

The CORE-MATH sinh is correctly rounded

Guillaume Melquiond and Paul Zimmermann

April 10, 2026

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 `d028f64`. Since `sinh` is an odd function, and the approximations used are odd too, it suffices to prove that 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 `fma(x, 2-55, x)`.

Lemma 1: For x a binary64 number, $0 \leq x < x_0$, `fma(x, 2-55, x)` yields the correct rounding of $\sinh(x)$, whatever the IEEE 754 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 `nextabove(x)`. For $x_1 < x < x_0$, x a binary64 number, the term $x^3/6 + 0.01x^5$ lies in $[2^{-80}, 2^{-79})$, and `ulp(x) = 2-78`, thus $x^3/6 + 0.01x^5 < \frac{1}{2}\text{ulp}(x)$, and the same rounding rule as above applies. ■

Note that x_0 is optimal, since `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 `nextabove(x0)`. And if we replace `fma(x, 2-55, x)` by `fma(x, m, x)` with a magic constant $m > 2^{-55}$ so that it rounds to `nextabove(x0)` for $x = x_0$, for example $m = 2^{-53}$, then for $x = x_0/2$, `fma(x, m, x)` will round to `nextabove(x0/2)`, which is wrong.

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

A. Fast path

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

```
prec = 64;
d=[0x1.7137449123ef7p-26, 0.25];
c0=0x1.5555555555555555p-3;
c1=0x1.1111111111151ep-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);
```

This proves $|q(x) - \sinh(x)| < 2^{-56.584}x^3$ on the whole range $[x_0, 1/4)$. (The function `dirtyinfnorm` not being rigorous, this result was confirmed with a rigorous script kindly provided by Sylvain Chevillard.) In fact, it is possible to get a more precise result, namely $s_\ell < (q(x) - \sinh(x))/x^3 < s_u$. Table I gives the corresponding values of s_ℓ, s_u on intervals covering $[x_0, 1/4]$.

interval	s_ℓ	s_u
$[x_0, 2^{-22}]$	$-10923 \cdot 2^{-70}$	$-10922 \cdot 2^{-70}$
$[2^{-22}, 2^{-18}]$	$-10923 \cdot 2^{-70}$	$-10922 \cdot 2^{-70}$
$[2^{-18}, 2^{-14}]$	$-10923 \cdot 2^{-70}$	$-10922 \cdot 2^{-70}$
$[2^{-14}, 2^{-10}]$	$-10923 \cdot 2^{-70}$	$-10920 \cdot 2^{-70}$
$[2^{-10}, 2^{-6}]$	$-10921 \cdot 2^{-70}$	$-10410 \cdot 2^{-70}$
$[2^{-6}, 2^{-2}]$	$-10411 \cdot 2^{-70}$	$2689 \cdot 2^{-70}$

TABLE I

FOR INTERVALS COVERING $[x_0, 1/4]$, SOLLYA BOUNDS s_ℓ, s_u SUCH THAT $s_\ell < (q(x) - \sinh(x))/x^3 < s_u$, CONFIRMED INDEPENDENTLY BY A ROCQ SCRIPT.

These values s_ℓ and s_u were double-checked using the `CoqInterval` library [3] and the `Rocq` proof assistant. Once the polynomial q is defined, it is just a matter of running the following `Rocq` script, where `XMIN` and `XMAX` are substituted with the bounds of the input interval.

```
Def res x `(XMIN <= x <= XMAX) :=
  interval ((q x - sinh x) / x^3 * 0x1p70)
  with (i_bisect x, i_taylor x, i_prec 160,
       i_depth 6, i_decimal).
```

Algorithm `FastPath` details how the approximation of $q(x)$ is performed. The notation $\circ(\cdot)$ denotes the current rounding mode. The rounding test is performed at line 9. (In the CORE-MATH code, ℓ_b and u_b denote the rounded values.)

