Towards a correctly rounded x^y in double precision

Tom Hubrecht (with Claude-Pierre Jeannerod, Paul Zimmermann)

November 07, 2023

LIP - Équipe AriC

Introduction

IEEE-754 rounding modes :

- Round to nearest, ties to even
- Round towards 0
- Round towards $+\infty$
- Round towards $-\infty$

Correct rounding :

- A given output is correctly rounded if it is the result of the selected rounding function applied to the infinitely precise value.
- Beware of special inputs: $\pm\mathbf{0}$, $\pm\infty$, NaN

Previous work

- Books from Markstein [1], and Beebe [2]
- MathLib from IBM (Ziv [3], 1991)
- libmcr (Sun, 2004)
- CRlibm (De Dinechin, Lauter et al. [4], 2006)
- RLIBM and LLVM libc [5] (without a binary64 power function yet)

MathLib (Ziv, 1991)

- Only works in round to nearest
- Integrated in the GNU libc, but slow path removed in 2018 (after v2.27)
- No longer maintained
- Algorithm not detailed for x^y

- Only works in round to nearest mode.
- Doe
  x=0
- Som
  x=0
  0x1
- No longer maintained

Floating point hexadecimal litteral :

`0x1.f80b060553772p-1` = $\left(1+\frac{4363619669456754}{2^{52}}\right)\times 2^{-1}$

Closer to the machine representation.

Libmcr (Sun, 2004)

- Only works in round to nearest mode.
- Does not terminate for some inputs e.g. :
  `x=0x1.470574d68e0afp+1`, `y=0x1.02e0706205c0ep+1`.
- Some wrong results e.g. :
  `x=0x1.f80b060553772p-1`, `y=0x1.99cp+13` gives
  `0x1.00001p+0` instead of `0x1.a2e7cca9cfd72p-297`.
- No longer maintained

CRlibm

- Only works in round to nearest mode.
- Power function still experimental
- Algorithm detailed in Ch. Lauter’s PhD thesis [6]
- On hard to round cases, returns -5
- No longer maintained

Our contribution

- Open-Source in the CORE-MATH project [7]
- All rounding modes are supported
- A paper in ARITH 23 [8], with complete proofs of the first phase
  (arguably very hard to read)
- Detailed explanations within the source code
- Performance comparable to incorrectly rounded math libraries

Performance

Library	GLIBC (v2.38)	LibUltim	CRLibm	CORE-MATH
Reciprocal throughput	23	60	110	30
Latency	60	105	150	72

Table 1: Timings (in number of cycles) for pow on an Intel Core i7-1260P

Methodology :

- Using `rdtsc`
- Randomly choose $x, y \in [0, 10]$

How to compute x^y

Polynomial approximations and decomposing the evaluation are required.

- Markstein and Beebe propose to express $x^y = 2^{y \cdot \log_2 x}$, but the coefficients of the Taylor expansions of 2^t and $\log_2(1+t)$ at 0 are not nice.
- We prefer to use $x^y = e^{y \cdot \log x}$, where the argument reduction is more complex, but the Taylor expansions have simpler coefficients [9].

Polynom

- Marks

of the Ta

- We pre ... ex,

but the

Taylor expansions :

$$\log_2(1+t) = \log(2)^{-1} \times \left(t - \frac{t}{2} + \frac{t^3}{3} - \cdots\right)$$

$$\log(1+t) = t - \frac{t}{2} + \frac{t^3}{3} - \cdots$$

$$2^t = 1 + t\log(2) + \frac{(t\log(2))^2}{2} + \cdots$$

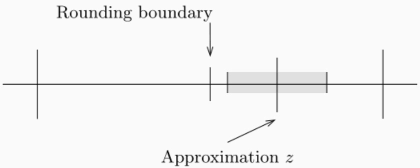
$$e^t = 1 + t + \frac{t^2}{2} + \cdots$$

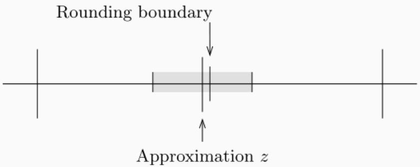
How to compute x^y

Polynomial approximations and decomposing the evaluation are required.

- Markstein and Beebe propose to express $x^y = 2^{y \cdot \log_2 x}$, but the coefficients of the Taylor expansions of 2^t and $\log_2(1+t)$ at 0 are not nice.
- We prefer to use $x^y = e^{y \cdot \log x}$, where the argument reduction is more complex, but the Taylor expansions have simpler coefficients [9].

