This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2690632, IEEE Transactions on Computers

Correctly Rounded Arbitrary-Precision Floating-Point Summation

Vincent Lefèvre

Abstract—We present a fast algorithm together with its low-level implementation of correctly rounded arbitrary-precision floating-point summation. The arithmetic is the one used by the GNU MPFR library: radix 2; no subnormals; each variable (each input and the output) has its own precision. We also give a worst-case complexity of this algorithm and describe how the implementation is tested.

Index Terms—summation, floating point, arbitrary precision, multiple precision, correct rounding.

1 Introduction

In a floating-point system, the summation operation consists in evaluating the sum of several floating-point numbers. The IEEE 754 standard for floating-point arithmetic introduced the `sum` reduction operation in its 2008 revision [1, Clause 9.4], but does not provide specifications except related to special inputs and exceptions; the only specified finite result is that the result of the sum of 0 numbers is defined as $+0$. The IEEE 1788-2015 standard for interval arithmetic goes further by completely specifying this `sum` operation for IEEE 754 floating-point formats [2, Clause 12.12.12], in particular requiring correct rounding and specifying the sign of an exact zero result, but in a way that is incompatible with IEEE 754-2008 since in particular, the result of the sum of 0 numbers is -0 in the roundTowardNegative rounding direction.

The articles in the literature on floating-point summation mainly focus on IEEE 754 arithmetic and consider the floating-point arithmetic operations ($+$, $-$, etc.) as basic blocks; in this context, inspecting bit patterns is generally not interesting. For instance, fast and accurate summation algorithms are presented by Demmel and Hida [3] and by Rump [4]. Correct rounding is not provided. On this subject, the class of algorithms that can provide a correctly rounded sum of $n \geqslant 3$ numbers is somewhat limited [5]. In [6], Rump, Ogita and Oishi present correctly rounded summation algorithms. Kulisch proposes a quite different solution: the use of a long accumulator covering the full exponent range (and slightly more to handle intermediate overflows) [7]. A survey of summation methods can be found in [8, Section 6.3].

In IEEE 754, the precision of each floating-point format is fixed. In this paper, we deal with the extension of the summation operation to arbitrary precision in radix 2, where each number has its own precision and results must be correctly rounded, as with the GNU MPFR library[1] [9], where this function is named `mpfr_sum`. This paper is an extended version of [10], with an enhanced specification of `mpfr_sum` (for backward compatibility with the one from MPFR 3.1 and to follow the usual MPFR rules concerning the function arguments, but also supporting precision 1, which is a recent change in the MPFR development) and much more details (in particular, some important parts of the proofs could not be given in the previous version of the paper).

Due to the requirements from MPFR, our algorithm is not based on any previous work, even though one can find similar ideas used in a different context such as in [11], which also uses blocks, but in some other way; indeed, this algorithm from Demmel and Nguyen does not have the same goals and the data are represented in a different way:

	Demmel/Nguyen	`mpfr_sum`
Model	parallel	sequential
Precision	fixed	arbitrary
Accuracy	error bound involving the maximum of the input numbers	correct rounding
Reproducibility	yes	yes, implied by correct rounding
Representation	floating point	based on arrays of integers

The condition on the accuracy makes a big difference. Like some other algorithms, Demmel and Nguyen's does not take a possible cancellation into account; this allows it to be always fast, but in case of large cancellation, the result will be very inaccurate in general (if not completely meaningless). Conversely, for `mpfr_sum`, we need to handle cancellation in order to always get an accurate result, which is the main difficulty; the correct-rounding requirement mainly adds more subcases, but it does not introduce additional issues from a theoretical point of view: we will see that guaranteeing a correctly rounded result in the difficult cases (i.e., solving the *Table Maker's Dilemma*) is equivalent to the computation of an accurate sum to a 1-bit target precision.

We first give some notation (Section 2). In Section 3, we present a brief overview of GMP and GNU MPFR. In

Vincent Lefèvre is with INRIA, LIP / CNRS / ENS Lyon / Université de Lyon, Lyon, France. E-mail: vincent@vinc17.net

1. http://www.mpfr.org/ and http://www.mpfr.org/mpfr-3.1.5/mpfr.html

Section 4, we describe the old `mpfr_sum` implementation and explain why a new one was needed. In Section 5, we give the complete specification of the summation operation in MPFR. In Section 6, we present the completely new algorithm and implementation; since this is a low-level algorithm, the context of MPFR is quite important for the details, but the main ideas could be reused in other contexts. We also give an example in §6.5, and an asymptotic upper bound on the time taken by this algorithm (worst-case complexity) in §6.6. In Section 7, we explain how `mpfr_sum` is tested.

This paper is based on the revision 11319 of `sum.c` in the trunk of the MPFR repository[2] for MPFR 4.0 (not released yet).

2 Notation

We will use $[\![$ and $]\!]$ for the bounds of integer intervals, e.g. $[\![0,3]\!] = \{0,1,2,3\}$ and $[\![0,3[\![= \{0,1,2\}$.

3 Overview of GMP and GNU MPFR

GNU MPFR is a free library for efficient arbitrary-precision floating-point computation with well-defined semantics (copying the good ideas from the IEEE 754 standard), in particular correct rounding. It is based on GNU MP (GMP)[3], which is a free library for arbitrary-precision arithmetic; MPFR mainly uses the low-level GMP layer called "mpn", and we will restrict to it here. As said on the GMP web page: "*Low-level positive-integer, hard-to-use, very low overhead functions are found in the mpn category. No memory management is performed; the caller must ensure enough space is available for the results.*"

In this layer, a natural number is represented by an array of words, called *limbs*, each word corresponding to a digit in high radix (2^{32} or 2^{64}). The main GMP functions that will be useful for us are: the addition (resp. subtraction) of two N-limb numbers, with carry (resp. borrow) out; ditto between an N-limb number and a limb; left shift; right shift; negation with borrow out; complement. For instance, `mpn_add_1` adds a limb to an N-word number, yielding an N-word number and a carry (0 or 1); this is particularly efficient when the source and the destination N-word numbers have the same memory location (in-place operation), which will always be the case in `mpfr_sum`.

Each MPFR floating-point object (even when it does not contain a number yet) has its own precision in bits, starting at 1 for the future MPFR 4.0, which is the target of this implementation (the minimum precision is 2 in MPFR up to 3.1). All arithmetic operations are correctly rounded to the precision of the destination number in one of the 5 supported rounding modes:

- `MPFR_RNDN` (to nearest, with the even rounding rule),
- `MPFR_RNDD` (toward $-\infty$),
- `MPFR_RNDU` (toward $+\infty$),
- `MPFR_RNDZ` (toward zero),
- `MPFR_RNDA` (away from zero).

Let us describe how MPFR data (numbers and NaN) are represented. In addition to the precision field (regarded mainly as a parameter), 3 fields are used to represent nonzero finite numbers, called *regular* data: a sign, a significand (always normalized, with the leading bit 1 represented, and any trailing bit in the least significant limb being 0) interpreted as being in $[1/2, 1[$, and an exponent field. Similarly to the IEEE-754 formats but mainly for a different reason (as detailed below), not all possible values of the exponent field correspond to valid exponents. Thus zeros, infinities and NaN, together called *singular* data, are represented with some special values of the exponent field.[4] Contrary to IEEE 754, MPFR has only a single kind of NaN and does not have subnormals (but a function `mpfr_subnormalize` is provided to emulate them). The sign field contains a boolean value and is handled in the same way as in IEEE 754: all floating-point numbers, including zeros and infinities, are signed; NaN is not signed, but its sign field can be used by some operations for (partial) compatibility with IEEE 754. For singular data, the significand contains garbage.

