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

Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, Paul Zimmermann

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

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

Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, Paul Zimmermann

N° 5753

November 2005

Thème SYM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--5753--FR+ENG

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

Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier,
Paul Zimmermann

Thème SYM — Systèmes symboliques
Projet SPACES

Rapport de recherche n° 5753 — November 2005 — 15 pages

Abstract: 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 ideas from the IEEE-754 standard to arbitrary precision, by providing *correct rounding* and *exceptions*. We demonstrate how these strong semantics are achieved — with no significant slowdown with respect to other tools — and discuss a few applications where such a library can be useful.

Key-words: multiple-precision arithmetic, IEEE-754 standard, floating-point arithmetic, correct rounding, elementary function, portable software

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy (France)
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

MPFR: une bibliothèque d'arithmétique flottante binaire en multiprécision avec arrondi correct

Résumé : Cet article présente une bibliothèque d'arithmétique flottante binaire en multiprécision, écrite en langage C ISO, et basée sur la bibliothèque GNU MP. Sa particularité est d'étendre les idées de la norme IEEE 754 à la précision arbitraire en fournissant l'*arrondi correct* et des *exceptions.* Nous montrons comment nous avons pu réaliser cette sémantique forte — sans ralentissement important par rapport aux autres outils — et discutons de quelques applications pour lesquelles une telle bibliothèque peut être utile.

Mots-clés : arithmétique multiprécision, norme IEEE 754, arithmétique à virgule flottante, arrondi correct, fonction élémentaire, logiciel portable

1 Introduction and Motivation

The IEEE-754 standard [16] for floating-point arithmetic, adopted in 1985, has now become a common standard, even if some features like *gradual underflow* (i.e. subnormals) are still discussed. 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 were able to design efficient algorithms using those formats and operations, and prove their correctness: interval arithmetic, floating-point expansions [24], correctly-rounded elementary functions [35, 32, 8, 31]. Thus, even if the IEEE-754 standard received several criticisms — both from hardware constructors that argued it would be too difficult to implement in a chip, and from software engineers that thought it would slow down significantly 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 [2], MPF [11], CLN [14], PARI/GP [1], but they do not provide clear semantics, or only claim “almost always correctly rounded” results like the FM package [27]. As pointed out by Ziv [35], 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 disastrous 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 [15], is written in the C language on top of the GNU MP library (GMP for short) [11], and freely distributed under the GNU Lesser General Public License (LGPL for short). MPFR provides correct rounding for all the operations and mathematical functions it implements, with an efficiency comparable to other 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 with respect to multiple-precision libraries with ill-defined semantics.

The paper is organized as follows. Section 2 presents existing software and related work. Section 3 describes the MPFR library, its user interface and its internals (data representation and algorithms). Section 4 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 5.

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

2 Existing Software and Related Work

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 of computer algebra systems like Maple [3] or Mathematica [34], 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 [11] 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 [13].

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], Arithmos [7] which implements correct rounding in several possible radices; and NTL [26], which guarantees correct rounding — to nearest only — for the four basic operations and the square root, and "almost correct rounding" for other mathematical functions. In radix 10, the `decNumber` package guarantees correct rounding, but only implements the four basic operations, the square root, and the integral power [6].

3 The MPFR Library

3.1 User Interface

The MPFR library is a smooth extension of the IEEE-754 standard [16], with radix 2. This choice of radix 2 follows from two requirements. For the sake of efficiency, we wanted to use 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

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

[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!

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 [25], IRRAM [22] and Mathematica [28] 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.

3.2 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 exponent e. 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):

$\boxed{11100}$	$\boxed{11000}$	$\boxed{11001}$	$\boxed{10\mathit{000}}$
limb 3	limb 2	limb 1	limb 0

3.3 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 [19]; an additional difficulty with respect to previous work, e.g. [5], is that all three operands of the addition routine — both input variables and the output variable — may have different precisions.

The multiplication of two n-bit numbers with a n-bit result is performed by a "short product", either using a naive algorithm [18] or Mulders' algorithm [20], 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 $\lfloor \beta n \rceil$ like in Mulders' original algorithm, but instead the optimal cutoff is determined by a tune 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 size of the inputs). This allows to get an 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).

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 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 a FFT-based algorithm, with thresholds tuned for the target processor.

3.4 Advanced Functions

Release 2.1.1 of MPFR implements 24 mathematical functions — logarithm and exponential in natural base, base 2 and 10, the $\log(1+x)$ and $(e^x - 1)$ functions, the six trigonometric and hyperbolic functions and their inverses, the gamma, zeta and error functions, the arithmetic-geometric mean — and 3 mathematical constants ($\log 2$, π, Euler's constant γ). One of the long-term objectives is to implement all functions from the ISO C99 standard mathematical library.

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 [9] 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, $\arccos 2$;
2. treat inputs that surely 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.

This approach requires to be able to identify all inputs that give an output which is exactly representable (step 3), otherwise Ziv's strategy will not terminate, unless the error bound ϵ is zero. 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.

