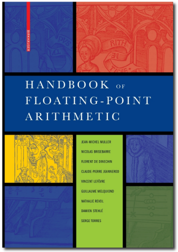
From CRLibm to Metalibm : assisting the production of high-performance proven floating-point code

Florent de Dinechin
AriC project

My research group

The AriC project @ École Normale Supérieure de Lyon :
Computer Arithmetic at large

- Hardware and software
- From addition to linear algebra
- Fixed point, floating-point, multiple-precision, finite fields, ...
- Pervasive concern of performance, numerical quality and validation



Outline

Introduction : performance versus accuracy

Elementary function evaluation

Correctly rounded functions computing just right

Open-source tools for FP coders

Formal proof of floating-point code for the masses

Two metalibm prototypes

Conclusion

Introduction : performance versus accuracy

Bottom line of this talk

Common wisdom

The more accurate you compute, the more expensive it gets

In practice

- We (hopefully) notice it when our computation is not accurate enough.
- But do we notice it when it is too accurate for our needs ?

Reconciling performance and accuracy ?

Or, regain performance by computing just right ?

Double precision spoils us

The standard binary64 format (formerly known as double-precision) provides roughly 16 decimal digits.

Why should anybody need such accuracy ?

Count the digits in the following

- Definition of the second : *the duration of 9,192,631,770 periods of the radiation corresponding to the transition between the two hyperfine levels of the ground state of the cesium 133 atom.*
- Definition of the metre : *the distance travelled by light in vacuum in 1/299,792,458 of a second.*
- Most accurate measurement ever (another atomic frequency) to 14 decimal places
- Most accurate measurement of the Planck constant to date : to 7 decimal places
- The gravitation constant G is known to 3 decimal places only

Parenthesis : then why binary64 ?

- This PC computes 10^9 operations per second (1 gigaflops)

An allegory due to Kulisch

- print the numbers in 100 lines of 5 columns double-sided : 1000 numbers/sheet
- 1000 sheets $\approx$ a heap of 10 cm
- 10^9 flops $\approx$ heap height speed of 100m/s, or 360km/h
- A teraflops (10^{12} op/s) prints to the moon in one second
- Current top 500 computers reach the petaflop (10^{15} op/s)

- each operation may involve a relative error of 10^{-16}, and they accumulate.

Doesn't this sound wrong ?

We would use these 16 digits just to accumulate garbage in them ?

Back to the point

… which was :

Mastering accuracy for performance

When implementing a "computing core"

- A goal : *never compute more accurately than needed*
- Two sub-goals
  - Know what accuracy you need
  - Know how accurate you compute

"Computing cores" considered so far : elementary functions, sums of products, linear algebra, Euclidean lattices algorithms.

By the way

"computing just right" implies "computing right"…

A general technique for computing just right

I've seen it for orientation predicates, area of a triangle, elementary functions...

Fast in average, always accurate

1. use a quick and dirty routine
2. runtime-test if it was accurate enough
3. launch an expensive, accurate routine only when needed

If done well, average time is close to that of the quick routine

Only works if you know how to implement step 2

... requires to understand/master/engineer the accuracy of your code.

Elementary function evaluation

How does your PC compute elementary functions ?

Rule of the game : use the hardware, i.e. $+$, $-$, $\times$

(and maybe $/$ and $\sqrt{\ }$ but they are expensive).

- Polynomial approximation works on a small interval
- Argument reduction : using mathematical identities, transform large arguments in small ones

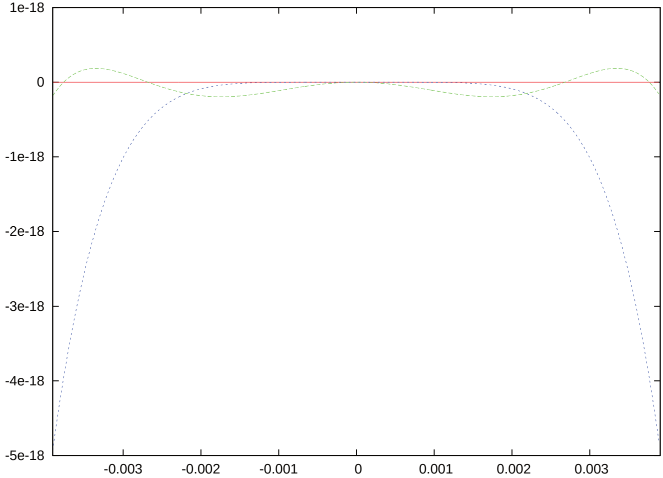
Simplistic example : an exponential

- identity : $e^{a+b} = e^a \times e^b$
- split $x = a + b$
  - a : k leading bits of x
  - b : lower bits of x $\quad b << 1$
- tabulate all the e^a $\quad(2^k$ entries)
- use a Taylor polynomial for e^b

Know how accurate you compute

- Approximation errors
  - example : approximate a function f with a polynomial p :
    $||p - f||_\infty$?
    (see next slide)
  - in general : approximate an object by another one
- Rounding errors
  - for data, often called quantization errors ;
  - for operations, each individual error well specified by IEEE-754
  - but their accumulation difficult to manage
- In physics : time discretization errors, etc

Approximation of a function by a polynomial



$||p - f||_\infty$ for Taylor and Remez approximation (exp on $[-2^{-8}, 2^{-8}]$)

What is an error ? What is accuracy ?

