The Metalibm project :

assisting the production of high-performance, proven elementary function code

Florent de Dinechin
CITI/Socrates project

Outline

Introduction : performance versus accuracy

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, regaining 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.

The rest of this talk is about this... elementary functions being used as an easy example.

Elementary function schizophrenia

A producer/consumer mismatch

On the producer side

- Specification : a mathematical function function (e.g. log), and a floating-point format (binary32 or binary64)
- Obviously, one should return the correct rounding of $\log(x)$ (IEEE-754-2008)
- Anything less means wasting bits, probably cheating for speed, and introducing system-specific idiosyncrasies.

On the consumer side

Code should

1. use all the available space but no more,
2. be as fast as possible, and
3. be "accurate enough"

Correctly rounded probably vast overkill for "accurate enough".

Not only a quantitative matter



The overkill is not only about the accuracy (the number of correct bits).
Just two examples :

- In binary floating-point, $\sin(x)$ is very difficult to evaluate, just because 2 and π are irrational.
  - IEEE754-2008 introduced the new function $\mathrm{sinPi}(x) = \sin(\pi x)$.
  - For most applications, it is just as good, only faster.
- Fixed-point versus floating-point (see next slide)

Fixed-point versus floating-point

What is wrong with the following computation in floating-point ?

$$s = \sum_{i=1}^{n} x_i \qquad y = e^s$$

A fix-point number is a binary integer, scaled by some constant 2^m

- cheaper to compute on (addition defines the PC's cycle time)
- simpler to analyze (addition is exact until it overflows)
- most sensors provide fixed-point data
- many computations are inherently fixed-point
  (i.e. do not take very large values nor need arbitrary resolution around 0).

Exponential and logarithm functions

... map a fixed-point range to a floating-point-range

I'm schizophrenic, but I'm under treatment

- On one side, I'm advocating perfectly accurate (correctly rounded) functions.
- On the other side, I'm advocating "computing just right" using less accurate functions

This MetaLibm in the title

- should provide all these variants
- ... if you provide it with explicit accuracy specifications
- (yes I'm trying to twist your arm into computing just right)

