CR-LIBM, and Arenaire's results on function implementation

Jean-Michel Muller
CNRS - Laboratoire LIP

http://perso.ens-lyon.fr/jean-michel.muller/

with the help of Nicolas Brisebarre, Florent de Dinechin, Vincent Lefèvre, Christoph Lauter...

Crlibm

Started in Arénaire in 2000 (David Defour's PhD: `http://gala.univ-perp.fr/~ddefour/research/thesis_dd.pdf`)

- correctly rounded transcendentals;
- use of our database of worst cases for rounding;
- Two steps only (fast step and accurate step)
- Emphasis on reliability: provide a proof which each function;
- Emphasis on efficiency: parameterized Maple scripts to explore the various tradeoffs (also part of the library)
- Four rounding modes
- several tools designed to help designing the libraries and proving error bounds. Maybe these tools are now more important than the library itself.

At the beginning there was The Table Maker's Dilemma

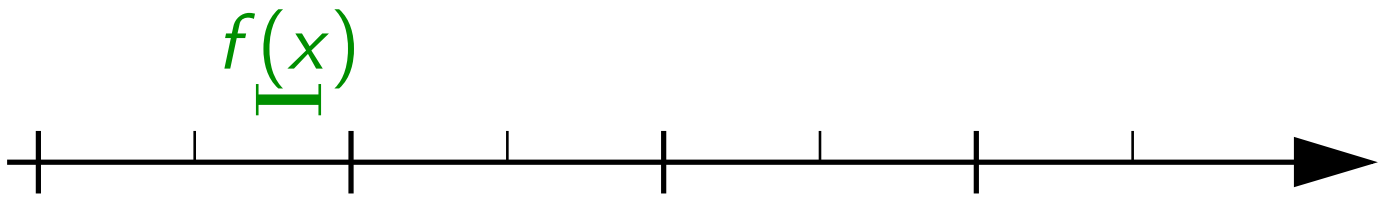
- Finite-precision algorithm for evaluating $f(x)$
- Approximation + rounding errors $\longrightarrow$ overall error bound ε.
- What we compute: y such that $f(x) \in [y(1-\varepsilon), y(1+\varepsilon)]$

At the beginning there was The Table Maker's Dilemma

- Finite-precision algorithm for evaluating $f(x)$
- Approximation + rounding errors $\longrightarrow$ overall error bound ε.
- What we compute: y such that $f(x) \in [y(1-\varepsilon), y(1+\varepsilon)]$

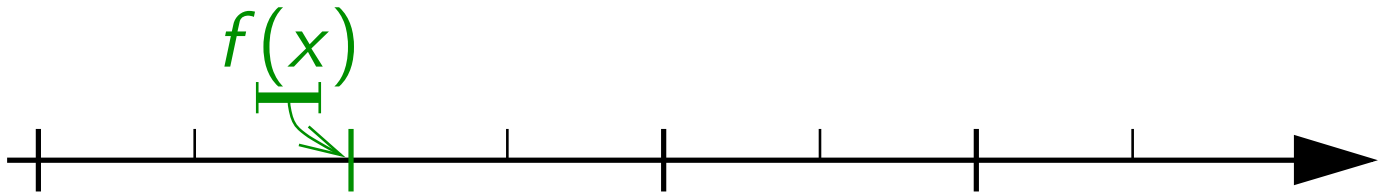
At the beginning there was The Table Maker's Dilemma

- Finite-precision algorithm for evaluating $f(x)$
- Approximation + rounding errors $\longrightarrow$ overall error bound ε.
- What we compute: y such that $f(x) \in [y(1-\varepsilon), y(1+\varepsilon)]$



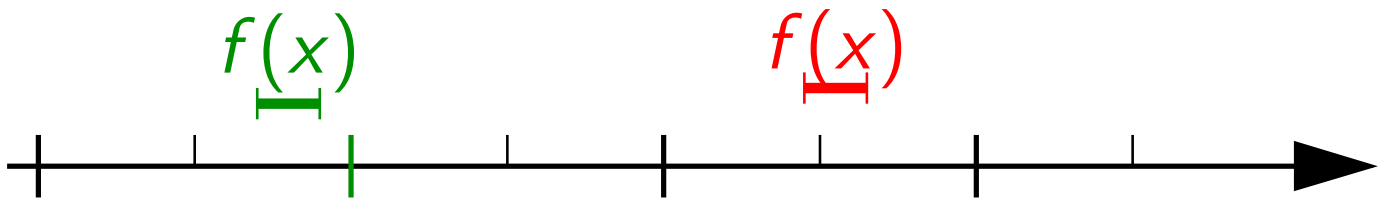
At the beginning there was The Table Maker's Dilemma

- Finite-precision algorithm for evaluating $f(x)$
- Approximation + rounding errors $\longrightarrow$ overall error bound ε.
- What we compute: y such that $f(x) \in [y(1-\varepsilon), y(1+\varepsilon)]$



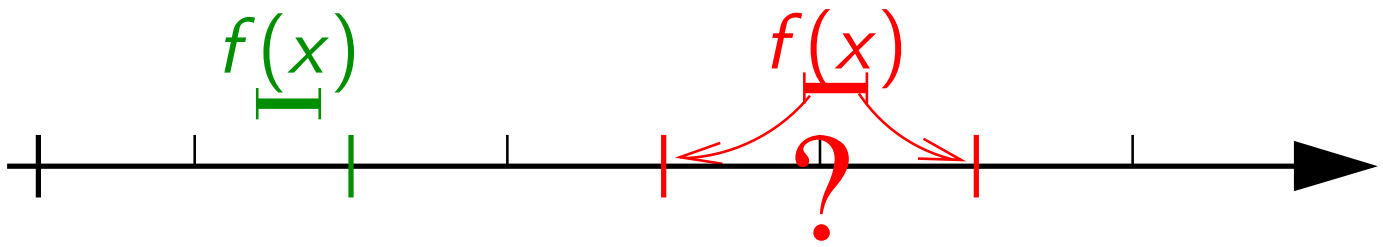
At the beginning there was The Table Maker's Dilemma

- Finite-precision algorithm for evaluating $f(x)$
- Approximation + rounding errors $\longrightarrow$ overall error bound ε.
- What we compute: y such that $f(x) \in [y(1-\varepsilon), y(1+\varepsilon)]$



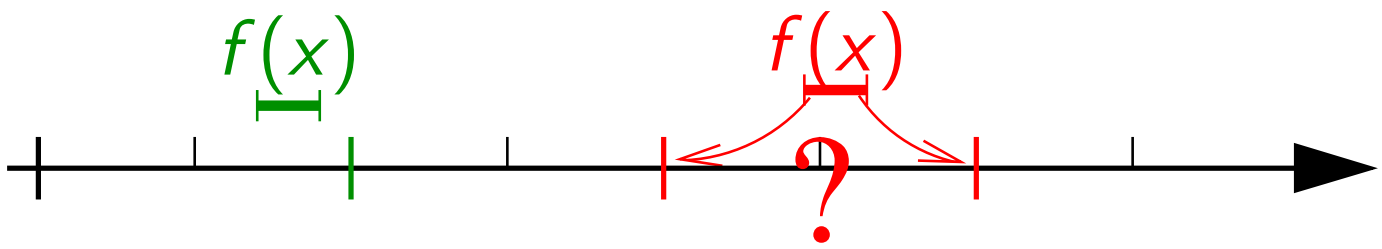
At the beginning there was The Table Maker's Dilemma

- Finite-precision algorithm for evaluating $f(x)$
- Approximation + rounding errors $\longrightarrow$ overall error bound ε.
- What we compute: y such that $f(x) \in [y(1-\varepsilon), y(1+\varepsilon)]$



At the beginning there was The Table Maker's Dilemma

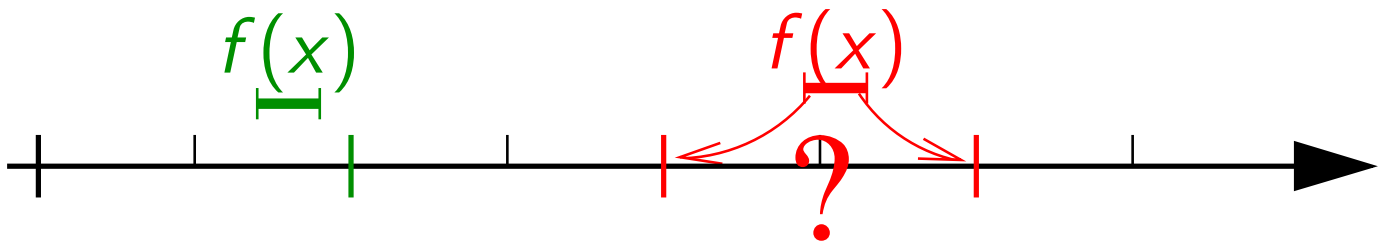
- Finite-precision algorithm for evaluating $f(x)$
- Approximation + rounding errors $\longrightarrow$ overall error bound ε.
- What we compute: y such that $f(x) \in [y(1-\varepsilon), y(1+\varepsilon)]$