The most important sentence of this talk

An error is a difference (absolute or relative) between two values, one being a reference for the other.

Examples :

- error of the FP addition is with reference of the real sum (easy)
- error of the polynomial is with reference to the function (easy)
- error of one FP addition within the polynomial evaluation ? (difficult because we have no direct reference in the function)
- yesterday : accuracy of the summation algorithms ?

Never say "the error of this term is ..." :

it doesn't mean anything without the reference.

If you are not able to define the reference value,

you will not be able to know how accurate you compute

Parenthesis : reproductibility and predictability

As soon as we are able to define the reference value,

- Who cares about exact reproductibility ?
- What matters is to be able to reproduce enough significant digits.
- Martyn's compiler will not help you there :

  his compiler has no access to the reference !

Let us take a simple example

This is part of the code of `sin`,

after y has been reduced to $[-\pi/256, \pi/256]$:

```
s3 = -0.16666666666666665741480812812369549646973609924;
s5 =  8.3333333262892793358300735917509882710874081e-3;
s7 = -1.98400103113668426196153360407947729981970042e-4;

y2 = y * y;
ts = y2 * (s3 + y2*(s5 + y2*s7));
r  = y + y*ts
```

- evaluation of sine as an odd polynomial
  $p(y) = y + s_3 y^3 + s_5 y^5 + s_7 y^7$
  (think Taylor for now)
- reparenthesized as $p(y) = y + yt(y^2)$ to save operations
- `y + y*ts` is more accurate than `y*(1+ts)` in floating-point, do you see why ?

Rounding errors piled over approximations

```
s3 = -0.1666666666666666574148081281236954964697360992 4;
s5 =  8.33333333262892793358300735917509882710874081e-3;
s7 = -1.98400103113668426196153360407947729981970042e-4;

y2 = y * y;
ts = y2 * (s3 + y2*(s5 + y2*s7));
r  = y + y*ts
```

- This polynomial is an approximation to $sin(y)$
- Oops, I wrote its coefficients in decimal !
- if x was not in $[-\pi/256, \pi/256]$, y is not the ideal reduced argument Y (such that $x = Y + k\frac{\pi}{256}$)
- We have a rounding error in computing y^2
- y2 already stacks two errors. We evaluate *ts* out of it
- There is a rounding error hidden in each operation.

How many correct bits at the end ?

What this code doesn't tell

The context

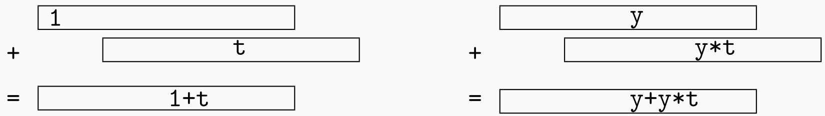
$y \in [-\pi/256, \pi/256]$

What it is supposed to compute

a sine accurate to 2^{-60}

My programmer expertise

`y*(1+ts)` is a bit less accurate than `y + y*ts` in floating-point
... because $|t| < 2^{-14}$ because $|y| < 2^{-7}$



On the positive side : combining errors is easy

Since an error is a difference :

$$F(x) - f(x) \quad = \quad F(x) - p(x) \quad + \quad p(x) - f(x)$$

(rounding error + polynomial approximation error)

$$|F(x) - f(x)| \quad \leq \quad |F(x) - p(x)| \quad + \quad |p(x) - f(x)|$$

... then recurse on $F(x) - p(x)$

Difficulties

- define "intermediate reference values"
- do not forget anything
- relative errors :

$$\frac{a-c}{c} \quad = \quad \frac{a-b}{b} + \frac{b-c}{c} + \frac{a-b}{b} \times \frac{b-c}{c}$$

Later in this talk : **Gappa, a tool that helps you with all this**

Correctly rounded functions computing just right

Know what accuracy you need ?

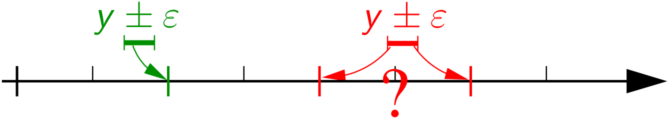
Correctly rounded elementary functions

- IEEE-754 floating-point single or double-precision
- Elementary functions : sin, cos, exp, log, implemented in the "standard mathematical library" (`libm`)
- Correctly rounded : As perfect as can be, considering the finite nature of floating-point arithmetic
  - same standard of quality as $+, \times, /, \sqrt{\ }$
- Now recommended by the IEEE754-2008 standard, but long considered too expensive

  because of the Table Maker's Dilemma

The Table Maker's Dilemma

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



Dilemma if this interval contains a midpoint between two FP numbers

The first digital signature algorithm

LOGARITHMICA. 25

Tabulæ inventioni Logarithmorum inserviens.