The art of implementing elementary functions

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

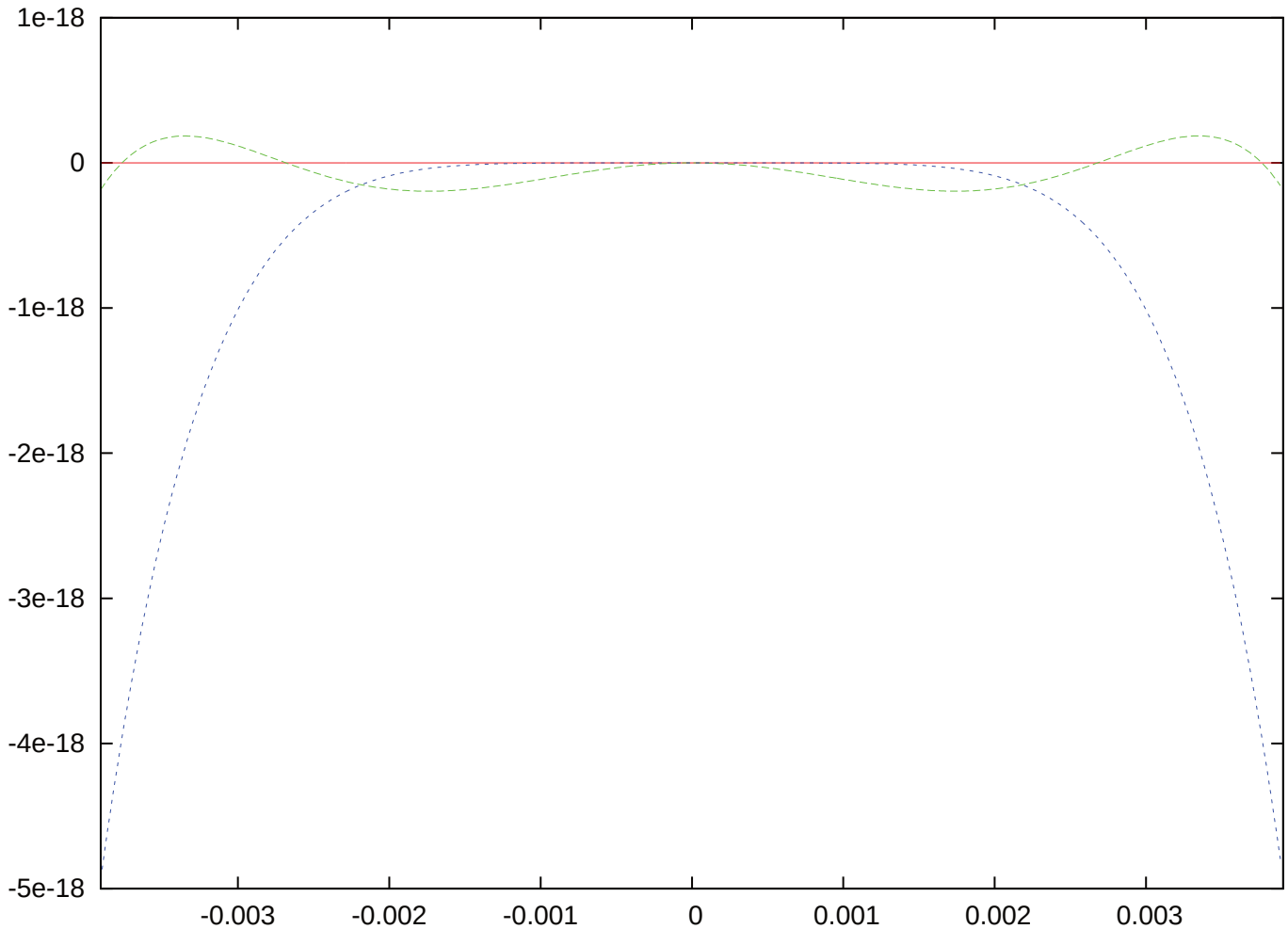
Simplistic example : an exponential

- identity : $e^{a+b} = e^a \times e^b$
- split $x = a + b$
  - a : k leading bits of x
  - b : lower bits of x $\qquad b << 1$
- 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 degree-5 Taylor and Remez approx. to 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 :

- $||p - f||_\infty$ above is an upper bound of the approximation error
- error of the polynomial is with reference to the function (easy)
- error of the FP addition is with reference of the real sum (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.
- The compiler will not help you there :

  No compiler has no access to the reference ! It is not in the code.

Let us take a simple example

This is part of the code of `sin`,
after y has been reduced to $[-\pi/256, \pi/256]$:

```
1  s3 = -0.16666666666666657414808128123695496469736099 24;
2  s5 =  8.33333332628927933583007359175098827108740 81e-3;
3  s7 = -1.98400103113668426196153360407947729981970042e-4;
4
5  y2 = y * y;
6  ts = y2 * (s3 + y2*(s5 + y2*s7));
7  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