If the dilemma happens, try again with smaller ε

At the beginning there was The Table Maker's Dilemma

- Finite-precision algorithm for evaluating $f(x)$
- Approximation + rounding errors $\longrightarrow$ overall error bound ε.
- What we compute: y such that $f(x) \in [y(1-\varepsilon), y(1+\varepsilon)]$



If the dilemma happens, try again with smaller ε

- Ziv's strategy: improve ε until we can round
- Fast in average (high accuracy is rarely needed)
- For most transcendental functions, it is proven to succeed eventually (Lindeman's theorem)
- First implemented in IBM's `libultim`

The last frontier

Now we are four:

- IBM's `libultim`
- Arenaire's `crlibm`
- Sun's `libmcr`

Plus `mpfr` (arbitrary precision arithmetic with correct rounding)

Why do we want a correctly rounded libm?

- Because it is the best mathematically possible
- Because it makes floating-point programs portable
- Because it helps building proofs and computing errors
- For a fair comparison of performance on functionally identical libraries

Why do we want a correctly rounded libm?

- Because it is the best mathematically possible
- Because it makes floating-point programs portable
- Because it helps building proofs and computing errors
- For a fair comparison of performance on functionally identical libraries

Our main point: Because it is barely more expensive

Our proud customers at CERN

- LHC@home: Large-scale distribution of the simulation of the future Large Hadron Collider
- The physical phenomenon is chaotic
- You need fully deterministic and portable computations to be able to merge results of the distributed computation
- Inconsistencies due to different mathematical libraries on Intel and AMD processors
  - all faithful with 90+% correct rounding,
  - all highly optimized for speed,
  - probably all micro-coded
- Six months of engineer work to track and solve the problem
  - starting with working single-process code...
  - Fortunately all they needed was exp, log and arctangent.
- Performance doesn't come first in this case.

From libultim to crlibm

- Worst case accuracy (WCA) required for correct rounding
  - There exists an ε which always guarantees correct rounding
  - Predicted by Gal at $\varepsilon \approx 2^{-64-53} = 2^{-117}$
  - Computed by a cleverly accelerated exhaustive search (Lefèvre's PhD);
  - Finished functions/domains: see next slide.
- Two steps are enough
  - first step accurate to $\varepsilon \approx 2^{-53-10}$
  - second step accurate to our computed WCA, not more

From libultim to crlibm

- Worst case accuracy (WCA) required for correct rounding
  - There exists an ε which always guarantees correct rounding
  - Predicted by Gal at $\varepsilon \approx 2^{-64-53} = 2^{-117}$
  - Computed by a cleverly accelerated exhaustive search (Lefèvre's PhD);
  - Finished functions/domains: see next slide.
- Two steps are enough
  - first step accurate to $\varepsilon \approx 2^{-53-10}$
  - second step accurate to our computed WCA, not more
- Two steps are better than n steps
  - optimisation
  - reliability
  - an "almost never used" step is also "almost never checked" $\longrightarrow$ too dangerous.

Worst cases so far (double precision)

Functions/domains for which either we have worst cases, or we don't need them (e.g. $\sin(x) = x$ in RN mode for very small x).

- e^x, $\ln(x)$, $e^x - 1$, $\ln(1+x)$, 2^x, $\log_2(x)$, 10^x, $\log_{10}(x)$, sinh, cosh, $\sinh^{-1}$, $\cosh^{-1}$, $\sin^{-1}(x)$, $\tan^{-1}(x)$: full domain;
- $\sin(x)$ for $|x| < 2 \times (1 + 4675/2^{13}) \approx 3.141357422$;
- $\cos(x)$ for $|x| < (1 + 4675/2^{13}) \approx 1.570678711$;
- $\cos^{-1}(x)$ for $|x| > 2^{-25}$;
- $\tan(x)$ for $|x| < 7074237752028441/4503599627370496$ (slightly $> \pi/2$);
- $\sin(2\pi x)$, $\cos(2\pi x)$ for $|x| \geq 2^{-25}$;
- $\tan(2\pi x)$ for $2^{-25} \leq |x| < 2^{-6}$;
- $1/\sqrt{x}$, x^3 and $x^{1/3}$: full domain.

Worst cases so far (double precision)

Functions/domains for which either we have worst cases, or we don't need them (e.g. $\sin(x) = x$ in RN mode for very small x).

- e^x, $\ln(x)$, $e^x - 1$, $\ln(1+x)$, 2^x, $\log_2(x)$, 10^x, $\log_{10}(x)$, sinh, cosh, $\sinh^{-1}$, $\cosh^{-1}$, $\sin^{-1}(x)$, $\tan^{-1}(x)$: full domain;
- $\sin(x)$ for $|x| < 2 \times (1 + 4675/2^{13}) \approx 3.141357422$;
- $\cos(x)$ for $|x| < (1 + 4675/2^{13}) \approx 1.570678711$;
- $\cos^{-1}(x)$ for $|x| > 2^{-25}$;
- $\tan(x)$ for $|x| < 7074237752028441/4503599627370496$ (slightly $> \pi/2$);
- $\sin(2\pi x)$, $\cos(2\pi x)$ for $|x| \geq 2^{-25}$;
- $\tan(2\pi x)$ for $2^{-25} \leq |x| < 2^{-6}$;
- $1/\sqrt{x}$, x^3 and $x^{1/3}$: full domain.

Still improving. Soon in the public domain

We started with probabilistic ideas (same as Gal's)

- the significant y of $f(x)$ has the form:

$$y = y_0.y_1y_2\cdots y_{p-1}\overbrace{01111111\cdots11}^{kbits}xxxxx\cdots$$

or

$$y = y_0.y_1y_2\cdots y_{p-1}\overbrace{10000000\cdots00}^{kbits}xxxxx\cdots$$

with $k \geq 1$.

- Assuming that after the p^{th} position, the "1s" and "0s" are equally probable, the "probability" of having $k \geq k_0$ is 2^{1-k_0};
- assuming N possible FP input numbers, we will observe around $N \times 2^{1-k_0}$ values for which $k \geq k_0$;
- vanishes as soon as k_0 is significantly larger than $\log_2(N)$, so we might expect the largest k_0 to be around $\log_2(N)$.

But this does not prove anything!

We continued with transcendental number theory... but wasn't that convincing

A result due to Baker (1975):

- $\alpha = i/j$, $\beta = r/s$, with $i, j, r, s < 2^p$;
- $C = 16^{200}$;

$$|\alpha - \log(\beta)| > (p2^p)^{-Cp \log p}$$

This is exactly the kind of result we need.

Application: To compute ln and exp correctly rounded in double precision, it suffices to compute an approximation y whose precision is around

We continued with transcendental number theory... but wasn't that convincing

A result due to Baker (1975):

- $\alpha = i/j$, $\beta = r/s$, with $i, j, r, s < 2^p$;
- $C = 16^{200}$;

$$|\alpha - \log(\beta)| > (p2^p)^{-Cp \log p}$$

This is exactly the kind of result we need.

