh()`	10^7	7.3%	1.1	96.6%
`atanh()`	10^5	34.4%	1.0	85.6%

long as the argument x is not so large that x^2 overflows. A better approach is to employ the three alternate forms involving logarithms of modified arguments. The mathcw library code does not use inline IA-32 assembly code for the hyperbolic functions, but it can do so for the needed logarithms, at the slight expense of an additional function call.

The Motorola 68000 has `fatanh`, `fcosh`, `fsinh`, and `ftanh` instructions. No single instructions exist for the inverse hyperbolic cosine and sine, but they can be computed with the help of the 68000 hardware instructions `flogn` and `flognp1` for the logarithms. The mathcw library uses those hardware instructions when it is built on a 68000 system with a compiler that supports inline assembly code, and the preprocessor symbol `USE_ASM` is defined. The only 68000 machine available to this author during the library development lacks support for the `long double` data type, so it has been possible to test the hyperbolic-function hardware support only for the 32-bit and 64-bit formats.

12.6 Summary

Simplistic application of the mathematical definitions of the hyperbolic functions and their inverses in terms of the exponential, logarithm, and square root does not lead to acceptable accuracy in software that uses floating-point arithmetic of fixed precision.

Plots of the errors of those functions as implemented in the PORT library [FHS78b] show reasonable accuracy, except for the `acosh()` pair, as summarized in **Table 12.1**. Errors in the `acosh()` functions rise sharply above 2 ulps for x in $[1, 1.1]$, and the errors increase as x decreases.

Results of tests of the inverse hyperbolic functions in the earlier FNLIB library [Ful81b, Ful81a] are shown in **Table 12.2**. Although 85% or more of the function values are correctly rounded, the tests find huge errors for $x \approx 1$ in the inverse hyperbolic cosine and sine, and for $x \approx 0.5$ in the inverse hyperbolic tangent.

Tests with the newer software technology in the GNU math library show worst case errors of 1.32 ulps in `tanhf()` and `tanh()`, and 1.11 ulps for the other hyperbolic functions. From 88% (`tanhf()`) to 98% (`acosh()`) of the results are correctly rounded. Tests of the fdlibm library show similar behavior.

Our implementations of the hyperbolic functions and their inverses provide results that are almost always correctly rounded, and have accuracy comparable to our code for the related exponential and logarithm functions. Achieving that goal requires considerable care in algorithm design, and in programming.