For example, we describe what happens in MPFR-2.1.1 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:

$$s = (0.111111111110000000000000101010101010100100111110100101011000011)_2.$$

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

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

The error bound ϵ corresponds to 2^{14} ulps (the boldface bit), and we see that $\mathrm{round}(y - \epsilon, p) = \mathrm{round}(y + \epsilon, p)$, 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 [4, 10, 29]. 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 [30]:

> *[...] 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 [12], 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" by Smith [27], and finally a binary splitting method for huge operands. As for basic operations, the corresponding thresholds are optimized by a tuning program.

3.5 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 0 (or other functions that return an exact infinite value from finite values). 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.

3.6 Testing

Testing is a major issue, especially for a library claiming correct rounding in arbitrary precision [33]. 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 $-1 \le \frac{|x|}{\sqrt{x^2+y^2}} \le 1$ for rounding to nearest

[17] 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 the rounding mode). 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.

4 Results

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

4.1 Efficiency

Table 1 compares MPFR 2.1.1, CLN 1.1.9, PARI 2.2.9-alpha, NTL 5.3.2, 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.

4.2 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. 2). For some random input x, let z be the value computed by the corresponding library[4]. 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: for CLN, the symmetry is sometimes not respected; for example for $x = 0.83070210528807542$ and $f = \sinh$, we have $f(-x) \neq -f(x)$.

4.3 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

[4] 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.

operation	digits	MPFR	CLN	PARI	NTL
$x \times y$	10^2	**0.00053**	0.00072	0.00057	0.00080
	10^4	**0.56**	0.82	0.57	0.57
x/y	10^2	**0.0011**	0.0013	0.0012	0.0017
	10^4	**1.2**	2.4	**1.2**	**1.2**
$\sqrt{x}$	10^2	0.0018	0.0016	**0.0015**	0.0039
	10^4	**0.83**	1.60	**0.83**	1.98
$\exp x$	10^2	**0.019**	0.055	0.032	0.148
	10^4	74	**71**	139	1730
$\log x$	10^2	0.040	0.068	**0.037**	0.806
	10^4	**39**	79	*40*	17790
$\sin x$	10^2	**0.024**	0.057	0.032	0.157
	10^4	**109**	131	133	1830
$\cos x$	10^2	**0.019**	0.050	0.029	0.167
	10^4	**107**	125	132	8440
$\operatorname{acos} x$	10^2	0.68	***0.078***	0.090	NA
	10^4	1210	*156*	**154**	NA
$\operatorname{atan} x$	10^2	0.65	**0.069**	0.080	NA
	10^4	1020	**150**	153	NA

Table 1: Timings in milliseconds for several operations on a 1.8 GHz Athlon (`laurent5.medicis.polytechnique.fr`) under Linux. 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: for 10^4 digits `glogagm` is faster in Pari than the default `glog`, and CLN only provides a complex $\operatorname{acos} x$ function. "NA" means that the corresponding function is not available.

library	$f()$	x or x, y	ulp error
NTL	$\sqrt{x}$	0.54143409767007922	-1.000
NTL	e^x	-0.97619125763993853	0.5003
NTL	$e^x - 1$	0.66066817942155287	1.001
NTL	$\log_{10} x$	0.76254888190805381	-0.5015
NTL	$\log(1 + x)$	0.52094640542487658	1.814
NTL	x^y	3.6710619463140368, 12.199014764594423	0.5007
PARI	$\sin x$	863.93798795269947	$-2.245 \cdot 10^4$
PARI	$\cos x$	783.82736362139860	$5.621 \cdot 10^4$
PARI	$\tan x$	-783.82736362139860	$-3.438 \cdot 10^4$
PARI	$\operatorname{acos} x$	0.99999589812050316	20.62
PARI	x^y	53602.864594407532, 1014.5693745541940	-15.23
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 2: Some incorrectly rounded values from CLN 1.1.9, PARI 2.2.9-alpha and NTL 5.3.2, with a precision of 53 bits. Inputs are the 53-bit numbers nearest from the given 17-digit values.

$k = 64$. Since its implementation uses integer types only, MPFR should correctly work on a non IEEE-754 compliant configuration, except the conversions to machine floating-point types.

5 Applications and Companion Tools

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

5.1 Companion Tools

Several companion tools are built on top of MPFR:

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

- MPC is a library for complex numbers, developed by Enge and Zimmermann[5]. MPC uses the Cartesian representation: a complex number $x+iy$ is stored as a pair of two MPFR variables (x, y). 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.

[5] `http://www.loria.fr/~zimmerma/free/`

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

5.2 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.[6] 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 [23]. Even a Matlab toolbox exists, to provide multiple-precision floating-point numbers within Matlab.

6 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 [27]:

> *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.

