Towards a correctly-rounded and fast power function in binary64 arithmetic

Tom Hubrecht
Département d'Informatique de l'ENS
École Normale Supérieure,
CNRS, PSL University
F-75005 Paris, France
tom.hubrecht@ens.fr

Claude-Pierre Jeannerod
Inria, Université de Lyon
CNRS, ENSL, UCBL, LIP
F-69342 Lyon, France
claude-pierre.jeannerod@inria.fr

Paul Zimmermann
Université de Lorraine
CNRS, Inria, LORIA
F-54000 Nancy, France
paul.zimmermann@inria.fr

***Abstract*—We design algorithms for the correct rounding of the power function x^y in the binary64 IEEE 754 format, for all rounding modes, modulo the knowledge of hardest-to-round cases. Our implementation of these algorithms largely outperforms previous correctly-rounded implementations and is not far from the efficiency of current mathematical libraries, which are not correctly-rounded. Still, we expect our algorithms can be further improved for speed. The proofs of correctness are fully detailed in the extended version [9] of this paper, with the goal to enable a formal proof of these algorithms. We hope this work will motivate the next IEEE 754 revision committee to require correct rounding for mathematical functions.**

***Index Terms*—IEEE 754, double precision, binary64 format, power function, correct rounding, efficiency.**

I. Introduction

The IEEE standard for floating-point arithmetic requires correct rounding for basic arithmetic operations since 1985, but in its latest revision (IEEE 754-2019) [10], it still does not require correct rounding for the most common mathematical functions such as $\log$, $\exp$, $\sin$, $\cos$, etc.

Among all these commonly used functions, one of the trickiest ones to implement is the power function $(x, y) \mapsto x^y$. One difficulty is that it is a bivariate function, thus it has many more possible inputs for a given IEEE format, and in particular its hardest-to-round cases are not known yet for the binary64 format (formerly called 'double precision' and where the precision is 53). Another difficulty of the power function is that it has many exact and midpoint cases (that is, cases where x^y is exactly representable in precision 54). Finally, as the relation $x^y = b^{y \log_b x}$ for $b > 1$ and $x > 0$ shows, the power function grows very fast in some regions. This requires that some intermediate steps be performed with a significant amount of extra precision which however must be kept small enough in order to achieve high efficiency in practice.

Current mathematical libraries do not provide a correctly rounded power function for the binary64 format. For this format, the maximal known error goes from 0.523 ulp for the GNU libc to 636 ulps for OpenLibm [11].

A. State of the Art

Detailed descriptions of how to implement the power function are given in Markstein's book [18, Chapter 12] and Beebe's book [1, Chapter 14]. Markstein suggests using the formula $x^y = 2^{y \log_2 x}$, a choice already advocated in [3]. We tried this approach first as it looked quite promising for binary arithmetic, but in our context it turned out to be less efficient than using the relation $x^y = e^{y \log x}$, essentially since the Taylor approximations of $\log x$ and e^x have simpler coefficients. Beebe also suggests evaluating x^y by means of $2^{y \log_2 x}$ for binary arithmetic. His algorithms are more complex, in particular since the fused multiply-add (FMA) instruction is not used for producing exact products. Finally, none of these two chapters discusses ways to ensure correct rounding.

The MathLib library [23] includes a correctly rounded power function for the binary64 format. However, the algorithm used is not fully detailed, and MathLib is no longer maintained. Some MathLib routines (including the power function) were included in the GNU libc up to release 2.27, but the "slow path" was gradually removed, and the latest release of the GNU libc does no longer provide a correctly rounded power [11]. Furthermore, MathLib only provides correct rounding for rounding to nearest-even.

Sun Microsystems developed a library called LIBMCR that included in its latest version (release 0.9, April 2004) seven functions with claimed correct rounding (presumably for rounding to nearest-even): exp, log, pow, atan, sin, cos, tan. LIBMCR still compiles with modern compilers but has both efficiency and correctness issues (see §VI).

CRLIBM [4] also provides a correctly rounded power function. Like for MathLib, this function is only implemented for rounding to nearest-even. Also, this function has remained tagged as "experimental" and has indeed some correctness issues (see §VI). Compared to MathLib, though, CRLIBM has a better treatment of exact cases and midpoint cases, based on works by Lauter and Lefèvre [13], [14].

More recently, the RLIBM project [16], the LLVM C Library [17], and the CORE-MATH project [19] have proposed implementations for various correctly-rounded functions. However, at the time of writing, RLIBM only provides binary32 functions, LLVM libc only provides the `hypot` and `log10` functions in binary64, and CORE-MATH provides about 20 binary64 functions, but not x^y.

B. Our Contributions

In this paper we present a new algorithm for the binary64 power function with correct rounding (modulo the knowledge of hardest-to-round cases) together with a very efficient implementation. Compared to previous work (MathLib, LIBMCR, and CRLIBM), this algorithm is not restricted to rounding to nearest-even, but also works for the directed rounding modes from IEEE 754.

Furthermore (and as far as we can tell), this is the first time an algorithm for binary64 powering with correct rounding is fully detailed and presented together with its rounding error analysis. The algorithm, which makes heavy use of the FMA instruction available on most modern processors, can be readily implemented.

Finally, our C implementation of the algorithm yields a two-fold to four-fold speedup with respect to MathLib, CRLIBM, and LIBMCR.

C. Outline

The paper is organized as follows. Section II gives a high-level description of our three-phase algorithm for computing correctly-rounded powers for the IEEE binary64 format, together with some notation to be used later on. Section III then provides a detailed description of all the building blocks and algorithms used in the first phase of the powering algorithm, together with explicit error bounds (each of which having a detailed proof given in [9]). Section IV only provides a coarse description of the second and third phases, since those follow essentially the same approach as the first phase (up to higher precisions, larger approximation polynomials, and more sophisticated range reduction). Section V discusses the correctness of our implementation on known worst cases as well as on random inputs. Section VI compares our implementation to the MathLib, LIBMCR and CRLIBM libraries, and to the incorrectly-rounded GNU libc power function. We conclude and discuss further work in Section VII.