Theorem 1: For $x_0 \leq x < 1/4$, let ℓ_b and u_b the values computed at line 8 of Algorithm `FastPath`. If ℓ_b and u_b round to the same value, it is the correct rounding of $\sinh(x)$.

Proof: In each interval from Table I, we prove:

$$(\ell_b - q(x)) + s_u x^3 \leq 0 \leq (u_b - q(x)) + s_\ell x^3,$$

where s_ℓ, s_u are respectively lower and upper bounds for $(q(x) - \sinh(x))/x^3$ on the corresponding interval. It follows:

$$\ell_b \leq \sinh(x) \leq u_b,$$

thus if ℓ_b and u_b round to the same value, so does $\sinh(x)$ due to the monotonicity of all rounding functions.

Algorithm 1 (FastPath)

Input: x a binary64 number, $x_0 \leq x < 1/4$ **Output:** the correct rounding $\circ(\sinh x)$ or FAIL

- 1: define c_0, \dots, c_4 as in the Sollya code
 - 2: $x_2 \leftarrow \circ(x \cdot x)$, $x_3 \leftarrow \circ(x_2 \cdot x)$
 - 3: $x_4 \leftarrow \circ(x_2 \cdot x_2)$
 - 4: $p_0 \leftarrow \circ(c_0 + \circ(x_2 c_1))$, $p_2 \leftarrow \circ(c_2 + \circ(x_2 c_3))$
 - 5: $p'_2 \leftarrow \circ(p_2 + \circ(x_4 c_4))$, $p'_0 \leftarrow \circ(p_0 + \circ(x_4 p'_2))$
 - 6: $p \leftarrow \circ(x_3 p'_0)$
 - 7: $e = \circ(\epsilon x_3)$ where $\epsilon = 28 \cdot 2^{-57}$
 - 8: $\ell_b = x + \circ(p - e)$, $u_b = x + \circ(p + e)$
 - 9: if $\circ(\ell_b) = \circ(u_b)$ return $\circ(\ell_b)$ else return FAIL
-

To show $0 \leq (u_b - q(x)) + s_\ell x^3$, we use the following Gappa script, with version 1.7.0 of Gappa [2], where we substitute RND by all four rounding modes, XMIN and XMAX by interval bounds, EPS by $28 \cdot 2^{-57}$, and BND by the lower bound s_ℓ :

```
@rnd = float<ieee_64,RND>;
c0 = 0x1.5555555555555555p-3;
c1 = 0x1.111111111111151ep-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 = EPS;
er rnd= eps*x3r;
ub = x + rnd(pr+er);
{ x in [XMIN,XMAX] -> (ub-q)+x3*BND >= 0 }
ub - q -> rnd(pr+er) - (pr+er)
+ (pr - p) + er;
```

The Gappa script to prove $(\ell_b - q(x)) + s_u x^3 \leq 0$ has the same initial instructions, and ends instead with the following, where BND is replaced by the Sollya upper bound s_u :

```
lb = x + rnd(pr-er);
{ x in [XMIN,XMAX] -> (lb-q)+x3*BND <= 0 }
lb - q -> (rnd(pr-er) - (pr-er))
+ (pr - p) - er;
```

Note that having asymmetrical values for s_ℓ and s_u is not needed to prove the correctness of the fast path. One could just as well use $s_u = 10923 \cdot 2^{-70}$. But having a much smaller value for s_u considerably speeds up the verification of $(\ell_b - q(x)) + s_u x^3 \leq 0$ by Gappa.

B. 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 $\circ(\sinh(x))$. This is only a *potential bug* since in this case, $x + \circ(p + e)$ rounds to the same value as $\sinh(x)$. It can become a real bug if $\sinh(x)$ is just above a rounding boundary, and $x + \circ(p + e)$ just below this boundary.

We found a similar example which defeats the condition $\sinh(x) \leq u_b$ for the polynomial now used in the fast path, $x = 0x1.01cb85ecb4ea9p-9$ and $\epsilon = 27 \cdot 2^{-57}$, still with rounding toward zero. This shows that the value $\epsilon = 28 \cdot 2^{-57}$ from Algorithm FastPath is close to optimal.