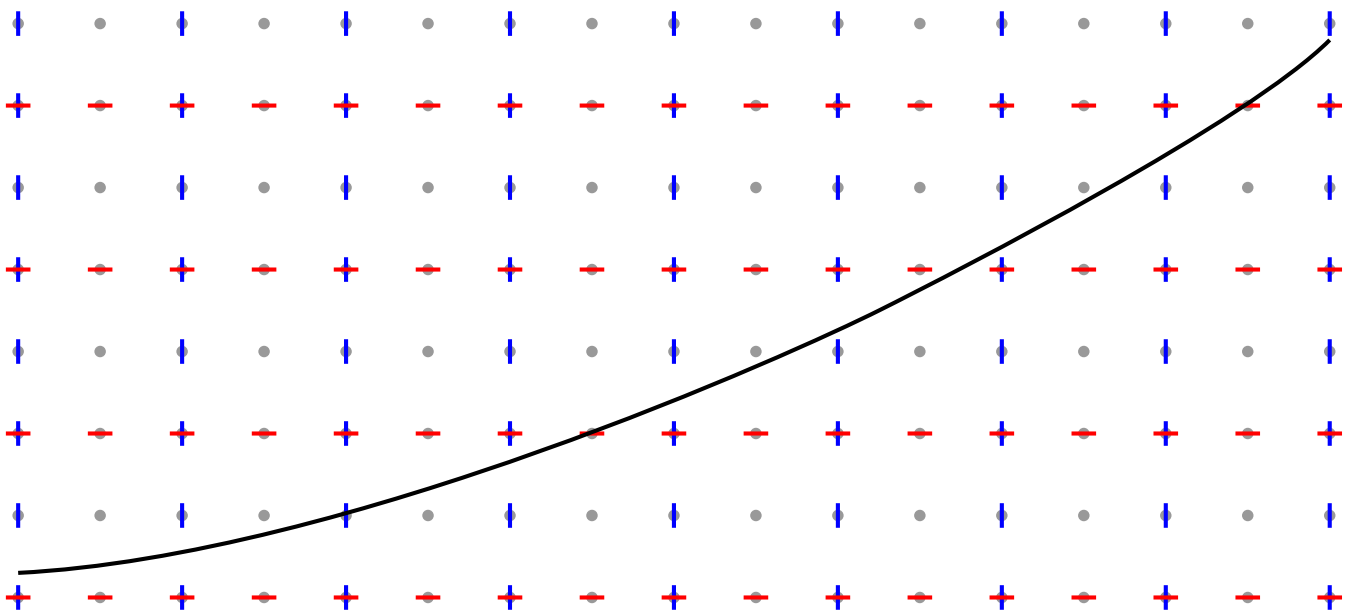
Application: To compute ln and exp correctly rounded in double precision, it suffices to compute an approximation y whose precision is around

$$10^{244} \text{ bits}$$

Lefèvre's PhD dissertation (2000)

`ftp.ens-lyon.fr/pub/LIP/Rapports/PhD/PhD2000/PhD2000-02.ps.Z`

- filtering: approximation of function by linear functions in very small intervals + algorithm for distance between a line and a regular grid $\longrightarrow$ very short list of intervals that could contain a worst case;
- accurate computation of the function in these intervals.



First example: natural logarithm

Table: *Worst cases for the natural (radix e) logarithm in the full range.*

Interval	worst case (binary)
$[2^{-1074}, 1)$	$\log(1.11101010011100011101100001011100111011100000000100000 \times 2^{-509})$ $= -101100000.00101001011010100110011010110100001011111111 \quad 1 \quad 1^{60}0000\ldots$
	$\log(1.10010100011101101110001100000100110011010111110001111 \times 2^{-384})$ $= -100001001.10110110000011001010111101000111101100110101 \quad 1 \quad 0^{60}1010\ldots$
	$\log(1.00100110111010011100010011010011001001111001011000000 \times 2^{-232})$ $= -10100000.10101011001011000010010111100110100001000000 \quad 0 \quad 0^{60}1001\ldots$
	$\log(1.01100001001110010101010111011100100000000010111110000 \times 2^{-35})$ $= -10111.11110000001011111001101110101111011000000011010 \quad 0 \quad 1^{60}0011\ldots$
$(1, 2^{1024}]$	$\log(1.01100010101010001000011000010011011000101001101101 10 \times 2^{678})$ $= 111010110.01000111100111101011101001111100100101110001 \quad 0 \quad 0^{64}1110\ldots$

Second example: 2^x

Table: *Worst cases for the radix-2 exponential function 2^x in the full range. Integer values of x are omitted.*

Interval	worst case (binary)
$[-1074, 0)$	$2 * *(-1.0010100001100011101010111010111010101111011110110010 \times 2^{-15})$ $= 0.11111111111111100110010100011111010001100000111101111 \quad 0 \quad 0^{57}1110\ldots$
	$2 * *(-1.0100000101101111011011000110010001000101101011001111 \times 2^{-20})$ $= 0.11111111111111111111001000010011001010111010011001110 \quad 1 \quad 1^{57}0000\ldots$
	$2 * *(-1.0000010101010110000000011100100010101011001111110001 \times 2^{-32})$ $= 0.11111111111111111111111111111111101001010110110110000 1 \quad 1 \quad 1^{57}0000\ldots$
	$2 * *(-1.0001100001011011100011011011011011010101100000011101 \times 2^{-33})$ $= 0.11111111111111111111111111111111100111101101010111100 \quad 0 \quad 0^{57}1100\ldots$
$(0, 1024]$	$2 * *(1.1011111110111011110111100100010011101101111111000101 \times 2^{-25})$ $= 1.00000000000000000000000010011011001011000011100000101 \quad 0 \quad 0^{59}1011\ldots$
	$2 * *(1.1110010001011001011001010010011010111111100101001101 \times 2^{-10})$ $= 1.00000000010100111111110000101110110000101011010100 11 \quad 0 \quad 1^{59}0100\ldots$

From crlibm to the post-ultimate libm

- Worst case accuracy required for correct rounding
  - Predicted by the probabilistic arguments at $\varepsilon \approx 2^{-64-53} = 2^{-117}$
  - Found at 2^{-130} to 2^{-160} for quite a few functions
  - This seems very improbable

From crlibm to the post-ultimate libm

- Worst case accuracy required for correct rounding
  - Predicted by the probabilistic arguments at $\varepsilon \approx 2^{-64-53} = 2^{-117}$
  - Found at 2^{-130} to 2^{-160} for quite a few functions
  - This seems very improbable
- When WCA much worse than the expected 2^{-117}, there is a mathematical explanation
  - high WCA due to the Taylor formula $\longrightarrow$ the repartition of digits is very regular (for instance $\exp(\varepsilon) \approx 1 + \varepsilon$) $\longrightarrow$ probabilistic assumptions no longer valid;
  - In these cases, the Taylor formula also allows to compute the function using intermediate accuracy of about 2^{-117}.

From crlibm to the post-ultimate libm

- Worst case accuracy required for correct rounding
  - Predicted by the probabilistic arguments at $\varepsilon \approx 2^{-64-53} = 2^{-117}$
  - Found at 2^{-130} to 2^{-160} for quite a few functions
  - This seems very improbable
- When WCA much worse than the expected 2^{-117}, there is a mathematical explanation
  - high WCA due to the Taylor formula $\rightarrow$ the repartition of digits is very regular (for instance $\exp(\varepsilon) \approx 1 + \varepsilon$) $\rightarrow$ probabilistic assumptions no longer valid;
  - In these cases, the Taylor formula also allows to compute the function using intermediate accuracy of about 2^{-117}.
- Second step always possible using double-double-extended
  - 128-bit precision
  - simple
  - efficient
  - well-known and well-proven
  - integration of the two steps

Various tools designed by the Arenaire team

`http://www.ens-lyon.fr/LIP/Arenaire/`

- worst cases: soon in the public domain
- Chevillard, Lauter: a "quick and dirty" Remez algorithm, but we don't need much more;
- Chevillard, Lauter: validated infnorm in C (uses multiple precision interval arithmetic) $\longrightarrow$ certain and tight bounds on

$$\max_{[a,b]} |p(x) - f(x)|$$

- Brisebarre, Chevillard, Muller, Tisserand, Torres: best polynomial among the ones that satisfy constraints on the size of the coefficients. Two approaches
  - general case: enumerate integer points in a polytope;
  - "big" difficult cases: LLL algorithm (lattice reduction).
- Melquiond: Gappa: error bounds on FP calculations, and formal proofs;

Best polynomial with constraints

- FP-polynomial: whose coefficients are exactly representable (variant: some are sum of 2 or 3 FP numbers);
- minimax polynomial $p \rightarrow$ error $\varepsilon = \max |f(x) - p(x)|$ (variant: relative error);
- $\hat{p}$ obtained by rounding to the nearest the coefficients of $p \rightarrow$ error $\hat{\varepsilon}$;
- $p^* =$ best approximation to f among the FP polynomials, and $\varepsilon^* = \max |f(x) - p^*(x)|$. obviously $\varepsilon \leq \varepsilon^* \leq \hat{\varepsilon}$.
- Parameter K, $\varepsilon < K \leq \hat{\varepsilon}$, and $d+1$ points x_0, x_1, ..., x_d;
- degree-n Polynomials: vector space of dimension $n+1$, coordinates=coefficients;
- if $d \geq n$, the constraints $f(x_i) - K \leq \sum_{j=0}^{n} a_j x_i^j \leq f(x_i) + K$ define a polytope (bounded polyhedron) $\rightarrow$ contains a finite number of FP-polynomials;

Best polynomial with constraints (cont.)

