The CORE-MATH sinh is correctly rounded

***Abstract*—We prove that the CORE-MATH binary64 hyperbolic sine function is correctly rounded.**

***Index Terms*—IEEE 754, binary64 format, correct rounding.**

We consider the CORE-MATH `sinh.c` file from commit `2d84534`. Since `sinh` is an odd function, and the approximations used are odd too, it suffices to prove the code is correctly rounded for $x \geq 0$.

I. Branch $0 \leq x < x_0$

For $0 \leq x < x_0$ with $x_0 =$ `0x1.7137449123ef7p-26`, the code simply returns $\mathrm{fma}(x, 2^{-55}, x)$.

Lemma 1: For $0 \leq x < x_0$, $\mathrm{fma}(x, 2^{-55}, x)$ yields the correct rounding of $\sinh(x)$, whatever the rounding mode.

Proof: First, for $x = \pm 0$, it yields ± 0 with the correct sign. Assume now $0 < x < x_0$. For $0 < x < 0.1$, we have:

$$x < \sinh(x) < x + \frac{x^3}{6} + 0.01x^5.$$

Let $x_1 =$ `0x1.3988e1409212ep-26` $\approx 1.83 \cdot 10^{-8}$. For $0 < x \leq x_1$, $x^2/6 + 0.01x^4 < 2^{-54}$, thus $x < \sinh(x) < x + 2^{-54}x$, and since there is no rounding boundary in $(x, x + 2^{-54}x)$, $\sinh(x)$ rounds to the same value as $x + 2^{-55}x$, i.e., x except for rounding upward where it rounds to $\mathrm{nextabove}(x)$. For $x_1 < x < x_0$, the term $x^3/6 + 0.01x^5$ lies in $[2^{-80}, 2^{-79})$, and $\mathrm{ulp}(x) = 2^{-78}$, thus $x^3/6 + 0.01x^5 < \frac{1}{2}\mathrm{ulp}(x)$, and the same rounding rule as above applies. ■

Note that x_0 is optimal, since $\mathrm{fma}(x, 2^{-55}, x)$ does not yield the correct rounding for $x = x_0$ and rounding to nearest: it yields x_0 instead of $\mathrm{nextabove}(x_0)$. And if we replace $\mathrm{fma}(x, 2^{-55}, x)$ by $\mathrm{fma}(x, m, x)$ with a magic constant $m > 2^{-55}$ so that it rounds to $\mathrm{nextabove}(x_0)$ for $x = x_0$, for example $m = 2^{-53}$, then for $x = x_0/2$, $\mathrm{fma}(x, m, x)$ will round to $\mathrm{nextabove}(x_0/2)$, which is wrong.

II. Branch $x_0 \leq x < 1/4$

In this branch, an odd degree-11 polynomial $q(x) = x + c_0x^3 + \cdots + c_4x^{11}$ is used. We use the following Sollya code to bound the mathematical error $|q(x) - \sinh(x)|$ in terms of x^3 [1]:

```
prec = 64;
d=[0x1.7137449123ef7p-26,0.25];
c0=0x1.5555555555555p-3;
c1=0x1.111111111151ep-7;
c2=0x1.a01a019d0c767p-13;
c3=0x1.71de444a96e11p-19;
c4=0x1.ae8465375242p-26;
q=x+c0*x^3+c1*x^5+c2*x^7+c3*x^9+c4*x^11;
dirtyinfnorm((q-sinh(x))/x^3, d);
9.25185853854214036e-18
```

