IEICE Proceeding Series

Accurate and Rigorous Logarithm Function Algorithm

Naoya Yamanaka, Shin'ichi Oishi

Vol. 1 pp. 820-823
Publication Date: 2014/03/17
Online ISSN: 2188-5079

2012 International Symposium on Nonlinear Theory and its Applications
NOLTA2012, Palma, Majorca, Spain, October 22-26, 2012

Accurate and Rigorous Logarithm Function Algorithm

Naoya Yamanaka† and Shin'ichi Oishi‡

†Research Institute for Science and Engineering, Waseda University
3-4-1 Okubo Shinjuku, Tokyo, 169-8555 Japan
‡Faculty of Science and Engineering, Waseda University
3-4-1 Okubo Shinjuku, Tokyo, 169-8555 Japan
Email: naoya_yamanaka@suou.waseda.jp, oishi@waseda.jp

Abstract—This paper is concerned with numerical algorithms retaining high reliability, high accuracy and high portability. In this paper, an algorithm with high reliability means a numerical algorithm which outputs a mathematically-rigorous result. An algorithm with high accuracy represents an algorithm which returns a result with high accuracy. Furthermore, an algorithm with high portability indicates an algorithm which calculates a result without relying on any numerical environment. In this paper, numerical logarithm algorithm retaining high reliability, high accuracy and high portability is discussed.

1. Introduction

Todays libraries for the approximation of elementary functions are very fast and the results are mostly of very high accuracy. For a good introduction and summary of state-of-art methods [6]. The achieved accuracy does not exceed one or two ulp[1] for almost all input arguments; however, there is no proof for that. In this paper, we discuss a reliable, accurate, portable implementation of the logarithm function. We intend to utilize the marvelous accuracy by a table approach. Proposed algorithm delivers rigorous bounds for the result for all floating-point input arguments. The order of evaluation of the formula is carefully chosen to diminish accumulation of rounding errors. As a result we obtain the relative accuracy of the logarithm function value is better than 0.5ulp.

Throughout this paper, we assume floating-point arithmetic adhering to IEEE standard 754-1985. IEEE standard 754-1985 is one of a technical standard established by the IEEE and the most widely-used standard for floating point computation, followed by many hardware and software implementations. Proposed algorithm is based on computer interval arithmetic adhering to the standard. Interval arithmetic is a numerical method developed by mathematicians since the 1950s and 1960s as an approach to putting bounds on rounding errors and measurement errors in mathematical computation and thus developing numerical methods that yield reliable results. It represents each value as a range of possibilities. The interval arithmetic on floating-point arithmetic with changing rounding mode is widely used. Rounding mode is one of defined parameter in IEEE standard 754-1985. The standard defines four rounding algorithms: rounding to nearest, round toward zero, round toward plus-infinity, round toward minus-infinity. However, changing rounding mode takes some computational costs, and the commands to change the mode vary widely depending on the numerical environment. Besides, numerical environments which do not have the commands to change the mode exist. For these problems, an algorithm only in rounding to nearest is proposed.

[1] ulp means "unit in last place", the relative error unit. In double format ulp= 2^{-52}.

2. Preliminary

2.1. Error Free Transformations

In this section, we briefly review some algorithms called *Error-Free Transformations*.

Throughout this paper, we assume floating-point arithmetic adhering to IEEE standard 754 double precision. Let $fl(\cdots)$ be the result of floating-point operations, where all operations inside parentheses are executed by ordinary floating-point arithmetic in rounding-to-nearest. We assume that neither overflow nor underflow occur. Furthermore, we use MATLAB-like programming description to describe algorithms.

First, we introduce the addition algorithm TwoSum. Knuth [4] presented **Algorithm 1** which transforms a pair (x, y) with $x, y \in \mathbb{F}$ into a new pair (a, b) with $a, b \in \mathbb{F}$ satisfying $x + y = a + b$ with $a = fl(x + y)$, $|b| \leq \mathrm{eps}\,|a|$. If

Algorithm 1 TwoSum

Error-free transformation of the sum of two floating-point numbers.[Knuth [4]]

function $[a, b] = \mathrm{TwoSum}(x, y)$
$a = fl(x + y)$
$c = fl(a - x)$
$b = fl((x - (a - c)) + (y - c))$
end

$|a| \leq |b|$ satisfies, **Algorithm 2** outputs the same results as TwoSum:

Algorithm 2 FastTwoSum

Error-free transformation of the sum of two floating-point numbers.[Knuth [4]]

function $[x, y] = \text{FastTwoSum}(a, b)$
$x = fl(a + b)$
$y = fl(b - (x - a))$
end

Next, we proceed to the dot product. We know a useful multiplication algorithm TwoProduct[5] , which transforms a pair (x, y) with $x, y \in \mathbb{F}$ into a new pair (a, b) with $a, b \in \mathbb{F}$ satisfying $x \cdot y = a + b$, $|b| \leq \text{eps}\,|a|$.

