MPFR: A Multiple-Precision Binary Floating-Point Library With Correct Rounding

Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, Paul Zimmermann
LORIA, 615 rue du jardin botanique, F-54602 Villers-lès-Nancy Cedex, France

This paper presents a multiple-precision binary floating-point library, written in the ISO C language, and based on the GNU MP library. Its particularity is to extend to arbitrary-precision ideas from the IEEE 754 standard, by providing *correct rounding* and *exceptions*. We demonstrate how these strong semantics are achieved — with no significant slowdown with respect to other arbitrary-precision tools — and discuss a few applications where such a library can be useful.

Categories and Subject Descriptors: D.3.0 [**Programming Languages**]: General—*Standards*; G.1.0 [**Numerical Analysis**]: General—*computer arithmetic, multiple precision arithmetic*; G.1.2 [**Numerical Analysis**]: Approximation—*elementary and special function approximation*; G 4 [**Mathematics of Computing**]: Mathematical Software—*algorithm design, efficiency, portability*

General Terms: algorithms, standardization, performance

Additional Key Words and Phrases: multiple-precision arithmetic, IEEE 754 standard, floating-point arithmetic, correct rounding, elementary function, portable software

Introduction and Motivation

The ANSI/IEEE 754-1985 standard for floating-point arithmetic (IEEE 754) has now become a common standard, even if some features like *gradual underflow* (i.e. subnormals) are still discussed [IEEE 1985]. An important consequence is that programs using the formats and operations specified by IEEE 754 have exactly the same behaviour on every configuration, as long as the processor, operating system and compiler are IEEE 754 compliant[1]. Another consequence is that researchers are able to design efficient algorithms using those formats and operations, and prove their correctness: interval arithmetic, floating-point expansions [Priest 1991], correctly-rounded elementary functions [Ziv 1991; The Arenaire project 2005; de Dinechin et al. 2005; Sun Microsystems 2004]. Thus, even if the IEEE 754 standard received several criticisms — both from hardware constructors who argued it would be too difficult to implement in a chip, and from software engineers who thought it would significantly slow down their programs — it is now accepted by everybody, and has enabled great progress in terms of correctness and portability of numerical software.

However, the IEEE 754 standard specifies fixed formats only, in particular single and double precision, with respectively 24 and 53 bits of mantissa. Several software tools exist for *multiple-precision* floating-point arithmetic, for example, MP [Brent 1978], GMP [Granlund 2004], CLN [Haible and Kreckel 2005], PARI/GP [Batut

[1]Assuming no extended precision is used internally, and the compiler does not over-optimize floating-point expressions.

et al. 2000], but they do not provide clear semantics, or only claim "almost always correctly rounded" results like the FM package [Smith 1991]. As pointed out by Ziv [Ziv 1991], although the accuracy provided by those packages is quite satisfactory, any slight change in the algorithm may produce changes in the output. Therefore, even an improvement in accuracy may have unwanted consequences for programs using those libraries.

This paper presents a library for multiple-precision floating-point arithmetic with such clear semantics, which extends IEEE 754. This library, called MPFR [Hanrot et al. 2005], is written in the C language on top of the GNU MP library (GMP) [Granlund 2004], and freely distributed under the GNU Lesser General Public License (LGPL). MPFR provides correct rounding for all the operations and mathematical functions it implements, with an efficiency comparable to other arbitrary-precision software — and even faster in most cases. As a consequence, applications using such a library inherit the same nice properties as programs using IEEE 754 — portability, well-defined semantics, possibility to design robust programs and prove their correctness — with no significant slowdown on average with respect to multiple-precision libraries with ill-defined semantics.

The paper is organized as follows. Section 1 presents existing software and related work. Section 2 describes the MPFR library, its user interface and its internals (data representation and algorithms). Section 3 presents the results obtained, in terms of efficiency, accuracy and portability, and compares them with other software. Finally, some companion tools and applications are discussed in Section 4.

1. EXISTING SOFTWARE AND RELATED WORK

As a floating-point package, MPFR only deals with real arguments. We thus restrict ourselves to software for *real numbers* here.

Several multiple-precision floating-point tools exist, so we do not aim at an exhaustive list here. We can mainly distinguish two classes of such software: libraries and interactive programs. The latter often provide other functionalities; this is the case for computer algebra systems like Maple [Char et al. 1991] or Mathematica [Wolfram 1996], or of number-theoretic programs like PARI/GP. Most floating-point libraries provide the four basic arithmetic operations $(+, -, \times, \div)$, but they differ in the underlying programming language (Fortran, C, C++), internal radix $(2, 2^{32}, 2^{64}, 10, 10^4, 10^9)$, the mathematical functions available, and the efficiency of the algorithms and/or implementation. For example, the MPF class from GMP [Granlund 2004] is quite efficient, but it provides basic arithmetic operations only. The PARI library, on top of which the GP program is written, implements several mathematical functions, including in the complex plane. The CLN library includes asymptotically fast algorithms for large numbers [Haible and Papanikolaou 1997].