This proves $|q(x) - \sinh(x)| < 2^{-56.584}x^3$ on the whole range $[x_0, 1/4)$,

interval	s
$[x_0, 2^{-25}]$	$10930 \cdot 2^{-70}$
$[2^{-25}, 2^{-24}]$	$10930 \cdot 2^{-70}$
$[2^{-24}, 2^{-23}]$	$10930 \cdot 2^{-70}$
$[2^{-23}, 2^{-22}]$	$10930 \cdot 2^{-70}$
$[2^{-22}, 2^{-21}]$	$10930 \cdot 2^{-70}$
$[2^{-21}, 2^{-20}]$	$10930 \cdot 2^{-70}$
$[2^{-20}, 2^{-19}]$	$10930 \cdot 2^{-70}$
$[2^{-19}, 2^{-18}]$	$10930 \cdot 2^{-70}$
$[2^{-18}, 2^{-17}]$	$10930 \cdot 2^{-70}$
$[2^{-17}, 2^{-16}]$	$10930 \cdot 2^{-70}$
$[2^{-16}, 2^{-15}]$	$10930 \cdot 2^{-70}$
$[2^{-15}, 2^{-14}]$	$10930 \cdot 2^{-70}$
$[2^{-14}, 2^{-13}]$	$10930 \cdot 2^{-70}$
$[2^{-13}, 2^{-12}]$	$10930 \cdot 2^{-70}$
$[2^{-12}, 2^{-11}]$	$10930 \cdot 2^{-70}$
$[2^{-11}, 2^{-10}]$	$10923 \cdot 2^{-70}$
$[2^{-10}, 2^{-9}]$	$10923 \cdot 2^{-70}$
$[2^{-9}, 2^{-8}]$	$10915 \cdot 2^{-70}$
$[2^{-8}, 2^{-7}]$	$10893 \cdot 2^{-70}$
$[2^{-7}, 2^{-6}]$	$10795 \cdot 2^{-70}$
$[2^{-6}, 2^{-5}]$	$10413 \cdot 2^{-70}$
$[2^{-5}, 2^{-4}]$	$8946 \cdot 2^{-70}$
$[2^{-4}, 2^{-3}]$	$4111 \cdot 2^{-70}$
$[2^{-3}, 2^{-2}]$	$2686 \cdot 2^{-70}$

TABLE I
FOR SEVERAL RANGES COVERING $[x_0, 1/4)$, SOLLYA BOUND s SUCH THAT $|q(x) - \sinh(x)| < s \cdot x^3$.

Algorithm FastPath details how the approximation of $q(x)$ is performed. The rounding test is performed at line 13. (In the CORE-MATH code, ℓ_b and u_b design the rounded values.) If it return FAIL, the accurate path is called.

Algorithm 1 (FastPath)

```
Input: $x$ a binary64 number, $x_0 \leq x < 1/4$
 1: $c_0 \leftarrow$ 0x1.5555555555555p-3
 2: $c_1 \leftarrow$ 0x1.111111111151ep-7
 3: $c_2 \leftarrow$ 0x1.a01a019d0c767p-13
 4: $c_3 \leftarrow$ 0x1.71de444a96e11p-19
 5: $c_4 \leftarrow$ 0x1.ae8465375242p-26
 6: $x_2 \leftarrow \circ(x \cdot x)$,    $x_3 \leftarrow \circ(x_2 \cdot x)$
 7: $x_4 \leftarrow \circ(x_2 \cdot x_2)$
 8: $p_0 \leftarrow \circ(c_0 + \circ(x_2 c_1))$,    $p_2 \leftarrow \circ(c_2 + \circ(x_2 c_3))$
 9: $p_2' \leftarrow \circ(p_2 + \circ(x_4 c_4))$,    $p_0' \leftarrow \circ(p_0 + \circ(x_4 p_2'))$
10: $p \leftarrow \circ(x_3 p_0')$
11: $e = \circ(\epsilon x_3)$ where $\epsilon = 33 \cdot 2^{-57}$
12: $\ell_b = x + \circ(p - e)$,    $u_b = x + \circ(p + e)$
13: if $\circ(\ell_b) = \circ(u_b)$ return $\circ(\ell_b)$ else return FAIL
```

Theorem 1: For $x_0 \leq x < 1/4$, let ℓ_b and u_b the values computed at line 12 of Algorithm FastPath. If ℓ_b and u_b round to the same value, it is the correct rounding of $\sinh(x)$.