The multiplication routine needs to split the input arguments into two parts. For the number t given by $\text{eps} = 2^{-t}$, we define $s := \lceil t/2 \rceil$; in double precision we have $t = 53$ and $s = 27$. The following **Algorithm 3** by Dekker [5] splits a floating point number $x \in \mathbb{F}$ into two parts x, y, where both parts have at most $(s - 1)$ nonzero bits.

Algorithm 3 Split

Split algorithm splits a t-bits floating-point number $x \in \mathbb{F}$ into $x_h, x_t \in \mathbb{F}$ such that $x = x_h + x_t$.

function $[x_h, x_t] = \text{Split}(x)$
$c = fl\left((2^{\lceil t/2 \rceil} + 1) \cdot x\right)$
$x_h = fl(c - (c - x))$
$x_t = fl(x - x_h)$
end

Using **Algorithm 3**, the following multiplication routine by G.W. Veltkamp [5] can be formulated (**Algorithm 4**).

Algorithm 4 TwoProduct

Error-free transformation of the product of two floating-point numbers [Veltkamp [5]].

function $[a, b] = \text{TwoProduct}(x,\ y)$
$a = fl(x \cdot y)$
$[x_1, x_2] = \text{Split}(x)$
$[y_1, y_2] = \text{Split}(y)$
$b = fl(x_2 \cdot y_2 - (((a - x_1 \cdot y_1) - x_2 \cdot y_1) - x_1 \cdot y_2))$
end

2.2. Proposed Algorithm for Logarithm Function

In this paper, we consider the calculation of logarithm function for a floating point number. All floating number x can be rewritten as follows:

$$x = m \times 2^{exponent}.$$

Here, an integer *exponent* must be fixed in order to satisfy

$$\frac{1}{\sqrt{2}} < m \leq \sqrt{2}. \tag{1}$$

Now, we consider the calculation of $\log(m)$. Let floating point numbers k_i $(1 \leq i \leq n)$ be floating point numbers which are all zeros except the leading 5 bits. The idea of the proposed algorithm is based on the finding the floating numbers k_i $(1 \leq i \leq n)$ in order to satisfy

$$k_1 (1 + k_2)(1 + k_3) \cdots (1 + k_n)\, m \approx 1. \tag{2}$$

$\log(m)$ can be written as

$$\log(m) = \log(k_1 \cdots (1 + k_n)\, m) - \log(k_1 \cdots (1 + k_n))$$

thus,

$$\log(k_1 \cdots (1 + k_n)\, m) \approx 0$$
$$\log(k_1 \cdots (1 + k_n)) = \log(k_1) + \sum_{i=2}^{n} \log(1 + k_i)$$

hold. From these, we can see that if we can find k_i $(1 \leq i \leq n)$, the logarithm value of m can be written by the summation of the logarithm values of k_i.

Furthermore, the proposed algorithm is based on a famous technique called "Table-Based Methods" [6]. We calculate the accurate value of $\log(k_i)$ and make a table in advance. Using this technique, the proposed algorithm can access the accurate values so that the algorithm is fast and rigorous.

2.3. Division Algorithm for Double-double Arithmetic

In a numerical calculation sometimes we need higher-than double-precision floating-point arithmetic to allow us to be confident a result. One alternative is to rewrite the program to use a software package implementing arbitrary-precision extended floating-point arithmetic such as MPFR [1] or ARPREC [2], and try to choose a suitable precision. There are possibilities intermediate between the largest hardware floating-point format and the general arbitrary-precision software which combine a considerable amount of extra precision with a relatively speaking modest factor loss in speed. An alternative approach is to store numbers in a multiple-component format, where a number is expressed as unevaluated sums of ordinary floating-point words, each with its own significand and exponent. The multiple-digit approach can represent a much larger range of numbers, whereas the multiple-component approach has the advantage in speed. Sometimes merely doubling the number of bits in a double-floating-point fraction is enough, in which case arithmetic on double-double operands would suffice.

A double-double number is an unevaluated sum of two double precision numbers, capable of representing at least 106 bits of significand. A natural idea is to manipulate

such unevaluated sums. This is the underlying principle of double-double arithmetic. It consisted in representing a number x as the unevaluated sum of two basic precision floating-point numbers:

$$x = x_h + x_l \tag{3}$$

such that the significands of x_h and x_l do not overlap, which means here that

$$|x_l| \leq \mathrm{eps}\,|x_h|\,. \tag{4}$$

To calculate the floating numbers k_i ($1 \leq i \leq n$), we sometimes need *verified and accurate* division algorithm for double-double arithmetic. In this paper we proposed an division algorithm for double-double arithmetic instead of the widely-used algorithm developed by Hida *et. al.* and we proved the maximum error bound of the proposed division algorithm.