```
1  s3 = -0.16666666666666657414808128123695496469736099 24;
2  s5 =  8.33333332628927933583007359175098827108740 81e-3;
3  s7 = -1.98400103113668426196153360407947729981970042e-4;
4
5  y2 = y * y;
6  ts = y2 * (s3 + y2*(s5 + y2*s7));
7  r  = y + y*ts
```

- This polynomial is an approximation to $sin(y)$
- Oops, I wrote its coefficients in decimal !
- In general, 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 ?

Important information NOT in the code

The context

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

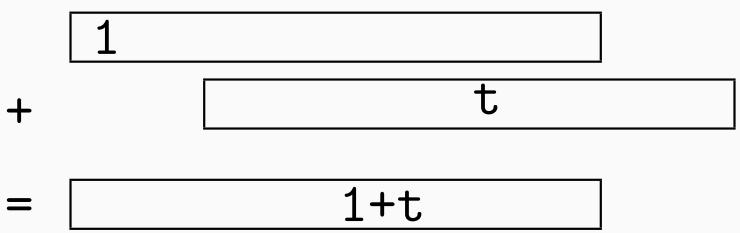
What it is supposed to compute

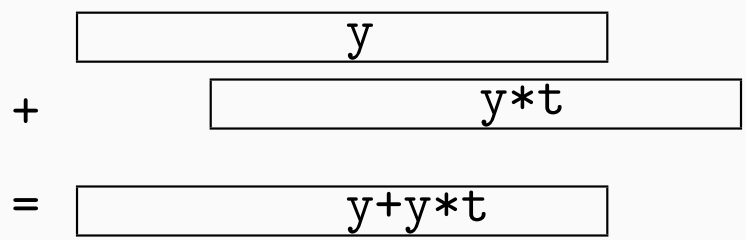
a sine accurate to 2^{-60}

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 :

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

$$(\text{rounding error} + \text{polynomial approximation error})$$

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

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

Difficulties

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

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

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

Correctly rounded functions computing just right

Know what accuracy you need ?

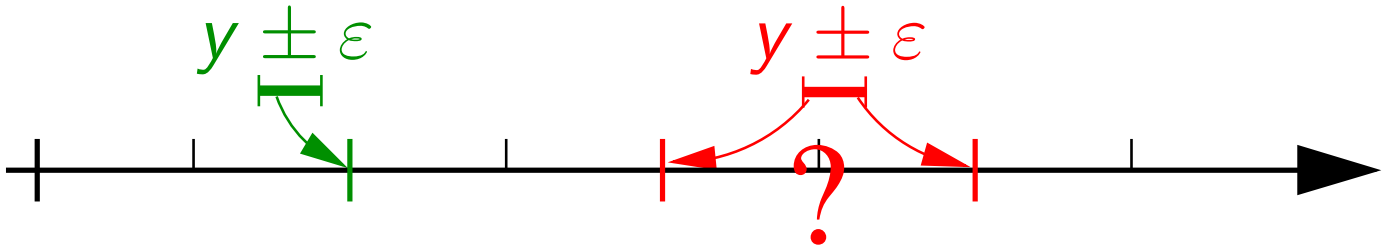
Correctly rounded elementary functions

- 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

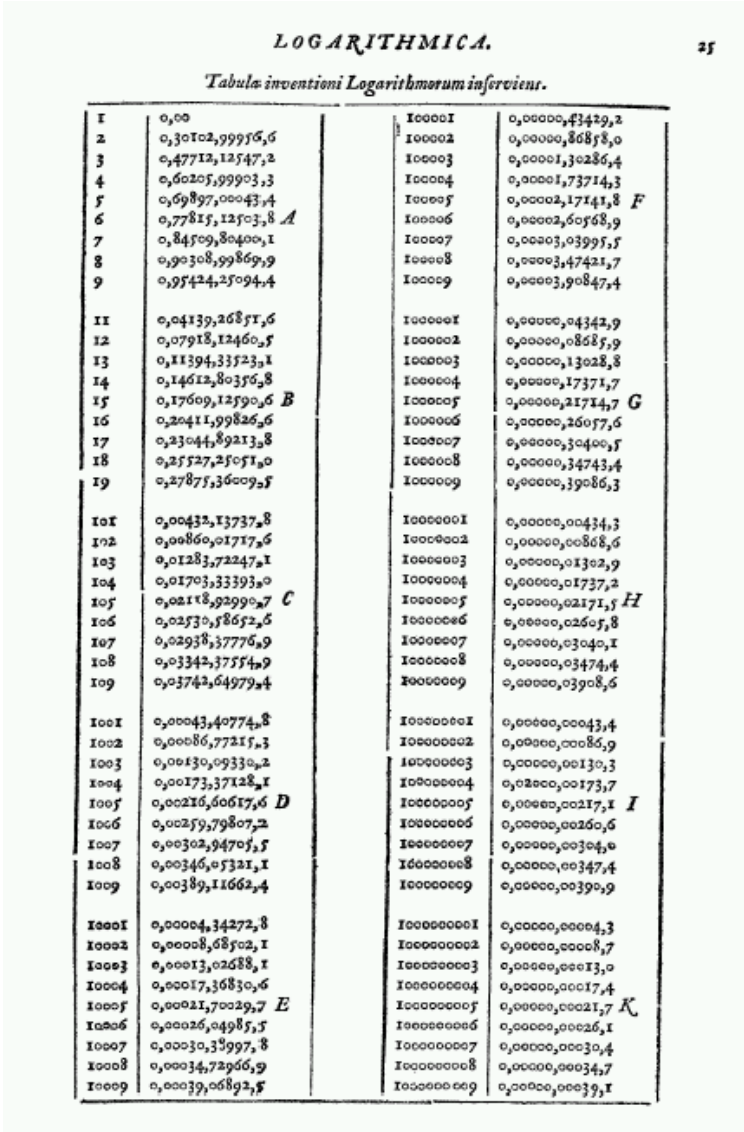
The Table Maker's Dilemma

- 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 inservientes.

1	0,00	10001	0,00000,43429,2
2	0,30102,99956,6	10002	0,00000,86858,0
3	0,47712,12547,2	10003	0,00001,30286,4
4	0,60205,99903,3	10004	0,00001,73714,3
5	0,69897,00043,4	10005	0,00002,17141,8 F
6	0,77815,12503,8 A	10006	0,00002,60568,9
7	0,84509,80400,1	10007	0,00003,03995,5
8	0,90308,99869,9	10008	0,00003,47421,7
9	0,95424,25094,4	10009	0,00003,90847,4
11	0,04139,26851,6	100001	0,00000,04342,9
12	0,07918,12460,5	100002	0,00000,08685,9
13	0,11394,33523,1	100003	0,00000,13028,8
14	0,14612,80356,8	100004	0,00000,17371,7
15	0,17609,12590,6 B	100005	0,00000,21714,7 G
16	0,20411,99826,6	100006	0,00000,26057,6
17	0,23044,89213,8	100007	0,00000,30400,5
18	0,25527,25051,0	100008	0,00000,34743,4