Proof: We use the following Gappa program, with version 1.7.0 of Gappa, where we substitute RND by all four rounding modes, XMIN and XMAX by interval bounds, and BND by the Sollya bound [2].

```
@rnd = float<ieee_64,RND>;
c0 = 0x1.5555555555555p-3;
c1 = 0x1.111111111151ep-7;
c2 = 0x1.a01a019d0c767p-13;
c3 = 0x1.71de444a96e11p-19;
c4 = 0x1.ae8465375242p-26;
x2 = x*x;
x2r rnd= x*x;
x3 = x2*x;
x3r rnd= x2r*x;
x4 = x2*x2;
x4r rnd= x2r*x2r;
p0 = c0+x2*c1;
p0r rnd= c0 + x2r*c1;
p2 = c2+x2*c3;
p2r rnd= c2 + x2r*c3;
pp2 = p2 + x4*c4;
pp2r rnd= p2r + x4r*c4;
pp0 = p0 + x4*pp2;
pp0r rnd= p0r + x4r*pp2r;
p = x3*pp0;
q = x + p;
pr rnd= x3r*pp0r;
eps = 0x2.1p-53;
er rnd= eps*x3r;
lb = x + rnd(pr-er);
ub = x + rnd(pr+er);
{ x in [XMIN,XMAX] -> (q-lb)/x3 >= BND }
```

For example, for the interval $[x_0, 2^{-25}]$, this Gappa program proves:

$$q(x) - \ell_b \geq 10930 \cdot 2^{-70} x^3.$$

Since in this interval, the mathematical error $|q(x) - \sinh(x)|$ is less than $10930 \cdot 2^{-70} x^3$ (cf. Table I), this proves $\ell_b < \sinh(x)$. A similar program checks the goal `(ub-q)/x3 >= 10930b-70`, which proves $\sinh(x) < u_b$. For all intervals from Table I, we have proven both $(q - \ell_b)/x^3 \geq s$ and $(u_b - q)/x^3 \geq s$ where s is the corresponding mathematical error bound computed by Sollya. It follows $\ell_b \leq \sinh(x) \leq u_b$ and by monotonicity of the rounding, $\circ(\ell_b) \leq \circ(\sinh(x)) \leq \circ(u_b)$. ■

In fact, we can show still with Gappa that ϵ can be lowered to $29 \cdot 2^{-57}$ for the left bound ℓ_b, and to $32 \cdot 2^{-57}$ for the right bound u_b for $x < 2^{-15}$. We don't know if $\epsilon = 32 \cdot 2^{-57}$ works for the right bound and $2^{-15} \leq x < 2^{-2}$.

A. A potential bug in a previous version

In commit 89539bd of CORE-MATH, a different polynomial was used in the fast path for $x_0 \leq x < 1/4$, with error tolerance $\epsilon = 25 \cdot 2^{-57}$ in the rounding test. Consider $x =$ `0x1.71c5b3515d069p-8`. For that value and rounding toward zero, we got a value $u_b = x + \circ(p + e)$ which was smaller than $\sinh(x)$, which defeats the logic of the rounding test: we should always have $\ell_b \leq \sinh(x) \leq u_b$, so that $\circ(\ell_b) \leq \circ(\sinh(x)) \leq \circ(u_b)$, and if $\circ(\ell_b) = \circ(u_b)$, we can conclude this equals $\circ(\sinh(x))$.

We found a similar example with $u_b < \sinh(x)$ for $x =$ `0x1.01cb85ecb4ea9p-9` and $\epsilon = 27 \cdot 2^{-57}$, still with rounding toward zero. Thus we need $\epsilon \geq 28 \cdot 2^{-57}$, assuming ϵ is an integer multiple of 2^{-57}.