- algorithm for scanning the "integer points" in a polytope (close to rational programming): for each point (polynomial) p, we compute $\max |f - p|$;
- problem: is K is too small, the polytope may contain no integer point. If K is too large (even very slightly), it may contain several zillion points;
- other solution: use the LLL algorithm (Lenstra, Lenstra and Lovasz, 1982). LLL finds lattice basis of "short" vectors. In our case: does not always give the "best" polynomial, but always gives a "very good" one $\longrightarrow$ either satisfactory solution, or starting point that gives a sharp K for the other algorithm.

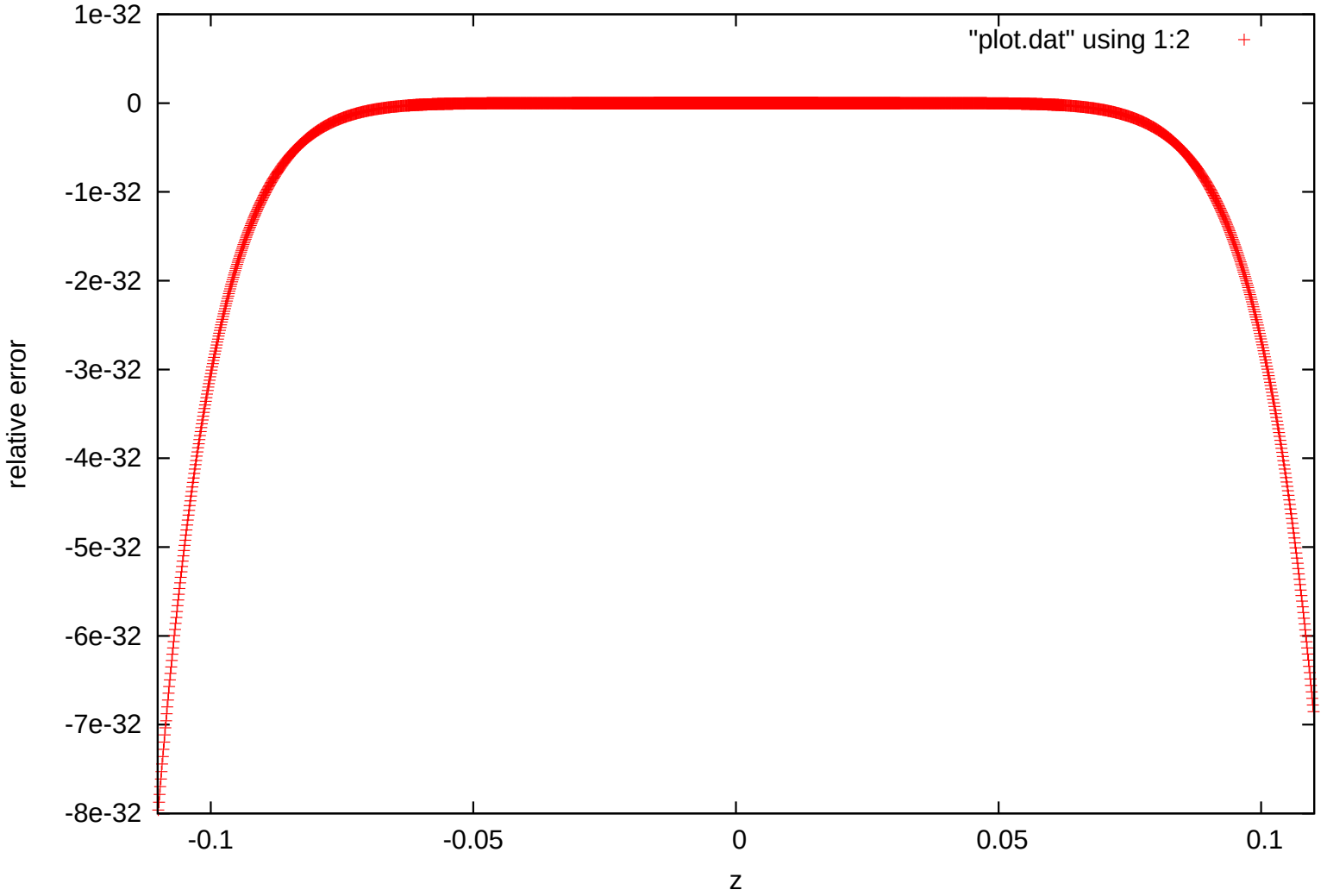
An example used in CRLIBM

- function arcsin near 1 with correct rounding;
- change of variables: we actually compute

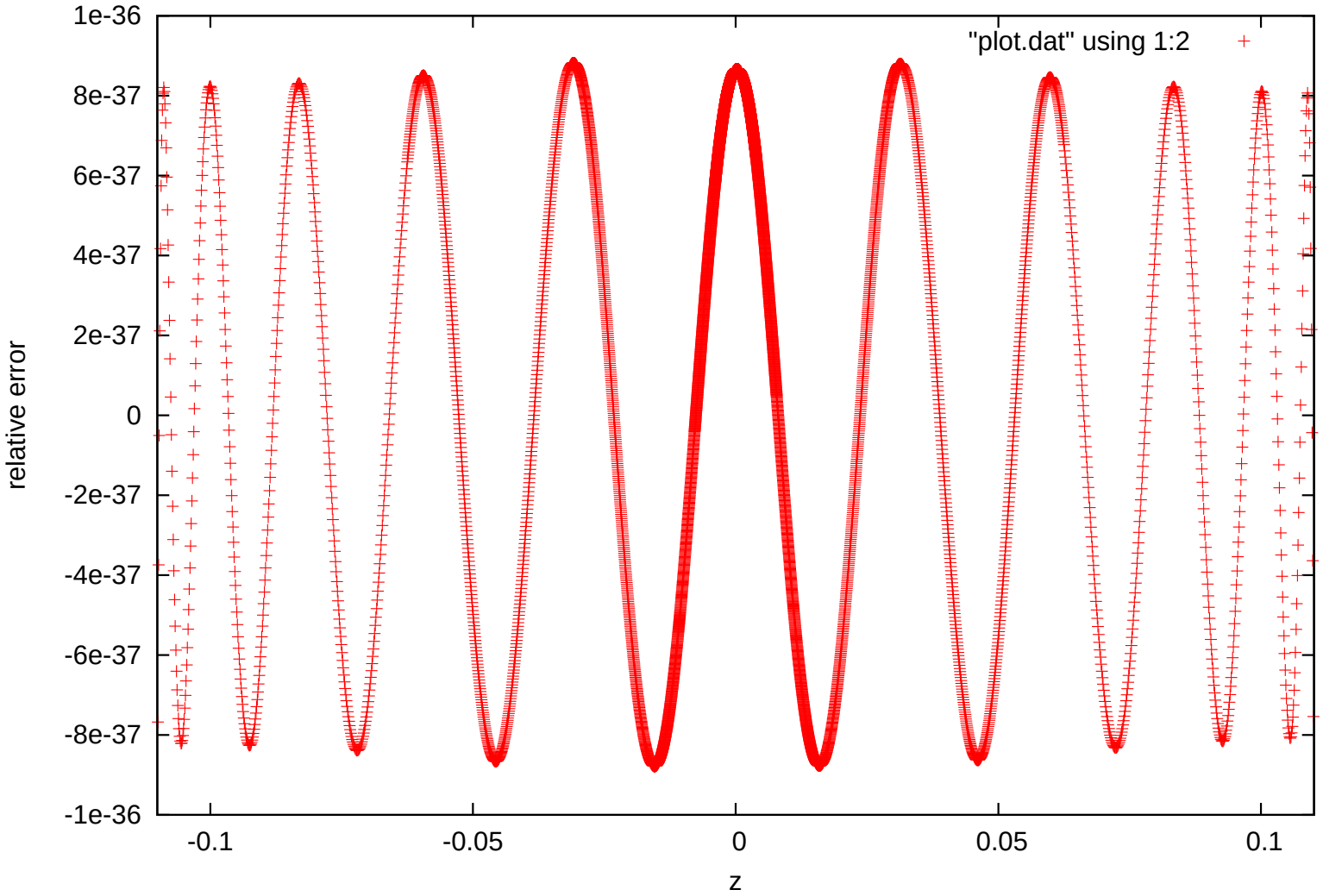
$$g(z) = \frac{\arcsin(1 - (z + m)) - \frac{\pi}{2}}{\sqrt{2 \cdot (z + m)}}$$

  where
  $0\text{xBFBC28F800009107} \leq z \leq 0\text{x3FBC28F7FFFF6EF1}$
  (roughly speaking $-0.110 \leq z \leq 0.110$) and
  $m = 0\text{x3FBC28F80000910F} \simeq 0.110$;
- degree-21 polynomial approximation.