An important point is that the exponent range can be very large in MPFR: up to $[\![1-2^{62}, 2^{62}-1]\!]$ on 64-bit machines. In addition to some theoretical issues for the evaluation of trigonometric functions, this introduces difficulties in the implementation of various functions (including `mpfr_sum`), but is more or less needed as a consequence of arbitrary precision. On this subject, Section *Extended and extendable precisions* of IEEE 754-2008 [1, Clause 3.7] requires the support of a maximum exponent to be at least 1000 times the precision for extendable precision formats.

In MPFR, exponents are stored in a signed integer type `mpfr_exp_t`. If this type has N value bits, i.e., the maximum value is $2^N - 1$, then the maximum exponent range is defined so that any valid exponent fits in $N-1$ bits (sign bit excluded), i.e., it is $[\![1-2^{N-1}, 2^{N-1}-1]\!]$; this choice has been made to allow the sum of two exponents to be representable in the type, which simplifies the implementation of some operations (such as the multiplication of two numbers). This implies a huge gap between the minimum value of the type `MPFR_EXP_MIN` $= -2^N$ (or $1-2^N$) and the minimum valid exponent `MPFR_EMIN_MIN` $= 1-2^{N-1}$. The maximum valid exponent is denoted `MPFR_EMAX_MAX` $= 2^{N-1}-1$.

This allows the following implementation to be valid in practical cases. Assertion failures could occur in cases involving extremely huge precisions (detected for security reasons). In short, the problem comes from the fact that the exponent of the k-th bit of a MPFR number of exponent e is $e-k$, and one may need to be able to represent this value. In practice, these failures are not possible with a 64-bit ABI due to memory limits. With a 32-bit ABI, users would probably reach other system limits first (e.g., on the address space); the best solution would be to switch to a 64-bit ABI for such computations. MPFR code of some other functions have similar requirements, which are often not documented. Here, the problem could be solved with some minor drawbacks, but this would not currently be interesting in practice.

2. https://gforge.inria.fr/scm/viewvc.php/mpfr/trunk/src/sum.c?revision=11319&view=markup

3. https://gmplib.org/

4. In the earliest versions of MPFR, these singular data were represented in another way, and changes were done in 2003 for MPFR 2.1.0 in order to reduce the overhead due to singular data, visible in low precision.

Note: *Unbounded floats*, whose exponent is an arbitrary-precision integer (GMP's `mpz_t` type), have been implemented recently by the author of this paper, for some basic operations. Such a number is like a MPFR number, but with an additional member to represent the exponent when the exponent field has the special value `MPFR_EXP_UBF`. So, little change to existing functions was needed to introduce this support, though it can slightly increase the overhead. This was useful for a correct implementation of the $ab + cd$ operation (`mpfr_fmma`) to avoid intermediate overflows or underflows, even in corner cases. In the same way, `mpfr_sum` could be changed to support unbounded floats; this could be useful to handle the most difficult cases of correctly rounded polynomial evaluation. Then, the problem mentioned in the above paragraph would disappear.

Moreover, most arithmetic operations return a ternary value, giving the sign of the rounding error. For instance, if one has:

```
r = mpfr_add (a, b, c, MPFR_RNDN);
```

meaning $a \leftarrow b+c$, where a has a 3-bit precision[5], $b = 5$ and $|c| < 1/2$, then one will get $a = 5$, and r will be 0 if $c = 0$, negative if $c > 0$, and positive if $c < 0$. With `MPFR_RNDD`, the ternary value is always negative (inexact result) or zero (exact result). With `MPFR_RNDU`, it is always positive (inexact result) or zero (exact result).

4 THE OLD `mpfr_sum` IMPLEMENTATION

The implementation of `mpfr_sum` from the current MPFR releases (up to version 3.1.5) is based on Demmel and Hida's accurate summation algorithm [3], which consists in adding the inputs one by one in decreasing magnitude. But here, this has several drawbacks:

- This is an algorithm using only high-level operations, mainly floating-point additions (in MPFR, `mpfr_add`). This is the right way to do to get an accurate sum in true IEEE 754 arithmetic implemented in hardware, but in MPFR, which uses integers as basic blocks, this introduces overheads, and more important problems mentioned just below.
- Due to the high-level operations, correct rounding had to be implemented with a Ziv loop: the working precision is increased until the rounding can be guaranteed [9]. In the case of summation, this gives a time and memory worst-case complexity exponential in the number of bits of the exponent field. In practice, this is very slow in some cases, and worse, since the exponent range can be large, this can yield a crash due to the lack of memory (and possible denial of service for other processes running on the machine).
- Demmel and Hida's algorithm is based on the fact that the precision is the same for all floating-point numbers, meaning that in the MPFR implementation, the maximum precision had to be chosen. An alternative would be to split the input numbers to numbers with the same precision, but this would introduce another overhead.

5. The target precision is attached to the variable (a one-element array, as in GMP, which is thus passed by reference, or pointer).

Moreover, the sign of an exact zero result is not specified and the ternary value is valid only when it is zero (a nonzero return value provides no information).

5 SPECIFICATION OF `mpfr_sum`

The prototype of the `mpfr_sum` function is:

```
int mpfr_sum (mpfr_ptr sum,
              const mpfr_ptr *x,
              unsigned long n,
              mpfr_rnd_t rnd)
```

where `sum` will contain the correctly rounded sum, `x` is an array of pointers to the inputs, `n` is the length of this array, and `rnd` is the rounding mode. The return value of type `int` will be the usual ternary value. Input pointers are now allowed to be reused for the output.[6]

If `n` $= 0$, then the result is $+0$, whatever the rounding mode. This is equivalent to `mpfr_set_ui` and `mpfr_set_si` on the integer 0, which both assign a MPFR number from a mathematical zero (not signed), and this choice is consistent with the IEEE 754 sum operation of vector length 0.

Otherwise the result (including the sign of zero) must be the same as the one that would have been obtained with:

- if `n` $= 1$: a copy with rounding (`mpfr_set`);
- if `n` > 1: a succession of additions (`mpfr_add`) done in infinite precision, then rounded (the order of these additions does not matter).

This is equivalent to apply the following ordered rules:

1) If an input is NaN, then the result is NaN.
2) If there are at least a $+\infty$ and a $-\infty$, then the result is NaN.
3) If there is at least an infinity (in which case all the infinities have the same sign), then the result is this infinity.
4) If the result is an exact zero:
   - if all the inputs have the same sign (thus all $+0$'s or all -0's), then the result has the same sign as the inputs;
   - otherwise, either because all inputs are zeros with at least a $+0$ and a -0, or because some inputs are nonzero (but they globally cancel), the result is $+0$, except for the `MPFR_RNDD` rounding mode, where it is -0.
5) Otherwise the exact result is a nonzero real number, and the conventional rounding function is applied.

6 NEW ALGORITHM AND IMPLEMENTATION

The new algorithm is carefully designed so that the time and memory complexity no longer depends on the value of the exponents of the inputs, i.e., the orders of magnitude of the inputs. Instead of being high level (based on `mpfr_add`), the algorithm/implementation is low level, based on integer operations, equivalently seen as fixed-point operations. Efficiency in case of cancellations and Table Maker's Dilemma is regarded as important as for cases without such issues. To be as fast as possible, we will use the mpn layer of GMP. The implementation is thread-safe (no use of global data).

6. This was not the case in [10]. So, the algorithm was a bit different there.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2690632, IEEE Transactions on Computers

As a bonus, this will also solve overflow, underflow and normalization issues, since everything is done in fixed point and the exponent of the result will be considered only at the end (early overflow detection could also be done, but this would probably not be very useful in practice).

The idea is the following. After handling special cases (NaN, infinities, only zeros, and fewer than 3 regular inputs), we apply the generic case, which more or less consists in a fixed-point accumulation by blocks: we take into account the bits of the inputs whose exponent is in some window $[\![\mathtt{minexp}, \mathtt{maxexp}[\![$, and if this is not sufficient due to cancellation, then we reiterate, using a new window with lower exponents. Once we have obtained an accurate sum, if one still cannot round correctly because the result is too close to a rounding boundary (i.e., a machine number or the middle of two consecutive machine numbers), which is the problem known as the Table Maker's Dilemma (TMD), then this problem is solved by determining the sign of the remainder by using the same method in a low precision.