B. Accurate path for $x_0 \leq x < 1/4$

The accurate path in this branch corresponds to the as_sinh_zero routine in the CORE-MATH code. An odd degree-17 polynomial $r(x) = x + c_0 x^3 + \cdots + c_7 x^{17}$ is used, with double-double coefficients for degrees 3 to 11, and double coefficients for degrees 13 to 17:

$$\begin{aligned}
c_0 &= \texttt{0x1.5555555555555p-3} + \texttt{0x1.555555555552fp-57}\\
c_1 &= \texttt{0x1.1111111111111p-7} + \texttt{0x1.11111115cf00dp-63}\\
c_2 &= \texttt{0x1.a01a01a01a01ap-13} + \texttt{0x1.a0011c925b85cp-73}\\
c_3 &= \texttt{0x1.71de3a556c734p-19} + \texttt{0x1.b4e2835532bcdp-73}\\
c_4 &= \texttt{0x1.ae64567f54482p-26} - \texttt{0x1.defcf17a6ab79p-81}\\
c_5 &= \texttt{0x1.6124613aef206p-33}\\
c_6 &= \texttt{0x1.ae7f36beea815p-41}\\
c_7 &= \texttt{0x1.95785063cd974p-49}
\end{aligned}$$

The relative error is bounded by Sollya on $[x_0, 1/4]$ with the following program:

```
prec = 64;
d=[0x1.7137449123ef7p-26,0.25];
c3=0x1.5555555555555p-3+0x1.555555555552fp-57;
c5=0x1.1111111111111p-7+0x1.11111115cf00dp-63;
c7=0x1.a01a01a01a01ap-13+0x1.a0011c925b85cp-73;
c9=0x1.71de3a556c734p-19-0x1.b4e2835532bcdp-73;
c11=0x1.ae64567f54482p-26-0x1.defcf17a6ab79p-81;
c13=0x1.6124613aef206p-33;
c15=0x1.ae7f36beea815p-41;
c17=0x1.95785063cd974p-49;
r=x+c3*x^3+c5*x^5+c7*x^7+c9*x^9
 +c11*x^11+c13*x^13+c15*x^15+c17*x^17;
dirtyinfnorm(r/sinh(x)-1, d);
3.8682632661548095912e-33
```

This proves:

$$|r(x)/\sinh(x) - 1| \leq 2^{-107.671}. \tag{1}$$

To avoid duplicating the coefficients, we denote them $c_i = c_{i,h} + c_{i,\ell}$ for double-double coefficients in Algorithm AccuratePath below. We denote roundings by $\circ(\cdot)$, where $\circ$ denotes the current rounding mode; and $\mathrm{fma}(a, b, c)$ denotes $\circ(ab + c)$. At line 2 of AccuratePath, y_2 is a double approximation of $c_5 x^2 + c_6 x^4 + c_7 x^6$, then after the polydd call at line 3, $y_1 + y_2$ is a double-double approximation of $c_0 + c_1 x^2 + \cdots + c_7 x^{14}$, after line 4 it is a double-double approximation of $c_0 x + \cdots + c_7 x^{15}$, after line 5 it is a double-double approximation of $c_0 x^3 + \cdots + c_7 x^{17}$, and the last two

Algorithm 2 (AccuratePath)

```
Input: $x$ a binary64 number, $x_0 \le x < 1/4$
Output: $y_0 + y_1 + y_2$ approximating $\sinh(x)$
1: $x_2 \leftarrow \circ(x \cdot x)$,    $x_{2\ell} \leftarrow \mathrm{fma}(x, x, -x_2)$
2: $y_2 \leftarrow \circ(x_2 \cdot \circ(c_5 + \circ(x_2 \cdot \circ(c_6 + \circ(x_2 \cdot c_7)))))$
3: $y_1, y_2 \leftarrow \mathrm{polydd}(x_2, x_{2\ell}, 5, y_2)$
4: $y_1, y_2 \leftarrow \mathrm{mulddd}(y_1, y_2, x)$
5: $y_1, y_2 \leftarrow \mathrm{muldd}(y_1, y_2, x_2, x_{2\ell})$
6: $y_0, y_1 \leftarrow \mathrm{fasttwosum}(x, y_1)$
7: $y_1, y_2 \leftarrow \mathrm{fasttwosum}(y_1, y_2)$
```

lines add the x term. The routines `polydd`, `fasttwosum`, `muldd` and `mulddd` are described below.