If we round to nearest the coefficients of the Remez polynomial



If we use our approach



Example 2: constrained polynomial for the logarithm

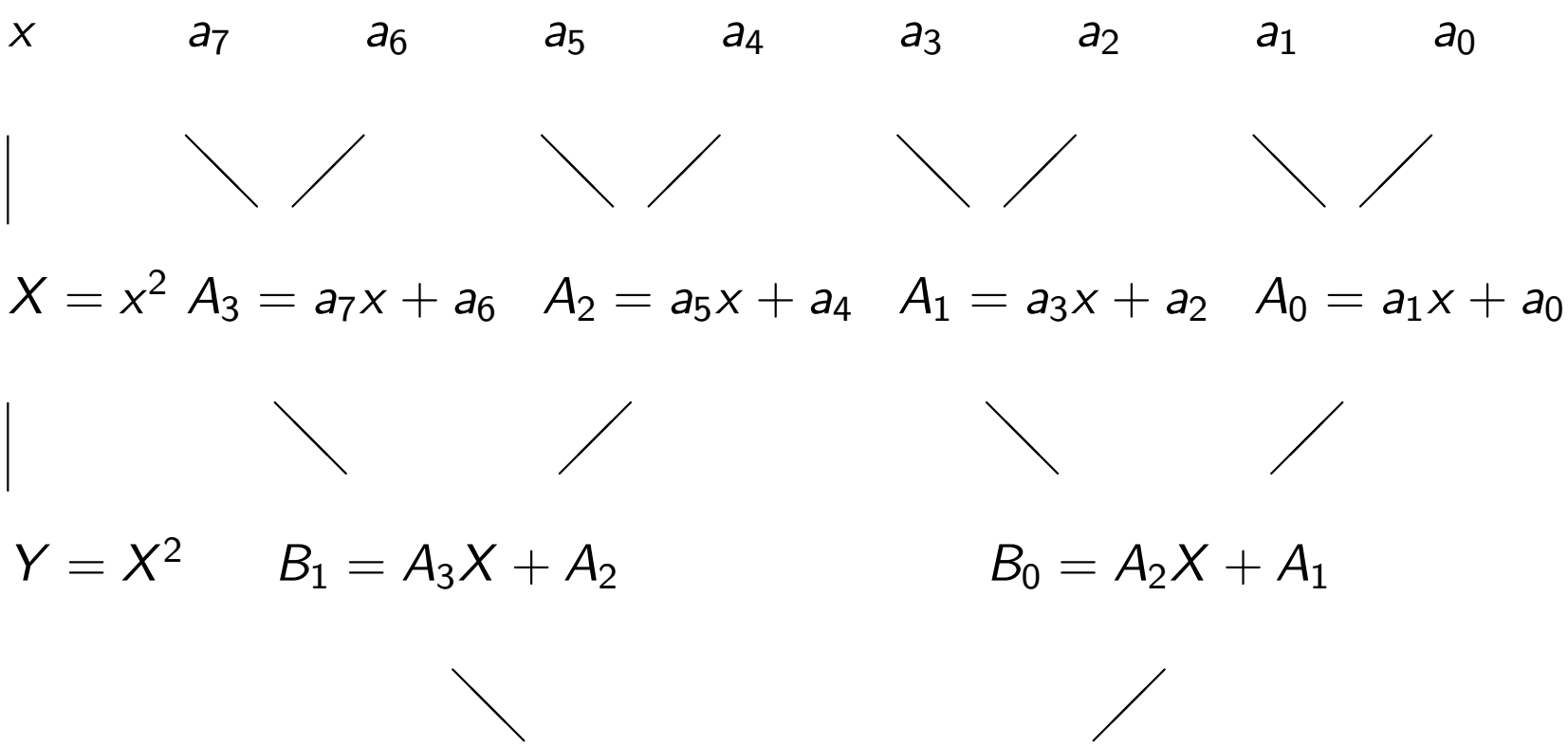
Code using double-extended arithmetic, Estrin evaluation.

- Before last week: a polynomial with double-ext coefficients, computed by Maple
  - Absolute error $2^{-70.9}$, says Maple (our tool says $2^{-72.65}$)
  - Relative error $2^{-62.99}$, says Maple (our tool says $2^{-64.69}$)
- Now: a polynomial with only double coefficients, and with $c_1 = 1$ and $c_2 = -0.5$ (to allow exact multiplications)
  - Absolute error: Our tool says $2^{-72.23}$
  - Relative error: Our tool says $2^{-64.11}$
- Saves 3 cycles on PIII, 18 on P4, how many on Itanium2?

$$p(x) = x * (1 + (x * ((-1b - 1) + (x * (6004799503160663b - 54 + (x * ((-9007199254173073b - 55) + (x * (3602879701310655b - 54 + (x * ((-6004904200786859b - 55) + (x * 40211673751819b - 48)))))))))))$$

Good polynomial $\longrightarrow$ evaluation scheme

Example: Estrin's scheme (one or several fmas and some pipeline)



Change evaluation scheme $\longrightarrow$ recompute new roundoff error bound.

Gappa (Guillaume Melquiond)

`http://lipforge.ens-lyon.fr/www/gappa/` (GNU GPL License)

- Melquiond's PhD defense: November 21, 2006
- manuscript: soon available at `http://www.ens-lyon.fr/LIP/Pub/PhD2006.php`

Gappa: bounds values and rounding errors of a straight-line program. Constraints:

1. robust enough to certify critical numerical applications: generates a formal proof of the bounds it computes, that can be verified by a proof checker like Coq;
2. powerful enough to assist in certifying CRLIBM: the engine handles more than just FP variables, it handles expressions on the set of real numbers. Errors are simply the difference of two expressions: $\tilde{f}(\tilde{x}) - f(x)$;

crlibm's current state

	State of the proof	
	Worst cases	Proof of the code
exp	complete	complete (mostly formal)
expm1	complete	partial
log	complete	complete (mostly formal)
log1p	complete	partial
log2	complete	partial
log10	complete	partial
sin	$[-3.1413, 3.1413]$	complete (paper+formal)
cos	$[-\pi/2, \pi/2]$	complete (paper+formal)
tan	$[-\pi/2, \pi/2]$	complete (paper+formal)
asin	complete	partial
acos	complete	partial
atan	complete	complete (paper)
sinh	complete	complete (paper)
cosh	complete	complete (paper)

crlibm's logarithm is our experimental testbench

- Worst case relative accuracy required: 2^{-118}.
- Initial version:
  - Range reduction by interval splitting,
  - First step in double-double
  - Second step using our Software Carry Save library
  - Manual proof
- Improvements since then
  - Tang's range reduction,
  - Optimized version using double-extended arithmetic
  - Portable version with second step in triple-double
  - Sharing of tables between the two steps
  - Estrin evaluation, then Estrin + Paterson & Stockmayer
  - Machine-assisted proofs using Gappa
  - Portable use of FMA
  - Turned into a "perfect" interval function
  - Validated and tighter infinite norm
  - Double-ext evaluation of a polynomial with only double coefficients

First argument reduction for log

- Extract exponent p and mantissa $1.F$ of x
  - If x is a normal number, $x = 2^p \cdot 1.F$
  - If x is subnormal, $2^{64}x = 2^{p+64} \cdot 1.F$
- Then we want to use $\log(x) = p \cdot \log(2) + \log(1.F)$.
- Pb: catastrophic cancellation for $x = 2^{-1}(1.9999999...)$
  Therefore, first center the mantissa around 1:

$$\begin{cases} E = p, & y = 1.F & \text{if} \quad 1.F < \sqrt{2} \\ E = p + 1 & y = \dfrac{1.F}{2} & \text{if} \quad 1.F > \sqrt{2} \end{cases}$$

  (all exact operations)
- Now $y \in [\frac{\sqrt{2}}{2}, \sqrt{2}]$ and $\log(y) \in [-0.3467, 0.3467]$.