C. 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 $r(x) = x + c_0 x^3 + \dots + c_7 x^{17}$ is used, with double-double coefficients for degrees 3 to 11, and double coefficients for degrees 13 to 17, here written using Sollya's syntax:

```
c0=0x1.5555555555555555p-3+0x1.555555555552fp-57;
c1=0x1.11111111111111p-7+0x1.11111115cf00dp-63;
c2=0x1.a01a01a01a01ap-13+0x1.a0011c925b85cp-73;
c3=0x1.71de3a556c734p-19-0x1.b4e2835532bcdp-73;
c4=0x1.ae64567f54482p-26-0x1.defcf17a6ab79p-81;
c5=0x1.6124613aef206p-33;
c6=0x1.ae7f36beea815p-41;
c7=0x1.95785063cd974p-49;
```

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

```
d=[0x1.7137449123ef7p-26,0.25];
r=x+c0*x^3+c1*x^5+c2*x^7+c3*x^9+c4*x^11
+c5*x^13+c6*x^15+c7*x^17;
supnorm(r,sinh(x),d,relative,1b-20);
```

This proves:

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

This property was also formally verified using the Rocq proof assistant and the CoqInterval library, thanks to the following Rocq script:

```
Lemma sinh_acc x `(x0 <= x <= 0.25) :
  Rabs ((r x - sinh x) / sinh x)
    <= exp (-107.671 * ln 2).
```

Proof. interval with (i_bisect x,
i_taylor x, i_prec 200). Qed.

For double-double coefficients, we write them $c_i = c_{i,h} + c_{i,\ell}$ below, where $c_{i,h}$ is the upper part, and $c_{i,\ell}$ the lower part; also, $\text{fma}(a, b, c)$ denotes $\circ(ab+c)$. At line 2 of AccuratePath,

Algorithm 2 (AccuratePath)

Input: x a binary64 number, $x_0 \leq x < 1/4$

Output: $y_0 + y_1$ approximating $\sinh(x)$

- 1: $x_2 \leftarrow \circ(x \cdot x)$, $x_{2\ell} \leftarrow \text{fma}(x, x, -x_2)$
 - 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: $y_1, y_2 \leftarrow \text{polydd}(x_2, x_{2\ell}, 5, y_2)$
 - 4: $y_1, y_2 \leftarrow \text{mulddd}(y_1, y_2, x)$
 - 5: $y_1, y_2 \leftarrow \text{muldd}(y_1, y_2, x_2, x_{2\ell})$
 - 6: $y_0, y_1 \leftarrow \text{fasttwosum}(x, y_1)$
 - 7: $y_1 \leftarrow \circ(y_1 + y_2)$
-

y_2 is a double approximation of $c_5x^2 + c_6x^4 + c_7x^6$, then after the `polydd` call at line 3, $y_1 + y_2$ is a double-double approximation of $c_0 + c_1x^2 + \dots + c_7x^{14}$, after line 4 it is a double-double approximation of $c_0x + \dots + c_7x^{15}$, after line 5 it is a double-double approximation of $c_0x^3 + \dots + c_7x^{17}$, and the last two 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 $c_0 + c_1 \cdot x^2 + \dots + (c_{n-1} + r)x^{2n-2}$

- 1: $h, t \leftarrow \text{fasttwosum}(c_{n-1}, h, r)$
 - 2: $\ell \leftarrow \circ(t + c_{n-1}, \ell)$
 - 3: for i from $n - 2$ downto 0
 - 4: $h, \ell \leftarrow \text{muldd}(x_h, x_\ell, h, \ell)$
 - 5: $h, e \leftarrow \text{fasttwosum}(c_{i,h}, h)$
 - 6: $\ell \leftarrow \circ(\ell + c_{i,\ell}, e)$
-

Algorithm 4 (fasttwosum)