In order to make the understanding of the algorithm easier, a simplified example will be given in Section 6.5.[7]

6.1 Preliminary Steps

We start by detecting the special cases. The `mpfr_sum` function does the following.

If $\mathtt{n} \leqslant 2$, we can use existing MPFR functions and macros, mainly for better efficiency since the algorithm described below can work with any number of inputs (only minor changes would be needed):

- if $\mathtt{n} = 0$: return $+0$ (by using MPFR macros);
- if $\mathtt{n} = 1$: use `mpfr_set` (which copies a number, with rounding to the target precision);
- if $\mathtt{n} = 2$: use `mpfr_add` (which adds two numbers, with rounding to the target precision).

Now, we have $\mathtt{n} \geqslant 3$. We iterate over the `n` input numbers to:

(A) detect singular values (NaN, infinity, zero);
(B) among the regular values, get the maximum exponent.

Such information can be retrieved very quickly and this does not need to look at the significand. Moreover, in the current internal number representation, the kind of a singular value is represented as special values of the exponent field, so that (B) does not need to fetch more data in memory after doing (A).

In detail, during this iteration, 4 variables will be set, but the loop will terminate earlier if one can determine that the result will be NaN, either because of a NaN input or because of infinity inputs of opposite signs:

- `maxexp`, which will contain the maximum exponent of the inputs. Thus it is initialized to `MPFR_EXP_MIN`.
- `rn`, which will contain the number of regular inputs, i.e., those which are nonzero finite numbers.
- `sign_inf`, which tracks the signs of infinite summands. It is initialized to 0, meaning no infinities yet. When the first infinity is encountered, this value is changed to the sign of this infinity ($+1$ or -1). When a new infinity is encountered, either it has the same sign of `sign_inf`, in which case nothing changes, or it has the opposite sign, in which case the loop terminates immediately and a NaN result is returned.
- `sign_zero`, which will contain the sign of the zero result *in the case where all the inputs are zeros*. Thanks to the IEEE 754 rules, this can be tracked with this variable alone: There is a weak sign (-1, except for `MPFR_RNDD`, where it is $+1$), which can be obtained only when all the inputs are zeros of this sign, and a strong sign ($+1$, except for `MPFR_RNDD`, where it is -1), which is obtained in all the other cases, i.e., when there is at least a zero of this sign. One could have initialized the value of `sign_zero` to the weak sign. But we have chosen to initialize it to 0, which means that the sign is currently unknown, and do an additional test in the loop. In practice, one should not see the difference; this second solution was chosen just because it was implemented first, and on a test, it made the code slightly shorter.

When the loop has terminated "normally", the result cannot be NaN. We do in the following order:

1) If $\mathtt{sign\_inf} \neq 0$, then the result is an infinity of this sign, and we return it.
2) If $\mathtt{rn} = 0$, then all the inputs are zeros, so that we return the result zero whose sign is given by `sign_zero`.
3) If $\mathtt{rn} \leqslant 2$, then one can use `mpfr_set` or `mpfr_add` as an optimization, similarly to what was done for $\mathtt{n} \leqslant 2$. We reiterate in order to find the concerned input(s), call the function and return.
4) Otherwise we call a function `sum_aux`, which implements the generic case. In addition to the parameters of `mpfr_sum`, we pass to this function:
   - the maximum exponent;
   - the number `rn` of regular inputs, i.e., the number of nonzero inputs. This number will be used instead of `n` to determine bounds on the sum (to avoid internal overflows) and error bounds.

6.2 Introduction to the Generic Case (`sum_aux`)

Let us define $\mathtt{logn} = \lceil \log_2(\mathtt{rn}) \rceil$.

The basic idea is to compute a truncated sum in the two's complement representation, by using a fixed-point accumulator stored in a fixed memory area.

Two's complement is preferred to the sign + magnitude representation because the signs of the temporary (and final) results are not known in advance, and the computations (additions and subtractions) in two's complement are more natural in this context. There will be a conversion to sign + magnitude (representation used by MPFR numbers) at the end, but this should not take much time compared to the other calculations.

The precision of the accumulator needs to be slightly larger than the output precision, denoted `sq`, for two reasons:

- We need some additional bits on the side of the most significant part due to the accumulation of `rn` values, which can make the sum grow and overflow without enough extra bits. The absolute value of the sum is less than $\mathtt{rn} \cdot 2^{\mathtt{maxexp}}$, thus takes up to `logn` extra bits; and one needs one more bit to be able to determine the sign due

7. This example is not given earlier because it uses variables introduced during the description of the algorithm, but it may help to look at it now.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2690632, IEEE Transactions on Computers

to two's complement. So, a total of $\mathtt{cq} = \mathtt{logn} + 1$ extra bits will be necessary.
- We need some additional bits on the side of the least significant part to take into account the accumulation of the truncation errors. The choice of this number dq of bits is quite arbitrary: the larger this value is, the longer an iteration will take, but conversely, the less likely a costly new iteration (due to cancellations and/or the Table Maker's Dilemma) will be needed. In order to make the implementation simpler, the precision of the accumulator will be a multiple of the limb size GMP_NUMB_BITS. Moreover, the algorithm will need at least 4 bits. The final choice should be done after testing various applications. In the current implementation, we chose the smallest value larger or equal to $\mathtt{logn} + 2$ such that the precision of the accumulator is a multiple of GMP_NUMB_BITS. Since $\mathtt{logn} \geqslant 2$, we have $\mathtt{dq} \geqslant 4$ as wanted.

As shown in the figure below, the precision of the accumulator is initially defined as:

$$\mathtt{wq} = \mathtt{cq} + \mathtt{sq} + \mathtt{dq}$$

The exponent of the least significant bit (LSB) of the accumulator is denoted by minexp, so that:

$$\mathtt{minexp} = \mathtt{maxexp} + \mathtt{cq} - \mathtt{wq}.$$

```
┌─────────────────── wq ───────────────────┐
[--------]----------------------------------]
│  cq  └─ maxexp      sq + dq      minexp ─┘
└─ sign bit (0 = positive, 1 = negative)
```

In the accumulation, the selected bits from the inputs will range from minexp (included) to maxexp (excluded), and the most significant cq bits can only be reached due to carry propagation.

When the Table Maker's Dilemma occurs, the needed precision for the truncated sum would grow. In particular, one could easily reach a huge precision with a few small-precision inputs: for instance, in directed rounding modes, sum(2^E,2^F) with F much smaller than E. We want to avoid increasing the precision of the accumulator. This will be done by detecting the Table Maker's Dilemma, and when it occurs, solving it consists in determining the sign of some error term. This will be done by computing an approximation to the error term in low precision. The algorithm to compute this error term is the same as the one to compute an approximation to the sum, the main difference being that we just need a 1-bit accuracy here. Thus we will define a function sum_raw, used for both computations; it is described in the next section.

6.3 The sum_raw Function

The sum_raw function will work in a fixed-point accumulator, having a fixed precision (a multiple of GMP_NUMB_BITS bits) and using a two's complement representation. An iteration will consist in accumulating the bits of the inputs whose exponents are in $[\![\mathtt{minexp}, \mathtt{maxexp}[\![$, where $\mathtt{maxexp} - \mathtt{minexp}$ is less than the precision of the accumulator: as said above, we need some additional bits in order to avoid overflows during the accumulation. On the entry, the accumulator may already contain a value from previous computations (it is the caller that clears it if need be): in some cases, some bits will have to be kept between the two sum_raw invocations.