1	0,00		100001	0,00000,43429,2	
2	0,30102,99956,6		100002	0,00000,86858,0	
3	0,47712,12547,2		100003	0,00001,30286,4	
4	0,60205,99903,3		100004	0,00001,73714,3	
5	0,69897,00043,4		100005	0,00002,17141,8	F
6	0,77815,12503,8	A	100006	0,00002,60568,9	
7	0,84509,80400,1		100007	0,00003,03995,5	
8	0,90308,99869,9		100008	0,00003,47421,7	
9	0,95424,25094,4		100009	0,00003,90847,4	
11	0,04139,26851,6		1000001	0,00000,04342,9	
12	0,07918,12460,5		1000002	0,00000,08685,9	
13	0,11394,33523,1		1000003	0,00000,13028,8	
14	0,14612,80356,8		1000004	0,00000,17371,7	
15	0,17609,12590,6	B	1000005	0,00000,21714,7	G
16	0,20411,99826,6		1000006	0,00000,26057,6	
17	0,23044,89213,8		1000007	0,00000,30400,5	
18	0,25527,25051,0		1000008	0,00000,34743,4	
19	0,27875,36009,5		1000009	0,00000,39086,3	
101	0,00432,13737,8		10000001	0,00000,00434,3	
102	0,00860,01717,6		10000002	0,00000,00868,6	
103	0,01283,72247,1		10000003	0,00000,01302,9	
104	0,01703,33393,0		10000004	0,00000,01737,2	
105	0,02118,92990,7	C	10000005	0,00000,02171,5	H
106	0,02530,58652,6		10000006	0,00000,02605,8	
107	0,02938,37776,9		10000007	0,00000,03040,1	
108	0,03342,37554,9		10000008	0,00000,03474,4	
109	0,03742,64979,4		10000009	0,00000,03908,6	
1001	0,00043,40774,8		100000001	0,00000,00043,4	
1002	0,00086,77215,3		100000002	0,00000,00086,9	
1003	0,00130,09330,2		100000003	0,00000,00130,3	
1004	0,00173,37128,1		100000004	0,02000,00173,7	
1005	0,00216,60617,6	D	100000005	0,00000,00217,1	I
1006	0,00259,79807,2		100000006	0,00000,00260,6	
1007	0,00302,94705,5		100000007	0,00000,00304,0	
1008	0,00346,05321,1		160000008	0,00000,00347,4	
1009	0,00389,11662,4		100000009	0,00000,00390,9	
10001	0,00004,34272,8		1000000001	0,00000,00004,3	
10002	0,00008,68502,1		1000000002	0,00000,00008,7	
10003	0,00013,02688,1		1000000003	0,00000,00013,0	
10004	0,00017,36830,6		1000000004	0,00000,00017,4	
10005	0,00021,70029,7	E	1000000005	0,00000,00021,7	K
10006	0,00026,04985,5		1000000006	0,00000,00026,1	
10007	0,00030,38997,8		1000000007	0,00000,00030,4	
10008	0,00034,72966,9		1000000008	0,00000,00034,7	
10009	0,00039,06892,5		1000000009	0,00000,00039,1	

- I want 12 significant digits
- I have an approximation scheme that provides 14 digits
- or,

$$y = \log(x) \pm 10^{-14}$$

- "Usually" that's enough to round

$$y = x, xxxxxxxxxx17 \pm 10^{-14}$$

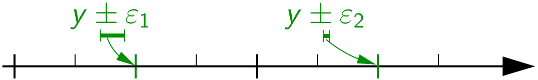
$$y = x, xxxxxxxxxx83 \pm 10^{-14}$$

- Dilemma when

$$y = x, xxxxxxxxxx50 \pm 10^{-14}$$

The first table-makers rounded these cases randomly, and recorded them to confound copiers.

Solving the table maker's dilemma



Ziv's onion peeling algorithm

1. Initialisation : $\varepsilon = \varepsilon_1$
2. Compute y such that $f(x) = y \pm \varepsilon$
3. Does $y \pm \varepsilon$ contain the middle point between two FP numbers ?
   - If no, return RN(y)
   - If yes,dilemma ! Reduce ε, and go back to 2

It is a *while* loop... we have to show it terminates, a topic in itself.

Accuracy versus performance

CRLibm : 2-step approximation process

- first step fast but accurate to $\overline{\varepsilon}_1$

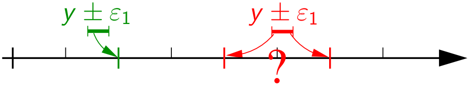
  sometimes not accurate enough
- (rarely) second step slower but always accurate enough

$$T_{\text{avg}} = T_1 + p_2 T_2$$

For each step, we need a tight bound on the error of the code :

$$\left|\frac{F(x) - f(x)}{f(x)}\right| \leq \overline{\varepsilon}$$

- Overestimating $\overline{\varepsilon}_2$ degrades T_2 ! (common wisdom)
- Overestimating $\overline{\varepsilon}_1$ degrades p_2 !