2.3.1. Hida et. al.'s Division Algorithm

Hida *et. al.* have proposed two division algorithms in their software QD/DD[3]: one is *accurate_div* and the other is *sloppy_div*. Using *accurate_div*, we can get an accurate result because the algorithm treats error carefully. The fact is that *accurate_div* uses two times of the multiplication algorithm for double-double arithmetic. *sloppy_div* is less accurate than *accurate_div*, but the computational speed of *sloppy_div* is faster than that of *accurate_div*. *sloppy_div* uses one-time multiplication algorithm for double-double arithmetic. The following **Algorithm 5** is *sloppy_div*.

Algorithm 5 *sloppy_div*

Division algorithm for double-double arithmetic based on double arithmetic [3].

function z = sloppy_div(x, y)
  $z_h = fl(x_h/y_h)$
  $[r_h, r_l]$ = mul(y, z_h)
  $[s_1, s_2]$ = TwoSum($x_h, -r_h$)
  $s_2 = fl(s_2 - r_l + x_l)$
  $z_l = fl((s_1 + s_2)/y_h)$
  $[z_h, z_l]$ = FastTwoSum(z_h, z_l)
end

Here, Algorithm *mul* is as follows (**Algorithm 6**):

2.3.2. Yamanaka and Oishi's Division Algorithm

Yamanaka and Oishi have proposed another division algorithm. As we have seen, because of using the multiplication algorithm for double-double arithmetic, it is difficult to speed up the computation time of *sloppy_div*. To construct an algorithm which is faster than *sloppy_div*, we've presented **Algorithm 7** which does not use the multiplication algorithm for double-double arithmetic.

Algorithm 6 *mul*

Multiplication algorithm for double-double arithmetic based on double arithmetic [3].

function $z = mul(x, y)$
  $[z_h, z_l]$ = TwoProduct(x_h, y_h)
  $z_l = fl(z_l + x_h \cdot y_l + x_l \cdot y_h)$
  $[z_h, z_l]$ = FastTwoSum(z_h, z_l)
end

Algorithm 7 Yamanaka and Oishi's Algorithm

Division algorithm for double-double arithmetic.

function z = modified_sloppy_div(x, y)
  $y_r = fl(1/y_r)$
  $z_h = fl(x_h \cdot y_r)$
  $[t_h, t_l]$ = TwoProduct(z_h, y_h)
  $z_l = fl(((x_h - t_h) - t_l) \cdot y_r + z_h \cdot (x_l/x_h - y_l \cdot y_r))$
  $[z_h, z_l]$ = FastTwoSum(z_h, z_l)
end

Algorithm 7 is based on the following approximation:

$$\frac{x_h + x_l}{y_h + y_l} = \frac{x_h(1+x_r)}{y_h(1+y_r)} = \frac{x_h}{y_h}(1+x_r)\left(1 - y_r + \frac{y_r^2}{1+y_r}\right) \tag{5}$$

$$\simeq \frac{x_h}{y_h} + \frac{x_h}{y_h}\left(x_r - y_r + O\left(\mathrm{eps}^2\right)\right), \tag{6}$$

where x_r, y_r denote

$$x_r = \frac{x_l}{x_h}, \qquad y_r = \frac{y_l}{y_h}. \tag{7}$$

It is easily seen that the proposed algorithm does not use *mul*, which is the multiplication algorithm for double-double arithmetic. Moreover, it also seen that the number of the floating point operation of the proposed algorithm is only 30 flops. By contrast, that of **Algorithm 5** is 36 flops.

2.3.3. Error bound of **Algorithm 7**

We now consider the problem of calculating basic operations of infimum-supremum double-double intervals in rounding to nearest; *i.e.* we are concerned with the problem of calculating an interval $Z = [\underline{z}, \overline{z}]$ with $\underline{z}, \overline{z} \in \mathbb{F}$ including $X \circ Y$ when double-double intervals X, Y are given. To solve this problem, it is obvious that the algorithms of basic operations of two floating point *numbers* are adequate from the definition of infimum-supremum interval operations:

$$X \circ Y := [\min(\underline{x}\circ\underline{y}, \underline{x}\circ\overline{y}, \overline{x}\circ\underline{y}, \overline{x}\circ\overline{y}),\ \max(\underline{x}\circ\underline{y}, \underline{x}\circ\overline{y}, \overline{x}\circ\underline{y}, \overline{x}\circ\overline{y})]. \tag{8}$$

Thus, we're considering the problem of calculating an interval Z containing $x \circ y$ for all double-double numbers x, y and $\circ \in \{+, -, \times, \div\}$ provided $fl(x \circ y)$ is finite.