During the accumulation, the bits of the i-th input x[i] whose exponents are strictly less than minexp form the *tail* of this input. When the tail of x[i] is not empty, its exponent e_i is defined as the minimum between minexp and the exponent of x[i]. Thus the absolute value of this tail is strictly less than 2^{e_i}. This will give an error bound on the computed sum at each iteration: $\mathtt{rn} \cdot 2^{\sup_i(e_i)} \leqslant 2^{\sup_i(e_i)+\mathtt{logn}}$.

At the end of an iteration, we do the following. If the computed result is 0 (meaning full cancellation), set maxexp to the maximum exponent of the tails, set minexp so that it corresponds to the least significant bit of the accumulator, and reiterate. Otherwise, let e and err denote the exponents of the computed result (in two's complement) and of the error bound respectively. While $\mathtt{e} - \mathtt{err}$ is less than some given bound denoted prec, shift the accumulator (as detailed later), update maxexp and minexp, and reiterate. For the caller, this bound must be large enough in order to reach some wanted accuracy. However, it cannot be too large since the accumulator has a limited precision: we will need to make sure that if a reiteration is needed, then the cause is a partial cancellation, so that the determined shift count is nonzero, otherwise the variable minexp would not change and one would get an infinite loop. Details and formal definitions are given later.

Notes:

- The reiterations will end when there are no more tails, but in the code, this is detected only when needed.
- This definition of the tails allows one to skip potentially huge gaps between inputs in case of full cancellation, e.g. $1 + (-1) + r$ where r is tiny.
- We choose not to include maxexp in the exponent interval in order to match the convention chosen to represent floating-point numbers in MPFR, where the significand is in $[1/2, 1[$, i.e., the exponent of a floating-point number is the one of the most significant bit + 1. Another advantage is that minexp at some iteration will be maxexp at the next iteration, unless there is a gap between the inputs (i.e., the exponent of each tail is less than minexp).

Now let us give the details about this sum_raw function. In addition to the pointers and sizes of the accumulator and a preallocated temporary area, it takes the following arguments:

- wq: precision of the accumulator.
- x: array of the input numbers.
- n: size of this array (number of inputs, regular or not).
- minexp: exponent of the LSB of the first window.
- maxexp: exponent of the first window (i.e., exponent of its MSB + 1).
- logn: $\lceil \log_2(\mathtt{rn}) \rceil$, rn being the number of regular inputs.
- prec: lower bound for $\mathtt{e} - \mathtt{err}$ (as described above).
- ep: pointer to mpfr_exp_t (see below).
- minexpp: pointer to mpfr_exp_t (see below).
- maxexpp: pointer to mpfr_exp_t (see below).

We require as preconditions (explanations are given later): $\mathtt{prec} \geqslant 1$ and $\mathtt{wq} \geqslant \mathtt{logn} + \mathtt{prec} + 2$.

This function returns 0 if the accumulator is 0 (which implies that the exact sum for this sum_raw invocation is

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2690632, IEEE Transactions on Computers

0), otherwise the number of *cancelled bits*, defined as the number of consecutive identical bits on the most significant part of the accumulator[8]. In the latter case, it also returns the following data in variables passed by reference (i.e., via pointers) unless these pointers are null (such data are useful only after the first invocation of `sum_raw`, i.e., after the main computation, not after the TMD resolution):

- for `ep`: the exponent `e` of the computed result;
- for `minexpp`: the last value of the variable `minexp`;
- for `maxexpp`: the last value of the variable `maxexp2` (which would be the new value of the variable `maxexp` for the next iteration, i.e. the first iteration of the second invocation of `sum_raw` in case of TMD resolution).

Some notation used below:

- `E(v)`: the exponent of a MPFR number `v`.
- `P(v)`: the precision of a MPFR number `v`.
- `Q(v)` $=$ `E(v)` $-$ `P(v)`: the exponent of the ulp of a MPFR number `v`.

A `maxexp2` variable will contain the maximum exponent of the tails. Thus it is initialized to the minimum value of the exponent type: `MPFR_EXP_MIN`; this choice means that at the end of the loop below, `maxexp2` $=$ `MPFR_EXP_MIN` if and only if there are no more tails (this case implies that the truncated sum is exact). If a new iteration is needed, then `maxexp2` will be assigned to the `maxexp` variable for this new iteration.

Then one has a loop over the inputs `x[i]`. Each input is processed with the following steps:

1) If the input is not regular (i.e., is zero), skip it. Note: if there are many zero inputs, it may be more efficient to have an array pointing to the regular inputs only, but such a case is assumed to be rare, and the number of iterations of this inner loop is also limited by the relatively small number of regular inputs.
2) If `E(x[i])` $\leqslant$ `minexp`, then no bits of `x[i]` need to be considered here. We set the `maxexp2` variable to $\max($`maxexp2`, `E(x[i])`$)$, then go to the next input.
3) Now, we have: `E(x[i])` $>$ `minexp`. If the tail of `x[i]` is not empty, i.e., if `Q(x[i])` $<$ `minexp`, then we set the `maxexp2` variable to `minexp`.
4) We prepare the input for the accumulation. In particular, if its significand is not aligned with the accumulator, then we need to align it by shifting a part of the significand (containing bits that will be accumulated at this iteration); the result is stored to the temporary area, which must be large enough, i.e., its bit size must be at least `maxexp` $-$ `minexp` $+$ `GMP_NUMB_BITS` $- 1$.
5) If `x[i]` is positive, an addition and carry propagation toward the most significant bit of the accumulator are done with `mpn_add_n` followed by `mpn_add_1`. There may be still be a carry out, but it is just ignored; this occurs when a negative value in the accumulator becomes nonnegative, and this fact is part of the usual two's complement arithmetic.

   If `x[i]` is negative, we do similar computations by using `mpn_sub_n` and `mpn_sub_1` for the subtraction and borrow propagation.

8. Note that a value larger than 1 does not necessarily mean that a cancellation really occurred, due to a possible bias, in particular at the first iteration because of the `cq` bits. What matters is that this value provides a measure of the relative accuracy.

Note: The steps 2, 3, and 4 above are currently done by distinguishing two cases:

- `Q(x[i])` $<$ `minexp`, covering cases like:

```
  [-+- accumulator ---]
[---|----- x[i] ------|--]
    |   [----- x[i] --|--]
    |                 |[----- x[i] -----]
    |                 |    [----- x[i] -----]
  maxexp           minexp
```

- `Q(x[i])` $\geqslant$ `minexp`, covering cases like:

```
            [-+- accumulator ---]
[- x[i] -]    |                 |
          [---|-- x[i] ------]  |
       [------|-- x[i] ---------]
              |   [- x[i] -]    |
            maxexp           minexp
```

It might be possible to merge these cases in a future version of the code.

After the loop over the inputs, we need to see whether the accuracy of the truncated sum is sufficient. We first determine the number of cancelled bits (the number of consecutive identical bits on the most significant part of the accumulator). At the same time, we can determine whether the truncated sum is 0 (all the bits are identical and their value is 0). If it is 0, we have two cases: if `maxexp2` $=$ `MPFR_EXP_MIN` (meaning no more tails), then we return 0, otherwise we reiterate at the beginning of `sum_raw` with `minexp` set to `cq` $+$ `maxexp2` $-$ `wq` and `maxexp` set to `maxexp2`.

We can now assume that the truncated sum is not 0.

Let us note that our computation of the number `cancel` of cancelled bits was limited to the accumulator representation, while from a mathematical point of view, the binary expansion is unlimited and the bits of exponent less than `minexp` are regarded as 0's:

```
[--- accumulator ---]000000...
            minexp ─┘└─ first nonrepresented bit = 0
```

So, we need to check that the value `cancel` matches this mathematical point of view:

- If the cancelled bits are 0's: the truncated sum is not 0, therefore the accumulator must contain at least a bit 1.
- If the cancelled bits are 1's: this sequence of 1's entirely fits in the accumulator, since the first nonrepresented bit is a 0.

