

# A proven correctly rounded logarithm in double-precision

Florent de Dinechin, Catherine Loirat, Jean-Michel Muller

*LIP, projet Arénaire, École Normale Supérieure de Lyon,  
46 allée d'Italie, 69364 Lyon Cedex 07, France*

---

## Abstract

This article is a case study in the implementation of a proven, portable, and efficient correctly rounded elementary function in double-precision. We describe the methodology used in the implementation of the natural logarithm in the `cribm` library. The discipline required to prove a tight bound on the overall evaluation error allows to design a very efficient implementation with moderate effort.

*Key words:* arithmetic, floating-point, correct rounding, elementary functions, logarithm, libm

---

## 1 Introduction

### 1.1 Correct rounding and elementary functions

The need for accurate elementary functions is important in many critical programs. Methods for computing these functions include table-based methods[9,20], polynomial approximations and mixed methods[4]. See the books by Muller[19] or Markstein[16] for recent surveys on the subject.

The IEEE-754 standard for floating-point arithmetic[2] defines the usual floating-point formats (single and double precision) and specifies precisely the behavior of the basic operators  $+$ ,  $-$ ,  $\times$ ,  $\div$  and  $\sqrt{\phantom{x}}$ . The standard defines four rounding

---

*Email addresses:* Florent.de.Dinechin@ens-lyon.fr (Florent de Dinechin),  
Catherine.Loirat@ens-lyon.fr (Catherine Loirat),  
Jean-Michel.Muller@ens-lyon.fr (Jean-Michel Muller).

modes (to the nearest, towards  $+\infty$ , towards  $-\infty$  and towards 0) and demands that these operators return the correctly rounded result according to the selected rounding mode. Its adoption and widespread use have increased the numerical quality of, and confidence in floating-point code. In particular, it has improved *portability* of such code and allowed construction of *proofs* of numerical behavior. Directed rounding modes (towards  $+\infty$ ,  $-\infty$  and 0) are also the key to enable efficient *interval arithmetic*[17,11].

However, the IEEE-754 standard specifies nothing about elementary functions, which limits these advances to code excluding such functions. Without a standard, until recently, only two options were available:

- One can use mathematical libraries (`libm`) provided by operating systems, which are efficient but without any warranty on the correctness of the results.
- When strict guarantees are needed, some multiple-precision packages like `mpfr` [18] offer correct rounding in all rounding modes, but are several orders of magnitude slower than the `libm` for the same precision (see Table3).

## 1.2 The Table Maker's Dilemma and the onion peeling strategy

An IBM team lead by Ziv has recently released IBM Ultimate Math Library[15] or `libultim`, which is both correctly rounded to the nearest, and fast. The method used in this library [21] is the following.

With a few exceptions, the image  $\hat{y}$  of a floating-point number  $x$  by a transcendental function  $f$  is a transcendental number, and can therefore not be represented exactly in standard numeration systems. The purpose here is to compute the floating-point number that is closest to (resp. immediately above or immediately below) the mathematical value, which we call the result *correctly rounded* to the nearest (resp. towards  $+\infty$  or towards  $-\infty$ ).

A computer may evaluate an approximation  $y$  to the real number  $\hat{y}$  with precision  $\bar{\epsilon}$ . This ensures that the real value  $\hat{y}$  belongs to the interval  $[y(1 - \bar{\epsilon}), y(1 + \bar{\epsilon})]$ . Sometimes however, this information is not enough to decide correct rounding. For example, if  $[y(1 - \bar{\epsilon}), y(1 + \bar{\epsilon})]$  contains the middle of two consecutive floating-point numbers, it is impossible to decide which of these two numbers is the correctly rounded to the nearest of  $\hat{y}$ . This is known as the Table Maker's Dilemma (TMD) [19].