Algorithm 3 (polydd)

```
Input: $x = x_h + x_\ell$ double-double, $n$ integer, $r$ binary64
Output: $h+\ell$ approximating $c_0+c_1 \cdot x^2+\cdots+(c_{n-1}+r)x^{2n-2}$
1: $h, t \leftarrow \mathrm{fasttwosum}(c_{n-1,h}, r)$
2: $\ell \leftarrow \circ(t + c_{n-1,\ell})$
3: for $i$ from $n-2$ downto 0
4:     $h, \ell \leftarrow \mathrm{muldd}(x_h, x_\ell, h, \ell)$
5:     $h, e \leftarrow \mathrm{fasttwosum}(c_{i,h}, h)$
6:     $\ell \leftarrow \circ(\circ(\ell + c_{i,\ell}) + e)$
```

Algorithm 4 (fasttwosum)

```
Input: $a, b$ binary64 numbers
Output: $h, \ell$ such that $h + \ell$ approximates $a + b$
1: $h \leftarrow \circ(a + b)$
2: $t \leftarrow \circ(h - a)$
3: $\ell \leftarrow \circ(b - t)$
```

Algorithm 5 (muldd)

```
Input: $x_h, x_\ell, c_h, c_\ell$ binary64 numbers
Output: $h, \ell$ approximating $(x_h + x_\ell)(c_h + c_\ell)$
1: $h \leftarrow \circ(x_h c_h)$
2: $\ell_1 \leftarrow \mathrm{fma}(x_h, c_h, -h)$
3: $\ell_2 \leftarrow \circ(x_h c_\ell)$
4: $\ell_3 \leftarrow \circ(x_\ell c_h)$
5: $\ell \leftarrow \circ(\circ(\ell_1 + \ell_2) + \ell_3)$
```

We wrote a Gappa program to analyze the rounding error of Algorithm AccuratePath. This program was generated automatically from a SageMath program, and is included in appendix. For example, we have a `gen_muldd` routine that generates the Gappa instructions for `muldd`, taking as argument the input and output variables, whose names are generated automatically to avoid name conflicts. The SageMath program also automatically generates Gappa hints, for example if `c=rnd(a*b)` is followed by `r = -(c-a*b)`, it generates the hint so that r is recognized by Gappa as the rounding error on c.

Algorithm 6 (mulddd)

```
Input: $x_h, x_\ell, c_h$ binary64 numbers
Output: $h, \ell$ approximating $(x_h + x_\ell)c_h$
1: $h \leftarrow \circ(x_h c_h)$
2: $\ell_1 \leftarrow \mathrm{fma}(x_h, c_h, -h)$
3: $\ell_2 \leftarrow \circ(x_\ell c_h)$
4: $\ell \leftarrow \circ(\ell_1 + \ell_2)$
```

This Gappa program proves that the following holds, for $x_0 \le x < 1/4$, and all rounding modes, where $Y = y_0 + y_1$:

$$|Y - r(x)|/r(x) \le 2^{-102.878}. \qquad (2)$$

Since $|r(x)| < 1.0001|\sinh(x)|$ by Eq. (1), and $2^{-107.671} + 1.0001 \cdot 2^{-102.878} < 2^{-102.826}$, it follows:

$$|Y - \sinh(x)| < 2^{-102.826} \sinh(x). \qquad (3)$$

Theorem 2: For any binary64 number x such that $\sinh(x)$ has less than 48 identical bits after the round bit, if $y_0 + y_1 + y_2$ is the triple-word approximation of $\sinh(x)$ computed by Algorithm AccuratePath, then $\circ(y_0 + y_1)$ is the correct rounding of $\sinh(x)$.