However, these software do not implement correct rounding. Noticeable exceptions are Maple, which includes since version 6 an environment variable to control the rounding of basic arithmetic operations[2], the MpIeee class from Arithmos [Cuyt et al. 2001] which implements correct rounding in several possible radices; and NTL 5.4 [Shoup 2005], which guarantees correct rounding — to nearest only

[2]However some bugs still remain, for example, in versions 6 to 10 of Maple, `1.0 - 9e-5` gives `1.0` with a precision of 3 decimal digits and rounding towards zero, where the expected result is `0.999`.

— for the four basic operations and the square root, and "almost correct rounding" — platform independent — for other mathematical functions. In radix 10, the `decNumber` guarantees correct rounding for the four basic operations, the square root, and the integer power, and "almost correct rounding" for a few transcendental functions (exp, log and base-10 log) [Cowlishaw 2005].

2. THE MPFR LIBRARY

2.1 User Interface

The MPFR library is a smooth extension of the IEEE 754 standard [IEEE 1985], with radix 2. This choice of radix 2 follows from two requirements. For the sake of efficiency, we wanted to use the GMP `mpn` layer: this required a radix of the form 2^k. A natural idea would have been to take for k the word size in bits. Several libraries, in particular MPF, made this choice; however, it leads to two problems. Firstly, floating-point mantissae using an odd number of words on a 32-bit computer — for example, 32 or 96 bits — would have no equivalent on a 64-bit computer, which would lead to portability problems. Secondly, it would not be possible to emulate the IEEE 754 formats of respectively 24, 53 bits (nor, for the same reason, quadruple precision — 113 bits).

The main idea is that any floating-point number has its own precision in bits, which can vary from 2 to the largest possible precision with the available memory[3]. Consider a floating-point number x of precision p, another number y of precision q, and a rounding mode r. Let `mpfr_f` be the library function corresponding to a mathematical function f. The result of `mpfr_f (y, x, r)` is to put the value of

$$\mathrm{round}(f(x), q, r)$$

into the floating-point number y, which means that the exact result $f(x)$ is rounded to precision q according to the direction r.

As an example, let $x = 601 \cdot 2^{-10} = (0.1001011001)_2$; then the correct rounding of $\exp x$ with rounding to nearest and a target precision of 17 bits is $58931 \cdot 2^{-15} = (1.1100110001100110)_2$.

As any arithmetic following the IEEE 754 standard, each function input is considered as exact by MPFR. In other words, correct rounding is provided for *atomic* operations only; no information is kept about the "accuracy" of intermediate results. Thus, for any sequence of operations, it is the user's responsibility to compute the corresponding error bounds. This work is simplified by the fact that each atomic operation yields rigorous error bounds. Some methods exist to solve (semi-)automatically this problem, for example interval arithmetic, the Real RAM model, or significance arithmetic, cf. the corresponding implementations in MPFI [Revol and Rouillier 2005], IRRAM [Müller 1997] and Mathematica [Sofroniou and Spaletta 2005] respectively.

Each MPFR function returns a ternary value, called the "inexact flag", which indicates the rounding direction with respect to the exact value: the inexact flag is negative (resp. positive, zero) when the rounded output is smaller than (resp. larger than, equal to) the exact value. This information is useful for some applications.

[3]With a precision of 1 bit, the *round-even* rule is not sufficient to completely define the rounding of the binary value $(0.11)_2$, since both surrounding numbers 1.0 and 0.1 have an odd mantissa!

2.2 Data Representation

The internal data representation used by MPFR is the following. A floating-point number x is represented by a mantissa m, a sign s and a signed[4] exponent e. If the (binary) precision of x is p, the mantissa m has p significant bits. Special numbers like NaN, infinities or zeroes have a special representation. The mantissa m is represented by an array of GMP "limbs" (an unsigned machine-integer type), and is interpreted as $\frac{1}{2} \leq m < 1$. The most significant bit of the mantissa is always 1: MPFR does not allow subnormal numbers, and does not use an implicit bit. The most significant bit of the mantissa corresponds to the most significant bit of the most significant limb; in other words, when the precision is not a multiple of the number of bits per word, the unused bits are in the least significant limb, and they are always zero. For example, the mantissa of the 17-bit number $(1.1100110001100110)_2$ would be stored on a 5-bit computer as follows (with the most significant limb written on the left, and in each limb, the most significant bit on the left):

$$\begin{array}{cccc} \boxed{11100} & \boxed{11000} & \boxed{11001} & \boxed{10\mathit{000}} \\ \text{limb 3} & \text{limb 2} & \text{limb 1} & \text{limb 0} \end{array}$$

The *unit in last place* (or simply ulp) of a non-zero floating-point number is the weight of its last mantissa bit; for example, $\mathrm{ulp}(1) = 2^{1-p}$ for a p-bit mantissa.

As mentioned in [Hull 1978], it is important that a 17-bit number is represented by *exactly* 17 bits, and not at least 17 bits; indeed, if computations with requested precisions of 17 and say 32 bits give similar results, the effect of the precision on roundoff errors cannot be measured.