Ziv's technique is to improve the precision  $\bar{\epsilon}$  of the approximation until the correctly rounded value can be decided. Given a function  $f$  and an argument  $x$ , the value of  $f(x)$  is first evaluated using a quick approximation of precision  $\bar{\epsilon}_1$ . Knowing  $\bar{\epsilon}_1$ , it is possible to decide if rounding is possible, or if more

precision is required, in which case the computation is restarted using a slower approximation of precision  $\bar{\varepsilon}_2$  greater than  $\bar{\varepsilon}_1$ , and so on. This approach leads to good average performance, as the slower steps are rarely taken.

### 1.3 Drawbacks of Ziv's approach

However there was until recently no practical bound on the termination time of Ziv's iteration: It may be proven to terminate for most transcendental functions, but the actual maximal precision required in the worst case is unknown. In `libultim`, the measured worst-case execution time is indeed two orders of magnitude slower than that of usual `libms`. This might prevent using this method in critical application. A related problem is memory requirement, which is, for the same reason, unbounded in theory, and much higher than usual `libms` in practice.

Probably for this reason, Ziv's implementation doesn't provide a proof of the correct rounding property, and indeed some values are still uncorrectly rounded<sup>1</sup>.

Finally, this library still lacks the directed rounding modes, which might be the most useful: Indeed, correct rounding provides a precision improvement over usual `libms` of only a fraction of a unit in the last place (*ulp*) in round-to-nearest mode. This may be felt of little practical significance. However, the three other rounding modes are needed to guarantee intervals in interval arithmetic. Without correct rounding in these directed rounding modes, interval arithmetic loses up to one *ulp* of precision in each computation.

The goal of the `crlibm` project is therefore to design a mathematical library which is

- correctly rounded in the four IEEE-754 rounding modes,
- proven,
- and sufficiently efficient in terms of performance (both average and worst-case) and resources

to enable the standardization of correct rounding for elementary functions.

---

<sup>1</sup> Try `sin(0.252113814338773334355892075109295547008514404296875)`

## 2 The Correctly Rounded Mathematical Library

Our own library, called `crlibm` for *correctly rounded mathematical library*, is based on the work of Lefèvre and Muller [14,12] who computed the worst-case  $\bar{\epsilon}$  required for correctly rounding several functions in double-precision over selected intervals in the four IEEE-754 rounding modes. For example, they proved that 157 bits are enough to ensure correct rounding of the exponential function on all of its domain for the four IEEE-754 rounding modes, and 118 bits for the logarithm.

### 2.1 Two steps are enough

Thanks to such results, we are able to guarantee correct rounding in two iterations only, which we may then optimize separately. The first of these iterations is relatively fast and provides between 60 and 80 bits of accuracy (depending on the function), which is sufficient in most cases. It will be referred throughout this document as the *quick* phase of the algorithm. The second phase, referred to as the *accurate* phase, is dedicated to challenging cases. It is slower but has a reasonably bounded execution time, being – contrary to Ziv – tightly targeted at Lefèvre’s worst cases.

Having a proven worst-case execution time lifts the last obstacle to a generalization of correctly rounded transcendentals. Besides, having only two steps allows us to publish, along with each function, a proof of its correctly rounding behavior.

### 2.2 Portable IEEE-754 FP for fast first step

The computation of a tight bound on the approximation error of the first step ( $\bar{\epsilon}_1$ ) is crucial for the efficiency of the onion peeling strategy: overestimating  $\bar{\epsilon}_1$  means going more often than needed through the second step. As we want the proof to be portable as well as the code, our first steps are written in strict IEEE-754 arithmetic. On some systems, this means preventing the compiler/processor combination to use advanced floating-point features such as fused multiply-and-add (FMA) or extended double precision. It also means that the performance of our portable library will be lower than optimized libraries using these features.

To ease these proofs, our first steps make wide use of classical, well proven results like Sterbenz’ lemma [10]. When a result is needed in a precision higher than double precision (as is the case of  $y_1$ , the result of the first step), it is

represented as the sum of two floating-point numbers, also called a *double-double* number. There are well-known algorithms for computing on double-doubles [8].