The analysis below virtually maps the truncated sum to the destination without considering rounding yet. Let us denote: `e` $=$ `minexp` $+$ `wq` $-$ `cancel` and `err` $=$ `maxexp2` $+$ `logn`.

Then `e` is the exponent of the least significant cancelled bit, thus the absolute value of the truncated sum is in $[2^{\mathtt{e}-1}, 2^{\mathtt{e}}]$ (binade closed on both ends due to two's complement). Since there are at most $2^{\mathtt{logn}}$ regular inputs and the absolute value of each tail is strictly less than $2^{\mathtt{maxexp2}}$, the absolute value of the error is strictly less than $2^{\mathtt{err}}$. If `maxexp2` $=$ `MPFR_EXP_MIN` (meaning no more tails), then the error is 0.

We need `prec` $\geqslant 1$ to be at least able to determine the sign of the result, hence this precondition. Moreover, the fact that `prec` is nonnegative allows us to use unsigned integer arithmetic in the test below in order to avoid a potential integer overflow.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2690632, IEEE Transactions on Computers

If $\mathtt{e} - \mathtt{err} \geqslant \mathtt{prec}$, then the `sum_raw` function returns as described above.

Otherwise, due to cancellation, we need to reiterate after shifting the value of the accumulator to the left and updating the `minexp` and `maxexp` variables. Let `shiftq` denote the shift count, which must satisfy: $0 < \mathtt{shiftq} < \mathtt{cancel}$. The left inequality must be strict to ensure termination, and the right inequality ensures that the value of the accumulator will not change with the updated `minexp`: `shiftq` is subtracted from `minexp` at the same time. The reiteration is done with `maxexp` set to `maxexp2`, as said above.

Let us give an example. If there is an additional iteration with $\mathtt{maxexp2} = \mathtt{minexp} - 4$ and a shift of $\mathtt{shiftq} = 26$ bits (due to cancellation), here is the accumulator before and after the shift:

```
Before: [--------]----------------------------------]
                 ├─ maxexp                  minexp ─┤
        0000000000000000000000000001----------------]
        <--- identical bits (0) --->
                                <------- 26 zeros ------->
After:  001-----------------00000000000000000000000000
This iteration:    minexp ─┘  ├─ maxexp2              │
Next iteration:               └─ maxexp     minexp ─┘
```

We now need to determine the value of `shiftq`. We prefer it to be as large as possible so that the next iteration will involve the largest possible number of additional bits of the summands: this is some form of normalization. Moreover, it must satisfy the above double inequality and be such that:

(A) the new value of `minexp` is smaller than the new value of `maxexp`, i.e., $\mathtt{minexp} - \mathtt{shiftq} < \mathtt{maxexp2}$, which is equivalent to: $\mathtt{shiftq} > \mathtt{minexp} - \mathtt{maxexp2}$;

(B) overflows will still be impossible in the new iteration.

Note that since $\mathtt{maxexp2} \leqslant \mathtt{minexp}$, (A) will imply $\mathtt{shiftq} > 0$. And (B) is an extension of $\mathtt{shiftq} < \mathtt{cancel}$. Thus it is sufficient to satisfy (A) and (B).

Since we prefer `shiftq` to be maximum (and defined with a simple formula), we focus on (B) first. To avoid an overflow, it is sufficient that the absolute value of the accumulator at the end of the next iteration be strictly less than $2^{\mathtt{minexp}-\mathtt{shiftq}+\mathtt{wq}-1}$ (and this condition is also necessary if the value is positive). This absolute value will be strictly bounded by: $2^{\mathtt{e}} + 2^{\mathtt{err}} \leqslant 2^{1+\max(\mathtt{e},\mathtt{err})}$. So, in order to satisfy (B), we can choose:

$$\mathtt{shiftq} = \mathtt{minexp} + \mathtt{wq} - 2 - \max(\mathtt{e}, \mathtt{err}).$$

Now, let us prove that for this value, (A) is satisfied.

- If $\mathtt{err} \geqslant \mathtt{e}$, then by using the precondition $\mathtt{prec} \geqslant 1$, we get: $\max(\mathtt{e}, \mathtt{err}) = \mathtt{err} < \mathtt{err} + \mathtt{prec}$.
- If $\mathtt{err} < \mathtt{e}$, then the error can be potentially small: to be able to prove (A), we need to use the fact that the stop condition was not satisfied, i.e., $\mathtt{e} - \mathtt{err} < \mathtt{prec}$. We get: $\max(\mathtt{e}, \mathtt{err}) = \mathtt{e} < \mathtt{err} + \mathtt{prec}$.

Thus $\mathtt{shiftq} > \mathtt{minexp} + \mathtt{wq} - 2 - \mathtt{err} - \mathtt{prec}$. By using $\mathtt{err} = \mathtt{maxexp2} + \mathtt{logn}$, we get:

$$\mathtt{shiftq} - (\mathtt{minexp} - \mathtt{maxexp2}) > \mathtt{wq} - \mathtt{logn} - \mathtt{prec} - 2 \geqslant 0.$$

The second inequality above comes from the precondition $\mathtt{wq} \geqslant \mathtt{logn} + \mathtt{prec} + 2$, which has been chosen for this purpose.

Note: It is expected in general that when a cancellation occurs so that a new iteration is needed, the cancellation is not very large (but this really depends on the problem), in which case the new additions will take place only in a small part of the accumulator, except in case of long carry propagation.

6.4 Back to the Generic Case

Let us recall that the accumulator for the summation is decomposed into three parts: $\mathtt{cq} = \mathtt{logn} + 1$ bits to avoid overflows, `sq` bits corresponding to the target precision, and `dq` additional bits to take into account the truncation error and improve the accuracy ($\mathtt{dq} \geqslant \mathtt{logn} + 2$ in the current implementation). Thus $\mathtt{wq} = \mathtt{cq} + \mathtt{sq} + \mathtt{dq}$.

A single chunk of memory is allocated for the accumulator and for the temporary area needed by `sum_raw`; since $\mathtt{maxexp} - \mathtt{minexp} \leqslant \mathtt{wq} - \mathtt{cq}$ at each `sum_raw` iteration (to avoid overflows), the size chosen for the temporary area is the smallest one with at least $\mathtt{wq} - \mathtt{cq} + \mathtt{GMP\_NUMB\_BITS} - 1$ bits. We also chose to allocate memory for the possible TMD resolution (as explained later) in the same chunk; this second accumulator will be useless in most cases (it is necessary only if the TMD occurs and some input is reused as the output), but in the current implementation, it takes at most two limbs in practice, so that this does not make a noticeable difference. For performance reasons, the memory is allocated in the stack instead of the heap if its size is small enough (with the `MPFR_TMP_LIMBS_ALLOC` macro, as often done in other functions for temporary allocations). No other memory allocation will be needed (except for auto variables).

Note: Having a small-size accumulator for `sum_raw`, either for the main computation or for the TMD resolution, is not the best choice for the worst-case complexity. For the time being, we focus on correctness and make sure that the implementation is fast on almost all cases and not too slow on corner cases. In the future, we may want to fix a minimal size for the accumulator or allow it to grow dynamically, for instance in a geometric progression after a few iterations (similarly to what is done for Ziv loops in the TMD resolution for mathematical functions).

The accumulator is zeroed and `sum_raw` is invoked to compute an accurate approximation of the sum. Among its parameters, `maxexp` was computed during the preliminary steps, $\mathtt{minexp} = \mathtt{maxexp} - (\mathtt{wq} - \mathtt{cq})$, and $\mathtt{prec} = \mathtt{sq} + 3$, which satisfies the $\mathtt{wq} \geqslant \mathtt{logn} + \mathtt{prec} + 2$ precondition: $\mathtt{wq} = \mathtt{cq} + \mathtt{sq} + \mathtt{dq} \geqslant \mathtt{logn} + 1 + \mathtt{sq} + 4 = \mathtt{logn} + \mathtt{prec} + 2$.