Second argument reduction for log

- define i = the 7 high magnitude bits of the mantissa of y
- use i to look up $r_i \approx \frac{1}{y}$ (on Itanium, Intel uses the frcpa instruction)
- Define $z = y \cdot r_i - 1$,
- reconstruction becomes

$$\log(x) = E \cdot \log(2) + \log(1 + z) - \log(r_i)$$

- $\log(r_i)$ also tabulated (for the first step, and for the second step: shared tables)
- The computation of z is errorless:
  - In the DE code, the r_i are single-precision numbers with at most 10 bits of mantissa. Then the product $y \cdot r_i$ fits in a DE number. The subtraction of 1 is exact (Sterbenz).
  - In the portable version, double-double have to be used, but still with the guarantee that the double-double z is exactly $z = y \cdot r_i - 1$.

Polynomial approximation

Approximate $\log(1+z)$ by a polynomial $p(z)$.

- Portable version:
  - $z = z_h + z_l$
  - First step:
    - degree 7, minimax with first coefficients forced to 1 and 0.5
    - $p(z) = z + 0.5 \cdot z^2 + z_h^3 p_3(z_h)$
  - Second step:
    - degree 14, minimax with first coefficients forced to 1 and 0.5
    - Horner , 5 iterations in double, 7 in DD and 3 in TD
    - C. Lauter has a tool that manages overlaps in TD.
- Double-extended version
  - First step: degree 7 with Estrin + Paterson & Stockmayer evaluation
  - Second step: degree 14 with Horner

Paterson & Stockmayer?

- See Knuth's *Art of Computer Programming*
- Evaluation of an n-degree polynomial in $n/2 + O(\log n)$ multiplications
- Contrary to Horner and Estrin, change of coefficients.

$\longrightarrow$ loss of accuracy

$\longrightarrow$ to be used only on the higher degrees ?

In the log,

$$p(x) = p_l(x) + x^4 p_h(x)$$

$$z^4 p_h(x) \;=\; (z^4 c_7)\left((x^2 + c)(x + \alpha) + (x + \beta)\right)$$

Same number of operations but more independant operations: 3 PIII cycles/18 P4 cycles less than our best Estrin.
Gappa proof available, too.
Still under investigation...

Computation of $E \cdot \log 2$

- $|E| < 1024$,
- The constant $\log 2$ is stored as the sum of two DE or three doubles, each with 10 LSB zeroes.

Current state

Linux, gcc-4.0. The default libm is the microcoded one.

Opteron (cycles)	avg time	max time
`crlibm` using double-extended	118	862
default `libm` (without correct rounding)	189	8050

Pentium 4 (cycles)	avg time	max time
`crlibm` using double-extended	298	3824
default `libm` (without correct rounding)	257	6640
Intel SSE2-based `libm` (RNC7 paper)	≈ 159	?

Pentium 3 (cycles)	avg time	max time
`crlibm` using double-extended	151	1108
default `libm` (without correct rounding)	154	1899

We are sorry we do not have recent results on Itanium, but we don't have a recent Itanium-based machine!

The triple-double version:

Power5 (arbitrary units)	avg time	max time
`crlibm` (without FMA)	50	259
`crlibm` (using FMA)	42	204
default libm (IBM's `libultim`)	52	28881

Pentium III (cycles)	avg time	max time
`crlibm`	252	1296
default `libm`	154	1899

The story so far

The *additional price* of correct rounding wrt existing `libms`:

- no longer poor performance or memory consumption of the function.

The story so far

The *additional price* of correct rounding wrt existing `libms`:

- no longer poor performance or memory consumption of the function.
- but poor performance and coffee consumption of the programmers who have to prove the correct rounding property.

The story so far

The *additional price* of correct rounding wrt existing `libms`:

- no longer poor performance or memory consumption of the function.
- but poor performance and coffee consumption of the programmers who have to prove the correct rounding property.
- "Proof" mostly means a detailed and tight error computation.
- Besides, any later code tweak will ruin most of the proof...

What do we want to prove exactly ?

Theorem

Under the following assumptions:

- *the worst-case relative accuracy needed to decide correct rounding to double-precision of the logarithm of a double-precision number is* 2^{-118}*;*
- *the code was compiled by a C99-compliant compiler;*
- *by default, floating-point operations in the system are double-precision (resp double-extended precision) IEEE-754-compliant;*

then for any double-precision number x*, a call to the portable (resp DE)* `crlibm` *function* `log(x)` *returns the correctly rounded logarithm of* x*.*

What we used to write (here, 3 lines of the sine)

```
yh2 = yh*yh;                                \
ts = yh2 * (s3.d + yh2*(s5.d + yh2*s7.d));          \
Add12(*psh,*psl,    yh,  yl+ts*yh);                 \
```

Upon entering DoSinZero, we have in $y_h + y_l$ an approximation to the ideal reduced value $\hat{y} = x - k\frac{\pi}{256}$ with a relative accuracy $\varepsilon_{\mathrm{argred}}$:

$$y_h + y_l = (x - k\frac{\pi}{256})(1 + \varepsilon_{\mathrm{argred}}) = \hat{y}(1 + \varepsilon_{\mathrm{argred}}) \tag{1}$$

with, depending on the quadrant, $\sin(\hat{y}) = \pm\sin(x)$ or $\sin(\hat{y}) = \pm\cos(x)$ and similarly for $\cos(\hat{y})$. This just means that $\hat{y}$ is the ideal, errorless reduced value.
In the following we will assume we are in the case $\sin(\hat{y}) = \sin(x)$, (the proof is identical in the other cases), therefore the relative error that we need to compute is

$$\varepsilon_{\mathrm{sinkzero}} = \frac{(*\mathtt{psh} + *\mathtt{psl})}{\sin(\mathtt{x})} - 1 = \frac{(*\mathtt{psh} + *\mathtt{psl})}{\sin(\hat{y})} - 1 \tag{2}$$

One may remark that we almost have the same code as we have for computing the sine of a small argument (without range reduction). The difference is that we have as input a double-double $\mathtt{yh} + \mathtt{yl}$, which is itself an inexact term.

At Line 4, the error of neglecting y_l and the rounding error in the multiplication each amount to half an ulp:

$\mathtt{yh2} = \mathtt{yh}^2(1 + \varepsilon_{-53})$, with $\mathtt{yh} = (\mathtt{yh} + \mathtt{yl})(1 + \varepsilon_{-53}) = \hat{y}(1 + \varepsilon_{\mathrm{argred}})(1 + \varepsilon_{-53})$

Therefore

$$\texttt{yh2} = \hat{y}^2(1 + \varepsilon_{\texttt{yh2}}) \tag{3}$$

with

$$\varepsilon_{\texttt{yh2}} = (1 + \varepsilon_{\texttt{argred}})^2(1 + \varepsilon_{-53})^3 - 1 \tag{4}$$

Line 5 is a standard Horner evaluation. Its approximation error is defined by:

$$P_{\texttt{ts}}(\hat{y}) = \frac{\sin(\hat{y}) - \hat{y}}{\hat{y}}(1 + \varepsilon_{\texttt{approxts}})$$

This error is computed in Maple as in **??**, only the interval changes:

$$\varepsilon_{\texttt{approxts}} = \left\| \frac{xP_{\texttt{ts}}(x)}{\sin(x) - x} - 1 \right\|_\infty$$

We also compute $\varepsilon_{\texttt{hornerts}}$, the bound on the relative error due to rounding in the Horner evaluation thanks to the `compute_horner_rounding_error` procedure. This time, this procedure takes into account the relative error carried by `yh2`, which is $\varepsilon_{\texttt{yh2}}$ computed above. We thus get the total relative error on `ts`:

$$\texttt{ts} = P_{\texttt{ts}}(\hat{y})(1 + \varepsilon_{\texttt{hornerts}}) = \frac{\sin(\hat{y}) - \hat{y}}{\hat{y}}(1 + \varepsilon_{\texttt{approxts}})(1 + \varepsilon_{\texttt{hornerts}}) \tag{5}$$

The final Add12 is exact. Therefore the overall relative error is:

$$\begin{aligned}
\varepsilon_{\mathrm{sinkzero}} &= \frac{((\mathtt{yh} \otimes \mathtt{ts}) \oplus \mathtt{yl}) + \mathtt{yh}}{\sin(\hat{y})} - 1 \\
&= \frac{(\mathtt{yh} \otimes \mathtt{ts} + \mathtt{yl})(1 + \varepsilon_{-53}) + \mathtt{yh}}{\sin(\hat{y})} - 1 \\
&= \frac{\mathtt{yh} \otimes \mathtt{ts} + \mathtt{yl} + \mathtt{yh} \ + \ (\mathtt{yh} \otimes \mathtt{ts} + \mathtt{yl}).\varepsilon_{-53}}{\sin(\hat{y})} - 1
\end{aligned}$$