Proof: Let $y = \sinh(x)$. If y has less than 48 identical bits after the round bit, then y is at distance at least $2^{-49}\mathrm{ulp}(y)$ from a rounding boundary z: $|y - z| \ge 2^{-49}\mathrm{ulp}(y)$. Since $\mathrm{ulp}(y) > 2^{-53}|y|$, this yields $|y - z| \ge 2^{-102}|y|$. Thus Eq. (3) shows that $|Y - \sinh(x)| < |\sinh(x) - z|$, i.e., $y_0 + y_1$ is closer from $\sinh(x)$ than any rounding boundary. Thus $\circ(y_0 + y_1) = \circ(\sinh(x))$. ■

Since CORE-MATH contains all hard-to-round cases of sinh with at least 43 identical bits after the round bit, due to Theorem 2, it only remains to check correct rounding for those with at least 48 identical bits after the round bit. The CORE-MATH code checks whether y_1 has its low 52 significant bits zero (i.e., it is $\pm 2^k$), and then adjusts the result according to the sign of the second order term y_2. This happens for 106 inputs, and this is enough to guarantee correct rounding for those.

III. Branch $x \ge 1/4$

This branch is proven correct by exhaustive testing, both with and without the use of FMA (fused-multiply-add). Indeed, in C code like `d = a*b+c`, the compiler might decide to emit a FMA if it is supported in hardware, or not to emit a FMA. Thus we have to check the result is correctly rounded in both cases.

Since $\sinh(x)$ overflows in binary64 for $|x| > 711$, we only have to check less than 12 binades, which is doable with academic resources, using the algorithm described in Section II.C from [3].

During this check, we found 2018290 failures with a previous commit (`d764c73`). These failures were fixed by commit `89539bd`. Since this commit only increases the error bounds, all inputs that were correctly rounded with commit `d764c73` are still correctly rounded; we thus only have to check again the 2018290 inputs that failed, and all pass with commit `89539bd`.

References

[1] Chevillard, S., Joldeş, M., and Lauter, C. Sollya: An environment for the development of numerical codes. In *Mathematical Software - ICMS 2010* (Heidelberg, Germany, September 2010), K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, Eds., vol. 6327 of *Lecture Notes in Computer Science*, Springer, pp. 28–31.

[2] de Dinechin, F., Lauter, C., and Melquiond, G. Certifying the floating-point implementation of an elementary function using Gappa. *Transactions on Computers 60*, 2 (2011), 242–253.

[3] Zimmermann, P. The GNU libc atanh is correctly rounded. preprint, 2026.

