Software techniques for perfect elementary functions in floating-point interval arithmetic

Florent de Dinechin, projet Arénaire
Sergey Maidanov, Intel

Real Numbers and Computers 2006, Nancy

Outline

Interval arithmetic, directed rounding and elementary functions

Perfect interval functions

Implementation

Conclusion and future work

Introduction: Interval arithmetic, directed rounding and elementary functions

Directed rounding modes

In this talk, we target double-precision computations.

- The IEEE-754 standard defines, for a real number x and a floating-point format $\mathbb{F}$,
  - the correct rounding up of x,
  - the correct rounding down of x
  - the correct rounding to zero of x
- Correct directed rounding supported by most processors at a cost for $+, -, \times, \div, \sqrt{\ }$
- Intended mostly for interval arithmetic (IA)
  - IA represents a real number x by an interval containing x.
  - Interval endpoints are machine-representable numbers, here in double-precision.
- Some interval arithmetic packages don't use directed rounding at all

Interval arithmetic does not require correct rounding

Important properties of interval operations are, *in this order:*

- Safety
  - For all real values in the input interval(s), the real result belongs to the output interval (containment property).
- Sharpness (of the output interval)
  - Finite number representation $\Longrightarrow$ output interval usually larger than the image of the input interval
  - Define the tight result as the smallest interval ensuring the containment property, considering the number format
  - Related to correct rounding in directed rounding modes
- Performance

Perfect interval elementary function:

- Safe
- Tight
- Efficient

This article

- Interval people have long been writing interval elementary functions which are
  - safe (usually),
  - sharp (but not tight),
  - and sometimes efficient
- In the `CRLibm` project, we have been writing correctly rounded elementary functions in directed rounding modes, wondering
  - *Why? Who cares*
  - *What should the interface be?*
- We should be writing perfect interval elementary functions instead.

Some IA packages

- Sun Microsystems' SunStudio
  - Very efficient integrated language/compiler approach (Fortran95 and C++)
  - Sharp (almost tight) elementary functions
    - paper by Priest in Arith 13
    - tight for more than 99% of input intervals
  - No published proof (and not open-source).
  - Not portable.
- `fi_lib` then C-XST then `filib++`
  - Portable, does not use hardware directed rounding
  - Neither sharp nor efficient, even for basic operations
  - Elementary functions written from scratch and carefully proven (in German).

Some IA packages (2)

- INTLAB (a Matlab package)
  - A trick by Rump: use the (underspecified) `libm` to build safe interval functions
  - But inefficient, non-sharp, and still requires development and proof work
- Boost
  - Empty shell : C++ templates and policies
  - Generic: FP intervals, but also multiple-precision if needed
  - Portable
  - Sharpness and efficiency depending on how you fill the shell
- MPFI
  - Safe and tight (on top of MPFR)
  - Not optimised for double-precision
    - but then you get multiple-precision IA...

Perfect interval functions

Interval arithmetic, directed rounding and elementary functions

Perfect interval functions

Implementation

Conclusion and future work

Implementing an interval function

Evaluate $f([x_-, x_+])$

- If the function f is monotonic increasing
  - Round down $f(x_-)$
  - Round up $f(x_+)$
  - That's all.
- If the function is monotonic decreasing
  - Left as an exercise for the reader
- If the function is not monotonic on the interval
  - Elementary functions are well-behaved:
    Such cases are handled in the argument reduction.
  - It may actually save work
    - $\sin([-42, 17])$

In theory, nothing difficult, really.

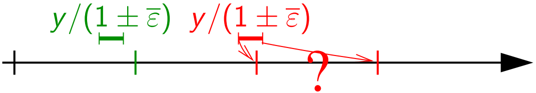
Rounding up a function

Evaluate $y \approx f(x)$ to some precision $\overline{\varepsilon}$

Is this information enough to ensure that rounding y is equivalent to rounding $f(x)$?

- Sometimes yes,
- Sometimes no.

Rounding up a function

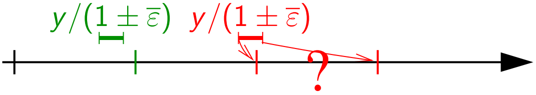


Two options:

- Round up $y/(1 - \overline{\varepsilon})$ (Krämer, Priest)
  - safe,
  - not tight (in the red case),
  - fast.
- Onion peeling strategy (`CRLibm`, idea by Ziv)
  1. *Rounding test* on y knowing $\overline{\varepsilon}$: Are we in the green case?
  2. If yes, return y rounded up.
  3. If no, compute y_2 with a smaller $\overline{\varepsilon}_2$, and go back to 1.

  - safe if termination is proven (elementary functions),
  - tight,
  - slower, but just a little
    - small cost of rounding test
    - restarting computation is expensive, but rare.

The Table Maker's Dilemma



- There are finitely many FP input numbers x
- Among them, finitely many are such that $f(x)$ is an FP number
  - 1 for exp, log, sine, cosine, tangent, ...
  - 23 for log10,
  - 2047 for log2,
  - ...
- Among the remaining ones, there is a minimum (worst-case) distance from $f(x)$ to an FP number.
- Choosing $\overline{\varepsilon}_2$ smaller than this distance guarantees correct rounding.
- Lefèvre and Muller have computed such "worst cases" for quite a few functions.