2.3 Semantics

The semantics chosen in MPFR is the following: for each assignment $a \leftarrow b \diamond c$ or $a \leftarrow f(b, c)$, the variables a, b, c may all have different precisions; the inputs b, c are considered to their full precision, and a correct rounding to the full target precision of a is computed. This semantics is less restrictive than other models: [Brent 1981a] uses a global precision, and first rounds the operands to that precision before performing the operation; on the contrary, [Hull 1978] defines a precision for each variable and computations are performed in blocks with (possibly) different precisions. If f_t represents the function f rounded to precision t, the MPFR semantics is:

$$a \leftarrow f_{t_a}(b, c),$$

where t_a is the precision of the variable a; the semantics from [Brent 1981a] is:

$$a \leftarrow f_t(\circ_t(b), \circ_t(c)),$$

where $\circ_t$ denotes rounding to the global precision t, and that from [Hull 1978] is:

$$a \leftarrow f_t(b, c),$$

[4]The exponent is represented by a machine word; as mentioned in [Brent 1981a], this limit is quite reasonable. Note that with an arbitrary-precision exponent, no underflow or overflow can occur.

where t is the precision of the current block. Both semantics from [Brent 1981a] and [Hull 1978] can be emulated by MPFR, either by first applying the `mpfr_set` function to round the inputs b and c to the target precision, or by first setting the precision of a to the block precision.

Yet, this semantics is not the most general possible. The radix is fixed to 2 for reasons of efficiency. Unlike [Hull 1978], in MPFR 2.2.0 there is no special value to distinguish unassigned variables from NaNs. Note that a higher-level language may extend or restrict the semantics. One may for example imagine that in C++, for an assignment `a = f(b)`, b is first rounded to the precision of a. Alternatively, a computer algebra system may define a different exponent range for each variable, as in [Hull 1978].

2.4 Basic Operations

We call "basic operations" those for which it is possible to directly compute the correct rounding, in contrast to other functions where Ziv's strategy has to be used (see next section). Among those basic operations are the four arithmetic operations (addition, subtraction, multiplication, division) and the square root. These operations admit a native implementation using the GMP `mpn` layer; for example, the addition is described in full detail in [Lefèvre 2004].

The multiplication of two n-bit numbers with a n-bit result is performed by a "short product", either using a naive algorithm [Krandick and Johnson 1993] or Mulders' algorithm [Mulders 2000], which gives a speedup up to about 20% with respect to a full product in the Karatsuba range. MPFR does not use a cutoff point of the form $\lfloor \beta n \rceil$ as in Mulders' original algorithm, but instead the optimal cutoff is determined by a tuning program, up to some limit (1024 words for example). Above that limit, a simple formula is used (e.g. $2n/3$ words, where n is the output size). This allows us to get close to optimal behaviour for the target processor: Mulders' algorithm is used up from 17 words on a Pentium 4, 8 words on an Opteron, 19 words on an Athlon, 11 words on a Pentium 3, 10 words on an Itanium 1. When the inputs have a much larger precision than the output, they are first truncated (of course, one checks at the end if the correct rounding is guaranteed, otherwise the full multiplication is performed). Indeed, if a short product of size m of two operands is computed (for instance by Mulders' algorithm), the relative error is at most

$$4 \sum_{k=m}^{\infty} (k+1)\beta^{-k},$$

where $\beta = 2^{32}$ or 2^{64} according to the processor type. For $m+4 \leq \beta$, this expression is bounded by $4(m+2)\beta^{-m}$; this means that in the practical case where the output size n is much smaller than β, taking $m = n + 1$ gives good chances that we will be able to round, since the computed value is within $\approx n/\beta$ ulps from the correct result.

The division and square root use the corresponding integer functions `mpn_divrem` and `mpn_sqrtrem` from the GMP `mpn` layer. As a consequence, the remainder is always exactly known, which enables one to compute the correct rounding.

For each basic operation, several auxiliary functions are available when one of the operands is of another type: `mpfr_add_ui` for an `unsigned long`, `mpfr_add_si` for an `signed long`, `mpfr_add_z` for a GMP multiple-precision integer.

Among those basic operations, the fused-multiply add, i.e. $xy+z$, is also provided. It is quite easy to implement in arbitrary-precision software: firstly compute $t := xy$ with t having a large enough precision so that the product is exact, then call the addition routine to correctly round $t + z$.

The efficiency of the basic operations is merely that of the corresponding routines from the GMP `mpn` layer. These routines use different algorithms depending on the number size; for example, the `mpn_mul_n` routine calls either the schoolbook method, Karatsuba's algorithm, Toom-Cook 3-way, or an FFT-based algorithm, with thresholds tuned for the target processor.

2.5 Advanced Functions

Release 2.2.0 of MPFR implements all mathematical functions from the ISO C99 standard mathematical library — except `modf` to extract integer and fractional parts, and `fmod` and `remainder` for floating-point remainder — and 4 mathematical constants ($\log 2$, π, Euler's and Catalan's constants).

The definition of these functions at special points (NaN, infinities) and the choice of the sign of the zero results are done according to Section F.9 of the ISO C99 standard for functions defined in this standard, and according to continuity rules [Defour et al. 2004] more generally.

Those functions for which a direct implementation is not possible are implemented using Ziv's strategy:

(1) treat special input values (NaN, infinities, zeroes), and values outside the function domain, for example, acos(2);
(2) treat inputs that clearly give an underflow or overflow, for example, $\exp(2^{99})$;
(3) treat inputs x such that $f(x)$ is exactly representable, for example, log(1);
(4) choose a working precision w slightly larger than the target precision p;
(5) compute an approximation y to $f(x)$ in precision w, together with a bound ϵ for the corresponding error;
(6) if $\mathrm{round}(y - \epsilon, p) = \mathrm{round}(y + \epsilon, p)$, return that common value;
(7) otherwise, increase w and go to step 5. In version 2.2.0 of MPFR, w is by default increased by the number of bits in a limb at the first iteration, and $w/2$ at the next iterations. A discussion about this choice in given in appendix.

