CORE-MATH Correctly Rounded Mathematical Functions ... Up to the End User

Today

```
#include <stdio.h>
#include <math.h>

int main()
{
  float x1 = 1.01027, x2 = 1.775031328;
  printf ("sinf(x1)=%.9f sinf(x2)=%.9f\n",
           sinf (x1), sinf (x2));
}
```

GNU libc 2.35:

```
sinf(x1)=0.846975386 sinf(x2)=0.979216456
```

Intel Math Library (oneAPI 2022.0.0):

```
sinf(x1)=0.846975446 sinf(x2)=0.979216397
```

Our Dream

```c
#include <stdio.h>
#include <math.h>

int main()
{
  float x1 = 1.01027, x2 = 1.775031328;
  printf ("sinf(x1)=%.9f sinf(x2)=%.9f\n",
          cr_sinf (x1), cr_sinf (x2));
}
```

GNU libc 3.4.5:

```
        sinf(x1)=0.846975386 sinf(x2)=0.979216397
```

Intel Math Library 23.1.4.217:

```
        sinf(x1)=0.846975386 sinf(x2)=0.979216397
```

How to make our dream possible?

- wait for the next IEEE-754 revision (2029), try to convince the revision committee that correct rounding is required for math functions, and wait another 5-10 years that math libraries do comply;
- or build yet another mathematical library with correct rounding, make it available to users, and maintain it over the different hardware, operating systems and compilers;
- or write efficient math routines with correct rounding, and contribute them to the already existing math libraries (GNU libc, Intel Math Library, AMD Libm, Redhat Newlib, OpenLibm, Musl, ...)

Current mathematical libraries

	binary32	binary64	binary80	binary128
GNU libc	√	√	√	√
Intel Math Library	√	√	√	√
AMD Libm	√	√		
Redhat Newlib	√	√		
OpenLibm	√	√	√	
Musl	√	√	√	

GNU libc is available in most Linux distributions.

The Intel Math Library is now freely available as a Docker image.

Musl is available in Alpine Linux.

IML and most of AMD Libm have their own code base, other libraries share some functions.

Also: Apple Math Library, LLVM libc, CUDA libm, RoCM

Why did previous attempts fail?

- MathLib/libultim (Ziv, 1991): almost succeeded, since Mathlib code was introduced in GNU libc, but the “slow path” was removed from 2018 to 2021. Main drawbacks: obscure code, large tables, very slow worst cases;
- CRlibm (Daramy-Loirat, Defour, de Dinechin, Gallet, Gast, Lauter, Muller, 2004-2006): much better worst-case performance, proofs of correctness, small tables, better use of FMA. Initial goal was only a “proof of concept” with good performance (and it was successful as such).

Why would it succeed now?

- the next C standard has reserved names `cr_sin` for correctly-rounded (CR) functions (see draft N2731);
- recent progress in the search for worst cases (TaMaDi project 2010-2013, BacSel tool, Serge Torres PhD thesis 2016). This will help in reducing the worst-case times;
- recent progress in the implementation (MetaLibm project, 2014-2018);
- very good knowledge of the field (Handbook of Floating-Point Arithmetic, 2010, 2nd edition 2018);
- good contacts with the developers of math libraries, with the C Floating-Point group working on C bindings for IEEE 754;
- numerical reproducibility is more and more a concern (cf NRE2019 workshop in 2019, ICERM workshop in 2020)

Workplan

Phase 1 (completed): publish efficient CR code for two sample functions (`cbrt` and `acos`).

Phase 2 (in progress): contact developers/vendors of math libraries and propose them to join the project (as observer, developer, counsellor, information provider, computing power provider, ...). Discuss with them of technical details (license, size of tables, code size, ...)

Phase 3 (started): implement highly optimized CR code with proofs, and help developers/vendors of math libraries to integrate it

Phase 4: provide support if bugs are found in our original code and/or in the proofs.

Which functions?