II. High-level Algorithm

Before approximating x^y using $\exp(y \log x)$, the case $x \leq 0$ gets a special treatment according to the IEEE 754-2019 specification [10, §9]. The same is done for the cases where x or y is $\pm\infty$ or NaN. (See also [12, §F.10.4.5].) Thus, we assume from now on that both x and $|y|$ are in the range $[\alpha, \Omega]$, where $\alpha := 2^{-1074}$ denotes the smallest positive subnormal binary64 number, and $\Omega := (1 - 2^{-53}) \cdot 2^{1024}$ denotes the largest finite binary64 number. Then, clearly, $\log x$ is either zero or in a subrange of the range $[-\Omega, -2^{-1022}] \cup [2^{-1022}, \Omega]$ of normal binary64 numbers.

Our powering algorithm has three phases. A first phase approximates x^y using double-double arithmetic (§III) in a very efficient way; if the rounding test of the first phase fails, then a second phase approximates x^y using 128-bit arithmetic (§IV); if the rounding test of the second phase fails again, then a third phase approximates x^y using 256-bit arithmetic (§IV). Since worst cases for the binary64 power function are still unknown, the third phase contains a final rounding test. In the very unlikely event where this last test fails, an error message is printed, which guarantees that no incorrect rounding is produced.

The three phases use the same algorithm:

- first we compute an approximation of $\log x$, by first rewriting $x = 2^e \cdot t$ with $\sqrt{2}/2 < t < \sqrt{2}$. Then we use Tang's algorithm [21]: a table value r approximates $1/t$ and yields a small value $z \approx r \cdot t - 1$. Finally we get $\log x \approx e \log 2 - \log r + \log(1 + z)$, where an approximation to $\log r$ is read from a table and where $\log(1 + z)$ is approximated using a small-degree polynomial;
- then we multiply the approximation of $\log x$ by y, obtaining a value called r;
- finally we approximate $\exp(r)$ by writing $r = (k + i/2^n) \log 2 + z$, where k is an integer, $i/2^n$ is a small rational, and z is small. This yields $\exp(r) = 2^k \cdot 2^{i/2^n} \cdot \exp(z)$, where an approximation to $2^{i/2^n}$ is read from a table and where $\exp(z)$ is approximated using a small-degree polynomial.

A. Notation and basic properties

We write $\mathbb{Z}$, $\mathbb{N}$, $\mathbb{F}$ to denote the set of integers, nonnegative integers, and finite binary64 numbers, respectively. We write $\mathrm{ulp}(x)$ to denote the unit in the last place of $x \in \mathbb{F}$, which for $x \neq 0$ gives the weight of the last bit of the significand of x. More generally, for any real number x such that $|x| \leq \Omega$, we define its ulp as $\mathrm{ulp}(x) := 2^{\max(-1022, \lfloor \log_2 |x| \rfloor) - 52}$, that is, $\max(\alpha, 2^{\lfloor \log_2 |x| \rfloor - 52})$, with the convention that $\mathrm{ulp}(0) = \alpha$.

We denote by $\circ$ the current rounding mode, which can be to nearest with ties to even, toward zero, toward $+\infty$, or toward $-\infty$. Our algorithm is division free and makes heavy use of operations of the form

$$t \leftarrow \circ(xy + z),$$

with a single rounding; in practice, such operations are performed efficiently by fused multiply-add (FMA) instructions. Unless stated explicitly, all variables such as x, y, z, t above represent finite binary64 numbers.

Our error bounds will be deduced in particular from the following accuracy properties of $\circ$: when $|xy + z| \leq \Omega$, the absolute error of t is bounded as

$$|t - (xy + z)| \leq \mathrm{ulp}(xy + z) \leq \mathrm{ulp}(t);$$

if in addition $|xy + z| \geq 2^{-1022}$ (normal range), then the relative error satisfies $|t/(xy+z) - 1| \leq \mathrm{ulp}(xy+z)/|xy+z|$ and is thus always at most 2^{-52}. Finally, in some proofs we shall also exploit the fact that if $|xy + z| \leq 2^e$ for some integer $e \geq -1021$, then $|t - (xy + z)| \leq 0.5 \cdot \mathrm{ulp}(2^e)$ instead of $\leq \mathrm{ulp}(2^e)$.

III. First Phase

The first phase uses double-double arithmetic for efficiency, and delivers an approximation to x^y with about 63 correct bits in the general case (see Algorithm 9). The only special cases are those discussed earlier: the cases $y \in \{0, 1/2, 1, 2\}$ are

only checked after the first phase, in order not to slow down the critical path.

In the first phase we make heavy use of the following ExactMul and FastTwoSum algorithms. In the absence of underflow and overflow, ExactMul rewrites the exact product of two finite binary64 numbers as a double-double value, that is, as the unevaluated sum of two finite binary64 numbers. The last instruction $\ell \leftarrow \circ(ab - h)$ of ExactMul is efficiently

Algorithm 1 (ExactMul)

Input: $a, b \in \mathbb{F}$
Output: h, ℓ such that $h + \ell = ab$
1: $h \leftarrow \circ(ab)$
2: $\ell \leftarrow \circ(ab - h)$

implemented using an FMA. If $ab \in \alpha\mathbb{Z}$ (that is, the exact product ab is an integer multiple of α) and if $|ab| \leq \Omega$, then $h + \ell = ab$ exactly.

FastTwoSum rewrites the exact sum of two finite binary64 numbers as a double-double value whose first component equals the rounded sum and whose second component is (an approximation to) the associated rounding error.