Input: a, b binary64 numbers

Output: h, ℓ 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_ℓ, c_h, c_ℓ binary64 numbers

Output: h, ℓ approximating $(x_h + x_\ell)(c_h + c_\ell)$

- 1: $h \leftarrow \circ(x_h c_h)$
 - 2: $\ell_1 \leftarrow \text{fma}(x_h, c_h, -h)$
 - 3: $\ell_2 \leftarrow \circ(x_h c_\ell)$
 - 4: $\ell_3 \leftarrow \circ(x_\ell c_h)$
 - 5: $\ell \leftarrow \circ(\ell_1 + \ell_2) + \ell_3$
-

We used Gappa to analyze the rounding error of Algorithm AccuratePath. The Gappa script was generated automatically

Algorithm 6 (mulddd)

Input: x_h, x_ℓ, c_h binary64 numbers

Output: h, ℓ approximating $(x_h + x_\ell)c_h$

- 1: $h \leftarrow \circ(x_h c_h)$
 - 2: $\ell_1 \leftarrow \text{fma}(x_h, c_h, -h)$
 - 3: $\ell_2 \leftarrow \circ(x_\ell c_h)$
 - 4: $\ell \leftarrow \circ(\ell_1 + \ell_2)$
-

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 and goals. For example, if some expression involves both a and $\circ(a)$, then a hint will tell Gappa how it can be rewritten to factor out $\circ(a) - a$, if not discovered by Gappa already.

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

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

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

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

Theorem 2: For any binary64 number x such that $\sinh(x)$ has less than 48 identical bits after the round bit, if $y_0 + y_1$ is the double-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 $2^{-49} \text{ulp}(y)$ from a rounding boundary z : $|y - z| \geq 2^{-49} \text{ulp}(y)$. Since $\text{ulp}(y) > 2^{-53} |y|$, this yields $|y - z| \geq 2^{-102} |y|$. 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 $\circ(y_0 + y_1) = \circ(\sinh(x))$. ■

D. Global proof of correctness

We now prove the correctness of the CORE-MATH function for $x_0 \leq x < 1/4$:

Theorem 3: For any binary64 number x , $x_0 \leq x < 1/4$, and any rounding mode, the value returned by the CORE-MATH \sinh function is correctly rounded.

Proof: The workflow of the CORE-MATH \sinh function is the following in this branch:

- if Algorithm FastPath does not yield FAIL, then it returns $\circ(\ell_b)$, which is correctly rounded from Theorem 1;
- if FastPath yields FAIL, Algorithm AccuratePath is called, which computes $y_0 + y_1$ approximating $\sinh(x)$. Then it is checked (routine `as_sinh_zero`) whether the last 52 bits of y_1 are all ones, and if so y_1 is adjusted, otherwise y_1 is left unchanged. This adjustment occurs for 106 numbers from the CORE-MATH test-suite, having from 51 to 72 identical bits after the round bit;

- the code then returns $\circ(y_0 + y_1)$.

The CORE-MATH test-suite (`sinh_wc`) contains all hard-to-round cases with at least 43 identical bits after the round bit. Assume $\sinh(x)$ has less than 48 identical bits after the round bit: then if x enters the accurate path, its y_1 value is not adjusted, thus $\circ(y_0 + y_1)$ is returned, which is correctly rounded from Theorem 2. If $\sinh(x)$ has 48 or more identical bits after the round bit, then x is included in the CORE-MATH test-suite, and since this test-suite gives no error (with or without FMA), $\sinh(x)$ is correctly rounded. ■

III. BRANCH $x \geq 1/4$

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. Thus we have to check that the result is correctly rounded in both cases.

Since $\sinh(x)$ overflows in binary64 for $|x| > 711$, we only have to check 12 binades, which is doable with academic resources, using the algorithm described in Section II.C from [4].

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`.