This approach requires to be able to identify all inputs that give an output which is exactly representable in step (3), otherwise Ziv's strategy will not terminate[5]. It also requires that the error bound ϵ is rigorous: both the mathematical error — for example, when truncating a Taylor series — and the roundoff error should be taken into account. However, the error bound does not need to be tight; it is sufficient that it converges to zero when the working precision w increases to infinity.

In practice the bound ϵ at step (5) is rounded to $2^k \,\mathrm{ulp}(y)$, which makes step (6) easier; indeed, it suffices to check whether adding or subtracting $2^k \,\mathrm{ulp}(y)$ to y

[5]Indeed, assume for example $f(x) = 1$, the rounding mode is towards zero, and we get the approximation $y = 1$; then even if $\epsilon > 0$ is very small, we cannot decide whether $f(x)$ rounds to 1 or to $1 - 2^{-p}$.

changes the rounded value, which reduces to search for runs of consecutive zeroes or ones between the round bit of y — for the target precision p — and the bit position corresponding to $2^k \operatorname{ulp}(y)$.

For example, we describe what happens in MPFR when one asks for $\cos x$ for $x = 1$ with a target precision of 42 bits, and rounding to nearest. The working precision is set to 63 bits. In a first step (argument reduction), one computes $x' = x/2^5$; then one computes an approximation to the Taylor expansion of $\cos x'$ up to order 6:

$$s = (0.111111111111000000000000001010101010101001001111101001010110000 11)_2.$$

Then the reconstruction performs 5 times $s \leftarrow 2s^2 - 1$, with result:

$$y = (0.\underbrace{100010100101000101000000011111011010100000}_{42}110100\underline{0}10100100110100)_2.$$

The error bound ϵ corresponds to 2^{14} ulps (the underlined bit), and we see that $\operatorname{round}(y - \epsilon, p) = \operatorname{round}(y + \epsilon, p)$, so the correct rounding to nearest is:

$$0.100010100101000101000000011111011010100001.$$

Input and output functions, i.e. base conversion functions, also implement correct rounding using Ziv's strategy, in the whole range of values supported by the library. For example, the 53-bit binary number

$$x = 6965949469487146 \cdot 2^{-249}$$

is correctly rounded to $0.77003665618896 \cdot 10^{-59}$ with 14 digits and rounding to $+\infty$. Previous work shows that this problem is difficult, even in fixed precision [Clinger 1990; Gay 1990; Steele and White 1990]. For example, IEEE 754 requires correct rounding for double precision for numbers in the range $[10^{-27}, 10^{44}]$ only, and allows an error of up to 0.97 ulps outside this range (for rounding to nearest). Steele and White say in their retrospective [Steele and White 2003]:

> *[. . .] can one derive, without exhaustive testing, the necessary amount of extra precision solely as a function of the precision and exponent range of a floating-point format? This problem is still open, and appears to be very hard.*

Indeed, this is related to the closest convergent p/q of a ratio $2^e/10^f$ for e, f in the exponent range, and as noted in [Hack 2004], to the size of the following partial quotient.

The implementation of advanced functions also relies on different algorithms depending on the target precision. For example, the exponential uses a naive series evaluation for small precision, then a baby-step/giant-step evaluation — called "concurrent series" in [Smith 1991] —, and finally a binary splitting method for huge operands. As for basic operations, the corresponding thresholds are optimized by a tuning program.

2.6 Exceptions

MPFR supports exceptions similar to those of the IEEE 754 standard: inexact, overflow, underflow, invalid operation (i.e., functions that return a NaN), but no exceptions yet for division by zero[6]. When an exception occurs, a global flag is set; it is sticky, i.e., it remains set as long as the user does not clear it explicitly. Unlike the IEEE 754 standard, MPFR does not provide trap mechanisms (this could be a future extension). It does not have subnormals either, but as the default exponent range is very large, subnormals are not very useful. However, in the case subnormals are necessary (e.g., for full IEEE 754 emulation), a special function `mpfr_subnormalize` can be used to generate them.

Care has been taken in the MPFR code to handle exceptions correctly. For instance, we avoid overflows on C integer types, and concerning the global flags, their state is saved before the internal computations (which can generate exceptions that the user must not see) and restored afterwards.

2.7 Testing

Testing is a major issue, especially for a library claiming correct rounding in arbitrary precision [Verdonk et al. 2001]. Checking that the results given by MPFR are correctly-rounded is quite a challenge, since except Arithmos, Maple, NTL and decNumber — the last three only for $+, -, \times, \div, \sqrt{\cdot}$ —, no other software can compute, and thus check, a correct rounding. As a consequence, we used standard software engineering testing strategies: internal consistency checks such as $\sqrt{x^2} = |x|$ for rounding to nearest, or $-1 \leq \frac{x}{\sqrt{x^2+y^2}} \leq 1$ for rounding to nearest [Kahan 1996] or toward $+\infty$, comparison with known or computed values in fixed precision, comparison with hard-to-round cases in fixed precision, comparison for random inputs evaluated with different target precisions... We also used some known properties of some operations, for example, the FastTwoSum property: If $|x| \geq |y|$, $u = \circ(x-y)$, $v = \circ(u-x)$, $w = \circ(v+y)$, then $x - y = u - w$ exactly (where $\circ$ denotes rounding to nearest). So the “inexact flag” of $u = \circ(x-y)$ should be coherent with the sign of w.

