Correctly rounded **pow** in double precision

Tom Hubrecht

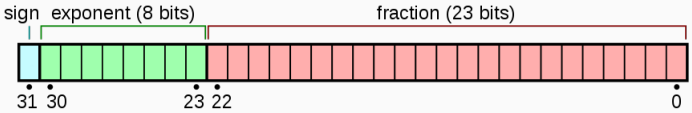
June 8, 2022

CERN, INRIA

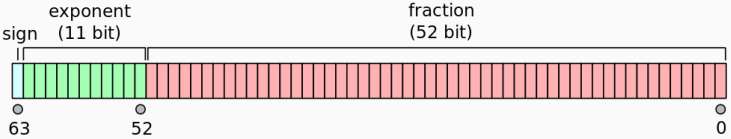
- Floating point numbers (A quick reminder)
  - Formats
  - Rounding
- The CORE-MATH project
- `pow` in double precision
  - Correct rounding
  - Previous work
  - Current approach
  - Details on `exp` and `log`

Floating point numbers (A quick reminder)

IEEE754 floating point numbers (in binary form)



$$\text{value} = (-1)^{sign} \times 2^{exponent-127} \times 1.b_{22}b_{21}\ldots b_0$$



$$\text{value} = (-1)^{sign} \times 2^{exponent-1023} \times 1.b_{51}b_{50}\ldots b_0$$

- Beware of subnormal numbers : $(-1)^{sign} \times 2^{e_{min}} \times 0.\bar{b}$

Rounding numbers

Only a small number of representable numbers.

Transcendental functions return real numbers (i.e. the return values are almost never in the set of floating-point values), hence the need to approximate :

$$e \approx 2.71828182845905 \quad (\texttt{0x1.5bf0a8b145769p+1})$$
$$log(2) \approx 0.693147180559945 \quad (\texttt{0x1.62e42fefa39efp-1})$$

IEEE754 rounding modes

Several rounding modes :

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

Correct rounding definition :

• An approximated value is correctly rounded if it is the result of the selected rounding function applied to the infinitely precise value.

IEEE754 requirements

Several functions are required to be correctly rounded :

- `add`, `subtract`, `multiply`, `divide`, `remainder`
- `sqrt`, `fma`

But not others (`sin`, `exp`, and of course `pow`), which leads to :

- GLIBC v2.27 :
  `pow(0×1.8p-214, 0×1.4p+2) = 0×0.0000000000079p-1022`
- LibUltim :
  `pow(0×1.8p-214, 0×1.4p+2) = 0×0.000000000007ap-1022`

GLIBC returns an incorrect result for half (8M) of the midpoint cases (where the results fits on 54 bits but not 53).

Known errors for **pow** in double precision

Library (Version)	GLIBC (2.35)	IML (2021.2.0)	AMD (3.8)	Newlib (4.2.0)	CUDA (11.5.0)
Max known error (ulps)	0.523	1.73	0.754	636	1.40

A "unit of least precision (ulp)" is the distance between two floating point numbers of the same binade.

- Vincenzo Innocente, Paul Zimmermann. Accuracy of Mathematical Functions in Single, Double, Extended Double and Quadruple Precision. 2022.

The CORE-MATH project

Different versions

```c
#include <stdio.h>
#include <math.h>
#include <gnu/libc-version.h>

int main() {
  printf("GNU libc version: %s\n", gnu_get_libc_version ());
  double x = -0x1.f8b791cafcdefp+4;
  printf ("sin(x)=%la\n", sin (x));
}
```

```
$ gcc -fno-builtin sin.c -lm
```

- GNU libc version 2.27 : `sin(x)=-0x1.073ca87470df9p-3`
- GNU libc version 2.33 : `sin(x)=-0x1.073ca87470dfap-3`

Different hardware

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

int main() {
  double x = 0x1.01825ca7da7e5p+0;
  printf ("x=%la y=%la\n", x, acosh (x));
}

$ icc -fno-builtin test_acosh.c # icc version 19.1.3.304
```

- Intel Xeon Gold 6142 :
  `x=0x1.01825ca7da7e5p+0 y=0x1.bc8c6186687cbp-4`
- AMD EPYC 7452 :
  `x=0x1.01825ca7da7e5p+0 y=0x1.bc8c6186687cap-4`

Observations

- Two different libms produce different results
- Two different versions of the same library produce different results
- Two different processors produce different results

All due to incorrect rounding.

Thus, results are not reproducible, and are less accurate than ideally.

How to address this issue

Several options in theory :

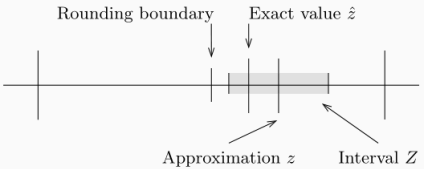
- Try to change the IEEE754 standard to impose correct rounding
- Create another math library
- Write fast (and correct) routines that can be included in all major math libraries

The last option is the most viable and is the object of the CORE-MATH project.

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

pow in double precision

The Table Maker's Dilemna



(a) Easy to round case

(b) Hard to round case

- Example :