### 2.3 Rounding test

At the end of the quick phase, a sequence of simple tests on  $y_l$  knowing  $\bar{\epsilon}_1$  allows to decide whether to go for the second step. We show here the rounding test theorem as an example, and because it is unpublished to our knowledge: The code sequence ubiquitous in IBM's `libultim`, but the way to compute the rounding constant is not explained. Defour published such an explanation in his thesis, but it is not as tight as ours: His test launches the accurate phase almost twice as often as actually needed.

#### Theorem 1 (Correct rounding of a double-double to the nearest double)

Let  $y$  be a real number, and  $\bar{\epsilon}$ ,  $e$ ,  $y_h$  and  $y_l$  be floating-point numbers such that

- $y_l < 2^{-53}y_h$  (which means that the mantissas of  $y_h$  and  $y_l$  do not overlap, or equivalently that the sum of  $y_h$  and  $y_l$ , rounded to the nearest, is equal to  $y_h$ ),
- none of  $y_h$  and  $y_l$  is a NaN.
- $|y_h| \geq 2^{-1022+53}$
- $|y_h + y_l - y| < \bar{\epsilon} \cdot |y|$
- $0 < \bar{\epsilon} \leq 2^{-53-k}$  with  $k \geq 2$  integer
- $e \geq 1 + \frac{2^{53+k+1}\bar{\epsilon}}{(2^k - 1)(1 - 2^{-53})}$

The following test (in C syntax, where `==` is the equality comparison operator) determines whether  $y_h$  is the correctly rounded value of  $y$  in round to nearest mode.

Listing 1: Test for rounding to the nearest

```

1 if (  $y_h == (y_h + (y_l * e))$  )
2   return  $y_h$ ;
3 else /* more accuracy is needed, launch accurate phase */

```

The proof is available in [3]. Note that the rounding test here depends on a constant  $e$  which is computed out of the overall relative error bound. Similar theorems are also given for directed rounding in [3], they depend directly on  $\bar{\epsilon}$ .

## 2.4 *Software Carry-Save for an accurate second step*

For the second step, we designed an ad-hoc multiple-precision library called Software Carry-Save library (*scslib*) which is lighter and faster than other available libraries for these specific precisions [6,5]. This choice is motivated by considerations of code size and performance, but also by the need to be independent of other libraries: Again, we need a library on which we may rely at the proof level. This library is included in `crlibm`, but also distributed separately [1]. The addition and multiplication in `scslib` offer a proven precision of 208 bits, a large overkill which facilitates the proof of the accurate phase.

## 2.5 *Tight and automated error computation*

The speed of an implementation of such a two-phase approach will be the average time of the quick phase, plus the average time of the accurate phase pondered by the percentage of calls to this accurate phase. This is the central performance tradeoff we have to handle: speed and accuracy of the first, quick phase will usually be antagonist, meaning that a faster quick phase will have a lower accuracy and launch the accurate phase more often, which will degrade average speed. On the other hand, a quick phase that is too accurate will also be too slow, and will impact the average speed, although the accurate phase is launched very rarely. For instance, the current exponential in `crlibm` has too accurate a quick phase [7].

Another problem raised here is that error analysis should be tight: if the computation is actually much more accurate than the proven error bound, it means that the slow accurate phase is launched more often than needed.

A design method crucial to the success of the implementation of efficient `crlibm` functions is therefore to have an implementation of the error analysis in Maple which is detailed enough to cover the whole of the computation, flexible enough to allow a quick navigation in the parameter space, and designed to output the values of all the constants appearing in the code (such as tabulated values, polynomial coefficients, constants used in various tests, and rounding constants) directly as a C include file. It also reduces the risk of an elusive error in a pencil-and-paper proof, and usually results in the tightest possible error bound. Of course, this Maple file becomes a part of the proof, and needs to be distributed along with the code.

Such an automated error analysis relies on a small set of robust general procedures which are described in [3]. For example, a very useful procedure computes a tight bound on the accumulated rounding errors of a polynomial whose coefficients are IEEE doubles, with a parameterized amount of double-double

arithmetic, with or without an error bound on the input. This procedure has been used extensively for the log and the trigonometric functions.