Algorithm 2 (FastTwoSum)

Input: $a, b \in \mathbb{F}$ with $a = 0$ or $|a| \geq |b|$
Output: h, ℓ such that $h + \ell$ approximates $a + b$
1: $h \leftarrow \circ(a + b)$
2: $t \leftarrow \circ(h - a)$
3: $\ell \leftarrow \circ(b - t)$

In the absence of underflow and overflow, it is well known that for rounding to nearest FastTwoSum is an error-free transform, that is, $h + \ell = a + b$ exactly, while for directed roundings we have in general only an approximation: $h + \ell \approx a + b$. It is shown in [22, Theorem 1] that the corresponding approximation error satisfies $|h + \ell - (a + b)| \leq 2^{-105}|h|$ and that it is zero when the exponent difference between a and b does not exceed 53. (See also [8] for a weaker bound.) Also, it is shown in [6, Theorem 3] that the middle computation $t \leftarrow \circ(h - a)$ is always exact, whatever the rounding mode.

Besides FastTwoSum, we shall also use the following FastSum algorithm, which allows us to add an extra error term to the low order term produced by FastTwoSum.

Algorithm 3 (FastSum)

Input: $a, b_h, b_\ell \in \mathbb{F}$ with $a = 0$ or $|a| \geq |b_h|$
Output: h, ℓ such that $h + \ell$ approximates $a + b_h + b_\ell$
1: $h, t \leftarrow \mathrm{FastTwoSum}(a, b_h)$
2: $\ell \leftarrow \circ(t + b_\ell)$

Lemma 1: In the absence of underflow and overflow, the pair (h, ℓ) computed by Algorithm FastSum satisfies

$$\left|h + \ell - (a + b_h + b_\ell)\right| \leq 2^{-105}|h| + \mathrm{ulp}(\ell).$$

Proof: The error of FastSum is bounded by the sum of the error of the FastTwoSum call, plus the rounding error in $t + b_\ell$. The first one is bounded by $2^{-105}|h|$ from [22], and the second one by $\mathrm{ulp}(\ell)$. ∎

$$\begin{aligned} P &= z - z^2/2 + \texttt{0x1.5555555555558p-2}\, z^3 \\ &- \texttt{0x1.0000000000003p-2}\, z^4 \\ &+ \texttt{0x1.999999981f535p-3}\, z^5 \\ &- \texttt{0x1.55555553d1eb4p-3}\, z^6 \\ &+ \texttt{0x1.2494526fd4a06p-3}\, z^7 \\ &- \texttt{0x1.0001f0c80e8cep-3}\, z^8 \end{aligned}$$

Fig. 1. Degree-8 polynomial $P(z)$ generated by Sollya [2] for approximating $\log(1 + z)$ when $|z| \leq 33 \cdot 2^{-13}$, with absolute error at most $2^{-81.63}$ and relative error at most $2^{-72.423}$.

Section III-A computes a double-double approximation $h+\ell$ of $\log x$, which is multiplied by y in Section III-B to obtain a double-double approximation $r_h + r_\ell$ of $y \log x$, and a double-double approximation of $\exp(r_h + r_\ell)$ is obtained in Section III-C. Finally Section III-D details how these three steps are used in the first phase.

A. Approximation of $\log x$

To approximate $\log x$ for $x \in [\alpha, \Omega]$ a binary64 number, after some argument reduction described below, we have to approximate $\log(1 + z)$ for some binary64 number z such that $|z| \leq 33 \cdot 2^{-13}$. For this task, we use Algorithm p_1 which computes a double-double approximation $p_h + p_\ell$ of $\log(1 + z) - z$, using a degree-8 polynomial (Fig. 1).

Algorithm 4 Algorithm p_1

Input: $z \in \mathbb{F}$
Output: $p_h + p_\ell$ approximating $\log(1 + z) - z$
1: $w_h, w_\ell \leftarrow \mathrm{ExactMul}(z, z)$
2: $t \leftarrow \circ(P_8 z + P_7)$
3: $u \leftarrow \circ(P_6 z + P_5)$
4: $v \leftarrow \circ(P_4 z + P_3)$
5: $u \leftarrow \circ(t w_h + u)$
6: $v \leftarrow \circ(u w_h + v)$
7: $u \leftarrow \circ(v w_h)$
8: $p_h \leftarrow -0.5 \cdot w_h$
9: $p_\ell \leftarrow \circ(uz - 0.5 \cdot w_\ell)$

Lemma 2: Given $|z| \leq 33 \cdot 2^{-13}$ with z an integer multiple of 2^{-61}, the double-double approximation $p_h + p_\ell$ to $\log(1+z) - z$ returned by Algorithm p_1 satisfies

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

with $|p_h| < 2^{-16.9}$ and $|p_\ell| < 2^{-25.446}$. If $z \neq 0$, and assuming further $|z| < 32 \cdot 2^{-13}$, the relative error satisfies

$$\left|\frac{z + p_h + p_\ell}{\log(1 + z)} - 1\right| < 2^{-67.441}.$$

Proof: See [9, Appendix A-A], and refinement via $|\delta_0| + |\delta_6| \leq 3.505\mathbf{u}$ for the relative error in the case $|z| < 32 \cdot 2^{-13}$. ∎

The approximation of $\log x$ is done using Algorithm 5. First, $x \in \mathbb{F} \cap [\alpha, \Omega]$ is written $2^e \cdot t$ with $e \in \mathbb{Z}$ and $t \in \mathbb{F} \cap (1/\sqrt{2}, \sqrt{2})$. Then, following [15], [21], we reduce the range of t even further by computing $z = \circ(rt-1)$, where r is a precomputed 9-bit approximation to $1/t$ obtained from $i = \lfloor 2^8 t \rfloor$, and denoted $\mathrm{INVERSE}_i$ below. (The value of r is explicitly defined in [9, Appendix A-B]; when $i \in \{255, 256\}$, we shall simply take $r = 1$ in order to avoid cancellation when approximating $\log z - \log r$.) Note that z can be produced efficiently using an FMA.