First function development in Arénaire

First correctly rounded elementary function in CRLibm

- `exp` by David Defour
- worst-case time $T_2 \approx 10{,}000$ cycles
- complex, hand-written proof
- duration : a Ph.D. thesis (2002)

Conclusion was :

- performance and memory consumption of CR elem function is OK
- problem now is : performance and coffee consumption of the programmer (and that is because of the need for tight error bounds)

Latest CR function developments in Arénaire

C. Lauter at the end of his PhD,

- development time for sinpi, cospi, tanpi : 2 days
- worst-case time $T_2 \approx 1{,}000$ cycles

(but as a result of three more PhDs)

Open-source tools for FP coders

The GMP family

- GMP (GNU Multiple Precision) and its beautiful C++ wrapper
  - integer arithmetic
  - best asymptotic algorithms + lower layers in hand-crafted assembly code
- MPFR : Multiple Precision Floating-point correctly Rounded
  - a floating-point layer on top of GMP
  - IEEE 754-like specification
  - Now a dependency of GCC, so you probably have it installed
- MPFI : interval arithmetic on top of MPFR

Sollya (1)

Open-source, LGPL, `http://sollya.gforge.inria.fr/`

The Swiss Army Knife of the libm developer (Lauter, Chevillard, Joldes)

Killer feature 1

apologizes each time it rounds something

```
fdedinec@krupnik: sollya
> 1+1;
2
> 1/3;
Warning: rounding has happened. The value displayed is a
    faithful rounding of the true result.
0.33333333333333333333333333333333333333333333333333
```

The Patriot bug

In 1991, a Patriot missile failed to intercept a Scud, and 28 people were killed.

- The code worked with time increments of 0.1 s.
- But 0.1 is not representable in binary.
- In the 24-bit format used, the number stored was 0.099999904632568359375
- The error was 0.0000000953.
- After 100 hours = 360,000 seconds, time is wrong by 0.34s.
- In 0.34s, a Scud moves 500m

In single, we don't have that many bits to accumulate garbage in them !

Test : which of the following increments should you use ?

10 5 3 1 0.5 0.25 0.2 0.125 0.1

Sollya (2)

Killer feature 2

multiple-precision, last-bit accurate evaluation of arbitrary expressions

```
fdedinec@krupnik: sollya
> e=exp(x) - (1+x+x^2/2+x^3/6);
> e(0.125);
Warning: rounding has happened. The value displayed is a
    faithful rounding of the true result.
1.04322334929834956738944784605392321697984118482926e-5
>
```

All these digits are meaningful ! This is better than Maple.

Sollya (3)

Killer feature 3

guaranteed infinite norm $||f(x)||_\infty$ even in degenerate cases

- $||f(x) - P(x)||_\infty$ is a degenerate case...

Sollya (4)

Killer feature 4

Machine-efficient polynomial approximation

- Remez' minimax algorithm finds the best polynomial approximation over the reals
- But we need polynomials with machine coefficients
  - `float`, `double`, fixed-point, ...
- Rounding Remez coefficients does not provide the best polynomial among polynomial with machine coefficients.
- Sollya does (almost).
  - this saves a few bits of accuracy
  - especially relevant for small precisions (FPGAs)
  - that's how we get our polynomials

Nice number theory behind. And needs all the previous.

6 guaranteed log polynomials on one slide

A sollya script that computes appproximations to the log of various qualities

```
f= log(1+y);
I=[-0.25;.5];
filename="/tmp/polynomials";
print("") > filename;
for deg from 2 to 8 do begin
  p = fpminimax(f, deg,[|0,23...|],I, floating, absolute);
  display=decimal;
  acc=floor(-log2(sup(supnorm(p, f, I, absolute, 2^(-40)))));
  print( "   // degree = ", deg,
         "  => absolute accuracy is ",  acc, "bits" ) >> filename;
  print("#if ( DEGREE ==", deg, ")") >> filename;
  display=hexadecimal;
  print("   float p = ", horner(p) , ";") >> filename;
  print("#endif") >> filename;
end;
```

Formal proof of floating-point code for the masses

crlibm.pdf 5 years ago : 124 pages of this

```
1    yh2 = yh*yh;                                    \
2    ts = yh2 * (s3.d + yh2*(s5.d + yh2*s7.d));      \
3    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 $\varepsilon_{\text{argred}}$:

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

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

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

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 $\texttt{yh} + \texttt{yl}$, 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 :

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

Therefore

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

with

$$\overline{\varepsilon}_{\mathtt{yh2}} = (1 + \overline{\varepsilon}_{\mathrm{argred}})^2(1 + \overline{\varepsilon}_{-53})^3 - 1 \tag{4}$$

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

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

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

$$\overline{\varepsilon}_{\mathrm{approxts}} = \left\| \frac{x P_{\mathtt{ts}}(x)}{\sin(x) - x} - 1 \right\|_{\infty}$$

We also compute $\overline{\varepsilon}_{\mathrm{hornerts}}$, 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 $\overline{\varepsilon}_{\mathtt{yh2}}$ computed above. We thus get the total relative error on `ts` :

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

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

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

Let us define for now

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

Then we have

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

Using (1) and (5) we get :

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

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

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

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 $\overline{\varepsilon}_{\mathrm{sinkzero}}$ as an infinite norm under Maple. We get an error smaller than 2^{-67}.

4 pages for 3 lines of code...

Two years of experience showed that nobody (including myself) should trust such a proof (and that nobody reads it anyway).

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, of course :

- Where, in your code, can you read what it is supposed to compute ?
- Most of the knowledge used to build the code is not in the code

Trusted error computation means : formal proof

but... automatic proof assistants are not there yet

- Research on formal proofs for arithmetic
  - John Harrison at Intel (HOL light)
  - Marc Daumas and Sylvie Boldo in the Arénaire project (Coq, PVS)
  - And many others...
- Proving Sterbenz Lemma (one operation) is worth a full paper.
- Here is the typical `crlibm` code for which I want the relative error :