### 3 Overview of `crlibm`'s correctly rounded logarithm function

The worst-case accuracy required to compute a logarithm function correctly rounded in double precision is 118 bits according to Lefvre and Muller [13]. The first, quick phase for the log is accurate only to 57 – 64 bits, depending on the execution path as detailed below. If this is not enough to decide correct rounding, a second phase, accurate to  $2^{-120}$  using the SCS library is launched. In the quick phase, the core of the computation is shared by the three functions `log_rn`, `log_ru` and `log_rd` (the function `log_rz` calls either `log_ru` or `log_rd`). This computation is done by a procedure that returns an approximation to the log as two double-precision numbers, and also returns an index in an array of constants for testing if correct rounding is possible. This array contains the relative error for directed rounding modes, and the rounding constant computed as per Theorem 1 for round-to-nearest.

Special cases are handled as follows: The natural logarithm is defined over positive floating point numbers. If  $x \leq 0$ , then  $\log(x)$  should return *NaN*. If  $x = +\infty$ , then  $\log(x)$  should return  $+\infty$ . This is true in all rounding modes.

Concerning denormals, the smallest exponent of the logarithm of a double-precision input number is  $-53$  (for the input values  $\log(1 + 2^{-52})$  and  $\log(1 - 2^{-52})$ , as  $\log(1 + \varepsilon) \approx \varepsilon$  when  $\varepsilon \rightarrow 0$ ). This will make it easy to ensure that no denormal ever appears in the computation of the logarithm of a double-precision input number.

## 4 Quick phase of the evaluation

### 4.1 Overview of the algorithm

The algorithm consists of an argument reduction using the well-known property of the logarithm, and a polynomial evaluation using a degree 12 polynomial.

First we need to handle denormal inputs: If  $x < 2^{-1022}$ , ie if  $x$  is a subnormal number, then we use the equation

$$\log(x) = -52 \times \log(2) + \log\left(\frac{x}{2^{-52}}\right)$$

where  $\frac{x}{2^{-52}}$  is now a normalized number.

Argument reduction starts with the decomposition of  $x$  into its mantissa  $m$  and its exponent  $E$ , computed without error by bit manipulation. The mantissa is in  $[1, 2[$  and we prefer an interval centered around 1, so if needed the mantissa is divided by 2 and the exponent incremented – both exact operations as well – to get

$$x = 2^E \cdot y \quad (1)$$

where  $E$  is an integer, and  $y$  satisfies  $\frac{11}{16} < y < \frac{23}{16}$ .

The final reconstruction will use the equation

$$\log(x) = E \times \log(2) + \log(y) \quad . \quad (2)$$

The interval  $[\frac{11}{16}, \frac{23}{16}]$  being too large for a polynomial approximation of acceptable degree, it is broken down into 8 intervals given in Table 1. Note that the first four intervals are of size  $2^{-4}$ , while the last four are of size  $2^{-3}$ . The value of  $i$ , the index of the interval  $X[i]$  to which  $y$  belongs, will be computed out of a few bits of  $y$ .

Noting  $\text{middle}[i]$  the middle of the  $i$ -th interval, the final range reduction consists in computing  $z = y - \text{middle}[i]$ . Again this range reduction can be computed exactly thanks to Sterbenz lemma. On each interval, a polynomial  $P[i](z)$  approximates  $\log(y)$ . In the following  $P[i]$  will be noted  $P$  when no ambiguity arises.

Each polynomial has coefficients which are exactly representable as IEEE doubles, with the two first coefficients being exactly representable as the sum of two doubles:  $c_0 = c_0^{hi} + c_0^{lo}$  and  $c_1 = c_1^{hi} + c_1^{lo}$ . Their relative approximation error is also given in Table 1.

The polynomials are evaluated thanks to a Horner scheme:

$$P(z) = c_0^{hi} + c_0^{lo} + z \cdot (c_1^{hi} + c_1^{lo} + z \cdot (c_2 + z \cdot (c_3 + \dots + z \cdot (c_{11} + z \cdot (c_{11} + (c_{12} \cdot z))))))))))$$

where the two last iterations may use double-double arithmetic if required by the overall target accuracy, as detailed below.