We also tried to construct test cases covering all the nasty parts of the source code of each function, in particular underflow and overflow, special values for input and output (± 0, $\pm\infty$, NaN), checking the inexact flag for exact results... Such a search is sometimes difficult; however in some cases, it allowed to simplify the code, by discovering that some branches could not be visited.

3. RESULTS

In this section we compare MPFR and other libraries concerning the following properties: efficiency, accuracy and portability.

3.1 Efficiency

Table I compares MPFR, CLN, PARI, NTL, all configured to use GMP-4.1.4. Those timings show that MPFR is quite efficient compared to other libraries, except for acos and atan where faster algorithms have still to be implemented.

[6]There is no difficulty here, however it is not clear if the “division by zero” exception should be extended to other functions that return an exact infinite value from finite values, like $\log 0$. We expect the revision of IEEE 754 will solve that issue.

operation	digits	MPFR	CLN	PARI	NTL
		2.2.0	1.1.11	2.2.12-beta	5.4
$x \times y$	10^2	**0.00048**	0.00071	0.00056	0.00079
	10^4	**0.48**	0.81	0.58	0.57
x/y	10^2	**0.0010**	0.0013	0.0011	0.0020
	10^4	**1.2**	2.4	**1.2**	**1.2**
$\sqrt{x}$	10^2	**0.0014**	0.0016	0.0015	0.0037
	10^4	**0.81**	1.58	0.82	1.23
$\exp x$	10^2	**0.017**	0.060	0.032	0.140
	10^4	**54**	70	68	1740
$\log x$	10^2	**0.031**	0.076	0.037	0.772
	10^4	**34**	79	40	17940
$\sin x$	10^2	**0.022**	0.056	0.032	0.155
	10^4	**78**	129	134	1860
$\cos x$	10^2	**0.017**	0.050	0.029	0.164
	10^4	**77**	123	133	8530
$\operatorname{acos} x$	10^2	0.32	***0.076***	0.085	NA
	10^4	720	*154*	**153**	NA
$\operatorname{atan} x$	10^2	0.28	**0.067**	0.076	NA
	10^4	610	**149**	151	NA

Table I. Timings in milliseconds for several operations on a 1.8 GHz Athlon under Linux (`laurent5.medicis.polytechnique.fr`). The inputs correspond to $x = \sqrt{3} - 1, y = \sqrt{5}$. Boldface values indicate the faster timings for a given function and precision, and italics the use of a non-standard function: CLN only provides a complex acos function. "NA" means that the corresponding function is not available.

3.2 Accuracy

For each of the CLN, PARI and NTL libraries, several functions f, and a precision of 53 bits, we have made the following experiment (Tab. II). For some random input x, let z be the value computed by the corresponding library[7]. We compared the ulp error between z — or its rounded value in the case of PARI — and $f(x)$, where $f(x)$ was computed with increased precision; this ulp error is given with four significant digits, and rounding away from zero. For rounding to nearest that ulp error should not exceed 0.5 in absolute value for a correct rounding.

Note: in case of argument reduction, PARI does not increase the internal working precision to guarantee the result accuracy, thus large errors of more than 10^4 ulps can be obtained, for example $\sin x$ for $x \approx 863.93798795269947$. For CLN, the symmetry is sometimes not respected; for example for $x = 0.83070210528807542$, we have $\sinh(-x) \neq -\sinh(x)$.

3.3 Portability

Since MPFR is built on top of GMP, it suffers from all portability problems of GMP. The main assumption is that the ISO C types `long` and `unsigned long` can represent 2^k different values. It was extensively tested for $k = 32$ and $k = 64$.

[7]We made sure that no error was made while translating x from the MPFR internal format to the target library format. When translating the computed value z back to the MPFR format, two cases are possible. Either the target library rounded z to 53 bits, as in the case of CLN and NTL, or it uses internally more bits, so we had to round to nearest the value of z.

library	$f()$	x or x, y	ulp error
NTL	e^x	0.60337592897831904	0.5009
NTL	$e^x - 1$	0.66690478331490088	1.002
NTL	$\log_{10} x$	0.59145421077101468	−0.5019
NTL	$\log(1 + x)$	0.61574695303550087	0.5005
NTL	x^y	22.172328425393630, 0.18478812559152935	0.5011
PARI	$\sin x$	0.092436882176912372	0.5006
PARI	$\tan x$	−0.74316551642999262	0.5008
PARI	$\operatorname{acos} x$	0.99999480067740643	8.245
PARI	x^y	0.0054835146127132361, 14.809742565349817	0.5249
CLN	$\operatorname{atan} x$	−0.92184053351615713	−0.5010
CLN	$\operatorname{asin} x$	0.70044840147400333	0.5013
CLN	$\sinh x$	0.90564218340505143	0.5005
CLN	$\operatorname{asinh} x$	0.44463173722234539	0.5016
CLN	x^y	3684.4155953211484, 85.582808072565101	−0.9812

Table II. Some incorrectly rounded values from CLN 1.1.11, PARI 2.2.12.beta and NTL 5.4, with a precision of 53 bits. Inputs are the 53-bit numbers nearest from the given 17-digit values, errors are rounded away from zero. Note: PARI computes with 64 bits.