Let us define for now

$$\delta_{\mathrm{addsin}} = (\mathtt{yh} \otimes \mathtt{ts} + \mathtt{yl}).\varepsilon_{-53} \tag{6}$$

Then we have

$$\varepsilon_{\mathrm{sinkzero}} = \frac{(\mathtt{yh} + \mathtt{yl})\mathtt{ts}(1 + \varepsilon_{-53})^2 + \mathtt{yl} + \mathtt{yh} \ + \ \delta_{\mathrm{addsin}}}{\sin(\hat{y})} - 1$$

Using (1) and (5) we get:

$$\varepsilon_{\mathrm{sinkzero}} = \frac{\hat{y}(1 + \varepsilon_{\mathrm{argred}}) \times \frac{\sin(\hat{y}) - \hat{y}}{\hat{y}}(1 + \varepsilon_{\mathrm{approxts}})(1 + \varepsilon_{\mathrm{hornerts}})(1 + \varepsilon_{-53})^2 + \mathtt{yl} + \mathtt{yh} \ + \ \delta_{\mathrm{addsin}}}{\sin(\hat{y})} - 1$$

To lighten notations, let us define

$$\varepsilon_{\mathrm{sin1}} = (1 + \varepsilon_{\mathrm{approxts}})(1 + \varepsilon_{\mathrm{hornerts}})(1 + \varepsilon_{-53})^2 \ - \ 1 \tag{7}$$

We get

$$\begin{aligned}\varepsilon_{\mathrm{sinkzero}} &= \frac{(\sin(\hat{y})-\hat{y})(1+\varepsilon_{\mathrm{sin1}})+\hat{y}(1+\varepsilon_{\mathrm{argred}}) \;+\; \delta_{\mathrm{addsin}} - \sin(\hat{y})}{\sin(\hat{y})} \\ &= \frac{(\sin(\hat{y})-\hat{y}).\varepsilon_{\mathrm{sin1}}+\hat{y}.\varepsilon_{\mathrm{argred}} \;+\; \delta_{\mathrm{addsin}}}{\sin(\hat{y})}\end{aligned}$$

Using the following bound:

$$|\delta_{\mathrm{addsin}}| = |(\mathtt{yh}\otimes\mathtt{ts}+\mathtt{yl}).\varepsilon_{-53}| \quad < \quad 2^{-53}\times|y|^3/3 \tag{8}$$

we may compute the value of $\varepsilon_{\mathrm{sinkzero}}$ as an infinite norm under Maple. We get an error smaller than 2^{-67}.

4 pages for 3 lines of code...

Nobody (including myself) should trust such a proof

4 pages for 3 lines of code...

Nobody (including myself) should trust such a proof
. . . and nobody reads it anyway.

4 pages for 3 lines of code...

Nobody (including myself) should trust such a proof
. . . and nobody reads it anyway.

- People (from Intel!) have found bugs in our first proven code!
- In the early version of `crlibm`, the paper proof was essentially used as
  - an extensive documentation
  - a means to fine-tune performance
  - a teaching tool

4 pages for 3 lines of code...

Nobody (including myself) should trust such a proof
. . . and nobody reads it anyway.

- People (from Intel!) have found bugs in our first proven code!
- In the early version of `crlibm`, the paper proof was essentially used as
  - an extensive documentation
  - a means to fine-tune performance
  - a teaching tool
- ... But can it be taken as a proof?

Lack of confidence $\Longrightarrow$ big players reluctant to claim correct rounding

First step to improve confidence (crlibm 0.8)

- For each function, a self-contained Maple script
  - produces all the constants in the code,
  - computes approximation errors,
  - and implements the error computation out of them
- Thus the error computation uses the same numerical values as the code.

First step to improve confidence (crlibm 0.8)

- For each function, a self-contained Maple script
  - produces all the constants in the code,
  - computes approximation errors,
  - and implements the error computation out of them
- Thus the error computation uses the same numerical values as the code.
- Still, three suspiscious layers of human intervention:
  - Write the code (*C*);
  - Write the (paper) error computation (*LaTeX*);
  - Implement it (*Maple*).

The Arénaire touch

We wish we had an automatic tool that

- takes a set of C files,
- parses them,
- and outputs "The overall error of the computation is ...".

It's hopeless. Our current approach (`crlibm`0.14β):

- Keep Maple for symbolic computation, but not more.
- Keep paper for code structure, but not more.
- Develop tools for polynomial approximations under constraints.
- Develop tools for validated approximation error.
- Develop tools for machine-assisted handling of rounding errors.

Approximation errors

- Until this year, we had to trust Maple's infnorm
  - usually overestimation... but sometimes lower than the actual infnorm
  - fair confidence by increasing the precision until digits agree... but this does not prove anything
- We now have a validated infinite norm (Chevillard, Lauter)
  - based on multiple-precision interval arithmetic
  - not universal, but good enough for elementary functions
  - example: for the `log-de` polynomial, Maple was pessimistic by 2 bits.
  - more confidence, and lower overall error bound.

Now for the rounding errors.

Automatic proof assistants are not there yet

Previous related work:

- Basic floating-point arithmetic (Daumas, Thierry, Boldo)
  - Proving Sterbenz Lemma (one operation) is worth a full paper.
- John Harrison at Intel (exponential function);
- Current proof assistants are comparable to assembly language.

From clean math to messy code, 1: the clean math

The math for `crlibm` portable log:

- Use i to look up $r_i \approx \frac{1}{y}$
- Define $z = y \cdot r_i - 1$
- Reconstruction becomes

$$\log(x) = E \cdot \log(2) + \log(1+z) - \log(r_i)$$

- $\log(r_i)$ also tabulated
- $\log(1+z)$ is approximated with a 7-degree polynomial $p(z)$

From clean math to messy code, 2: the mess is coming

The implementation:

$$r_h + r_l \approx E \cdot \log(2) + p(z) - \log(r_i)$$

- $\log(2)$ approximated by a special overlapping double-double
- $z = z_h + z_l$ is an (exact) double-double
- $\log(r_i)$ approximated by a double-double
- Evaluation of $p(z)$:
  - degrees 3 to 7 evaluated using Horner's scheme
  - $p(z) \approx z - \frac{1}{2} \cdot z^2 + z^3 \cdot q(z)$
  - z^2 approximated as by $z^2 = (z_h + z_l)^2 \approx z_h^2 + 2 \cdot z_h \otimes z_l$
  - z_h^2 is computed exactly as a double-double, and multiplied exactly by $1/2$
  - $q(z)$ approximated as $q(z_h)$
  - and z^3 approximated as $z^3 \approx (z_h \otimes z_h) \otimes z_h$ where $z_h \otimes z_h$ has already been computed

From clean math to messy code, 3: the messy code

Mul12 and Add12 are aka FastTwoSum and TwoProd, Add22 and Mul22 are double-double operations

```
polyHorner = c3 + zh * (c4 + zh * (c5 + zh * (c6 + zh * c7)));
Mul12(&zhSquareh, &zhSquarel, zh, zh);
polyUpper = polyHorner * (zh * zhSquareh);
zhSquareHalfh = zhSquareh * -0.5;
zhSquareHalfl = zhSquarel * -0.5;
Add12(t1h, t1l, polyUpper, -1 * (zh * zl));
Add22(&t2h, &t2l, zh, zl, zhSquareHalfh, zhSquareHalfl);
Add22(&ph, &pl, t2h, t2l, t1h, t1l);
Add12(log2edh, log2edl, log2h * ed, log2m * ed);
Add22Cond(&logTabPolyh, &logTabPolyl, logih, logim, ph, pl);
Add22Cond(&logh, &logm, log2edh, log2edl, logTabPolyh, logTabPolyl);
```

What is the error of what with regard to what?