The reconstruction computes in double-double arithmetic:

$$\log(x) \approx E \times \log(2) + P(z)$$

polynomial	interval	middle[ $i$ ]	$\max( z )$	$\log_2(\bar{\varepsilon}_{\text{approx}}[i])$
P[0]	$[\frac{11}{16}, \frac{12}{16}]$	$\frac{23}{32}$	$2^{-5}$	68.38
P[1]	$[\frac{12}{16}, \frac{13}{16}]$	$\frac{25}{32}$	$2^{-5}$	70.79
P[2]	$[\frac{13}{16}, \frac{14}{16}]$	$\frac{27}{32}$	$2^{-5}$	67.49
P[3]	$[\frac{14}{16}, \frac{15}{16}]$	$\frac{29}{32}$	$2^{-5}$	66.47
P[4]	$[\frac{15}{16}, \frac{17}{16}]$	1	$2^{-4}$	61.72
P[5]	$[\frac{17}{16}, \frac{19}{16}]$	$\frac{18}{16}$	$2^{-4}$	64.54
P[6]	$[\frac{19}{16}, \frac{21}{16}]$	$\frac{20}{16}$	$2^{-4}$	65.77
P[7]	$[\frac{21}{16}, \frac{23}{16}]$	$\frac{22}{16}$	$2^{-4}$	70.51

Table 1

The polynomials and their intervals

where  $P(z)$  has been computed by the previous step as the sum of two double-precision numbers.

The computation of  $E \times \log(2)$  as a double-double exploits the fact that  $E$  is a small integer: The constant  $\log(2)$  is stored as the sum of two double-precision numbers  $l_h + l_l = \log(2)(1 + \varepsilon_{-101})$ , with  $l_h$  having the 11 lower bits of its mantissa equal to zero, so that the computation of  $E \otimes l_h$  is exact. Now  $E \times \log(2)$  may be computed as  $\text{Fast2Sum}(E \otimes l_h, E \otimes l_l)$ . There  $\otimes$  denotes the machine FP multiplication operator, and the  $\text{Fast2Sum}$  procedure computes the exact sum of two double-precision numbers as a double-double. It is exact as well as the computation of  $E \otimes l_h$ , so the only errors are the representation error in  $l_l$  and the rounding error in  $E \otimes l_l$ , the sum of which amounts to a total relative error smaller than  $\bar{\varepsilon}_{\text{reconstruction}} = 2^{-100}$  for the computation of  $E \times \log(2)$ .

#### 4.2 Error analysis

Bounds on the polynomial approximation errors (both relative  $\bar{\varepsilon}_{\text{approx}}[i]$  and absolute  $\bar{\delta}_{\text{approx}}[i]$ ) are easily computed under Maple as infinite norms.

The rounding tests need a bound on the relative error, which can be computed as follows:

- If  $E \neq 0$ , then the result is computed as  $E \times \log(2) + P(z)$ . We compute a bound on the relative error by dividing the sum of all absolute errors in the approximation scheme by the minimum value of the result:

$$\bar{\varepsilon}_{\text{total}} = \frac{\bar{\delta}_{\text{approx}} + \bar{\delta}_{\text{rounding-poly}} + \bar{\delta}_{\text{reconstruction}}}{\min(|E \log(2) + P(x)|)}$$

Here,  $\bar{\delta}_{\text{reconstruction}} = \max(E \log(2)) \times \bar{\varepsilon}_{\text{reconstruction}}$  as computed previously,

and  $\bar{\delta}_{\text{approx}}$  and  $\bar{\delta}_{\text{rounding-poly}}$  can be computed by generic Maple procedures for several variants of the polynomial evaluation which are detailed below (this procedure also checks that no denormal ever appear in this evaluation, and similar conditions for the correct execution of the double-double operations in the code. It is detailed in [3]).