Lemma 3: For any $t \in \mathbb{F} \cap (1/\sqrt{2}, \sqrt{2})$, let $i = \lfloor 2^8 t \rfloor$ and $r = \mathrm{INVERSE}_i$. Then $z = \circ(rt-1)$ is exact, $|z| \le 33 \cdot 2^{-13}$, and $z \in 2^{-61}\mathbb{Z}$.

Proof: See [9, Appendix A-B]. ∎

In Lemma 3, it can be checked that the upper bound $33 \cdot 2^{-13}$ is optimal if we want $z = \circ(rt-1)$ to be exact, when using a table indexed by $\lfloor 2^8 t \rfloor$.

In Algorithm log_1, the table value LOGINV_i is a double-double approximation $\ell_1 + \ell_2$ to $-\log r$, such that $\ell_1 \in \mathbb{F}$ is an integer multiple of 2^{-42} nearest $-\log r$, and $\ell_2 \in \mathbb{F}$ is nearest $(-\log r) - \ell_1$. Similarly, LOG2H+LOG2L is a double-double approximation to $\log 2$, to nearest and with LOG2H an integer multiple of 2^{-42}.

Algorithm 5 Algorithm log_1

Input: a binary64 value $x \in [\alpha, \Omega]$
Output: $h + \ell$ approximating $\log x$

1: write $x = t \cdot 2^e$ with $t \in (1/\sqrt{2}, \sqrt{2})$ and $e \in \mathbb{Z}$
2: $i \leftarrow \lfloor 2^8 t \rfloor$ ▷ i integer
3: $r \leftarrow \mathrm{INVERSE}_i, \quad \ell_1, \ell_2 \leftarrow \mathrm{LOGINV}_i$
4: $z \leftarrow \circ(rt-1)$
5: $t_h \leftarrow \circ(e\,\mathrm{LOG2H} + \ell_1)$
6: $t_\ell \leftarrow \circ(e\,\mathrm{LOG2L} + \ell_2)$
7: $h, \ell \leftarrow \mathrm{FastSum}(t_h, z, t_\ell)$
8: $p_h, p_\ell \leftarrow \mathrm{p\_1}(z)$
9: $h, \ell \leftarrow \mathrm{FastSum}(h, p_h, \circ(\ell + p_\ell))$
10: **if** $e = 0$ **then** $h, \ell \leftarrow \mathrm{FastTwoSum}(h, \ell)$

Lemma 4: Given $x \in [\alpha, \Omega]$, Algorithm log_1 computes (h, ℓ) such that $|\ell| \le 2^{-23.89}|h|$, and

$$|h + \ell - \log x| \le \varepsilon_{\log} \cdot |\log x| \tag{1}$$

with $\varepsilon_{\log} = 2^{-73.527}$ if $x \notin (1/\sqrt{2}, \sqrt{2})$, and $\varepsilon_{\log} = 2^{-67.0544}$ otherwise.

Proof: If $x = 1$, then it is easy to see that $e = 0$, $t = 1$, $i = 256$ thus $r = 1$ (remember for $i \in \{255, 256\}$ we use $r = 1$ to avoid cancellation), $z = 0$, so that Algorithm p_1 returns $p_h = p_\ell = 0$ and Algorithm log_1 returns $h = \ell = 0$. The proof is thus finished in this case.

Let us now assume $x \neq 1$. Firstly, from the proof of Lemma 3, the value z computed at line 4 fulfills the conditions of Lemma 2. The main analysis of Algorithm log_1 is performed in [9, Appendix A-C], where we distinguish three cases:

1) case $e \neq 0$, that is, $x < \sqrt{2}/2$ or $\sqrt{2} < x$. This case is detailed in [9, Appendix A-D];
2) case $e = 0$ and $i \neq \{255, 256\}$, that is, $\sqrt{2}/2 < x < 255/256$ or $257/256 \le x < \sqrt{2}$. This case is detailed in [9, Appendix A-E];
3) case $e = 0$ and $i \in \{255, 256\}$, that is, $255/256 \le x < 257/256$. This case is detailed in [9, Appendix A-F].

∎

B. Multiplication by y

Once we have computed a double-double approximation $h + \ell$ to $\log x$, we multiply it by y in order to obtain an approximation to $y \log x$, as in Algorithm 6 below. As pointed out in [18, Chapter 12], what is important is to bound the *absolute* error in the approximation to $y \log x$.

Algorithm 6 Algorithm mul_1

Input: a double-double value $h + \ell$, and a double y
Output: $r_h + r_\ell$ approximating $y(h + \ell)$

1: $r_h, s \leftarrow \mathrm{ExactMul}(y, h)$
2: $r_\ell \leftarrow \circ(y\ell + s)$

Lemma 5: If x, h, ℓ are as in Lemma 4 and if the exact product yh satisfies $2^{-969} \le |yh| \le 709.7827$, then Algorithm 6 computes (r_h, r_ℓ) such that $|r_h| \in [2^{-970}, 709.79]$, $|r_\ell| \le 2^{-14.4187}$, $|r_\ell / r_h| \le 2^{-23.8899}$, $|r_h + r_\ell| \le 709.79$, and

$$|r_h + r_\ell - y \log x| \le \varepsilon_{\text{mul}}$$

with $\varepsilon_{\text{mul}} = 2^{-63.799}$ if $x \notin (1/\sqrt{2}, \sqrt{2})$, and $\varepsilon_{\text{mul}} = 2^{-57.580}$ otherwise.