Why perfection is not very expensive

Both approaches for rounding up a function need

- An algorithm for evaluating y,
- with a proven error bound $\overline{\varepsilon}$
  - That's the difficult bit

Additional cost for a perfect interval function:

- Compute the TMD worst cases
  - (once for all)
- Design a rounding test
  - (already studied in projects such as MPFR and `CRLibm`)
- Write a second, more accurate computation (and prove its $\overline{\varepsilon}_2$)
  - We have it in `CRLibm`, too
  - This second step is easier to write and prove than the first
  - Its average runtime cost is very low

Bonus

- TMD worst case still missing for some functions on some intervals.
  - Statistical arguments gives us fair hope that our code always returns correctly rounded result.
  - For IA, hope isn't enough.
- Never mind. Round up $y_2/(1-\overline{\varepsilon}_2)$ at the end of the second step (or, second rounding test)
  - Statistics predict that it will probably never change the result.
  - If it does, it will add one ulp.
  - Not expensive in average, because in the second step.
- For such functions with missing TMD worst cases, we get a probably perfect interval functions.
  - The "probably" is on tightness, not on safety nor performance.
  - More TMD work may prove tightness *a posteriori*.

Performance of an interval function

Performance should be more or less in a factor two
with respect to the scalar libm (Priest, Arith13)

- (more): context changes (argument passing, processor status)
- (less): work sharing (argument reduction, table prefetches, cache misses)
- (less): intrinsic parallelism, better pipeline usage

Implementation on an Itanium® 2-based platform

Interval arithmetic, directed rounding and elementary functions

Perfect interval functions

Implementation

Conclusion and future work

The present work

Convert Itanium-optimised correctly rounded `exp` and `log` into a perfect interval function.

Parallel Estrin evaluation

Itanium has two parallel FMAs *(fused multiply and add)*,
and many many registers.

```
z2i = zi*zi;                    z2s = zs*zs;
p67i = c6 + zi*c7;              p67s = c6 + zs*c7;
p45i = c4 + zi*c5;              p45s = c4 + zs*c5;
p23i = c2 + zi*c3;              p23s = c2 + zs*c3;
p01i = logirhi + zi*c1;         p01s = logirhs + zs*c1;
z4i  = z2i*z2i;                 z4s  = z2s*z2s;
p47i = p45i + z2i*p67i;         p47s = p45s + z2s*p67s;
p03i = p01i + z2i*p23i;         p03s = p01s + z2s*p23s;
p07i = p03i + z4i*p47i;         p07s = p03s + z4s*p47s;
logi = p07i + Ei*log2h;         logs = p07s + Es*log2h;
```

Combined rounding test

Itanium allows to mix rounding modes and precisions at no cost.

```
/* Tentatively set the result */
result.INF = ROUND_EXT_TO_DOUBLE_DOWN(logi);
result.SUP = ROUND_EXT_TO_DOUBLE_UP(logs);

/* Rounding test  */
mantissai = GET_EXT_MANTISSA(logi);
mantissas = GET_EXT_MANTISSA(logs);
bitsi =  mantissai & (0x7ff&(accuracymask));
bitss =  mantissas & (0x7ff&(accuracymask));
infDone= (bitsi!=0) && (bitsi!=(0x7ff&(accuracymask)));
supDone= (bitss!=0) && (bitss!=(0x7ff&(accuracymask)));

/* Only one test, expected true */
if(__builtin_expect(infDone && supDone, TRUE))
  return result;

/* otherwise lauch accurate computation */
```

Avoiding a squaring of the number of tests

In each function, a few tests for special cases

(subnormals, infinities, etc.)

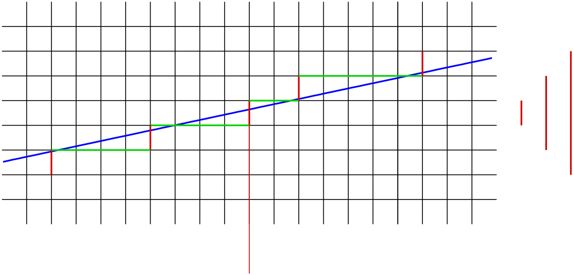
- Make common case fast:
  Fuse all the tests for both endpoints into one, with static prediction.
- If this branch is taken, simply call the existing CRLibm functions that round up and down.

Compared Itanium 2 timings

Library	exp	interval exp	log	interval log
`libm`	42	n/a	31	n/a
`fi_lib`	586	1038	619	1158
`CRlibm`	60	**69**	66	**96**

An exp/log iteration

$$\begin{cases} y_{n+1} = \log(x_n) \\ x_{n+1} = \exp(y_{n+1}) \end{cases}$$



Library	Interval bloat in ulps
`fi_lib`	13
`CRlibm`, RU and RD functions	2
`CRlibm` merged interval function	2

Conclusion and future work

Conclusions

- Itanium is nice for interval arithmetic
  - Nice floating-point unit for basic operations
  - Many many registers for parallel evaluation of elementary functions.
  - Predication, etc, for fused tests.
- `CRLibm` is nice for writing perfect interval functions
  - Safety: We are very happy to recycle our proofs
  - Tightness thanks to correct rounding
  - Performance-oriented algorithms

Future work

- Argue a lot: Is it worth writing tight functions ?
  - Yes in `CRLibm` (why not ? No additional work)
  - Not sure if you start from scratch
- A few implementation questions
  - The basic structure for FP intervals
  - Trigonometric argument reduction
  - Portability on a register-starved Pentium
- Publish `CRLibm0.14beta1`

Questions ?