In this formula one sees several ways to handle the tradeoff between accuracy (which translates as percentage of cases where the slow, accurate phase will be needed) and performance of the quick phase:

- The absolute value of  $P(z)$  is bounded by 0.37. A bound on the denominator can therefore easily be deduced from the value of  $|E|$ . In other terms, all things being equal, there will be less chances to go to the accurate phase for larger  $E$ . It is therefore worth adding a test to the code which selects a rounding constant according to the value of  $|E|$ .

The “normal” execution path of our algorithm performs the Horner polynomial evaluation with the three last operations (two additions and one multiplication) performed in double-double arithmetic. This leads to two values of  $\bar{\varepsilon}_{\text{total}}$ :

If  $|E| \geq 24$  then we have  $\bar{\varepsilon}_{\text{total}} = 2^{-64.8}$ .

If  $0 < |E| < 24$  then we have  $\bar{\varepsilon}_{\text{total}} = 2^{-60.2}$ .

- For even larger values of  $|E|$ , we can afford to have a large  $\bar{\delta}_{\text{rounding-poly}}$  and still get an acceptable  $\bar{\varepsilon}_{\text{total}}$ . This allows us to evaluate the polynomial faster, without using double-double arithmetic. This situation will be referred to as the *fast path* in the following.

Actually, performing only the last Horner addition in double-double on the fast path turns out to improve the average performance: It slows down the quick phase only a little, and greatly reduces  $\bar{\delta}_{\text{rounding-poly}}$ , hence the chance of taking the slow, accurate phase. This in turn allows to take the fast path more often, *i.e* for more values of  $E$ .

- If  $E = 0$  then the reconstruction step is skipped, and the error  $\bar{\varepsilon}_{\text{total}}$  is given by the generic Maple procedures already mentionned.

It turns out that the polynomial  $P[5]$  (the one which is centered on 1 and has therefore a null coefficient of degree 0) leads to a relative rounding error which is much worse than that of the other polynomials. To still get an acceptable percentage of accurate phase in this case, and as the code contains a test if  $E = 0$  anyway, we chose to perform the polynomial evaluation in this case with one double-double multiplication more than in the normal case.

Actual values for this error analysis are summed up in Table 2. In the implementation, we take as rounding constant the worst case in each column. One remarks that the worst case for the three first columns is for  $P[5]$ , which would suggest yet another improvement: having specific rounding constants for  $i = 5$ . A closer look at the expected average improvement shows that the benefit will be minimal.

$P$	$\ P\ ^\infty$	$E = 0$	$1 \leq E < 24$	$24 \leq E < 186$	fast path
P[1]	0.375	61.10	61.26	66.93	63.30
P[2]	0.288	60.89	61.87	67.20	63.46
P[3]	0.208	60.47	62.35	67.43	63.39
P[4]	0.134	59.62	62.77	67.65	63.73
P[5]	0.065	57.68	60.16	64.88	62.91
P[6]	0.172	58.12	61.22	66.20	62.78
P[7]	0.272	59.93	61.22	66.50	62.94
P[8]	0.363	60.90	61.19	66.81	63.23

Table 2

Error analysis for the various path in the algorithm. The table gives  $-\log_2(\bar{\epsilon}_{\text{total}})$  in the different execution path.

## 5 Accurate phase

The first argument reduction is the same as in the quick phase. Therefore the functions of the accurate phase take as arguments  $y$  and  $E$ , computed in the quick phase (similarly, exceptional cases are not considered again). As in the quick phase, we will compute the log using  $\log(x) = E \times \log(2) + \log(1 + f)$ , however we perform a second table-based argument reduction [16]: We define  $w_i = 1 + i \times 2^{-4}$ , for  $i = -6 \dots 6$ , and we select the  $w_i$  closest to  $1 + f$ , in order to have:

$$\log(1 + f) = \log(w_i) + \log\left(1 + \frac{1 + f - w_i}{w_i}\right)$$

where  $R = \frac{1+f-w_i}{w_i} \leq 2^{-5}$ .