Proof: See [9, Appendix A-G]. ∎

C. Final exponentiation

Finally we approximate $\exp(r_h + r_\ell)$. After some argument reduction described in Algorithm 8, we have to approximate $\exp(z)$ for z a double-double value with $|z| < 0.000130273$. (This bound comes from the paragraph "About the values z_h and z_ℓ" in [9, Appendix A-I].) We use a degree-4 polynomial (Fig. 2), that is evaluated using Algorithm q_1.

$$\begin{aligned} Q &= 1 + z + z^2/2 \\ &+ \texttt{0x1.5555555995d37p-3}\, z^3 \\ &+ \texttt{0x1.55555558489dcp-5}\, z^4 \end{aligned}$$

Fig. 2. Degree-4 polynomial $Q(z)$ generated by Sollya for approximating $\exp(z)$ when $|z| \le 0.000130273$, with absolute error at most $2^{-74.346}$.

Lemma 6: Given (z_h, z_ℓ) such that $|z_h + z_\ell| < 0.000130273$ and $|z_\ell| \le 2^{-42.7260}$, Algorithm q_1 returns (q_h, q_ℓ) such that

$$\left| \frac{q_h + q_\ell}{\exp(z_h + z_\ell)} - 1 \right| < 2^{-74.169053}$$

and $|\ell| \le 2^{-42.7096}$.

Proof: See [9, Appendix A-H]. ∎

Algorithm 7 Algorithm q_1

Input: a double-double value $z_h + z_\ell$
Output: $q_h + q_\ell$ approximating $\exp(z_h + z_\ell)$

1: $z \leftarrow \circ(z_h + z_\ell)$
2: $q \leftarrow \circ(Q_4 z_h + Q_3)$
3: $q \leftarrow \circ(qz + Q_2)$
4: $h_0, \ell_0 \leftarrow \mathrm{FastTwoSum}(Q_1, \circ(q \cdot z))$
5: $h_1, s \leftarrow \mathrm{ExactMul}(z_h, h_0)$
6: $t \leftarrow \circ(z_\ell h_0 + s)$
7: $\ell_1 \leftarrow \circ(z_h \ell_0 + t)$
8: $q_h, q_\ell \leftarrow \mathrm{FastSum}(Q_0, h_1, \ell_1)$

Algorithm 8 Algorithm exp_1

Input: a double-double value $r_h + r_\ell$
Output: $e_h + e_\ell$ approximating $\exp(r_h + r_\ell)$

1: $\rho_0 =$ `-0x1.74910ee4e8a27p+9` ≈ -745.133
2: $\rho_1 =$ `-0x1.577453f1799a6p+9` ≈ -686.909
3: $\rho_2 =$ `0x1.62e42e709a95bp+9` ≈ 709.78267
4: $\rho_3 =$ `0x1.62e4316ea5df9p+9` ≈ 709.78276
5: **if** $\rho_3 < r_h$ **then** return $e_h = e_\ell = \Omega$
6: **if** $r_h < \rho_0$ **then** return $e_h = \alpha$, $e_\ell = -\alpha$
7: **if** $r_h < \rho_1$ or $\rho_2 < r_h$ **then** return $e_h = e_\ell = \mathrm{NaN}$
8: INVLN2 $\leftarrow$ `0x1.71547652b82fep+12`
9: $k \leftarrow \lfloor \circ(r_h \cdot \mathrm{INVLN2}) \rceil$ ▷ nearest integer
10: LN2H $\leftarrow$ `0x1.62e42fefa39efp-13`
11: LN2L $\leftarrow$ `0x1.abc9e3b39803fp-68`
12: $k_h, k_\ell \leftarrow \mathrm{ExactMul}(k, \mathrm{LN2H})$
13: $k_\ell \leftarrow \circ(k \cdot \mathrm{LN2L} + k_\ell)$
14: $z_h, z_\ell \leftarrow \mathrm{FastSum}(\circ(r_h - k_h), r_\ell, -k_\ell)$
15: write $k = e \cdot 2^{12} + i_2 \cdot 2^6 + i_1$ with $0 \le i_2, i_1 < 2^6$
16: let $h_2 + \ell_2$ approximate $2^{i_2/64}$
17: let $h_1 + \ell_1$ approximate $2^{i_1/2^{12}}$
18: $p_h, s \leftarrow \mathrm{ExactMul}(h_1, h_2)$
19: $t \leftarrow \circ(\ell_1 h_2 + s)$
20: $p_\ell \leftarrow \circ(h_1 \ell_2 + t)$
21: $q_h, q_\ell \leftarrow \mathrm{q\_1}(z_h, z_\ell)$
22: $h, s \leftarrow \mathrm{ExactMul}(p_h, q_h)$
23: $t \leftarrow \circ(p_\ell q_h + s)$
24: $\ell \leftarrow \circ(p_h q_\ell + t)$
25: $e_h, e_\ell \leftarrow \circ(2^e \cdot h), \circ(2^e \cdot \ell)$

Lemma 7: In the case $\rho_1 \le r_h \le \rho_2$, if $|r_\ell / r_h| < 2^{-23.8899}$ and $|r_\ell| < 2^{-14.4187}$, then the value $e_h + e_\ell$ returned by Algorithm exp_1 satisfies

$$\left| \frac{e_h + e_\ell}{\exp(r_h + r_\ell)} - 1 \right| < 2^{-74.16}.$$

Moreover, $|e_\ell / e_h| \le 2^{-41.7}$.

Proof: Appendix A-J of [9] shows that for $\rho_1 \le r_h \le \rho_2$, both e_h, $2^e \cdot h$ and $2^e \cdot (h + \ell)$ lie in $[2^{-991}, \Omega]$.