To solve this problem, we present the following theorem. Now we consider the error of the division algorithm (**Algorithm 7**).

Theorem 1
Let x, y be double-double numbers consisting of x_h, x_l and y_h, y_l. Denote z be the result of **Algorithm 7**, *then*

$$\left| z - \frac{x}{y} \right| \leqq eps^2(22 + 45eps) \left| \frac{x_h}{y_h} \right|. \tag{9}$$

holds when we assume that no overflow and underflow occur.

2.4. Decomposition of A Floating Point Number

Let floating point numbers k_i $(1 \leq i \leq n)$ be floating point numbers which are all zeros except the leading 5 bits. Then for a floating point number f, we propose an algorithm to decompose of f as

$$f = t_1 (1 + t_2)(1 + t_3) \cdots (1 + t_n) + \delta. \tag{10}$$

Algorithm 8 Decomposition of a floating point number

```
function  t_{1:n} = Decomposition (f)
    s_1 = f;     t_1 = zero_5 (s_1) ; u_1 = s_1 − t_1;
    s_2 = u_1/t_1; t_2 = zero_5 (s_2) ; u_2 = s_2 − t_2;
    for  i = 3 : n
        s_i = modified_sloppy_div ([u_i, 0], [1, t_i]) ;
        t_i = zero_5 (s_i) ;
        u_i = s_i − t_i;
    end
end
```

Here "$\text{zero}_5(\cdot)$" means a function to cover zeros except the leading 5 bits. t_1 implies an approximation of f and t_i $(2 \leq i \leq n)$ involve approximations of relative errors of an approximate value by the decomposition using $t_1, t_2, \cdots, t_{i-1}$, and for them, $t_1 \geq t_2 \geq \cdots \geq t_n$ holds.

3. Numerical Results

In this section, we present the numerical experiments. These experiments have been done by a computer having Intel Core 2 Duo 2.13 GHz CPU with 4G Byte Memory. We use C++ language under the Mac OS X.

We compare some errors and computational costs of the following two kinds of outputs.

1. Comparison with a result of verified interval of double-double number

	theo. max err	time ratio
Proposed Method	8.0×10^{-30}	0.10
MPFR	1.2×10^{-32}	(1.0)

In this experiment, we've set the mantissa of MPFR as 106-bit.

2. Comparison with a result of verified interval of double number

We're set 10,000 random points of $[1, 100]$ as the inputted floating point numbers. The unit of errors in the following table is ulp.

	ave. err	max. err	theo. max err	time ratio
Prop.	0.251	0.499	0.501	4.6
CRlibm	0.500	0.500	0.500	(1.0)
Intlab	1.051	2.480	3.000	–*

* We could not make the program on C++ based on Rump's algorithm, so we could not compare the computational time of it.

Acknowledgments

This work was supported by Grant-in-Aid for Young Scientists (B) 24700015, and further, this paper is a part of the outcome of research performed under a Waseda University Grant for Special Research Projects (Project number: 2012A-508).

References

[1] L. Fousse, G. Hanrot, V. Lefévre, P. Pélissier, P. Zimmermann, "MPFR: A multiple-precision binary floating-point library with correct rounding", ACM Transactions on Mathematical Software (TOMS), 33:2, No.13. 2007.

[2] D. H. Bailey, Y. Hida, X. S. Li and B. Thompson. "ARPREC: an arbitrary precision computational package", Lawrence Berkeley National Laboratory. Berkeley. CA94720. 2002.

[3] Y. Hida, X. S. Li and D. H. Bailey. "Quad-Double Arithmetic: Algorithms, Implementation, and Application" October 30, 2000 Report LBL-46996.

[4] D. E. Knuth: The Art of Computer Programming: Seminumerical Algorithms, vol. 2, Addison-Wesley, Reading, Massachusetts, 1969.

[5] T. J. Dekker: A floating-point technique for extending the available precision, Numer. Math., 18 (1971), pp. 224–242.

[6] J.M.Muller, Elementary Functions, Birkhauser Boston, 2nd edition, 2006.

[7] S.M.Rump, Rigorous and portable standard functions. BIT Numerical Mathematics, 41(3), pp. 540–562, 2001.

[8] CRlibm – Correctly Rounded mathematical library: `http://lipforge.ens-lyon.fr/www/crlibm/`