Target the 39 functions defined in IEEE 754-2019:

- `exp`, `expm1`, `exp2`, `exp2m1`, `exp10`, `exp10m1`;
- `log`, `log2`, `log10`, `logp1`, `log2p1`, `log10p1`;
- `hypot`;
- `rSqrt`;
- `compound`;
- `rootn`, `pown`, `pow`, `powr`;
- `sin`, `cos`, `tan`, `sinPi`, `cosPi`, `tanPi`;
- `asin`, `acos`, `atan`, `asinPi`, `acosPi`, `atanPi`;
- `atan2`, `atan2Pi`;
- `sinh`, `cosh`, `tanh`;
- `asinh`, `acosh`, `atanh`.

Which target formats

- single precision (`binary32`);
- double precision (`binary64`);
- extended double precision (`long double` on `x86_64`);
- quadruple precision (`binary128`).

How to Round Correctly?

Assume the target type has n bits.

Let m be a bound on the HR (hardest-to-round) cases (usually $m \approx 2n$), and $\circ()$ the current rounding mode.

$\exp(0.09407822313572878) = 1.098645682066338500000000000000000278$

- [quick phase] compute an approximation y with say $n + 10$ correct bits, and maximal error ε
- [rounding test] if $\circ(y - \varepsilon) = \circ(y + \varepsilon)$, return that number
- [accurate phase] otherwise compute an approximation y' with more than m correct bits, then $\circ(y')$ is always CR

Workplan for a given function and a given target format

Compute hardest-to-round (HR) cases and exact cases (if any).

Try several implementations of the quick phase and of the accurate phase: different kinds of argument reduction, different internal formats (`double`, `int64_t`), optional use of FMA. Check their correctness on the HR and exact cases.

Keep the fastest one, optionally prove its correctness, possibly with the use of some exhaustive search, and tools like Sollya, Gappa.

Publish the code. Help the math lib vendors/developers to integrate the code.

How to do for missing HR cases?

In some cases (atan2/hypot/pow in single precision, sin in double precision, all functions in quadruple precision), it might be very difficult to compute HR cases.

At the end of the accurate phase, since we know the maximal error, we can perform a second rounding test.

If it succeeds, we are sure the computed value is CR.

Otherwise, it might be wrong. Either return it (default mode), or launch an exception at the user request.

Possibly offer bug bounties for non-CR cases, or inputs giving such an exception.

Cf Brisebarre/Hanrot estimates on the number of HR cases.

How to deal with rounding modes?

One routine for each mathematical function:

```
fesetround (FE_TOWARDZERO);
y = cr_sin (x);
```

This is already how it works for the `sqrt` function.

Current progress

Cf `https://core-math.gitlabpages.inria.fr/`

MIT license to make integration easier.

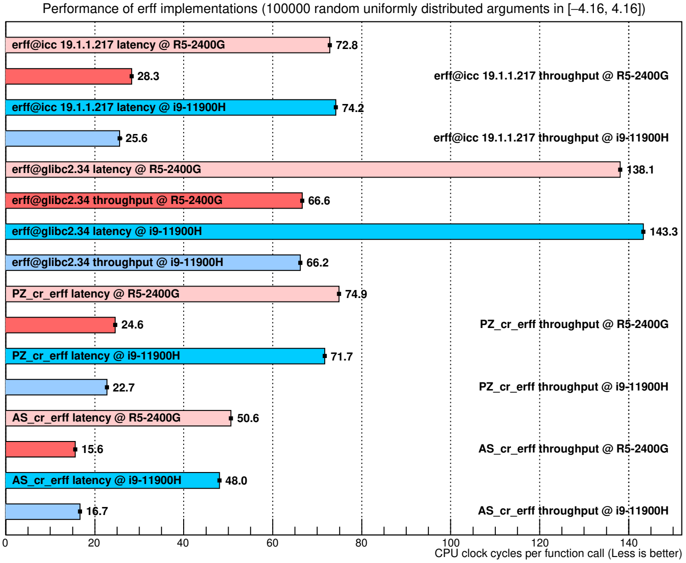
binary32: all C99 functions implemented (except powf)

binary64: acos, cbrt, exp (to nearest)

binary80: acos, cbrt

binary128: acos, cbrt

And what about performance?



How can I contribute?

- find a bug in the published functions
- submit a faster CR implementation
- find hard-to-round cases for binary64, binary80 or binary128
- make a formal proof of some implementation
- work on the integration in some libm

References

`https://core-math.gitlabpages.inria.fr/`: main page

`https://gitlab.inria.fr/core-math/core-math/`: git page