```
pow(0×8.5f83191fa0e78p-4,0×5p-4) ≈ 0×d.119a338f53fb8000000000000003a35p-4
```

How to round correctly ?

Let $\circ$ be the rounding function.

The idea is to compute an approximation z with an error ε, then if $\circ(z-\varepsilon) = \circ(z+\varepsilon)$, the approximation is correctly rounded and we can return it. If not, we need to compute the result with more precision and try again. This is called Ziv's algorithm.

But, for **pow**, there are cases where z is exact or a midpoint value and therefore for any precision large enough, $\circ(z-\varepsilon) \neq \circ(z+\varepsilon)$, hence we need to treat them specially.

Previous work

- IBM's LibUltim (1991) : Correctly rounded implementation using Ziv's algorithm
- CRLibm (2007) : Theoretical results and start of the implementation but still incomplete

Current approach

- A "fast path" algorithm computes quickly an approximation with ~ 69 bits of accuracy
- If the rounding test fails, we check for easy cases ($y \in \{1, 2, 3, 4, 0.5, 0\}$ or $x = 1$)
- Then we compute another approximation with at least 120 bits of precision
- If the rounding test fails again, we deal with the exact and midpoint cases (Based on the algorithm from [1])
- If we are not with a midpoint or exact case, then we compute an approximation with at least 250 bits of accuracy and return its rounding.

Approximation algorithms

For all approximations, the broad algorithm is the same, we write `pow(x,y)` = `exp(y * log(x))`, and compute the logarithm then the exponential. The only difference is the data type used.

How to compute `log(x)`

- Let $x = 2^{E'} \times m$ with $m \in [\![1, 2[\![$ and $E' \in \mathbf{Z}$
- If $m \geq \sqrt{2}$ then let $E = E' + 1$ and $y = \frac{m}{2}$ else, $E = E'$ and $y = m$ (This is to avoid cancellation)
- Let $z = y \times r - 1$ (Where r is an approximation of $\frac{1}{y}$ such that $|z| < 2^{-6}$)
- Let $l = -\log(r)$, then $\log(x) = E \times \log(2) + l + \log(1 + z)$

We get the values l and r from a table, and approximate $\log(1 + z)$ with a polynomial.

How to calculate **exp(x)**

- Let $k = \left\lfloor x \times \frac{2^{12}}{\log(2)} \right\rceil$ and $y = z - k \times \frac{\log(2)}{2^{12}}$

  Write $k = M \times 2^{12} + i_2 \times 2^6 + i_1$, and let $t_2 = 2^{\frac{i_2}{2^6}}$, $t_1 = 2^{\frac{i_1}{2^{12}}}$
- We then have : $\exp(x) = 2^M \times t_1 \times t_2 \times \exp(y)$

As y is very small, we approximate $\exp(y)$ with a polynomial and get the values for t_1 and t_2 from a table.

Fast path

To represent the values, we use two doubles, i.e. $b = b_h + b_l$, the main issue is the overlap between b_h and b_l as it decreases precision.

Second iteration

We use a representation of floating point numbers with 128 bits of mantissa (two `uint64_t`), and implement base functions with a small error to improve simplicity.

Exact and midpoint cases

Let $\mathbf{D}$ be the set of double precision floating-point numbers, all the exact and midpoint cases lie in the set

$$S = \{(x, y) \in \mathbf{D}^2 | y \in \mathbf{N}, 2 \leq y \leq 35\}$$
$$\cup \{(m, 2^F n) \in \mathbf{D}^2 | F \in \mathbf{Z}, -5 \leq F < 0, n \in 2\mathbf{N} + 1, 3 \leq n \leq 35, m \in 2\mathbf{N} + 1\}$$

As the worst case (not exact or midpoint) in S needs at most 117 bits of precision [1], if the previous rounding test fails, either $(x, y) \notin S$, or it is an exact or midpoint case. Therefore, we can filter the later cases and treat them specially.

Last iteration

The last iteration of rounding uses floating-point numbers with 256 bits of mantissa (four `uint64_t`). No test is done after this iteration, so it is possible in theory to have cases where the result returned by our function is incorrect, but to the best of our knowledge, it is not the case and computing all the hard-to-round cases is not currently feasible.

Current work

The implementation is finished, what is left to do is :

- Error analysis

Performance

On an Intel Core i7-9750H :

```
$ CORE_MATH_PERF_MODE=perf ./perf.sh pow
120.018
60.416

$ PERF_ARGS=--latency CORE_MATH_PERF_MODE=perf ./perf.sh pow
170.681
110.228
```

The number of CPU cycles according to `perf`, the first number is my implementation and the second the GLIBC's implementation (not correctly rounded).

References

1 Christoph Lauter, Vincent Lefèvre. An efficient rounding boundary test for pow(x,y) in double precision. 2007.

2 Muller, J.M., Brisebarre, N., de Dinechin, F., Jeannerod, C.P., Lefèvre, V., Melquiond, G., Revol, N., Stehlé, D., Torres, S.: Handbook of Floating-Point Arithmetic. Birkhäuser, Boston (2010)

3 Paul Zimmermann, CORE-MATH : quand pourra-t-on calculer correctement ? 2021.