The values of  $\log(w_i)$ ,  $1/w_i$  and  $\log(2)$  are tabulated. We therefore compute the reduced argument  $R = (1 + f - w_i)/w_i$  by a subtraction in double precision (exact thanks to Sterbenz Lemma) followed by a conversion to SCS (also exact) and an SCS multiplication with a relative error smaller than  $2^{-207}$  including the error in tabulating  $1/w_i$ .

$\log\left(1 + \frac{1+f-w_i}{w_i}\right)$  is then approximated by a polynomial  $Q(R)$  of degree 20, with an overall relative error less than  $2^{-130}$ . We use the same Maple procedure as previously to compute the relative rounding error of this polynomial, the difference being that here we have a relative error of  $2^{-207}$  on the input due to range reduction.

The reconstruction step computes, all in SCS:

$$\mathbf{result} = E \times \log(2) + \log(w_i) + Q(R) \quad (3)$$

where each operation has a relative error smaller than  $2^{-207}$ , again including the errors in tabulating  $\log(w_i)$  and  $\log(2)$ . The overall error of the SCS evaluation is well below  $2^{-120}$  bits, ensuring correct rounding in all modes. The SCS value `result` is returned to the caller, where it is rounded to a double by the appropriate `scslib` function.

## 6 Analysis of the logarithm performance

The input numbers for the performance tests given here are random positive double-precision numbers with a normal distribution on the exponents. More precisely, we take random 63-bit integers and cast them into double-precision numbers.

In average, the second step is taken in 0.23% of the calls, which seems a rather good balance considering the respective costs of the first and second steps (seen in the table as the min and max times, respectively).

### 6.1 Speed

Table 3 (produced by the `crlibm_testperf` executable) gives absolute timings for a variety of processors and operating systems. The time units are arbitrary.

As expected, our `log` has the best worst-case execution time, with a factor 10 improvement over the other correctly rounded functions in `libultim` and `mpfr`. In average time, it is a few percent slower than `libultim`. This is the price to pay for having three rounding modes: If we inline the code of `log_quick`, then our `log` becomes faster in average than Ziv's. Inlining saves a function call, but also allows other minor improvements, like removing the need for the constant index, and loading the constants for the rounding test in advance to hide their loading time. However, as it increases the size of the code and degrades its readability, we believe the current approach makes more sense in `crlibm`.

On the PowerPC, the fused multiplied-and-add of the processor is used only to improve double-double multiplication operations [16]. The code of the `log` itself is unchanged.

Pentium 4 Xeon / Linux Debian sarge / gcc 3.3			
	min time	max time	avg time
libm (without correct rounding)	180	6612	193
mpfr	8	266808	46698
libultim	440	498976	521
crlibm	332	52392	528
PowerPC G4 / MacOS X / gcc2.95			
	min time	max time	avg time
libm (without correct rounding)	14	16	15
mpfr	4610	8620	4895
libultim	20	19890	22
crlibm (without FMA)	5	1241	32
crlibm (using FMA)	5	1144	24
Pentium III / Linux Debian woody / gcc 3.0			
	min time	max time	avg time
libm (without correct rounding)	184	699	184
mpfr	239	184801	51279
libultim	182	201400	293
crlibm	173	19404	319

Table 3  
Absolute timings for the logarithm (arbitrary units)

## 6.2 Memory requirements

Table size is

- for the quick phase,  $8 \times 15 \times 8 = 960$  bytes for the eight polynomials, plus another 64 bytes for the rounding constants, or a total of exactly 1kB.
- for the accurate phase,  $1 + 13 + 13 + 20$  SCS constants:  $\log(2)$ , the 13  $\log(w_i)$  and the 13  $\frac{1}{w_i}$ , the polynomial of degree 20 with a null first coefficient. This amounts to a little more than 2kB.

## 7 Conclusion and perspectives

In the log we have a fairly good balance between both evaluation phases, which was obtained thanks to automated testing of various scenarii. The main lesson learned here was that designing a rigorous proof along with the code helped tuning performance, by helping managing the tradeoff between speed and accuracy of the first, quick phase.