```
yh2 = yh*yh ;
ts = yh2 * (s3 + yh2*(s5 + yh2*s7));
tc = yh2 * (c2 + yh2*(c4 + yh2*c6 ));
Mul12(&cahyh_h,&cahyh_l, cah, yh);
Add12(thi, tlo, sah,cahyh_h);
tlo = tc*sah+(ts*cahyh_h+(sal+(tlo+(cahyh_l+(cal*yh +
    cah*yl)))))) ;
Add12(*psh,*psl,  thi, tlo);
```

... and it changes all the time as we optimize it.

Gappa

Written by Guillaume Melquiond, Gappa is a tool that

- takes an input that closely matches your C file,
- forces you to express what this code is supposed to compute
- ... and some numerical property to prove (expressed in terms of intervals)
- and eventually outputs a proof of this property suitable for checking by Coq or HOL Light

Try it, it's free software

`gappa.gforge.inria.fr/`

Should I present interval arithmetic ?

Using a machine's finite precision, manipulate reals safely

- represent a real x in a machine as an interval $[x_l, x_r]$
  guaranteed to enclose it
  - x_l and x_r are finitely representable numbers (e.g. floating-point)
  - Example : π represented by $[3.14, 3.15]$
- Operation $\oplus$ on the reals $\rightarrow$ its interval counterpart

Guarantees based on the inclusion property

$I_x \oplus I_y$ must be an interval I_z such that

$$\forall x \in I_x, \forall y \in I_y, \quad x \oplus y \in I_z$$

- Example : interval addition using floating-point arithmetic

$$[a, b] + [c, d] \quad \text{is} \quad [\text{RoundDown}(a + c),\ \text{RoundUp}(b + d)]$$

- (multiplication, division similar but more complex)

A Gappa tutorial

```
# Convention: uncapitalized variables match the variables in the C code.

y = float<ieee_64,ne>(dummy); # y is a double

#---------------- Transcription of the C code --------------------------

s3 float<ieee_64,ne>= -1.6666666666666665741480812812369549646974e-01;
s5 float<ieee_64,ne>= 8.3333333333333321768510160154619370587170e-03;
s7 float<ieee_64,ne>= -1.9841269841269841252631711547849135968136e-04;

y2 float<ieee_64,ne>= y * y;
ts float<ieee_64,ne>= y2 * (s3 + y2*(s5 + y2*s7));
r  float<ieee_64,ne>= y + y*ts;

#-------- Mathematical definition of what we are approximating --------
#    (The same expression as in the code, but without rounding errors)

Y2 = Y * Y;
Ts = Y2 * (s3 + Y2*(s5 + Y2*s7));
R = Y + Y*Ts;

#----------------------- The theorem to prove --------------------------
{
  # Hypotheses (numerical values computed by Sollya)
     Y           in [1b-1000, 6.15e-3] #  Pi/512, rounded up
  /\ y - Y       in [-2.53e-23, 2.53e-23] # max abs. range reduction error
  /\ R-SinY  in [-3.55e-23, 3.55e-23] # approximation error (this defines SinY)
->
 r-SinY in ?                # A goal: absolute error
 /\
 (r-SinY)/SinY in ?         # Another goal: relative error
/\ SinY in ?
}
```

tutorial1.gappa

```
$ gappa < tutorial1.gappa
Results for Y in [-0.00615, 0.00615] and y - Y in [-2.53e-23, 2.53
r - SinY in [-2^(-60.9998), 2^(-60.9998)]
Warning: some enclosures were not satisfied.
Missing (r - SinY) / SinY
$
```

- A tight bound on the absolute error
- No bound for the relative error
  - of course ! I have to prove that SinY cannot come close to zero...
  - that's formal proof for you

We should now try gappa -Bcoq

How does Gappa work ?

- Gappa tries to associate an interval with each expression.
- Interval arithmetic is used to combine these intervals, until the goal is reached.
- Naively, it would lead to interval bloat. Here for instance
  - $\mathtt{r} \approx \mathtt{SinY} \in [-2^{-7}, 2^{-7}]$
  - so $\mathtt{r} - \mathtt{SinY} \in [-2^{-6}, 2^{-6}]$ using naive IA.
- Gappa uses rewriting of expressions

  As `r = float64ne(E);`

  try and use the rule

  `float64ne(E)) - SinY -> (float64ne(E) - E) + (E - SinY);`

  (hopefully now the sum of two smaller intervals)
- When Gappa is stuck, add user-defined rewriting rules
  - That's how you explain your floating-point tricks to the tool
- Internally, construction of a proof graph
  - Branches are cut when a shorter path or a better bound are found.
  - The final graph will be used to generate the formal proof.

Gappa's theorem library

- Predefined set of rewriting rules :
  - `float64ne(a)- b ->(float64ne(a)- a)+ (a - b);`
  - ...
- Support library of theorems (with their Coq proofs) :
  - Theorems giving the errors when rounding
    - `a in [...] ->(float64ne(a)-a)/a in [...]`
      Note how this takes care of dangerous cases (subnormal numbers, over/underflows...)
  - Classical theorems like Sterbenz Lemma
  - ...

To obtain a good relative error, Gappa will demand to prove that y may not be subnormal...

y + y*ts is a bit more accurate than y*(1+ts)