19	0,27875,36009,5	100009	0,00000,39086,3
101	0,00432,13737,8	1000001	0,00000,00434,3
102	0,00860,01717,6	1000002	0,00000,00868,6
103	0,01283,72247,1	1000003	0,00000,01302,9
104	0,01703,33393,0	1000004	0,00000,01737,2
105	0,02118,92990,7 C	1000005	0,00000,02171,5 H
106	0,02530,58652,6	1000006	0,00000,02605,8
107	0,02938,37776,9	1000007	0,00000,03040,1
108	0,03342,37554,9	1000008	0,00000,03474,4
109	0,03742,64979,4	1000009	0,00000,03908,6
1001	0,00043,40774,8	10000001	0,00000,00043,4
1002	0,00086,77215,3	10000002	0,00000,00086,9
1003	0,00130,09330,2	10000003	0,00000,00130,3
1004	0,00173,37128,1	10000004	0,00000,00173,7
1005	0,00216,60617,6 D	10000005	0,00000,00217,1 I
1006	0,00259,79807,2	10000006	0,00000,00260,6
1007	0,00302,94707,5	10000007	0,00000,00304,0
1008	0,00346,05321,1	10000008	0,00000,00347,4
1009	0,00389,11662,4	10000009	0,00000,00390,9
10001	0,00004,34272,8	100000001	0,00000,00004,3
10002	0,00008,68502,1	100000002	0,00000,00008,7
10003	0,00013,02688,1	100000003	0,00000,00013,0
10004	0,00017,36830,6	100000004	0,00000,00017,4
10005	0,00021,70029,7 E	100000005	0,00000,00021,7 K
10006	0,00026,04985,5	100000006	0,00000,00026,1
10007	0,00030,39997,8	100000007	0,00000,00030,4
10008	0,00034,72966,9	100000008	0,00000,00034,7
10009	0,00039,06892,5	100000009	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

$$y = x, xxxxxxxxxxx17 \pm 10^{-14}$$

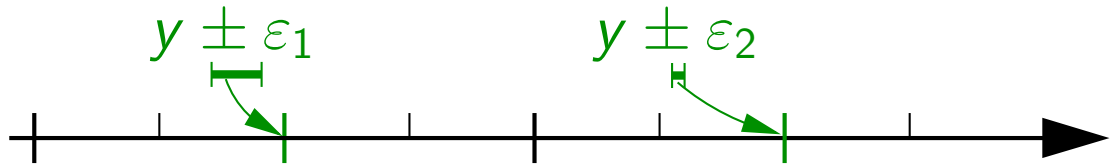
$$y = x, xxxxxxxxxxx83 \pm 10^{-14}$$

- Dilemma when

$$y = x, xxxxxxxxxxx50 \pm 10^{-14}$$

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 $\overline{\varepsilon}_1$

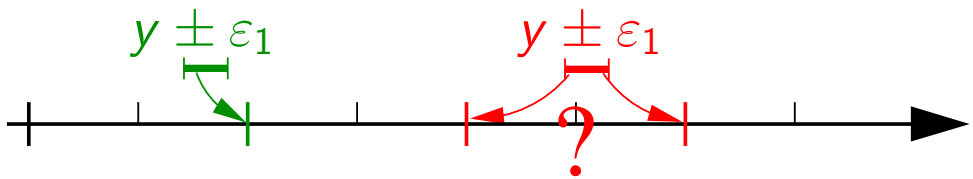
  sometimes not accurate enough
- (rarely) second step slower but always accurate enough

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

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 our group

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)

With MetaLibm prototype

- development time for 16 variants of sinpi, cospi, sincospi : 1 day

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
  - Unfortunately, no hope of a beautiful C++ wrapper that would always take the proper decision as of where to round
- 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