Since its implementation uses integer types only, MPFR should correctly work on a non IEEE 754 compliant configuration, except for the conversions from/to machine floating-point types.

4. APPLICATIONS AND COMPANION TOOLS

The MPFR library is distributed under the LGPL, which allows one to use it in any software. We list here some "companion tools" and some applications that use MPFR.

4.1 Companion Tools

Several companion tools are built on top of MPFR:

—MPFI is an interval library, developed by Revol and Rouillier [Revol and Rouillier 2005]; MPFR++ is a C++ interface for MPFR developed by Revol; both are available at `http://perso.ens-lyon.fr/nathalie.revol/software.html`;

—MPC (`http://www.lix.polytechnique.fr/Labo/Andreas.Enge/Mpc.html`) is a library for complex numbers, developed by Enge and Zimmermann. MPC uses the Cartesian representation: a complex number $x + iy$ is stored as a pair (x, y) of two MPFR variables. Similarly, a complex rounding mode is a pair of two real rounding modes, giving a total of 16 modes from the four ones from IEEE 754.

—MPCHECK (`http://www.loria.fr/~zimmerma/mpcheck/`) is a program that checks properties of mathematical libraries in fixed precision (correct rounding, monotonicity...), developed by Pélissier, Revol and Zimmermann.

4.2 Other Applications

A multiple-precision library with correct rounding is useful for many applications. For instance, the Fortran compiler (`gfortran`) distributed within GCC-4.0 uses MPFR to convert in double-precision constant expressions that can be computed statically. The Magma Computational Algebra System makes use of MPFR for its floating-point arithmetic[8]. The Computational Geometry Algorithms Library (CGAL, `www.cgal.org`) uses MPFR to convert rationals into double-precision intervals, while ensuring the input rational lies in the computed interval. Stehlé uses MPFR in his guaranteed floating-point LLL implementation [Nguyen and Stehlé 2005]. Even a Matlab toolbox exists, to provide multiple-precision floating-point numbers within Matlab[9].

In [Booker et al. 2006], the authors use MPFR — through the MPFI interval arithmetic library — for the effective computation of Maass cusp forms; similarly in [Booker 2005], the author performs rigorous computations with 30-digit accuracy for the verification of Artin's conjecture.

The FLUCTUAT static analyzer developed at CEA [Goubault 2001] represents the roundoff errors at step i of the program with a linear term l_i; computations are performed using a higher precision with MPFR, to ensure rigorous and tight error bounds.

5. CONCLUSION

This paper shows that correct rounding for arbitrary-precision floating-point numbers can be achieved at low cost. Moreover, a software implementing correct rounding enables one to build other applications with well-defined floating-point foundations, as quoted in [Smith 1991]:

> *Kahan [...] has pointed out that even if rounding is only slightly sloppy, it can sometimes lead to highly inaccurate results. He also notes that it is a great boon to the user to know that the results are correctly rounded. The fact that identities are true and bounds on the errors are known simplifies any analysis of a computation enough to justify a small time penalty.*

We hope this work will motivate developers of multiple-precision floating-point software to provide well-defined semantics. Ultimately, we may dream of a standard for multiple-precision floating-point arithmetic, so that a given multiple-precision computation would give the same result with any software; in the same way that thanks to the IEEE 754 standard, a given double-precision computation now gives the same result on any hardware.

Concerning the set of implemented functions, as said in [Brent 1981b]:

> *A never-ending project is to implement multiple-precision versions of ever more special functions, and to improve the efficiency of those multiple-precision routines already implemented.*

ACKNOWLEDGMENTS

The development of MPFR was supported by INRIA Lorraine and LORIA, and by the *Conseil Régional de Lorraine.* Apart from the authors of this paper, other people contributed to the library: Sylvie Boldo, David Daney, Emmanuel Jeandel, Mathieu Dutour, Fabrice Rouillier. We thank Richard Kreckel and Karim Belabas

[8] `https://magma.maths.usyd.edu.au/magma/export/mpfr_gmp.shtml`

[9] `http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=6446`

for their help with the CLN and PARI libraries respectively, Jean-Luc Szpyrka who set up the MPFR CVS archive, and finally all users of MPFR for their feedback which enables to improve the library.

REFERENCES

Batut, C., Belabas, K., Bernardi, D., Cohen, H., and Olivier, M. 2000. *User's Guide to PARI/GP*. `http://pari.math.u-bordeaux.fr/pub/pari/manuals/2.1.6/users.pdf`.

Booker, A. R. 2005. Artin's conjecture, Turing's method and the Riemann hypothesis. `http://www.arxiv.org/abs/math.NT/0507502`. 37 pages.

Booker, A. R., Strömbergsson, A., and Venkatesh, A. 2006. Effective computation of Maass cusp forms. `http://www.math.uu.se/~astrombe/papers/papers.html`. Preprint. 29 pages.