Acknowledgements. The authors thank Sylvain Chevillard for providing Sollya code to compute the minimum and maximum of a function over an interval, which enabled us to get the values s_ℓ, s_u of Table I, and to confirm the `dirtyinfnorm` results. They also thank Pavel Pancheckha for his feedback on an earlier version of this article. The exhaustive tests from Section III were performed thanks to the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several universities as well as other organizations (see <https://www.grid5000.fr/>).

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] MARTIN-DOREL, É., AND MELQUIOND, G. Proving tight bounds on univariate expressions with elementary functions in Coq. *Journal of Automated Reasoning* 57, 3 (2016), 187–217.
- [4] ZIMMERMANN, P. The GNU libc `atanh` is correctly rounded. preprint, 2026.

APPENDIX

A. Gappa script for the accurate path

The Gappa script to bound the error in the accurate path is the following. It was automatically generated from a SageMath program, including comments that help to link with Algorithm AccuratePath.

Note that $h, \ell = \text{fasttwosum}(a, b)$ is translated into `h rnd= a+b and l = rnd(-(h-(a+b)))`, which says that $\ell = \circ(a + b - h)$. This requires for example $|a| \geq |b|$, which is checked by the Gappa goal `b // a in [-1, 1]`.

For the `muldd` and `mulddd` calls, the Gappa script assumes $\ell_1 = x_h c_h - h$, which holds whenever $x_h c_h$ is an integer multiple of 2^{-1074} , which is checked by the Gappa goal `@FIX(xh*ch, -1074)`.

```
@rnd = float<ieee_64,RND>;
x = rnd(x_);
x2 = x*x;
x2r rnd= x*x;
x2l = -(x2r-x*x);
c0h = 0x1.5555555555555p-3;
c0l = 0x1.5555555555552fp-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;
ll = -(hh-x2r*h);
l2 rnd= x2r*l;
l3 rnd= x2l*h;
t1 rnd= ll+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;
ll1 = -(hh1-x2r*h1);
```

```

l21 rnd= x2r*l4;
l31 rnd= x2l*h1;
t3 rnd= l11+l21;
l11 rnd= t3+l31;
D2 = x2*C3; # hh1+l11 approximates D2
# h2,e1 = fasttwosum (c2h,hh1)
h2 rnd= c2h+hh1;
e1 = rnd(-(h2-(c2h+hh1)));
t4 rnd= l11+c2l;
l5 rnd= t4+e1;
C2 = c2+D2;
C2r = h2+l5; # h2+l5 approximates C2
# hh2,l12 = 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;
l12 rnd= t5+l32;
D1 = x2*C2; # hh2+l12 approximates D1
# h3,e2 = fasttwosum (c1h,hh2)
h3 rnd= c1h+hh2;
e2 = rnd(-(h3-(c1h+hh2)));
t6 rnd= l12+c1l;
l6 rnd= t6+e2;
C1 = c1+D1;
C1r = h3+l6; # h3+l6 approximates C1
# hh3,l13 = 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;
l13 rnd= t7+l33;
D0 = x2*C1; # hh3+l13 approximates D0
# y1_3,e3 = fasttwosum (c0h,hh3)
y1_3 rnd= c0h+hh3;
e3 = rnd(-(y1_3-(c0h+hh3)));
t8 rnd= l13+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 rnd= y1_6+y2_5;
Y = x+Y5;
Yr = y0+y1;

# goals
{x in [XMIN,XMAX] -> y2_2r // c4h in [-1,1] /\

```

```

@FIX(x2r*h, -1074) /\ hh // c3h in [-1,1] /\
@FIX(x2r*h1, -1074) /\ hh1 // c2h in [-1,1] /\
@FIX(x2r*h2, -1074) /\ hh2 // c1h in [-1,1] /\
@FIX(x2r*h3, -1074) /\ hh3 // c0h in [-1,1] /\
@FIX(y1_3*x, -1074) /\ @FIX(y1_4*x2r, -1074) /\
y1_5 // x in [-1,1] /\ |Yr - Y|/Y in ? }

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

```