```
1 fdedinec@krupnik: sollya
2 > 1+1;
3 2
4 > 1/3;
5 Warning: rounding has happened. The value displayed is a
      faithful rounding of the true result.
6 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

```
1 fdedinec@krupnik: sollya
2 > e=exp(x) - (1+x+x^2/2+x^3/6);
3 > e(0.125);
4 Warning: rounding has happened. The value displayed is a
      faithful rounding of the true result.
5 1.0432233492983495673894478460539232169798411848292 6e-5
6 >
```

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;
```

Gappa : motivation

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

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

with

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

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 previously, only the interval changes :

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

We also compute $\overline{\varepsilon}_{\texttt{hornerts}}$, 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 $\overline{\varepsilon}_{\texttt{yh2}}$ 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 :

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

Let us define for now

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

Then we have

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

Using (1) and (5) we get :

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

To lighten notations, let us define

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

We get

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

Using the following bound :

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

we may compute the value of $\overline{\varepsilon}_{\text{sinkzero}}$ 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, because the code doesn't include

- what it is supposed to compute
- the knowledge used to build 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

`gappa.gforge.inria.fr/`

CGPE

Code generation for polynomial evaluation

- explores different parallelizations of a polynomial on a VLIW processor
- generates code and Gappa proof of the evaluation error