Appendix

A. Gappa program for the accurate path

```
@rnd = float<ieee_64,RND>;
x = rnd(x_);
x2 = x*x;
x2r rnd= x*x;
x2l = -(x2r-x*x);
c0h = 0x1.5555555555555p-3;
c0l = 0x1.555555555552fp-57;
c0 = c0h+c0l;
c1h = 0x1.1111111111111p-7;
c1l = 0x1.11111115cf00dp-63;
c1 = c1h+c1l;
c2h = 0x1.a01a01a01a01ap-13;
c2l = 0x1.a0011c925b85cp-73;
c2 = c2h+c2l;
c3h = 0x1.71de3a556c734p-19;
c3l = 0x1.b4e2835532bcdp-73;
c3 = c3h+c3l;
c4h = 0x1.ae64567f54482p-26;
c4l = -0x1.defcf17a6ab79p-81;
c4 = c4h+c4l;
c5 = 0x1.6124613aef206p-33;
c6 = 0x1.ae7f36beea815p-41;
c7 = 0x1.95785063cd974p-49;
s6 = x2*c7;
s6r rnd= x2r*c7;
u6 = c6+s6;
u6r rnd= c6+s6r;
s5 = x2*u6;
s5r rnd= x2r*u6r;
u5 = c5+s5;
u5r rnd= c5+s5r;
y2_2 = x2*u5;
y2_2r rnd= x2r*u5r;
C4 = c4+y2_2;
# h,t = fasttwosum (c4h,y2_2r)
h rnd= c4h+y2_2r;
t = rnd(-(h-(c4h+y2_2r)));
l rnd= t+c4l;
C4r = h+l; # h+l approximates C4
# hh,ll = muldd (x2r,x2l,h,l)
hh rnd= x2r*h;
l1 = -(hh-x2r*h);
l2 rnd= x2r*l;
l3 rnd= x2l*h;
t1 rnd= l1+l2;
ll rnd= t1+l3;
D3 = x2*C4; # hh+ll approximates D3
# h1,e = fasttwosum (c3h,hh)
h1 rnd= c3h+hh;
e = rnd(-(h1-(c3h+hh)));
t2 rnd= ll+c3l;
l4 rnd= t2+e;
C3 = c3+D3;
C3r = h1+l4; # h1+l4 approximates C3
# hh1,ll1 = muldd (x2r,x2l,h1,l4)
hh1 rnd= x2r*h1;
l11 = -(hh1-x2r*h1);
l21 rnd= x2r*l4;
l31 rnd= x2l*h1;
t3 rnd= l11+l21;
ll1 rnd= t3+l31;
D2 = x2*C3; # hh1+ll1 approximates D2
# h2,e1 = fasttwosum (c2h,hh1)
h2 rnd= c2h+hh1;
e1 = rnd(-(h2-(c2h+hh1)));
t4 rnd= ll1+c2l;
l5 rnd= t4+e1;
C2 = c2+D2;
C2r = h2+l5; # h2+l5 approximates C2
# hh2,ll2 = muldd (x2r,x2l,h2,l5)
hh2 rnd= x2r*h2;
l12 = -(hh2-x2r*h2);
l22 rnd= x2r*l5;
l32 rnd= x2l*h2;
t5 rnd= l12+l22;
ll2 rnd= t5+l32;
D1 = x2*C2; # hh2+ll2 approximates D1
# h3,e2 = fasttwosum (c1h,hh2)
h3 rnd= c1h+hh2;
e2 = rnd(-(h3-(c1h+hh2)));
t6 rnd= ll2+c1l;
l6 rnd= t6+e2;
C1 = c1+D1;
C1r = h3+l6; # h3+l6 approximates C1
# hh3,ll3 = muldd (x2r,x2l,h3,l6)
hh3 rnd= x2r*h3;
l13 = -(hh3-x2r*h3);
l23 rnd= x2r*l6;
l33 rnd= x2l*h3;
t7 rnd= l13+l23;
ll3 rnd= t7+l33;
D0 = x2*C1; # hh3+ll3 approximates D0
# y1_3,e3 = fasttwosum (c0h,hh3)
y1_3 rnd= c0h+hh3;
e3 = rnd(-(y1_3-(c0h+hh3)));
t8 rnd= ll3+c0l;
y2_3 rnd= t8+e3;
C0 = c0+D0;