Appendix A-I of [9] analyzes Algorithm exp_1 and proves the bound $|e_\ell / e_h| \le 2^{-41.7}$, while Appendix A-K of [9] establishes the relative error bound $2^{-74.16}$ for $e_h + e_\ell$. ■

D. Main algorithm

The main algorithm for the first phase is Algorithm 9 (phase_1), which calls in turn Algorithms log_1, mul_1, and exp_1 seen above. The notation $\mathrm{RU}(2^{-63.797})$ means the rounding upwards of the error bound $2^{-63.797}$; this value is precomputed or obtained at compile-time, and any value in $\mathbb{F}$ larger than $2^{-63.797}$ works. For the rounding test, we cannot use the algorithms from [5] which assume rounding to nearest-even, whereas our goal is to design algorithms working for all rounding modes. Instead, we compute a left bound u and a right bound v with the current rounding mode, and if both u and v round to the same value, x^y rounds to that value.

Algorithm 9 Algorithm phase_1

Input: two binary64 numbers x, y with $x > 0$
Output: the correct rounding of x^y, or FAIL

1: $\ell_h, \ell_\ell \leftarrow \mathrm{log\_1}(x)$
2: $r_h, r_\ell \leftarrow \mathrm{mul\_1}(\ell_h, \ell_\ell, y)$
3: $e_h, e_\ell \leftarrow \mathrm{exp\_1}(r_h, r_\ell)$
4: **if** $\sqrt{2}/2 < x < \sqrt{2}$ **then** $\varepsilon \leftarrow \mathrm{RU}(2^{-57.579})$
5: **else** $\varepsilon \leftarrow \mathrm{RU}(2^{-63.797})$
6: $u \leftarrow \circ(e_h + \circ(e_\ell - \varepsilon e_h))$
7: $v \leftarrow \circ(e_h + \circ(e_\ell + \varepsilon e_h))$
8: **if** $u = v$ **then** return u
9: **else** return FAIL

Theorem 1: The value returned by Algorithm phase_1 (if not FAIL) is correctly rounded.

Proof: If $r_h < \rho_1$, see [9, Appendix A-L]. If $\rho_2 < r_h$, see [9, Appendix A-M]. If $\rho_1 \le r_h \le \rho_2$, see [9, Appendix A-N]. ■

For the first phase, the following four tables are used, which occupy a total of 6416 bytes:

- a table INVERSE used in Algorithm log_1, with $r = \mathrm{INVERSE}_i$ a 9-bit approximation of $1/t$ for $i \cdot 2^{-8} \le t < (i+1) \cdot 2^{-8}$, $181 \le i \le 362$, such that $rt - 1$ is exactly representable in binary64. When the interval $[i \cdot 2^{-8}, (i+1) \cdot 2^{-8}]$ contains 1, that is, for $i \in \{255, 256\}$, we use $r = 1$ to avoid cancellation in the computation of $\log z - \log r$. This table has 182 binary64 entries, thus occupies 1456 bytes;
- a table LOGINV used in Algorithm log_1 such that for $181 \le i \le 362$, LOGINV_i is a double-double approximation to nearest of $-\log r$, where r is defined above. This table has 182 entries with two binary64 values, thus occupies 2912 bytes;
- a table T_2 such that for $0 \le i < 64$, $T_2[i]$ is a double-double approximation of $2^{i/2^6}$, used in line 16 of Algorithm exp_1. It has 64 entries with two binary64 values, thus occupies 1024 bytes;
- a table T_1 such that for $0 \le i < 64$, $T_1[i]$ is a double-double approximation of $2^{i/2^{12}}$, used in line 17 of Algorithm exp_1. It has 64 entries with two binary64 values, thus occupies 1024 bytes.

IV. Second and Third Phases

The second phase uses 128-bit arithmetic (using two 64-bit integer words for the significands), and delivers an approximation with about 113 correct bits. It is very similar to the first phase, except for the computation of $\log x$, where we use a two-step (instead of single-step) argument reduction $z = r_1 r_2 t - 1$ so that $|z| \le 2^{-13}$; then we use a degree-9 polynomial for $\log(1+z)$, another two-step argument reduction and a degree-7 polynomial for the computation of $\exp(r)$. When the rounding test of the second phase fails, exact and midpoint cases are detected using the algorithm from [14], where it is shown that all exact and midpoint cases belong to the following set:

$$\begin{aligned}\mathbb{S} &= \{(x,y) \in \mathbb{F}^2 \mid y \in \mathbb{N}, 2 \le y \le 35\} \\ &\cup \{(m, 2^F n) \in \mathbb{F}^2 \mid F \in \mathbb{Z}, -5 \le F < 0, \\ &\quad n \in 2\mathbb{N}+1, 3 \le n \le 35, m \in 2\mathbb{N}+1\}.\end{aligned}$$

We have computed hard-to-round cases for all pairs (x, y) from $\mathbb{S}$. Our relative error bound for the second phase is slightly worse than the bound 2^{-117} from [14, Algorithm 1], but no remaining worst case at that point defeats that algorithm. (Note that for the new IEEE mode 'to nearest with ties to away', our algorithms should be easy to adapt, since the only difference with 'to nearest-even' is how to round midpoint cases, thus it suffices to adapt the routine handling them.)

The third phase uses 256-bit arithmetic (using four 64-bit integer words for the significands), and delivers an approximation with about 240 correct bits. For the computation of $\log x$, we use the same two-step argument reduction as in the second phase, with a degree-18 polynomial for $\log(1+z)$, and another two-step argument reduction with a degree-14 polynomial for the computation of $\exp(r)$. Figure 3 summarizes the parameters of the three phases. In particular, the error bound of $2^{-63.797}$ of the first phase when $x \notin (1/\sqrt{2}, \sqrt{2})$ corresponds to a probability of about $1/1000$ of calling the second phase, thus to an average overhead of less than one cycle with respect to the first phase and corresponding rounding test.