Gappa overview

Written by Guillaume Melquiond at Arénaire, Gappa is a tool that

- uses *interval arithmetic* to manage ranges and errors

Gappa overview

Written by Guillaume Melquiond at Arénaire, Gappa is a tool that

- uses *interval arithmetic* to manage ranges and errors
- takes an input which may closely match your C file

Gappa overview

Written by Guillaume Melquiond at Arénaire, Gappa is a tool that

- uses *interval arithmetic* to manage ranges and errors
- takes an input which may closely match your C file
- and outputs "The overall error of the computation is somewhere in this interval"
  - or any other theorem that can be expressed in terms of intervals

Gappa overview

Written by Guillaume Melquiond at Arénaire, Gappa is a tool that

- uses *interval arithmetic* to manage ranges and errors
- takes an input which may closely match your C file
- and outputs "The overall error of the computation is somewhere in this interval"
  - or any other theorem that can be expressed in terms of intervals
- also outputs a proof suitable for checking by Coq
  - (or PVS or possibly others, says Guillaume)

`http://lipforge.ens-lyon.fr/projects/gappa/`

Automatic ? No way

Gappa input:

- First you take your C code and convert it to Gappa input
- Then you add a mathematical definition of what this code is supposed to approximate
- Then you provide a theorem to prove

Automatic ? No way

Gappa input:

- First you take your C code and convert it to Gappa input
- Then you add a mathematical definition of what this code is supposed to approximate
- Then you provide a theorem to prove

Then you run Gappa

Automatic ? No way

Gappa input:

- First you take your C code and convert it to Gappa input
- Then you add a mathematical definition of what this code is supposed to approximate
- Then you provide a theorem to prove

Then you run Gappa, and Gappa says "No proof"

Gappa initial input for log-de.c

```
@IEEEdouble = float<ieee_64,ne>;
@IEEEext = float<x86_80,ne>;

# Actual values will be replaced by sed scripts
c2 = IEEEext(_c2);
c3 = IEEEext(_c3);
(...)
c7 = IEEEext(_c7);
log2h = IEEEext(_log2h);
log2l  = IEEEext(_log2l);
r      = IEEEext(_rval);    # ri
logirh = IEEEext(_logirh); # log(1/ri)
logirl = IEEEext(_logirl);

# Transcription of the code, NOT using FMA
z2   IEEEext= z*z;
p67  IEEEext= c6 + z*c7;
p45  IEEEext= c4 + z*c5;
p23  IEEEext= c2 + z*c3;
p01  IEEEext= logirh + z;
z4   IEEEext= z2*z2;
p47  IEEEext= p45 + z2*p67;
p03  IEEEext= p01 + z2*p23;
p07  IEEEext= p03 + z4*p47;
logz IEEEext= p07 + E*log2h;

# The mathematical polynomial in Estrin form
Mz2 = z*z;
Mz4 = Mz2*Mz2;
PolyLog1pz = z + Mz2*(c2+z*c3) + Mz4*( (c4+z*c5) + Mz2*(c6+z*c7)  );

# Exact mathematical definition of the log
Mlogz = Log1pz + Logir + E*Mlog2;

epsilon = (logz - Mlogz)/Mlogz;
```

Gappa initial input for log-de.c

```
1  # A theorem to prove { |E| in [1,1024] /\  |z| in [1b-200, _zabsmax]
2  /\  Log2hl - Mlog2    in [-1b-129, 1b-129] /\  Logirhl - Logir in
3  [-1b-129, 1b-129] /\  (PolyLog1pz - Log1pz) in [-_deltaApproxQuick,
4  _deltaApproxQuick] -> epsilon in [-1b-62, 1b-62] }
```

Automatic ? No way

Gappa input (continued):

- Run Gappa. Gappa says "No proof"

Automatic ? No way

Gappa input (continued):

- Run Gappa. Gappa says "No proof"
- Ask him "why can't you find a proof ?" (in Gappa language)

Automatic ? No way

Gappa input (continued):

- Run Gappa. Gappa says "No proof"
- Ask him "why can't you find a proof ?"
  (in Gappa language)
- Analyse and give a little more help
  - which usually reflects some knowledge about the problem which was not explicit yet

Automatic ? No way

Gappa input (continued):

- Run Gappa. Gappa says "No proof"
- Ask him "why can't you find a proof ?"
  (in Gappa language)
- Analyse and give a little more help
  - which usually reflects some knowledge about the problem which was not explicit yet
- Go back to step 1

Automatic ? No way

Gappa input (continued):

- Run Gappa. Gappa says "No proof"
- Ask him "why can't you find a proof ?"
  (in Gappa language)
- Analyse and give a little more help
  - which usually reflects some knowledge about the problem which was not explicit yet
- Go back to step 1
- After a few iterations you have an input which is 50 times the size of your C code (but grew in small, easy steps)

Automatic ? No way

Gappa input (continued):

- Run Gappa. Gappa says "No proof"
- Ask him "why can't you find a proof ?" (in Gappa language)
- Analyse and give a little more help
  - which usually reflects some knowledge about the problem which was not explicit yet
- Go back to step 1
- After a few iterations you have an input which is 50 times the size of your C code (but grew in small, easy steps)
- Then suddenly the proof is complete.

Automatic ? No way

Gappa input (continued):

- Run Gappa. Gappa says "No proof"
- Ask him "why can't you find a proof ?" (in Gappa language)
- Analyse and give a little more help
  - which usually reflects some knowledge about the problem which was not explicit yet
- Go back to step 1
- After a few iterations you have an input which is 50 times the size of your C code (but grew in small, easy steps)
- Then suddenly the proof is complete.

To Florent's complete astonishment, it is very easy to use *without understanding in detail how it works*.

How does Gappa work ?

Use interval arithmetic, and explore expression rewriting to remove decorrelations.

- Predefined set of rewriting rules:
  - `IEEEdouble(a)- b ->(IEEEdouble(a)- a)+ (a - b);`
  - ...
  - (a *convergent* set of rewriting rules)

How does Gappa work ?

Use interval arithmetic, and explore expression rewriting to remove decorrelations.

- Predefined set of rewriting rules:
  - `IEEEdouble(a)- b ->(IEEEdouble(a)- a)+ (a - b);`
  - ...
  - (a *convergent* set of rewriting rules)
- Support library of theorems (*with their Coq proofs*):
  - Theorems giving the errors when rounding
    - `a in [...] ->(IEEEdouble(a)-a)/a in [...]`
    - Note how this takes care of dangerous cases (subnormal numbers, over/underflows...)
  - Classical theorems like Sterbenz Lemma
- To complete a proof, all you need to add is the relevant rewriting rules
  - mathematical identities which will lead to smaller intervals

Proof graph

- Internally, a proof graph is built to remember the logical progression.
- Branches are cut when either a shorter path, or a better interval are found.

Proof graph

- Internally, a proof graph is built to remember the logical progression.
- Branches are cut when either a shorter path, or a better interval are found.
  - Rewriting rules should either be convergent, or contract the interval

Proof graph

- Internally, a proof graph is built to remember the logical progression.
- Branches are cut when either a shorter path, or a better interval are found.
  - Rewriting rules should either be convergent, or contract the interval
- The final graph will be used to generate the formal proof.

Conclusion: it does improve confidence

Why we can attack real-size problems:

- there is an incremental path to complex proofs
  - add rewriting rules
- in the process, the tool is easy to question
  - add to the theorem to prove a goal of the form `x in ?`

Why we trust the result:

- Confidence in interval arithmetic
- Syntax such that we are confident that the initial input was a good description of the C code
- Confidence that nothing was forgotten
- The rewriting rule you need to provide are all mathematical identities (Gappa checks)
- And if there is a bug, the final proof is checked by Coq anyway

A higher-level language for formal proofs

- You don't have to write Coq
  - (the assembly language of formal proofs)
- You end up with a proof faster than if you had done it by hand
- It teaches a few good habits in the process
  - (how to write better paper proofs)
- And it's even fun...

In crlibm

- Already Gappa-powered:
  - bits of the first step of the trigs
  - First step of `log-de`
    - Including some Estrin (parallel Horner) polynomial evaluation
  - bits of `log-td, exp-td, asin-td`
    - Including triple-double arithmetic

In crlibm

- Already Gappa-powered:
  - bits of the first step of the trigs
  - First step of `log-de`
    - Including some Estrin (parallel Horner) polynomial evaluation
  - bits of `log-td, exp-td, asin-td`
    - Including triple-double arithmetic
- Work on the *proof style* still in progress...

In crlibm

- Already Gappa-powered:
  - bits of the first step of the trigs
  - First step of `log-de`
    - Including some Estrin (parallel Horner) polynomial evaluation
  - bits of `log-td, exp-td, asin-td`
    - Including triple-double arithmetic
- Work on the *proof style* still in progress...
- Confidence that it will help everywhere,

In crlibm

- Already Gappa-powered:
  - bits of the first step of the trigs
  - First step of `log-de`
    - Including some Estrin (parallel Horner) polynomial evaluation
  - bits of `log-td, exp-td, asin-td`
    - Including triple-double arithmetic
- Work on the *proof style* still in progress...
- Confidence that it will help everywhere,
- but a new surprise (and more work for Guillaume) at each new function.