It might be interesting for some applications to add a fast execution paths for values around 1, as Ziv does: This is the situation where our quick phase is slowest and our second phase most often taken. However it will not show any improvement in our test protocol. Besides, it is probably more important to implement the standard  $\log(1+x)$  function, which is designed specifically for this case.

The accurate phase also has plenty of room for improvement: For instance the current version is 10 bits more accurate than needed, while `sclib` precision could also be degraded to 178 bits instead of the current 208.

Finally, we did not try to implement the argument reduction used in the accurate phase for the quick phase, mostly because it is not exact (contrary to the one we used), which complicates error analysis [16]. This is, however, another obvious path to explore.

The complete code with a more detailed proof, including the Maple programs mentioned above, is available from [3].

## References

- [1] SCS, Software Carry-Save multiprecision library.
- [2] ANSI/IEEE. Standard 754-1985 for binary floating-point arithmetic, 1985.
- [3] CR-LIBM, a library of correctly rounded elementary functions in double-precision. <http://lipforge.ens-lyon.fr/projects/crlibm/>.
- [4] Marc Daumas and Claire Moreau-Finot. Exponential: implementation trade-offs for hundred bit precision. In *Real Numbers and Computers*, pages 61–74, Dagstuhl, Germany, 2000.
- [5] F. de Dinechin and D. Defour. Software carry-save: A case study for instruction-level parallelism. In *Seventh International Conference on Parallel Computing Technologies*, Nizhny Novgorod, Russia, September 2003.
- [6] D. Defour and F. de Dinechin. Software carry-save for fast multiple-precision algorithms. In *35th International Congress of Mathematical Software*, Beijing, China, 2002. Updated version of LIP research report 2002-08.
- [7] D. Defour, F. de Dinechin, and J.M. Muller. Correctly rounded exponential function in double precision arithmetic. Technical Report RR2001-26, LIP, École Normale Supérieure de Lyon, July 2001. Available at <ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/RR/RR2001/RR2001-26.ps.Z>.
- [8] Theodorus J. Dekker. A floating point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.

- [9] P. M. Farmwald. High bandwidth evaluation of elementary functions. In K. S. Trivedi and D. E. Atkins, editors, *Proceedings of the 5th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society Press, Los Alamitos, CA, 1981.
- [10] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–47, March 1991.
- [11] R. Klatte, U. Kulisch, C. Lawo, M. Rauch, and A. Wiethoff. *C-XSC a C++ class library for extended scientific computing*. Springer Verlag, 1993.
- [12] V. Lefèvre. *Moyens arithmétiques pour un calcul fiable*. PhD thesis, École Normale Supérieure de Lyon, Lyon, France, 2000.
- [13] V. Lefèvre and J.-M. Muller. Worst cases for correct rounding of the elementary functions in double precision. <http://perso.ens-lyon.fr/jean-michel.muller/Intro-to-TMD.htm>, 2004.
- [14] V. Lefèvre, J.M. Muller, and A. Tisserand. Towards correctly rounded transcendentals. *IEEE Transactions on Computers*, 47(11):1235–1243, November 1998.
- [15] IBM Accurate Portable Math. Library. <http://oss.software.ibm.com/mathlib/>.
- [16] P. Markstein. *IA-64 and Elementary Functions : Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000. ISBN: 0130183482.
- [17] R.E. Moore. *Interval analysis*. Prentice Hall, 1966.
- [18] MPFR. <http://www.mpfr.org/>.
- [19] J.-M. Muller. *Elementary Functions, Algorithms and Implementation*. Birkhauser, Boston, 1997.
- [20] P. T. P. Tang. Table lookup algorithms for elementary functions and their error analysis. In P. Kornerup and D. W. Matula, editors, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 232–236, Grenoble, France, June 1991. IEEE Computer Society Press, Los Alamitos, CA.
- [21] A. Ziv. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software*, 17(3):410–423, September 1991.