For a result in double precision, we need to use more precise intermediate values.





Hard to round case :

`pow(0x1.0bf06323f41cfp-1, 0x1.4p-2) ≈ 0x1.a2334671ea7f70000000000000074p-1`

Midpoint case :

`pow(0x1.4p-44, 0x1.7p+4) = 0x1.52d02c7e14af68p-1005`

Exact case :

`pow(0x1.8p-33, 0x1.fp+4) = 0x1.18e2a5afb5158p-1005`

Exact and midpoint cases

x^y may be representable with 53 bits (an **exact** double) or 54 bits (a **midpoint**).

In theory, we need an infinitely precise result to compute the correctly rounded result. Hence filtering them is necessary.

Thanks to Ch. Lauter and V. Lefèvre ([10], 2009), we have an efficient algorithm to treat those cases.

Workflow

- Three-phase approach, with increasing precision
- Filter the exact and midpoint cases between the second and third phase
- Probability to use the second phase roughly 2^{prec-p}, where :
  $prec$ is the computation precision and p the number of bits of the significand
  Thus, we target ~ 67 bits of precision at first.
- Because of compounded errors, we need more precision for the logarithm

Specification for the first phase

Operations use a mix of double values and double doubles ($b = b_h + b_\ell$)

- `log_1` (Approx. of $\log(x)$) takes a double and returns a double double
- `p_1` (Approx. of $\log(1+z)$) also
- `exp_1` (Approx. of $\exp(y)$) takes a double double and returns a double double
- `q_1` (Approx. of $\exp(z)$) takes a double and returns a double double

Approximation of $\log(x)$

Algorithm 1 log(x)

let $x = 2^E \cdot y$ ▷ With $1 \leq y < 2$
if $y > \sqrt{2}$ **then**
  $y \leftarrow \frac{y}{2}$
  $E \leftarrow E + 1$
end if
$i \leftarrow \lfloor y \times 2^8 \rfloor$
$z \leftarrow y \cdot r_i - 1$ ▷ z is computed without rounding thanks to an FMA
return $E \cdot \ln(2) - \ln(r_i) + \ln(1 + z)$

- $\ln(1 + z)$ is computed with a polynomial approximation

Approximation of $\log(1+z)$

- FPminimax polynomial approximation, found with Sollya [11]
- Estrin method for evaluation allows parallelism via hardware

- FPminimax polynomial approximation, found with Sollya [11]
- Estr

$$P(X) = \sum_{i=0}^{n} a_i X^i$$

Horner method :

$$P(X) = a_0 + X \times (a_1 + X \times (a_2 + X \times \cdots))$$

Estrin method :

$$\begin{aligned} P(X) = &\left[a_0 + X^2 \times (a_2 + X^2 \times \cdots)\right] \\ &+ X \times \left[a_1 + X^2 \times (a_3 + X^2 \times \cdots)\right] \end{aligned}$$

- FPminimax polynomial approximation, found with Sollya [11]
- Estrin method for evaluation allows parallelism via hardware

Lemma

Given $|z| \leq 33 \cdot 2^{-13}$, with $z \in 2^{-61} \cdot \mathbf{Z}$, the approximation $p_h + p_\ell$ returned by `p_1` satisfies :

$$|p_h + p_\ell - (\log(1+z) - z)| < 2^{-75.492}$$

with $|p_h| < 2^{-16.9}$, $|p_\ell| < 2^{-25.446}$.

If $z \neq 0$, $|z| \leq 32 \cdot 2^{-13}$, the relative error is smaller than $2^{-67.441}$

Approximation of $\mathbf{exp}(y \cdot \mathbf{log}(x))$

Algorithm 2 exp(x)

$k \leftarrow \left\lfloor x \cdot \frac{2^{12}}{\ln(2)} \right\rceil$

$y \leftarrow x - k \cdot \frac{\ln(2)}{2^{12}}$

let $k = M \cdot 2^{12} + i_2 \cdot 2^6 + i_1$ ▷ With $0 \leq i_1, i_2 < 2^6$

$t_1 \leftarrow 2^{i_2/2^6}, \quad t_2 \leftarrow 2^{i_1/2^{12}}$

return $2^M \cdot t_1 \cdot t_2 \cdot e^y$

Formal proofs for bounds

Thanks to a huge and incredible work from Laurence Rideau and Laurent Théry, the error bounds obtained by hand in 10 pages of annexes for the ARITH paper are now formally proven in Coq for the first phase of the algorithm.