Used to generate the code for the division and square root of FLIP, a Floating-Point Library for Integer Processors
(collaboration with ST Microelectronics)
Should help for vector code as well.

`cgpe.gforge.inria.fr/`

Three metalibm prototypes

Challenges of elementary function coding

- Multiple targets
  - numerous architectures (Intel's x86/Xeon-Phi, Kalray's K1, ARM)
  - numerous constraints (throughput, latency, precision, memory consumption)
  - numerous programming styles (e.g. vector versus scalar)
- More function out there than the ones listed in C11 !

libm paradigm poorly addresses this complexity.

Metalibm : generate function code on demand for a given context

The MetaLibm open-ended vision

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

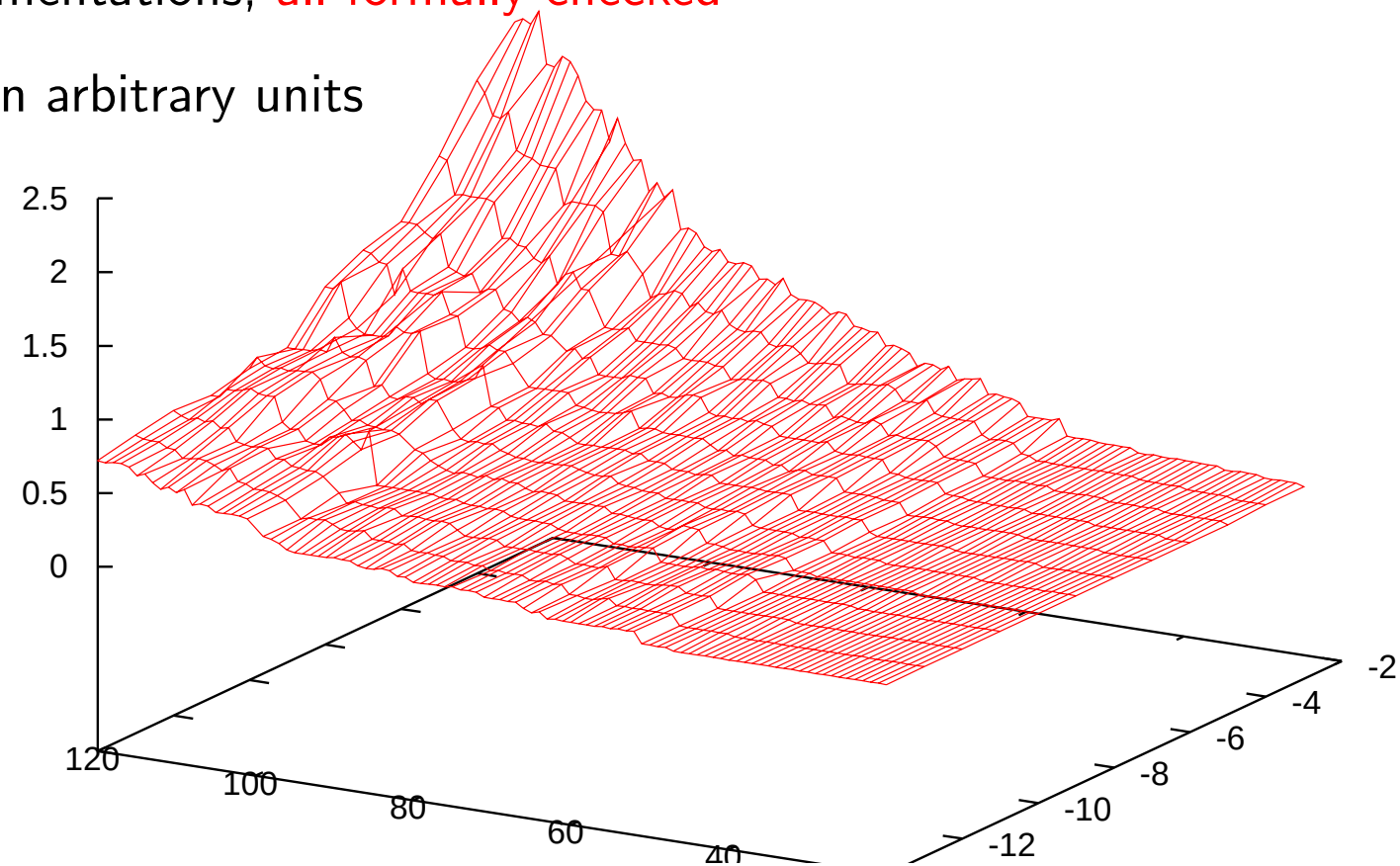
An ANR-funded project

- 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

1/ 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



Current experiments with this prototype

Connect it to the Dynamic Dictionary of Mathematical Functions

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

... to obtain code for a function defined by a differential equation + limit conditions

2/ MetalibmC11 : an ad-hoc approach

Philosophy : take good working C code, wrap it in `printfs`, then generalize it.

- low abstraction, but succes guaranteed
- ... with perf identical to hand-written code

All scripted in Python.

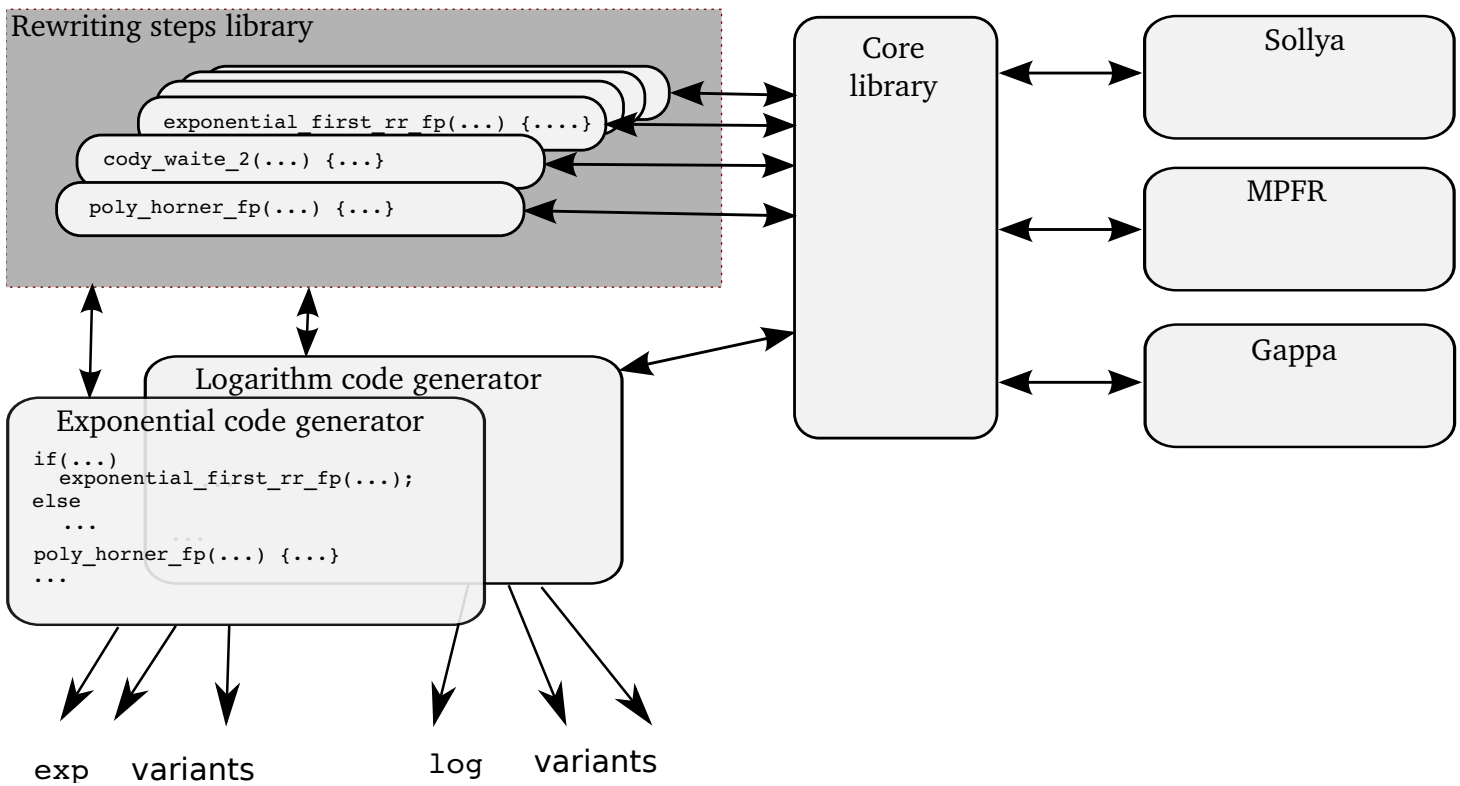
Helper framework

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 that manages both approximation and evaluation
- A `CFunction` class for libm functions
  - automatically generates test programs

Metaexp in one slide

- Already 8 useful implementations, each for arbitrary accuracy p (float/double, subnormals or not, Estrin or Horner)
- A case study for structuration as a metaskeleton



- No Gappa generation yet

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
- No Gappa generation yet

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;
```