Lauter estimates to 2^{112} the number of pairs (x, y) such that x^y lies in the binary64 range [13]. Thus a full search for hardest-to-round cases is not possible, even with clever algorithms like SLZ [20]. As a consequence, it is possible, albeit extremely unlikely, that the rounding test of the third phase fails. In such a case, an error message is printed, with the corresponding inputs x and y, which will enrich the knowledge of hard-to-round cases. This guarantees that whenever the function does not print this error message, the returned value is correctly rounded. If worst-case information was available, one could avoid the rounding test of the third phase, by adding some exceptional cases if needed. This would ensure the function never yields an error message and always returns a correctly-rounded result, but this would have little impact on its efficiency.

phase	error bound	throughput
1	$2^{-63.797}/2^{-57.579}$	63
2	$2^{-113.17}$	543
3	$2^{-240.44}$	1857

Fig. 3. Relative error bound and reciprocal throughput (in i7-8700 cycles) of the different phases. For the first phase, the error bound differs depending on whether $x \in (1/\sqrt{2}, \sqrt{2})$ or not. The second phase includes the exact/midpoint detection.

V. Correctness

We have tested the correctness of our implementation, which is publicly available at https://gitlab.inria.fr/zimmerma/core-math-power-b64, for all rounding modes, on a set of 917231 input pairs (x, y). This set includes: (a) worst cases for exponents y that might yield exact or midpoint cases, including cases where x^y lies in the subnormal range; (b) special values specified by IEEE 754-2019, for example $(x, y) = (1, -0)$ or $(\pm\infty, -0)$; (c) pairs with $x < 0$ and y an integer; (d) pairs with x near 1 and x^y near underflow/overflow; (e) exact and midpoint cases; (f) inputs exhibiting bugs in other libraries; (g) non-regression tests; (h) additional worst cases found in the literature, in particular from Section 14.14 in [1], and worst cases computed by Vincent Lefèvre for x^n and $x^{1/n}$, n integer. We also tested our implementation on a set of 10^8 random inputs pairs, again for all rounding modes. For all these tests, we used as reference the value given by GNU MPFR [7], with exponent range matching that of binary64, and emulating subnormals using `mpfr_subnormalize`.

VI. Efficiency

In this section, we measure the efficiency of our implementation in the C language of the algorithms described in this paper. We measure both the reciprocal throughput and the latency, using the CORE-MATH `perf.sh` tool [19]. For the power function, CORE-MATH generates random x and y uniformly in $[0, 20]$, so that x^y lies in $[0, 10^{26}]$ approximately. We compare our implementation to the GNU libc (which is not correctly rounded), to MathLib, CRLIBM, and CRLIBM, for rounding to nearest-even.

For MathLib, we used the non-official copy from https://github.com/dreal-deps/mathlib, which we compiled with the default compile flags (using `-O3 -march=native` yields plenty of incorrect roundings). Our tests confirm that the MathLib `upow` routine does not provide correct rounding for directed rounding modes.

For LIBMCR, we used the non-official copy from https://github.com/simonbyrne/libmcr/, which matches our copy of release 0.9 from 2004. For rounding to nearest-even, on our set of about one million tests, the pow function from LIBMCR does not terminate for 15 inputs pairs, for example $x =$ `0x1.470574d68e0afp+1` and $y =$ `0x1.02e0706205c0ep+1`, and we get about 96% of failures on the remaining tests, for example for $x =$ `0x1.f80b060553772p-1` and $y =$ `0x1.99cp+13`, LIBMCR yields `0x1.00001p+0`.

GNU libc 2.36	MathLib	LIBMCR	CRLIBM	this work
43	123	256	211	66
79	166	285	275	111

Fig. 4. Comparison of the reciprocal throughput (top) and latency (bottom) of some implementations of the binary64 power function, in terms of cpu cycles, on an Intel(R) Core(TM) i7-8700 with gcc 12.2.0, for rounding to nearest-even.

The power function from CRLIBM is explicitly said as experimental, and only works for rounding to nearest-even. We also noticed some issues, for example for $x =$ `0x1.524ebae943097p+1` and $y =$ `0x1.ep-2`, it returns -5 instead of `0x1.93bd0cd47eb5fp+0`. Looking at the code, it appears that the strange return value of -5 corresponds to a failure in the last rounding test, which is not treated (here, x^y has 68 identical bits after the round bit).

Figure 4 shows that our implementation is only about 50% slower than the GNU libc, which is not correctly rounded (about 40% for the latency). (On a i7-1260P, we get a reciprocal throughput of 35 cycles and a latency of 78 cycles, against 23 and 59 respectively for the GNU libc.) While Figure 4 only considers rounding to nearest-even, timings of our implementation are very similar for other rounding modes. For the reciprocal throughput, our implementation outperforms that of MathLib by 86%, that of CRLIBM by a factor larger than 3, and that of LIBMCR by a factor of almost 4.

VII. Conclusion

We have proposed algorithms for computing the power function x^y with correct rounding, for the binary64 floating-point format and the four main rounding modes (to nearest-even as well as the three directed roundings of IEEE 754), and up to the knowledge of hardest-to-round cases. Our approach makes heavy use of the FMA instruction (either in hardware or emulated in software), and leads to a very efficient implementation, which is 2x to 4x faster than the power functions from MathLib, CRLIBM, and LIBMCR, claiming correct rounding.

When designing these algorithms, our aim has been to ensure both correctness and high efficiency. Efficiency follows from a three-phase approach whose first phase handles most binary64 inputs (x, y) very fast thanks to carefully designed FMA-based double-double computations. Correctness is obtained by returning either the correctly-rounded value of x^y (for which detailed error analysis has been done for the first phase), or switch to the next (more accurate) phase. Our implementation handles all the special input cases (such as $x \leq 0$, x or y NaN or infinite, etc.) according to the current IEEE 754 specification [10, §9.2.1].