If `sum_raw` returns 0, then the exact sum is 0, so that we just set the target `sum` to 0 with the correct sign according to the IEEE 754 rules (positive, except for `MPFR_RNDD`, where it is negative), and return with ternary value 0.

Now, the accumulator contains the significand of a good approximation to the nonzero exact sum. The corresponding exponent is `e` and the sign is determined from one of the cancelled bits. The exponent of the ulp for the target precision is denoted $\mathtt{u} = \mathtt{e} - \mathtt{sq}$. The exponent stored at `maxexpp` (i.e., the last value of the variable `maxexp2` in `sum_raw`) is denoted `maxexp2`. We have:

- $\mathtt{err} = \mathtt{maxexp2} + \mathtt{logn}$ as in `sum_raw`;
- $\mathtt{e} - \mathtt{err} \geqslant \mathtt{prec} = \mathtt{sq} + 3$.

Thus $\mathtt{err} \leqslant \mathtt{u} - 3$, i.e., the absolute value of the error is strictly less than 2^{-3} times the ulp of the computed value: $2^{\mathtt{u}-3}$.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2690632, IEEE Transactions on Computers

Here is a representation of the accumulator and the cancelled bits, with the two cases depending on the sign of the truncated sum, where the x's correspond to the $\mathtt{sq}-1$ represented bits following the initial value bit (1 if positive sum, 0 if negative), `r` is the *rounding bit*, and the bits `f` are the following bits:

```
][--------------- accumulator ---------------]
][--- cancel ---]----------------------------]
]0000000000000001xxxxxxxxxxxxxxxxxxrffffffffff0000...
]1111111111111110xxxxxxxxxxxxxxxxxxrffffffffff0000...
                └─ e              u ─┘  minexp ─┘
```

Note that the iterations in `sum_raw` could have stopped even in case of important cancellation: it suffices that the error term be small enough, i.e., where the tails for the last iteration consisted only of inputs `x[i]` whose exponent was very small compared to `minexp`. In such a case, the bit `r` and some of the least significant bits `x` may fall outside of the accumulator, in which case they are regarded as 0's (still due to truncation). In the following, we will make sure that we do not try to read nonrepresented bits.

When $\mathtt{maxexp2} \neq \mathtt{MPFR\_EXP\_MIN}$, i.e., when some bits of the inputs have still not been considered, we will need to determine whether the TMD occurs. In this case, we will compute $\mathtt{d} = \mathtt{u} - \mathtt{err}$, which is larger or equal to 3 (see above) and can be very large if `maxexp2` is very small; nevertheless, `d` is representable in a `mpfr_exp_t` since:

- If $\mathtt{maxexp2} < \mathtt{minexp}$, then `maxexp2` is the exponent of an input `x[i]`, so that $\mathtt{maxexp2} \geqslant \mathtt{MPFR\_EMIN\_MIN}$; and since $\mathtt{u} \leqslant \mathtt{MPFR\_EMAX\_MAX}$ (the maximum valid exponent), we have $\mathtt{d} \leqslant \mathtt{MPFR\_EMAX\_MAX} - \mathtt{MPFR\_EMIN\_MIN}$, which is representable in a `mpfr_exp_t` as per definition of the `MPFR_EMIN_MIN` and `MPFR_EMAX_MAX` macros in MPFR (see Section 3 about the exponent range).
- If $\mathtt{maxexp2} = \mathtt{minexp}$, then

$$\mathtt{d} \leqslant (\mathtt{minexp} + \mathtt{wq}) - \mathtt{maxexp2} = \mathtt{wq},$$

which is representable in a `mpfr_exp_t` since this type can contain all precision values (type `mpfr_prec_t`).

The TMD occurs when the sum is close enough to a *breakpoint*, which is defined as a discontinuity point of the function that maps a real input to the correctly rounded value and the ternary value. This is either a machine number (i.e., a number whose significand fits on `sq` bits) or a midpoint between two consecutive machine numbers, depending on the rounding mode:

Rounding mode	Breakpoint
to nearest	midpoint
to nearest	machine number
directed	machine number

(when the sum is close to an `sq`-bit number and the rounding mode is to nearest, the correctly rounded sum can be determined, but not the ternary value, and this is why the TMD occurs). More precisely, the TMD occurs when:

- in directed rounding modes: the `d` bits following the ulp bit are identical;
- in round-to-nearest mode: the $\mathtt{d}-1$ bits following the rounding bit are identical.

Several things need to be considered for the significand, in arbitrary order:

- the copy of the significand to the destination (if the destination is used by an input, the TMD may need to be resolved first);
- a shift (for the normalization), if the shift count is nonzero (this is the most probable case);
- a negation/complement if the value is negative (cancelled bits = 1), since the significand of MPFR numbers uses the conventional sign + absolute value representation;
- rounding (the TMD needs to be resolved first if it occurs).

It is more efficient to merge some of these operations, i.e., do them at the same time, and this possibility depends on the operations provided by the mpn layer of GMP. Ideally, all these operations should be merged together, but this is not possible with the current version of GMP (6.1.1).

For practical reasons, the shift should be done before the rounding, so that all the bits are represented for the rounding. The copy itself should be done together with the shift or the negation, because this is where most of the limbs are changed in general. We chose to do it with the shift as it is assumed that the proportion of nonzero shift counts is higher than the proportion of negations.

Moreover, for negative values, the difference between negation and complement is similar to the difference between rounding directions (these operations are identical on the real numbers, i.e., in infinite precision), so that negation/complement and rounding can naturally be merged, as detailed later.

Taking the above remarks into account, we will do the following:

1) Determine how the result will be rounded. If the TMD occurs, it is resolved at this step.
2) Copy the truncated accumulator (shifted) to the destination. For simplicity, after this step, the trailing bits of the destination (present when the precision is not a multiple of `GMP_NUMB_BITS`) contain garbage. Since rounding needs a specific operation on the least significant limb, these trailing bits (located in this limb) will be zeroed in the next step.
3) Take the complement if the result is negative, and at the same time, do the rounding and zero the trailing bits.
4) Set the exponent and handle a possible overflow or underflow.

Details for each of these four steps are given below.

6.4.1 *Rounding Information / TMD Resolution*

The values of three variables are determined:

- `inex`: 0 if the final sum is *known* to be exact (which can be the case only if $\mathtt{maxexp2} = \mathtt{MPFR\_EXP\_MIN}$), otherwise 1.
- `rbit`: the rounding bit (0 or 1) of the truncated sum, changed to 0 for halfway cases that will round toward $-\infty$ if the rounding mode is to nearest (so that this bit gives the rounding direction), as explained below.
- `tmd`: three possible values: 0 if the TMD does not occur, 1 if the TMD occurs on a machine number, 2 if the TMD occurs on a midpoint.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2690632, IEEE Transactions on Computers

Note: The value of inex will be used only if the TMD does not occur (i.e. tmd $= 0$). So, inex could be left indeterminate when tmd $\neq 0$, but this would not simplify the current code.