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, with tests and all.

Weakness of previous attempts

- Lauter's too generic, will miss many function-specific tricks we need for performances.
- Dinechin's not abstract enough, misses "what's being computed". As a consequence, automatic proof generation mostly hopeless.

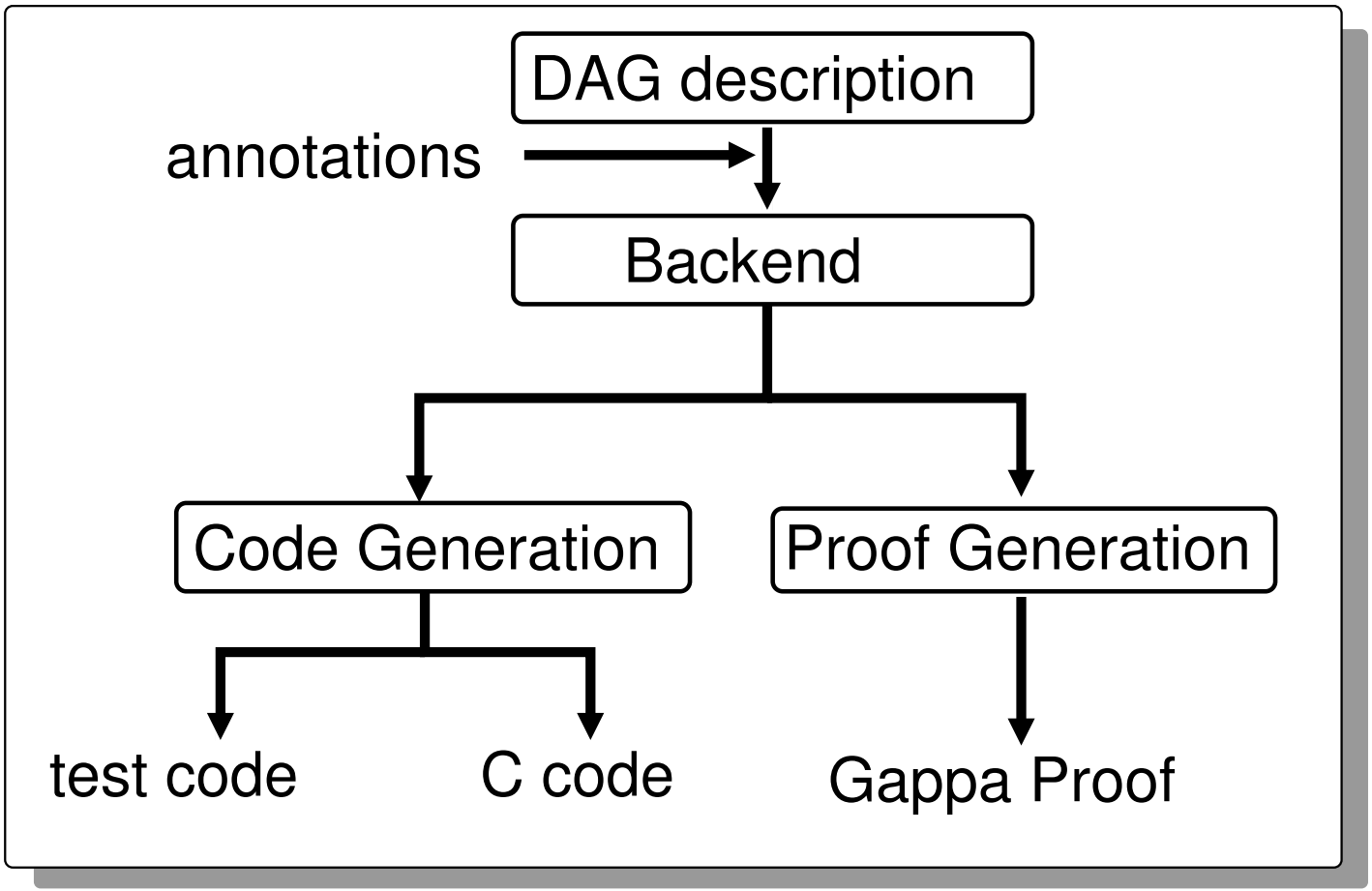
3/ Le troisième sera le bon

New Metalibm developed by N. Brunie

- DAG representation of the function, and of the implementation
- abstract target description

disconnect description, optimization and code+proof generation

Metalibm process



DAG description

- describe function implementation with operation nodes
  - **VARIABLE, Constant**
  - **TableLoad, TableLoad_HL**
  - **Addition, Multiplication, Modulo ...**
  - **Test, SpecOp, ExponentInsertion ...**
  - **ConditionBlock**
- DAG can be built two ways :
  - by composing constructors
  - by overloaded python expressions

example of DAG description

```
vx = VARIABLE("x", precision = fpformat)

# reduced argument
red_x = NearestInt(vx / log(2), precision = int32, tag = "red_x")

# HIGH qnd LOW part log(2) generation
log2_hi = round(log(2), fpformat.sollya_name - 10, RN)
log2_lo = round(log(2) - log2_hi, fpformat.sollya_name, RN)
r = (vx - (red_x * log2_hi)) - red_x * log2_lo
r.set_attributes(tag = "r", exact = True)

red_int = Interval(-log(2)/2, log(2)/2)

poly = Polynomial.generate_fpminimax(exp(x), 5, red_int, [ML_Binary64]*6,
poly_scheme = PolySchemeGenerator.generate_horner(poly, r)

result = Return(ExponentInsertion(red_x) * poly_scheme)

backend_scheme = Backend(processor).backend_float(result, ML_Binary64)
source_code = CodeGenerator(processor).generate_expr(backend_scheme)
```

The annotation system

- to facilitate generated code reading
- to optimize DAG
- to enforce numerical constraints

Some examples of annotations :

- **tag, debug**
- **precision**
- **likely**
- **exact, interval**

Backend processing

The main operations performed by the backend are :

- instanciating every undetermined format
- introducing necessary conversions
- optimizations at the level of abstract operations

Other tasks include :