C0r = y1_3+y2_3; # y1_3+y2_3 approximates C0
# y1_4,y2_4 = mulddd (y1_3,y2_3,x)
y1_4 rnd= y1_3*x;
l14 = -(y1_4-y1_3*x);
l24 rnd= y2_3*x;
y2_4 rnd= l14+l24;
Y4 = C0*x;
Y4r = y1_4+y2_4;
# y1_5,y2_5 = muldd (y1_4,y2_4,x2r,x2l)
y1_5 rnd= y1_4*x2r;
l15 = -(y1_5-y1_4*x2r);
l25 rnd= y1_4*x2l;
l34 rnd= y2_4*x2r;
t9 rnd= l15+l25;
y2_5 rnd= t9+l34;
Y5 = Y4*x2;
Y5r = y1_5+y2_5;
# y0,y1_6 = fasttwosum (x,y1_5)
```

```
y0 rnd= x+y1_5;
y1_6 = rnd(-(y0-(x+y1_5)));
# y1,y2 = fasttwosum (y1_6,y2_5)
y1 rnd= y1_6+y2_5;
y2 = rnd(-(y1-(y1_6+y2_5)));
Y = x+Y5;
Yr = y0+y1;

# goals
{x in [XMIN,XMAX] -> |Yr - Y|/Y in ? }

# automatic hints
(x2r+x2l) - (x*x) -> x2l - (-(x2r-x*x));
(h+t) - (c4h+y2_2r) -> t - (-(h-(c4h+y2_2r)));
(hh+l1) - (x2r*h) -> l1 - (-(hh-x2r*h));
(hh+ll) - (x2r+x2l)*(h+l) -> ((hh+l1)-(x2r*h))
  + (l2 - (x2r*l)) + (l3 - (x2l*h))
  + (t1 - (l1+l2)) + (ll - (t1+l3)) - x2l*l;
(h1+e) - (c3h+hh) -> e - (-(h1-(c3h+hh)));
(hh1+l11) - (x2r*h1) -> l11 - (-(hh1-x2r*h1));
(hh1+ll1) - (x2r+x2l)*(h1+l4) -> ((hh1+l11)-(x2r*h1))
  + (l21 - (x2r*l4)) + (l31 - (x2l*h1))
  + (t3 - (l11+l21)) + (ll1 - (t3+l31)) - x2l*l4;
(h2+e1) - (c2h+hh1) -> e1 - (-(h2-(c2h+hh1)));
(hh2+l12) - (x2r*h2) -> l12 - (-(hh2-x2r*h2));
(hh2+ll2) - (x2r+x2l)*(h2+l5) -> ((hh2+l12)-(x2r*h2))
  + (l22 - (x2r*l5)) + (l32 - (x2l*h2)) + (t5 - (l12+l22))
  + (ll2 - (t5+l32)) - x2l*l5;
(h3+e2) - (c1h+hh2) -> e2 - (-(h3-(c1h+hh2)));
(hh3+l13) - (x2r*h3) -> l13 - (-(hh3-x2r*h3));
(hh3+ll3) - (x2r+x2l)*(h3+l6) -> ((hh3+l13)-(x2r*h3))
  + (l23 - (x2r*l6)) + (l33 - (x2l*h3))
  + (t7 - (l13+l23)) + (ll3 - (t7+l33)) - x2l*l6;
(y1_3+e3) - (c0h+hh3) -> e3 - (-(y1_3-(c0h+hh3)));
(y1_4+l14) - (y1_3*x) -> l14 - (-(y1_4-y1_3*x));
(y1_4+y2_4) - (y1_3+y2_3)*x -> ((y1_4+l14)-(y1_3*x))
  + (l24-(y2_3*x)) + (y2_4-(l14+l24));
(y1_5+l15) - (y1_4*x2r) -> l15 - (-(y1_5-y1_4*x2r));
(y1_5+y2_5) - (y1_4+y2_4)*(x2r+x2l) ->
  ((y1_5+l15)-(y1_4*x2r)) + (l25 - (y1_4*x2l))
  + (l34 - (y2_4*x2r)) + (t9 - (l15+l25))
  + (y2_5 - (t9+l34)) - y2_4*x2l;
(y0+y1_6) - (x+y1_5) -> y1_6 - (-(y0-(x+y1_5)));
(y1+y2) - (y1_6+y2_5) -> y2 - (-(y1-(y1_6+y2_5)));
Yr - Y -> ((y0+y1_6)-(x+y1_5)) + (y1 - (y1_6+y2_5))
  + (Y5r - Y5);
```