```
r1 float<ieee_64,ne>= y*(1+ts);
r2 float<ieee_64,ne>= y+y*ts;

yts float<ieee_64,ne>= y*ts;   # for lighter hints

#——————— Mathematical definition of what we are approximating ———————
#    (The same expression as in the code, but without rounding errors)
Y2 = y*y;
Ts = Y2 * (s3 + Y2*(s5 + Y2*s7));
Poly = y*(1+Ts);
#—————————————————— The theorem to prove ——————————————————————
{
  # Hypotheses (numerical values computed by Sollya)
y  in [1b-200, 6.15e-3] #   left: Kahan/Douglas algorithm. Right: Pi/512, rounded up
->
 r1-/Poly in ?          #  relative error
 /\
 r2-/Poly in ?          #  relative error
}

#————————————————Loads of rewriting hints needed for r2 ————————————————
y+yts -> y* ( (1+ts) + ts*((yts-y*ts) / (y*ts)))   {y*ts <> 0};

(r2-Poly)/Poly  -> ((r2 - (y+yts))/(y+yts) + 1)   * (   ((y+yts)/y) / (1+Ts)) -1 {1+Ts
     <>0};

(y+yts)/y ->
          # (y+y*ts-y*ts+yts) /y;
          # 1+ts + (yts-y*ts)/y;
          1+ts + ts*( (yts-y*ts)/(y*ts)  )   {y*ts <> 0};

((y+yts)/y) / (1+Ts) -> (1+ts)/(1+Ts) + ts*( (yts-y*ts)/(y*ts)  )/(1+Ts)  {1+Ts<>0};

(1+ts)/(1+Ts) -> 1 + (Ts*((ts-Ts)/Ts))/(1+Ts)  {1+Ts<>0};
```

tutorial2.gappa

```
$ gappa < tutorial2.gappa

Results for y in [7.88861e-31, 0.00615]:
(r1 - Poly) / Poly in [-2^(-52.415), 2^(-52.415)]
(r2 - Poly) / Poly in [-2^(-52.9777), 2^(-52.9339)]