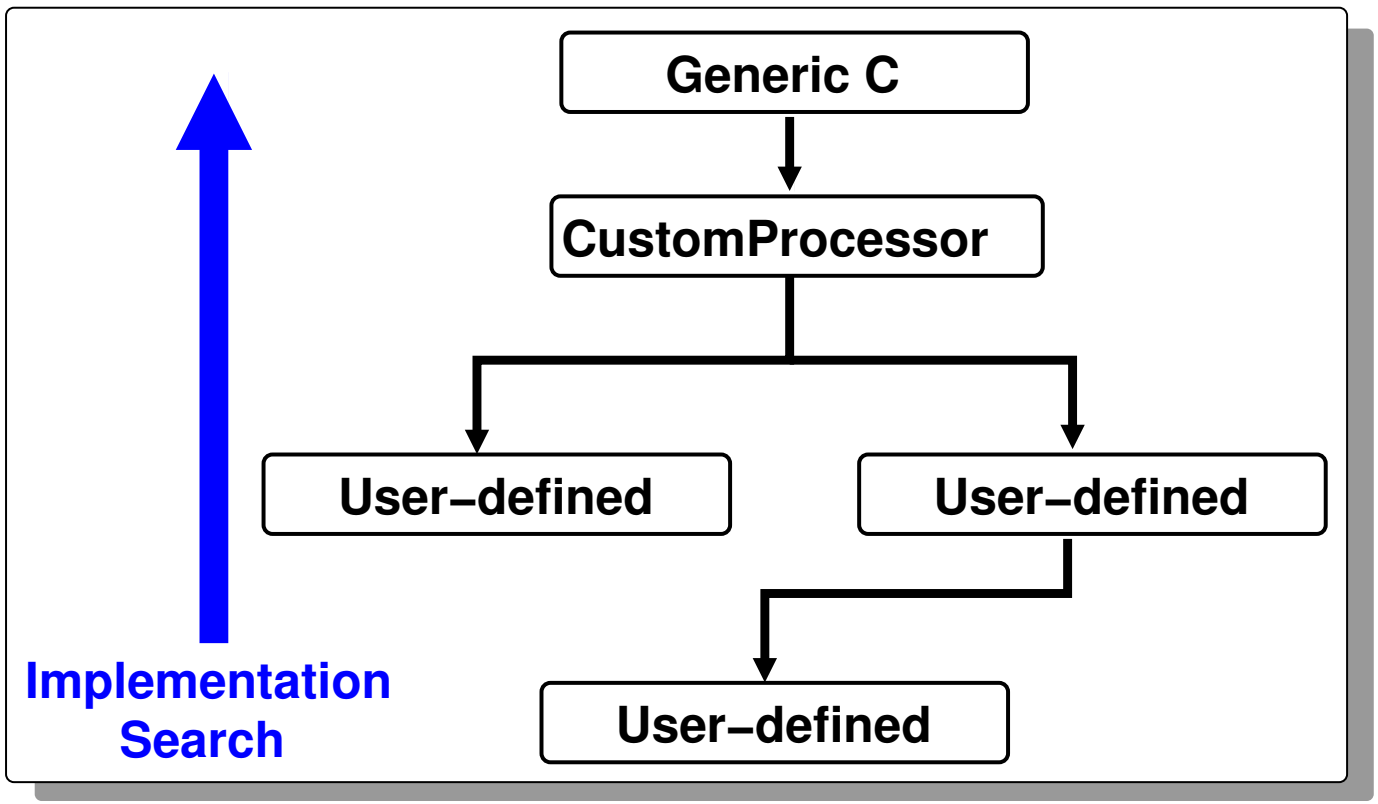
- dynamic support library expansion
- pre-vectorization processing

memoization is used for resource sharing

Code Generation

- generated from fully type-instanciated description
- constants, tables and core code generation
- several targets are available
- support for processor-specific code generation
- Gappa proof generation

Processor-specific code generation



Challenges of vectorization

- introduction or vector ISA : x86's MMX, SSE (1, 2, 3, 4, 4.1, 4.2), AVX (1 and 2), Xeon-Phi, ARM's NEON
- heavily reliant on branch uniformity
- compiler support can be unreliable
- generate branchless code
  - blending returns
  - extract most common path
  - generating callouts
- use intrinsics to force use of vector instructions

All this still preliminary

Challenges :

- Genericity versus ad-hoc efficiency (it is easy to write a generator that works for only one function)
- Draw the line between the compiler and Metalibm. Do not reimplement a full compiler !
- Open the code ! (some at Intel, some at Kalray, all locked for the moment)

Conclusion

Main messages

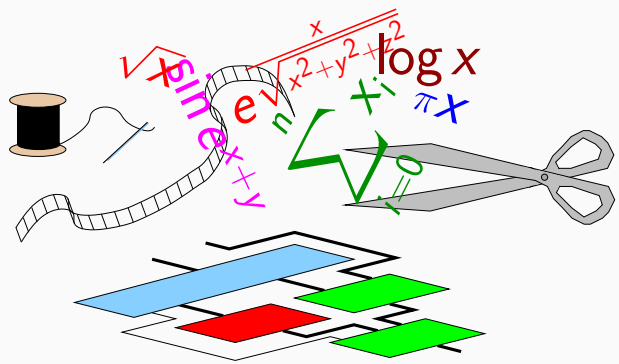
- If you're computing accurately enough, you're probably computing too accurately.

- 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

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

Introduction : performance versus accuracy

Elementary function schizophrenia

The art of implementing elementary functions

Correctly rounded functions computing just right

Open-source tools for FP coders

Three metalibm prototypes

Conclusion