This is done by considering two cases:

- u $>$ minexp. The rounding bit, which is represented, is read. Then there are two subcases:
  - Subcase maxexp2 $=$ MPFR_EXP_MIN. The sum in the accumulator is exact. Thus inex will be the logical OR between the rounding bit and the sticky bit, where the *sticky bit* is 0 if and only if the bits following the rounding bit are all 0's (i.e., the value is a breakpoint in some rounding mode). In round to nearest, rbit $= 1$ will mean that the value is to be rounded toward $+\infty$, even for halfway cases as it is easier to handle these cases now. The variable rbit is initially set to the value of the rounding bit. We need to determine the sticky bit (which involves a loop) only if:
    - rbit $= 0$, or
    - rbit $=$ 1 and rnd is MPFR_RNDN and the least significant bit of the truncated sq-bit significand (i.e., the bit before the rounding bit) is 0; in such a case, if the sticky bit is 0, this halfway value will have to be rounded toward $-\infty$, so that rbit is changed to 0. Note that for sq $\geqslant 2$, the parity of the rounded significand does not depend on the representation (two's complement or sign + magnitude); that is why, even though the significand is currently represented in two's complement, we round to even. To illustrate this point, let us give an example with a negative value:

1110.1100[100000]	(two's complement)
1110.1100	(rounded to even)
0001.0100	(magnitude)

      where the bits inside the brackets are those after the truncated sq-bit significand. If we had converted the accumulator first, we would have obtained:

0001.0011[100000]	(magnitude)
0001.0100	(rounded to even)

      i.e., the same result. For sq $= 1$, the IEEE 754 rule for halfway cases is to choose the value larger in magnitude, i.e., round away from zero;[9] therefore, in this case, we want to keep rbit to 1 for positive values, and set it to 0 for negative values, but it happens that this corresponds to the rule chosen for sq $\geqslant 2$ (since the only bit of the truncated significand is 1 for positive values and 0 for negative values), so that there is no need to distinguish cases in the code.

    And tmd is set to 0 because one can round correctly, knowing the exact sum.
  - Subcase maxexp2 $\neq$ MPFR_EXP_MIN. We do not know whether the final sum is exact, so that we set inex to 1. We also determine the value of tmd as briefly described above (the code is quite complex since we need to take into account the fact that not all the bits are represented).
- u $\leqslant$ minexp. The rounding bit is not represented (its value is 0), thus rbit is set to 0. If maxexp2 $=$ MPFR_EXP_MIN, then both inex and tmd are set to 0; otherwise they are set to 1 (the bits following the ulp bit are not represented, thus are all 0's, implying that the TMD occurs on a machine number).

We also determine the sign of the result: a variable neg is set to the value of the most significant bit of the accumulator, and a variable sgn to the corresponding sign. In short:

number	neg	sgn
positive	0	$+1$
negative	1	-1

Now we seek to determine how the value will be rounded, more precisely, what correction will be done to the significand that will be copied. We currently have a significand, a trailing term t in the accumulator (bits whose exponent is in $[\![\texttt{minexp}, \texttt{u}[\![$) such that $0 \leqslant t < 1\,\text{ulp}$ (nonnegative thanks to the two's complement representation), and an error on the trailing term bounded by $t' \leqslant 2^{\texttt{u}-3} = 2^{-3}\,\text{ulp}$ in absolute value, so that the error ε on the significand satisfies $-t' \leqslant \varepsilon < 1\,\text{ulp} + t'$. Thus one has 4 correction cases, denoted by an integer value corr between -1 and 2, which depends on ε, the sign of the significand, rbit, and the rounding mode:

-1: equivalent to nextDown;
0: no correction;
$+1$: equivalent to nextUp;
$+2$: equivalent to two consecutive nextUp.

At the same time, we will also determine the ternary value and store it in inex. This will be the ternary value *before* the check for overflow and underflow, which is done at the very end of sum_aux with the mpfr_check_range function (see Section 6.4.4).

To determine corr and the ternary value, we distinguish two cases:

- tmd $= 0$. The TMD does not occur, so that the error has no influence on the rounding and the ternary value (one can assume $t' = 0$). One has inex $= 0$ if and only if $t = 0$, so that inex is currently the absolute value of the ternary value. Therefore we set corr as follows:
  - for MPFR_RNDD, corr $= 0$;
  - for MPFR_RNDU, corr $=$ inex;
  - for MPFR_RNDZ, corr $=$ inex && neg;
  - for MPFR_RNDA, corr $=$ inex && !neg;
  - for MPFR_RNDN, corr $=$ rbit.

  We now correct the sign of the ternary value: if inex $\neq 0$ (i.e., inex $= 1$) and corr $= 0$, we set inex to -1.
- tmd $\neq 0$. The TMD occurs, the exact sum being a breakpoint $+$ a small secondary term, and will be resolved by determining the sign (-1, 0 or $+1$) of this secondary term thanks to a second sum_raw invocation with a low-precision accumulator.

  Note: In the code written before the support of reused inputs as the output, the accumulator had already been copied to the destination, so that a part of the memory

9. See the discussion http://grouper.ieee.org/groups/754/email/msg03907.html started by the author in 2009, and the IEEE 754 errata on http://speleotrove.com/misc/IEEE754-errata.html.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2690632, IEEE Transactions on Computers

of this accumulator could be reused for the small-size accumulator for the TMD resolution. This is no longer possible, but currently not a problem since the accumulator for the TMD resolution takes at most only 2 limbs in practice; however, in the future, we might want the accumulators to grow dynamically, as explained above. We set up a new accumulator of size $\texttt{cq} + \texttt{dq}$ $(= \texttt{wq} - \texttt{sq})$ rounded up to the next multiple of the word size (GMP_NUMB_BITS); let us call this size zq (it will correspond to the variable wq in sum_raw). From the old accumulator, bits whose exponent is in $[\![\texttt{minexp}, \texttt{u}[\![$ (when $\texttt{u} > \texttt{minexp}$) will not be copied to the destination; these bits will be taken into account as described below.

Let us recall that the $\texttt{d} - 1$ bits from exponent $\texttt{u} - 2$ to $\texttt{u} - \texttt{d}$ $(= \texttt{err})$ are identical. We distinguish two subcases:

– Subcase $\texttt{err} \geqslant \texttt{minexp}$. The last two of the $\texttt{d} - 1$ identical bits and the following bits, i.e., the bits from $\texttt{err} + 1$ to minexp, are copied (possibly with a shift) to the most significant part of the new accumulator. The minexp value of this new accumulator is thus defined as $\texttt{minexp} = \texttt{err} + 2 - \texttt{zq}$, so that

$$\begin{aligned}\texttt{maxexp2} - \texttt{minexp} \\ &= (\texttt{err} - \texttt{logn}) - (\texttt{err} + 2 - \texttt{zq}) \\ &= \texttt{zq} - \texttt{logn} - 2 \\ &\leqslant \texttt{zq} - \texttt{cq}.\end{aligned}$$

Therefore the temporary area for sum_raw is still large enough.

– Subcase $\texttt{err} < \texttt{minexp}$. Here at least one of the identical bits is not represented, meaning that it is 0 and all these bits are 0's. Thus the accumulator is set to 0. The new minexp value is determined from maxexp2, with cq bits reserved to avoid overflows, just like in the main sum.

Then sum_raw is called with $\texttt{prec} = 1$, satisfying the first sum_raw precondition $(\texttt{prec} \geqslant 1)$. And we have:

$$\texttt{zq} \geqslant \texttt{cq} + \texttt{dq} \geqslant \texttt{logn} + 3 = \texttt{logn} + \texttt{prec} + 2,$$

corresponding to the second sum_raw precondition.

The sign of the secondary term (-1, 0, or $+1$), corrected for the halfway cases, is stored in a variable sst. In details: If the value returned by sum_raw (i.e., the number of cancelled bits) is not 0, then the secondary term is not 0, and its sign is obtained from the most significant bit of the accumulator: positive if it is 0, negative if it is 1. Otherwise the secondary term is 0, and so is its sign; however, for the halfway cases $(\texttt{tmd} = 2)$, we want to eliminate the ambiguity of their rounding due to the even-rounding rule by choosing a nonzero value for the sign: -1 if the truncated significand (in two's complement) is even, $+1$ if it is odd.

Then, from the values of the variables rnd (rounding mode), tmd, rbit (rounding bit), sst (sign of the secondary term, corrected for the halfway cases), and sgn (sign of the sum), we determine:

– the correction case corr (integer from -1 to $+2$);
– the ternary value inex (negative, zero, or positive).