The repository containing the proof is at :

`https://github.com/thery/ExpFloat`

Software floating-point formats

- Use 64 bit integers for the exponent, unsigned integers for the significand (128 or 256 bits)
- Allows for a wider range of values
- Multiple addition and multiplication algorithms implemented (tailored to the width of data used)
- Better hardware implementations

Extended floating point formats

- First phase uses double double values
- Experiment using triple double values in the second phase
  - Format is closer to the inputs
  - More complex algorithms,
    based on previous work by N. Fabiano, J. Picot, J.-M. Muller [12]
  - Taking into account drastic optimisation,
    at least twice slower than using emulated floating point formats
  - The exact phase requires integer significands, so conversion needed

Main achievements and future perspectives

- Correct results in all rounding modes (up to the knowledge of worst-cases)
- Fully compliant with IEEE-754 (regarding special cases)
- Detailed proofs and error analysis
- Faster execution times than previous work ($2\times$ improvement)
- Single execution path for all rounding modes
- A new version, 25% faster than in the paper

- Work is still required to find worst cases
- Would be great to have more automation in the proofs,
  (e.g. use Gappa to formalize the errors in the second and third phases)

References

[1] Peter Markstein. *IA-64 and Elementary Functions: Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000.

[2] Nelson H. F. Beebe. *The Mathematical-Function Computation Handbook - Programming Using the MathCW Portable Software Library*. Springer, 2017. ISBN: 978-3-319-64109-6. DOI: 10.1007/978-3-319-64110-2.

[3] A. Ziv. "Fast evaluation of elementary mathematical functions with correctly rounded last bit". In: 17.3 (1991), pp. 410–423.

[4] Catherine Daramy-Loirat et al. *CR-LIBM: A library of correctly rounded elementary functions in double-precision*. Research Report. https://hal-ens-lyon.archives-ouvertes.fr/ensl-01529804. LIP, 2006.

[5] *The LLVM C Library*. https://libc.llvm.org/.

[6] Christoph Q. Lauter. "Arrondi correct de fonctions mathématiques. Fonctions univariées et bivariées, certification et automatisation". PhD thesis. Université de Lyon - École Normale Supérieure de Lyon, 2008.

[7] Alexei Sibidanov, Paul Zimmermann, and Stéphane Glondu. "The CORE-MATH Project". In: *ARITH 2022 - 29th IEEE Symposium on Computer Arithmetic*. virtual, France, 2022. URL: https://hal.inria.fr/hal-03721525.

References

[8] Tom Hubrecht, Claude-Pierre Jeannerod, and Paul Zimmermann. "Towards a correctly-rounded and fast power function in binary64 arithmetic". URL: https://inria.hal.science/hal-04159652.

[9] Jean-Michel Muller et al. *Handbook of Floating-Point Arithmetic, 2nd edition*. ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-3-319-76525-9. Birkhäuser Boston, 2018, p. 632.

[10] Christoph Quirin Lauter and Vincent Lefèvre. "An Efficient Rounding Boundary Test for $\mathrm{pow}(x, y)$ in Double Precision". In: 58.2 (2009), pp. 197–207.

[11] Sylvain Chevillard, Mioara Maria Joldes, and Christoph Lauter. "Sollya: an environment for the development of numerical codes". In: *Third International Congress on Mathematical Software - ICMS 2010*. Ed. by Komei Fukuda et al. Vol. 6327. Lecture Notes in Computer Science. Kobe, Japan: Springer, 2010, pp. 28–31. DOI: 10.1007/978-3-642-15582-6_5.

[12] Nicolas Fabiano, Jean-Michel Muller, and Joris Picot. "Algorithms for Triple-Word Arithmetic". In: *IEEE Transactions on Computers* 68.11 (2019), pp. 1573–1583. DOI: 10.1109/TC.2019.2918451.

Reasons for slow-path removal

Wilco Dijkstra <wdijkstr@arm.com> Mon, 12 Feb 2018

Remove the slow paths from pow. Like several other double precision math functions, pow is exactly rounded.
This is not required from math functions and causes major overheads as it requires multiple fallbacks using higher precision arithmetic if a result is close to 0.5ULP.
Ridiculous slowdowns of up to 100000x have been reported when the highest precision path triggers.