Brent, R. P. 1978. A Fortran multiple-precision arithmetic package. *ACM Trans. Math. Softw. 4,* 1, 57–70.

Brent, R. P. 1981a. An idealist's view of semantics for integer and real types. Technical Report TR-CS-81-14, Australian National University, 12 pages.

Brent, R. P. 1981b. MP user's guide. Technical Report TR-CS-81-08, Australian National University. 4th edition. 73 pages.

Char, B. W., Geddes, K. O., Gonnet, G. H., Leong, B. L., Monagan, M. B., and Watt, S. M. 1991. *Maple V: Language Reference Manual.* Springer-Verlag.

Clinger, W. D. 1990. How to read floating point numbers accurately. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation.* White Plains, NY, 92–101.

Collins, G. E. and Krandick, W. 2000. Multiprecision floating point addition. In *Proceedings of the 2000 International Symposium on Symbolic and Algebraic Computation (ISSAC'2000)*, C. Traverso, Ed. ACM Press, 71–77.

Cowlishaw, M. 2005. *The decNumber C library*, 3.32 ed. IBM UK Laboratories. 55 pages.

Cuyt, A., Kuterna, P., Verdonk, B., and Vervloet, J. 2001. Arithmos: a reliable integrated computational environment. `http://www.cant.ua.ac.be/arithmos/index.html`.

de Dinechin, F., Ershov, A. V., and Gast, N. 2005. Towards the post-ultimate libm. In *Proceedings of 17th IEEE Symposium on Computer Arithmetic.* Cape Cod, USA.

Defour, D., Hanrot, G., Lefèvre, V., Muller, J.-M., Revol, N., and Zimmermann, P. 2004. Proposal for a standardization of mathematical function implementation in floating-point arithmetic. *Numerical Algorithms 37,* 1-4, 367–375.

Gay, D. M. 1990. Correctly rounded binary-decimal and decimal-binary conversions. Numerical Analysis Manuscript 90-10, AT&T Bell Laboratories.

Goubault, E. 2001. Static analyses of the precision of floating-point operations. In *Proceedings of SAS'01.* Lecture Notes in Computer Science, vol. 2126. Springer-Verlag, 234–259.

Granlund, T. 2004. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 4.1.4 ed. `http://www.swox.se/gmp/`.

Hack, M. 2004. On intermediate precision required for correctly-rounding decimal-to-binary floating-point conversion. In *Proceedings of 6th Conference Real Numbers and Computers (RNC'6).* Schloss Dagstuhl, Germany.

Haible, B. and Kreckel, R. 2005. CLN, a class library for numbers. `http://www.ginac.de/CLN/`. Version 1.1.11.

Haible, B. and Papanikolaou, T. 1997. Fast multiprecision evaluation of series of rational numbers. Technical Report TI-7/97, Darmstadt University of Technology.

Hanrot, G., Lefèvre, V., Pélissier, P., and Zimmermann, P. 2005. The MPFR library. `http://www.mpfr.org/`. Version 2.2.0.

Hull, T. E. 1978. Desirable floating-point arithmetic and elementary functions for numerical computation. In *Proceedings of the 4th IEEE Symposium on Computer Arithmetic (Arith'4).* 63–69.

IEEE 1985. IEEE standard for binary floating-point arithmetic. ANSI/IEEE Std 754-1985.

KAHAN, W. 1996. Lecture notes on the status of IEEE standard 754 for binary floating-point arithmetic. http://http.cs.berkeley.edu/~wkahan/ieee754status/ieee754.ps. 30 pages.

KRANDICK, W. AND JOHNSON, J. R. 1993. Efficient multiprecision floating point multiplication with optimal directional rounding. In *Proceedings of the 11th IEEE Symposium on Computer Arithmetic (Arith'11)*. Windsor, Ontario.

KREINOVICH, V. AND RUMP, S. 2006. Towards optimal use of multi-precision arithmetic: A remark. http://www.ti3.tu-harburg.de/~rump/. 4 pages.

LEFÈVRE, V. 2004. The generic multiple-precision floating-point addition with exact rounding (as in the MPFR library). In *Proceedings of 6th Conference on Real Numbers and Computers (RNC'6)*. Schloss Dagstuhl, Germany.

MULDERS, T. 2000. On short multiplications and divisions. *AAECC 11,* 1, 69–88.

MULLER, J.-M. 2005. *Elementary Functions. Algorithms and Implementation*, 2nd ed. Birkhäuser.

MÜLLER, N. T. 1997. Towards a real RealRAM: a prototype using C++. In *Proc. 6th International Conference on Numerical Analysis*. Plovdiv.

NGUYEN, P. AND STEHLÉ, D. 2005. Floating-point LLL revisited. In *Proceedings of Eurocrypt 2005*. Lecture Notes in Computer Science, vol. 3494. Springer-Verlag, 215–233.

PRIEST, D. M. 1991. Algorithms for arbitrary precision floating point arithmetic. In *Proceedings of the 10th Symposium on Computer Arithmetic*, P. Kornerup and D. Matula, Eds. IEEE Computer Society Press, Grenoble, France, 132–144.

REVOL, N. AND ROUILLIER, F. 2005. MPFI, a multiple precision interval arithmetic library based on MPFR. http://mpfi.gforge.inria.fr/.