$
```

Conclusion on Gappa

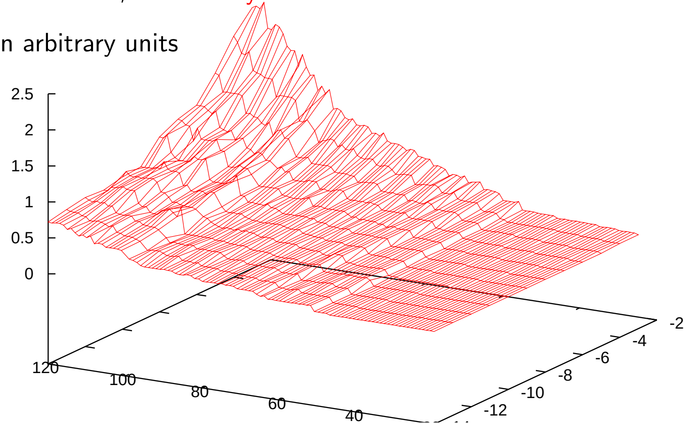
- I probably failed to convey this, but...
  Gappa is surprisingly easy to use.
  (if you didn't understand my Gappa proof, you just don't understand my C code)
  - if you don't know where it is stuck, ask it (by adding goals)
  - then add rewriting rules to help it
- It is built upon very solid theoretical fundations
- **MetaLibm is a generator of code + Gappa proof**
  - The same RR work for large classes of generated codes.
- Also support for arbitrary-precision fixed-point.

Two metalibm prototypes

Christoph Lauter's metalibm

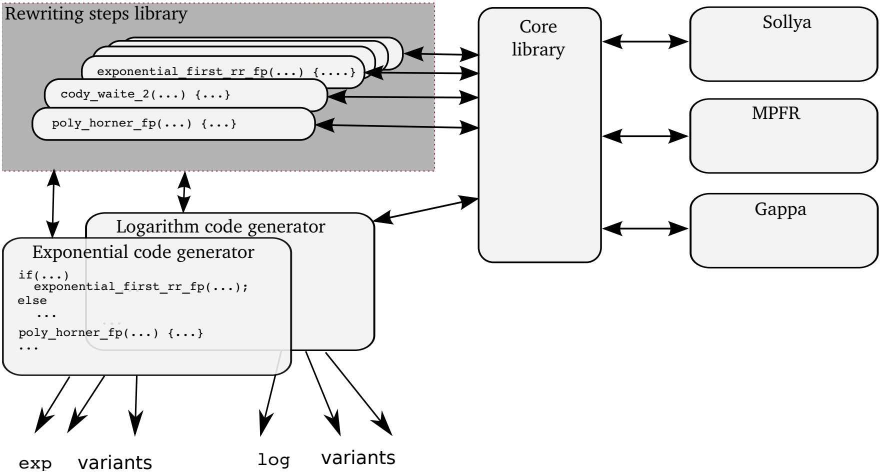
- Example : $\log(1 + x)$
- Two parameters
  - k from 1 to 13, defines table size
  - target accuracy, between 20 and 120 bits
- 1203 implementations, all formally checked

z axis : timings in arbitrary units



MetalibmC11 : an ad-hoc approach

How to develop/retarget functions in lower time ?



Metalibm framework

All this is work in progress

- A `Processor` class and its subclasses
  - encapsulates processor-specific code generation and tricks
  - still tinkering a lot there
- A `Format` class and its subclasses
- A `Polynomial` class
- A `CFunction` class for libm functions
  - automatically generates test programs

Metaexp in one slide

- inputs :

  `fp_format, processor, verbose=True,`
  `manage_subnormals=True, eval_Estrin=False,`
- Already 8 useful implementations
  (float/double, subnormals or not, Estrin or Horner)
- Trivial to add a precision input
- A case study for structuration as a metaskeleton
- No Gappa generation yet
- Current code doesn't autovectorize with GCC
- experimental generator of fixed-point code

Perfs for exp

- My laptop : Intel(R) Core(TM)2 Duo CPU U9600 @ 1.60GHz
- My desktop : Intel(R) Xeon(R) CPU E5-1620 0 @ 3.60GHz

Both running XUbuntu 12.10 with gcc 4.7.2

	Core2 U9600	Xeon E5-1620
stock expf	193	45
expf Horner	87	24
expf Estrin	77	27
stock exp	108	60
exp Horner	130	28
exp Estrin	89	36

Last-bit accuracy verified by exhaustive test for the expf's

Disclaimers :

- timings using `__rdtsc()`, usual caveats apply.
- inlining switched on for our code, not for the stock function.

Metalog in one slide

- experiment with optimized for latency / optimized for throughput
  - using autovectorisation with gcc 4.7
  - works for single but not for double
    (no `%ymmi` in the generated assembly ? ! ?)
    Either AVX doesn't replicate all SSE2 functions, or GCC is not ready
- I'm not sure I understand how a degree-20 Horner polynomial is evaluated in 37 cycles
- Estrin evaluation would be useful here
  - but current implementation not modular enough
  - short-term TODO

Perfs for log (see metalibm/tests/perftests.cc)

	Core2 U9600	Xeon E5-1620
stock logf	99	36
logf_horner(opt. for latency)	88	30
the same, autovectorized for SSE2	35	30
logf_horner_v (opt. for throughput)	107	33
the same, autovectorized for SSE2	11	11
stock log	132	86
log_horner (opt. for latency)	171	37

Last-bit accuracy verified by exhaustive test for the logf's

A glance at generated code

```
/* Exceptional case filtering, vectorizable */
minfty.ui = 0xff800000; /* minus infinity */
nan.ui = 0x7fc00000; /* nan */
ret_minfty = ((xx.ui & 0x7fffffff) == 0) ? minfty.f : 0.0f; /* x == +/-0 ?
ret_nan = (xx.ui > 0x80000000) ? nan.f : 0.0f;  /* x<0 ?*/
x_is_inf_or_nan = ((xx.ui &  0x7fffffff) >= 0x7f800000) ? xx.f : 0.0f;  /*
exn = ret_minfty + ret_nan + x_is_inf_or_nan;   /* 0.0 if normal or subnor
/* Now remains to add exn somewhere where it will propagate to the result
x_subnormal = (xx.ui < 0x00800000) && (xx.ui > 0);
subnormal_scale = x_subnormal ? 0x1.p48f : 1.0f; /* scale mantissa*/
e_x = x_subnormal ? -127-48 : -127; /* ... and initialize exponent*/
xx.f *= subnormal_scale;
/* Now decompose x into fraction and exponent */
e_x += ((xx.i) >> 23) & ((1<<8)-1); /* extract exponent*/
fraction.i = (xx.i & 0x007fffff); /* extract fraction bits*/
adjust = (fraction.i>>22); /* first non-implicit bit of the fraction, tell
fraction.i = fraction.i |0x3f800000; /* add the exponent of one */
fraction.i -= adjust << 23; /* if m>1.5, divide fraction by 2 (exact opera
e_x += adjust;     /* and update exponent so we still have x = 2^e_x * fra
```