As can be seen from the material in [9], we have systematically based our algorithms and implementations on very detailed proofs, which allows us to carefully control the accuracy of each of the routines involved as well as to predict and handle the (im)possibility of underflow and overflow. Furthermore, in the unlikely case where we cannot conclude that the correct value is returned for x^y, returning instead an error message along with the corresponding input (x, y) makes it possible to usefully augment the list of known hardest-to-round cases for that function.

Although our current implementation is already very fast, we do not claim that it cannot be improved further for speed. We hope that our results will contribute to turn the current IEEE recommendation of correct rounding for the mathematical functions listed in [10, Table 9.1] into a requirement. Our approach also easily extends to other rounding modes (such as round to nearest with ties to away) and to other formats (such as double extended and binary128, or decimal formats). Finally, the high level of detail of our proofs will hopefully make it easier to develop formal proofs of our algorithms, and we are planning to investigate this in the near future as well.

Acknowledgements

The authors are grateful to the three anonymous referees for their useful feedback, to Laurence Rideau and Laurent Théry who found some small issues in the initial proof of Lemma 2 while trying to convert it into a formal proof, and to Vincenzo Innocente for his feedback at early stages of this research. The first author is partially funded by CERN, and the search for worst cases was performed using computer resources from CERN and Grid 5000.

References

[1] Beebe, N. H. F. *The Mathematical-Function Computation Handbook - Programming Using the MathCW Portable Software Library*. Springer, 2017.

[2] Chevillard, S., Joldes, M. M., and Lauter, C. Sollya: an environment for the development of numerical codes. In *Third International Congress on Mathematical Software - ICMS 2010* (Kobe, Japan, 2010), K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, Eds., vol. 6327 of *Lecture Notes in Computer Science*, Springer, pp. 28–31.

[3] Clark, N. W., and Cody, W. J. Self-contained exponentiation. In *AFIPS Conference Proceedings* (1969), vol. 35, ACM, pp. 701–706.

[4] Daramy-Loirat, C., Defour, D., de Dinechin, F., Gallet, M., Gast, N., Lauter, C., and Muller, J.-M. CR-LIBM: A library of correctly rounded elementary functions in double-precision. Research report, LIP, 2006. https://hal-ens-lyon.archives-ouvertes.fr/ensl-01529804.

[5] de Dinechin, F., Lauter, C., Muller, J.-M., and Torres, S. On Ziv's rounding test. *ACM Trans. Math. Softw. 39*, 4 (2013), 25:1–25:19.

[6] Demmel, J., and Nguyen, H. D. Fast reproducible floating-point summation. In *21st IEEE Symposium on Computer Arithmetic* (2013), pp. 163–172.

[7] Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., and Zimmermann, P. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw. 33*, 2 (2007), article 13.

[8] Graillat, S., and Jézéquel, F. Tight interval inclusions with compensated algorithms. *IEEE Transactions on Computers 69*, 12 (2020), 1774–1783.

[9] Hubrecht, T., Jeannerod, C.-P., and Zimmermann, P. Towards a correctly-rounded and fast power function in binary64 arithmetic. Extended version of this ARITH 2023 paper, with detailed proofs in appendix. Original version of July 12, 2023: https://inria.hal.science/hal-04159652v1. Most recent version: https://inria.hal.science/hal-04159652.

[10] IEEE. *IEEE Standard for Floating-Point Arithmetic (IEEE Std 754-2019)*. 2019.

[11] Innocente, V., and Zimmermann, P. Accuracy of mathematical functions in single, double, extended double and quadruple precision. https://members.loria.fr/PZimmermann/papers/accuracy.pdf, 2023. Version of February 14, 21 pages.

[12] ISO/IEC. C programming language – N3096, working draft of the standard, 2023. https://en.wikipedia.org/wiki/C2x.

[13] LAUTER, C. Q. *Arrondi correct de fonctions mathématiques. Fonctions univariées et bivariées, certification et automatisation*. PhD thesis, Université de Lyon - École Normale Supérieure de Lyon, 2008.

[14] LAUTER, C. Q., AND LEFÈVRE, V. An efficient rounding boundary test for $\mathrm{pow}(x, y)$ in double precision. *IEEE Trans. Comput. 58*, 2 (2009), 197–207.

[15] LE MAIRE, J., BRUNIE, N., DE DINECHIN, F., AND MULLER, J. Computing floating-point logarithms with fixed-point operations. In *23nd IEEE Symposium on Computer Arithmetic, ARITH 2016* (2016), P. Montuschi, M. J. Schulte, J. Hormigo, S. F. Oberman, and N. Revol, Eds., IEEE Computer Society, pp. 156–163.

[16] LIM, J. P., AND NAGARAKATTE, S. High performance correctly rounded math libraries for 32-bit floating point representations. In *PLDI'21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event* (2021), S. N. Freund and E. Yahav, Eds., ACM, pp. 359–374.

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

[18] MARKSTEIN, P. *IA-64 and Elementary Functions: Speed and Precision*. Prentice Hall, 2000. Hewlett-Packard Professional Books.

[19] SIBIDANOV, A., ZIMMERMANN, P., AND GLONDU, S. The CORE-MATH Project. In *ARITH 2022 - 29th IEEE Symposium on Computer Arithmetic* (virtual, France, 2022). https://hal.inria.fr/hal-03721525.

[20] STEHLÉ, D., LEFÈVRE, V., AND ZIMMERMANN, P. Searching worst cases of a one-variable function using lattice reduction. *IEEE Transactions on Computers 54*, 3 (2005), 340–346.

[21] TANG, P. T. P. Table-driven implementation of the logarithm function in IEEE floating-point arithmetic. *ACM Trans. Math. Softw. 16*, 4 (1990), 378–400.

[22] ZIMMERMANN, P. Note on FastTwoSum with Directed Roundings. Working paper or preprint, available at https://hal.inria.fr/hal-03798376, 2023.

[23] ZIV, A. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Trans. Math. Softw. 17*, 3 (1991), 410–423.