SHOUP, V. 2005. NTL: A library for doing number theory. http://www.shoup.net/ntl/. Version 5.4.

SMITH, D. M. 1991. Algorithm 693. a Fortran package for floating-point multiple-precision arithmetic. *ACM Trans. Math. Softw. 17,* 2, 273–283.

SOFRONIOU, M. AND SPALETTA, G. 2005. Precise numerical computation. *Journal of Logic and Algebraic Programming. Special Issue on Practical Development of Exact Real Number Computation 64,* 1, 113–134.

STEELE, G. L. AND WHITE, J. L. 1990. How to print floating-point numbers accurately. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*. White Plains, NY, 112–126.

STEELE, G. L. AND WHITE, J. L. 2003. How to print floating-point numbers accurately. In *20 Years of the ACM/SIGPLAN Conference on Programming Language Design and Implementation (1979-1999): A Selection*. Retrospective. 3 pages.

SUN MICROSYSTEMS. 2004. Libmcr 0.9 beta: A reference correctly-rounded library of basic double-precision transcendental elementary functions. http://www.sun.com/download/products.xml?id=41797765.

THE ARENAIRE PROJECT. 2005. CR-Libm, a library of correctly rounded elementary functions in double-precision. http://lipforge.ens-lyon.fr/projects/crlibm/. Version 0.8.

VERDONK, B., CUYT, A., AND VERSCHAEREN, D. 2001. A precision- and range-independent tool for testing floating-point arithmetic I: Basic operations, square root, and remainder. *ACM Trans. Math. Softw. 27,* 1, 92–118.

WOLFRAM, S. 1996. *The Mathematica Book*, 3rd ed. Cambridge University Press.

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

A. PRECISION INCREMENT IN ZIV'S STRATEGY

We may wonder if the chosen precision increment in Ziv's strategy is the "optimal" one. It really depends on unknown data, but we can consider a probabilistic point of view: We assume that if we know that the result y is in some small interval of radius ε, then the probability of y being in some small subinterval of radius η is

η/ε. Under this probabilistic hypothesis (and a high enough precision increment, e.g. larger than 30, such as the number of bits in a limb), it is extremely unlikely to need to go beyond a second iteration in Ziv's strategy. If this is not the case, then this means that the probabilistic hypothesis may no longer hold. So, the choice of the increment for the second and next iterations should not be regarded as critical. However one may want to do a cost analysis in average under this probabilistic hypothesis; we do not try to be rigorous here.

We assume that the cost of a computation in precision n is Cn^{α}, where the constants C and α satisfy $C > 0$ and $\alpha \geq 1$. The constant C would appear in factor of every cost expression, so we can assume that $C = 1$. Now let us assume that we are at some iteration of Ziv's strategy; n denotes the precision chosen in the previous iteration (the target precision at the first iteration). Let k be the increment; the goal is to determine the best value of k. The next increment is denoted i; we assume that i is chosen in such a way that there exists some small constant $\beta \geq 1$ such that $i \leq \beta n$ (intuitively, this is not a bad choice, and this will be confirmed below). So, the total cost can be written:

$$(n+k)^{\alpha} + 2^{-k}(n+i)^{\alpha} + 2^{-i}(\ldots)$$

where the unwritten expression depends only on the following increments. We assume that i and the following increments have been fixed, so k will be expressed as a function of n, α and i. We seek to minimize the part that depends on k:

$$(n+k)^{\alpha} + 2^{-k}(n+i)^{\alpha}.$$

Its derivative is:

$$\alpha(n+k)^{\alpha-1} - 2^{-k}(n+i)^{\alpha}\log 2, \tag{1}$$

which is an increasing function of k on $[-n, +\infty)$, where it has a unique zero (since it is negative for $k = -n$ and goes to 1 or $+\infty$ as $k \to +\infty$); κ now denotes this zero. One has:

$$2^{\kappa} \leq \frac{(n+\beta n)^{\alpha}\log 2}{\alpha n^{\alpha-1}} = \frac{(1+\beta)^{\alpha}\log 2}{\alpha}\,n.$$

Therefore $\kappa \leq \log_2(n) + \delta$, where $\delta = \log_2(\log 2) - \log_2\alpha + \alpha\log_2(1+\beta)$, and $\log_2$ denotes the base-2 logarithm. Thus for reasonable values of α and β, we have: $\kappa \leq \beta n$. Then:

$$2^{\kappa} \geq \frac{n^{\alpha}\log 2}{\alpha(n+\beta n)^{\alpha-1}} = \frac{\log 2}{\alpha(1+\beta)^{\alpha-1}}\,n.$$

Therefore $\kappa \geq \log_2(n) + \gamma$, where $\gamma = \log_2(\log 2) - \log_2\alpha - (\alpha-1)\log_2(1+\beta)$.

This shows that a precision increment $n \to n + \log_2 n$ at each iteration is a good choice. This differs from the model of [Kreinovich and Rump 2006] — where the optimal increment is $n \to cn$ — because we take here into account the probability of failure of Ziv's strategy, which vanishes exponentially with the working precision.

Note also that this strategy may be suboptimal in some rare cases, like $\sqrt{x^2+y^2}$ for $y \ll x$, where the working precision w has to be much larger than the target precision p to be able to round correctly. Those cases should be detected before applying Ziv's strategy.