```
/* Now back to floating-point */
y = fraction.f - 1.0f; /* Sterbenz-exact; may cancel but we don't care  */
y += exn; /* exn is either 0.0, or an inf or NaN that will propagate to th
/* Now y in [-0.25, 0.5], and we must evaluate log(1+y) */
/* Horner evaluation */
y2 = y*y;
p9 = c9;
p8 = c8 + y*p9;
p7 = c7 + y*p8;
p6 = c6 + y*p7;
p5 = c5 + y*p6;
p4 = c4 + y*p5;
p3 = c3 + y*p4;
p2 = c2 + y*p3;
p = y + y2*p2;
r = e_x*log_2 + p;
return r;
```

Horner autovectorized to SSE2

thanks to `gcc -O3 -msse2 -finline-limit=1000 -S`

Without	With
`mulss %xmm2, %xmm1`	`mulps %xmm1, %xmm2`
`subss %xmm10, %xmm0`	`subps .LC50(%rip), %xmm0`
`mulss %xmm2, %xmm0`	`mulps %xmm1, %xmm0`
`addss %xmm9, %xmm0`	`addps .LC51(%rip), %xmm0`
`mulss %xmm2, %xmm0`	`mulps %xmm1, %xmm0`
`subss %xmm8, %xmm0`	`subps .LC52(%rip), %xmm0`
`mulss %xmm2, %xmm0`	`mulps %xmm1, %xmm0`
`addss %xmm7, %xmm0`	`addps .LC53(%rip), %xmm0`
`mulss %xmm2, %xmm0`	`mulps %xmm1, %xmm0`
`subss %xmm6, %xmm0`	`subps .LC54(%rip), %xmm0`
`mulss %xmm2, %xmm0`	`mulps %xmm1, %xmm0`
`addss %xmm5, %xmm0`	`addps .LC55(%rip), %xmm0`
`mulss %xmm2, %xmm0`	`mulps %xmm1, %xmm0`
`subss %xmm4, %xmm0`	`subps .LC56(%rip), %xmm0`

Room for improvement by interleaving two iterations ?

This is evaluated in 37 cycles ?

```
mulsd %xmm2, %xmm1
movapd %xmm2, %xmm3
mulsd .LC19(%rip), %xmm0
mulsd %xmm2, %xmm3
addsd .LC21(%rip), %xmm1
mulsd %xmm2, %xmm1
subsd .LC22(%rip), %xmm1
mulsd %xmm2, %xmm1
addsd .LC23(%rip), %xmm1
mulsd %xmm2, %xmm1
subsd .LC24(%rip), %xmm1
mulsd %xmm2, %xmm1
addsd .LC25(%rip), %xmm1
mulsd %xmm2, %xmm1
subsd .LC26(%rip), %xmm1
mulsd %xmm2, %xmm1
addsd .LC27(%rip), %xmm1
mulsd %xmm2, %xmm1
subsd .LC28(%rip), %xmm1
mulsd %xmm2, %xmm1
addsd .LC29(%rip), %xmm1
```

```
mulsd %xmm2, %xmm1
subsd .LC30(%rip), %xmm1
mulsd %xmm2, %xmm1
addsd .LC31(%rip), %xmm1
mulsd %xmm2, %xmm1
subsd .LC32(%rip), %xmm1
mulsd %xmm2, %xmm1
addsd .LC33(%rip), %xmm1
mulsd %xmm2, %xmm1
subsd .LC34(%rip), %xmm1
mulsd %xmm2, %xmm1
addsd .LC35(%rip), %xmm1
mulsd %xmm2, %xmm1
subsd .LC36(%rip), %xmm1
mulsd %xmm2, %xmm1
addsd .LC37(%rip), %xmm1
mulsd %xmm2, %xmm1
subsd .LC38(%rip), %xmm1
mulsd %xmm1, %xmm3
addsd %xmm2, %xmm3
addsd %xmm3, %xmm0
ret
```

Metatrigpi in one slide

- $\sin(\pi x)$ and $\cos(\pi x)$ recommended by IEEE 754-2008
  - No costly range reduction
  - Correct rounding proven feasible
- `sincospif(float x, float *s, float *c)`
  computes both in one function
- `sincospio2f(float x, float *s, float *c)`
  computes $\sin(\frac{\pi}{2}x)$ and $\cos(\frac{\pi}{2}x)$ even faster

Developed in one day.

Conclusion

Main messages

- If you're computing accurately enough, you're probably computing too accurately.

- Are you able to express what your code is supposed to compute ? If yes,
  - we can help you sort out the gory floating-point issues
  - we can provide functions computing just right for you

The MetaLibm open-ended vision

- We needed to automate the development of code+proof for the elementary functions
- Now that this is (almost) done, we may *open up the set of functions/precisions/performance constraints*

An ANR funding proposal under review

- metalibm/OpenEnded
  - genericity in input
- metalibm/C11
  - focus on performance (match hand-coded libraries)
  - genericity in target processor
  - hand-code what we are unable (yet) to automate : range reductions, floating-point trickery, ...
- FPGAs, DSP filters for good measure

Open-ended input

- As an arbitrary expression + interval + range
- As a differential equation (see Dynamic Dictionary of Mathematical Functions)

  `http://ddmf.msr-inria.inria.fr/`

- ...

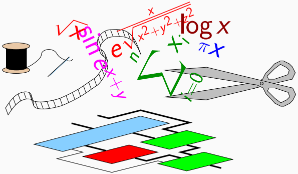
Beyond the horizon

Functions of several variables

My other research project

Computing just right for FPGAs

... but I was given another advertising slot for this.



http://flopoco.gforge.inria.fr/

Thank you for your attention