7 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 MPFR: Sylvie Boldo, David Daney, Emmanuel Jeandel, Mathieu Dutour, Fabrice Rouillier. We thank Richard Kreckel and Karim Belabas for their help with the CLN and Pari libraries respectively,

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

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

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

[2] Richard P. Brent. A Fortran multiple-precision arithmetic package. *ACM Transactions on Mathematical Software*, 4(1):57–70, 1978.

[3] Bruce W. Char, Keith O. Geddes, Gaston H. Gonnet, Benton L. Leong, Michael B. Monagan, and Stephen M. Watt. *Maple V: Language Reference Manual.* Springer-Verlag, 1991.

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

[5] George E. Collins and Werner Krandick. Multiprecision floating point addition. In Carlo Traverso, editor, *Proceedings of the 2000 International Symposium on Symbolic and Algebraic Computation (ISSAC'2000)*, pages 71–77. ACM Press, 2000.

[6] Mike Cowlishaw. The decNumber package. `http://www2.hursley.ibm.com/decimal/`, 2005. Version 3.25.

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

[8] F. de Dinechin and Nicolas Gast. Towards the post-ultimate libm. In *Proceedings of 17th IEEE Symposium on Computer Arithmetic*, Cape Cod, USA, 2005.

[9] David Defour, Guillaume Hanrot, Vincent Lefèvre, Jean-Michel Muller, Nathalie Revol, and Paul Zimmermann. Proposal for a standardization of mathematical function implementation in floating-point arithmetic. *Numerical Algorithms*, 37(1-4):367–375, 2004. `http://www.kluweronline.com/issn/1017-1398`.

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

[11] Torbjörn Granlund. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 4.1.4 edition, 2004. `http://www.swox.se/gmp/`.

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

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

[14] Bruno Haible and Richard Kreckel. CLN, a class library for numbers. `http://www.ginac.de/CLN/`, 2005. Version 1.1.9.

[15] Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. The MPFR library. `http://www.mpfr.org/`, 2005. Version 2.1.1.

[16] IEEE standard for binary floating-point arithmetic. ANSI/IEEE Std 754-1985, 1985.

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

[18] Werner Krandick and Jeremy R. Johnson. Efficient multiprecision floating point multiplication with optimal directional rounding. In *Proceedings of 11th IEEE Symposium on Computer Arithmetic*, Windsor, Ontario, 1993.

[19] V. Lefèvre. The generic multiple-precision floating-point addition with exact rounding (as in the MPFR library). In *Proc. 6th Conference on Real Numbers and Computers*, 2004.

[20] T. Mulders. On short multiplications and divisions. *AAECC*, 11(1):69–88, 2000.

[21] Jean-Michel Muller. *Elementary Functions. Algorithms and Implementation.* Birkhäuser, 1997. 232 pages.

[22] N. Th. Müller. Towards a real RealRAM: a prototype using C++. In *Proc. 6th International Conference on Numerical Analysis*, Plovdiv, 1997.

[23] P. Nguyen and D. Stehlé. Floating-point LLL revisited. In *Proceedings of Eurocrypt 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 215–233. Springer-Verlag, 2005.

[24] Douglas M. Priest. Algorithms for arbitrary precision floating point arithmetic. In Peter Kornerup and David Matula, editors, *Proceedings of the 10th Symposium on Computer Arithmetic*, pages 132–144, Grenoble, France, 1991. IEEE Computer Society Press.

[25] Nathalie Revol and Fabrice Rouillier. MPFI, a multiple precision interval arithmetic library based on MPFR. `http://perso.ens-lyon.fr/nathalie.revol/software.html`, 2001.

[26] Victor Shoup. NTL: A library for doing number theory. `http://www.shoup.net/ntl/`, 2004. Version 5.3.2.

[27] David M. Smith. Algorithm 693. a Fortran package for floating-point multiple-precision arithmetic. *ACM Transactions on Mathematical Software*, 17(2):273–283, 1991.

[28] Mark Sofroniou and Giulia Spaletta. Precise numerical computation. *Journal of Logic and Algebraic Programming. Special Issue on Practical Development of Exact Real Number Computation*, 64(1):113–134, July 2005. `http://dx.doi.org/10.1016/j.jlap.2004.07.007`.

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

[30] Guy L. Steele and Jon L. White. 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*, 2003. Retrospective. 3 pages.

[31] Sun Microsystems. 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`, 2004.

[32] The Arenaire project. CR-Libm, a library of correctly rounded elementary functions in double-precision. `http://lipforge.ens-lyon.fr/projects/crlibm/`, 2005. Version 0.8.

[33] Brigitte Verdonk, Annie Cuyt, and Dennis Verschaeren. A precision- and range-independent tool for testing floating-point arithmetic I: Basic operations, square root, and remainder. *ACM Transactions on Mathematical Software*, 27(1):92–118, 2001.

[34] Stephen Wolfram. *The Mathematica Book*. Cambridge University Press, 3rd edition, 1996.

[35] A. Ziv. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software*, 17(3):410–423, 1991.

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399