The different cases are summarized in Table 1. The two lines with "n/a" correspond to halfway cases and are not possible since sst has been changed to an equivalent nonzero value as said above. The rounding modes MPFR_RNDZ and MPFR_RNDA are not in this table since they are handled like MPFR_RNDD and MPFR_RNDU depending on the value of sgn (MPFR provides internal macros MPFR_IS_LIKE_RNDD and MPFR_IS_LIKE_RNDU for this purpose).

As an example, $(\texttt{tmd}, \texttt{rbit}) = (1, 1)$ means that the truncated sum (i.e., the approximation) is just below a machine number; moreover, if sst is 0, the exact sum is this machine number. Thus $\texttt{inex} = 0$, and $\texttt{corr} = +1$ to get this machine number.

rnd	tmd	rbit	sst	corr	inex
N	1	0	$-$	0	$+$
N	1	0	0	0	0
N	1	0	$+$	0	$-$
N	1	1	$-$	$+1$	$+$
N	1	1	0	$+1$	0
N	1	1	$+$	$+1$	$-$
N	2	0	$-$	0	$-$
N	2	0	0	n/a	n/a
N	2	0	$+$	$+1$	$+$
N	2	1	$-$	0	$-$
N	2	1	0	n/a	n/a
N	2	1	$+$	$+1$	$+$
D	1	0	$-$	-1	$-$
D	1	0	0	0	0
D	1	0	$+$	0	$-$
D	1	1	$-$	0	$-$
D	1	1	0	$+1$	0
D	1	1	$+$	$+1$	$-$
U	1	0	$-$	0	$+$
U	1	0	0	0	0
U	1	0	$+$	$+1$	$+$
U	1	1	$-$	$+1$	$+$
U	1	1	0	$+1$	0
U	1	1	$+$	$+2$	$+$

TABLE 1
Correction case (corr) and ternary value (inex) depending on rnd, tmd, rbit, and sst.

At this point, the variable inex contains the correct ternary value (before the overflow/underflow detection) and we know the correction that needs to be applied to the significand.

6.4.2 *Copy/Shift to the Destination*

First, we can set the sign of the MPFR number from the value of sgn.

The bits of the accumulator that need to be taken into account for the destination are those of exponents in the interval $[\![\max(\texttt{u}, \texttt{minexp}), \texttt{e}[\![$ (if $\texttt{u} < \texttt{minexp}$, the nonrepresented bits are seen as 0's). We distinguish two cases:

- $\texttt{u} > \texttt{minexp}$. We need to copy the bits of exponents in $[\![\texttt{u}, \texttt{e}[\![$, i.e., all the bits are represented in the accumulator. One just has a left shift or a copy. In the process, some bits of exponent less than u can be copied to the trailing bits; they are seen as garbage. Since rounding will need a specific operation on the least significant limb, these trailing bits (located in this limb) will be zeroed at the same time in the next step.

- `u` $\leqslant$ `minexp`. We just have a left shift (bits that are shifted in are 0's as specified by GMP, which is what we want) or a copy, and if there are remaining low significant limbs in the destination, they are zeroed.

Note: By definition of `e`, the most significant bit that is copied is the first bit after the cancelled bits: 1 for a positive number, 0 for a negative number.

6.4.3 Complement and Rounding

For the moment, let us assume that `sq` $\geqslant 2$. We distinguish two cases:

- `neg` $= 0$ (positive sum). Since the significand can contain garbage in the trailing bits (present when the precision is not a multiple of `GMP_NUMB_BITS`), we set these trailing bits to 0 as required by the format of MPFR numbers. If `corr` > 0, we need to add `corr` to the significand (we can see that this remains valid even if `corr` $= 2$ and the significand contains all 1's, which was not obvious). This is done with `mpn_add_1`, but `corr` must be shifted by `sd` bits to the left, where `sd` is the number of trailing bits. If `corr` $= 2$ and `sd` $=$ `GMP_NUMB_BITS` $- 1$, the mathematical result of the shift does not hold in the variable; in this case, the value 1 is added with `mpn_add_1` starting at the second limb, which necessarily exists, otherwise this would mean that the precision of the MPFR number would be 1, and this is not possible (we assumed `sq` $\geqslant 2$). In case of carry out, meaning a change of binade, the most significant bit of the significand is set to 1 without touching the other bits (this is important because if `corr` $= 2$ and the significand has only one limb, the least significant nontrailing bit may be 1), and the variable `e` is incremented. If `corr` < 0, then it is -1, so that we subtract 1 from the significand with `mpn_sub_1`. If the MSB of the significand becomes 0, meaning a change of binade, then it is set back to 1 so that all the (nontrailing) bits of the significand are 1's, and the variable `e` is decremented.
- `neg` $= 1$ (negative sum). In the positive case, we could add or subtract a limb to/from a mpn number with a GMP operation. But here, we want to be able to subtract a limb from a mpn number, and GMP does not provide such an operation. However, we will show that this can be emulated (efficiently, though probably not as much as with just a native operation implemented with highly optimized assembly code, as usually provided by GMP) with `mpn_neg`, which does a negation, and `mpn_com`, which does a complement. This allows us to avoid the naive use of separate `mpn_com` (or `mpn_neg`) and `mpn_add_1` (or `mpn_sub_1`) operations, which could yield two loops in some particular cases involving a long sequence of 0's in the low significant bits.

  Let us focus on the negation and complement operations and what happens at the bit level. For the complement operation, all the bits are inverted and there is no dependency between them. The negation of an integer is equivalent to its complement plus 1: $\mathtt{neg}(x) = \mathtt{com}(x) + 1$. Said otherwise, after an initial carry propagation on the least significant sequence of 1's in $\mathtt{com}(x)$, the bits are just inverted, i.e., one has a complement operation on the remaining bits. This is why we will regard complement as the core operation in the following.

  Now, we want to compute:

$$\begin{aligned}\mathtt{abs}(x+\mathtt{corr}) &= \mathtt{neg}(x+\mathtt{corr})\\ &= \mathtt{neg}(x)-\mathtt{corr}\\ &= \mathtt{com}(x)+(1-\mathtt{corr})\end{aligned}$$

  where $-1 \leqslant 1 - \mathtt{corr} \leqslant 2$. We consider two subcases, leading to a nonnegative case for the correction, and a negative case:

  – Subcase `corr` $\leqslant 1$, i.e., $1 - \mathtt{corr} \geqslant 0$. We first compute the least significant limb by setting the trailing bits to 1, complementing the limb, and adding the correction term $1 - \mathtt{corr}$ properly shifted. This can generate a carry. In the case where `corr` $= -1$ (so that $1 - \mathtt{corr} = 2$) and the shift count `sd` is `GMP_NUMB_BITS` $- 1$, the shift of the correction term overflows, but this is equivalent to have a correction term equal to 0 and a carry.
    - ∗ If there is a carry, we apply `mpn_neg` on the next limbs (if the significand has more than one limb). If there is still a carry, i.e., if the significand has exactly one limb or if there is no borrow out of the `mpn_neg`, then we handle the change of binade just like in the positive case for `corr` > 0.
    - ∗ If there is no carry, we apply `mpn_com` on the next limbs (if the significand has more than one limb). There cannot be a change of binade in this case since a complement cannot have a carry out.

  – Subcase `corr` $= 2$, i.e., $1 - \mathtt{corr} = -1$. Here we want to compute $\mathtt{com}(x) - 1$, but GMP does not provide an operation for that. The fact is that a sequence of low significant bits 1 is invariant, and we need to do the loop ourselves in C instead of using an optimized assembly version from GMP. However, this may not be a problem in practice, as the difference is probably not noticeable (anyway, the source should here be simple enough to get good code generation by the compiler). When a limb with a zero is reached (there is at least one since the most significant bit of the significand is a 0), we compute its complement minus 1 (the "$-$ 1" corresponds to a borrow in). If there are remaining limbs, we complement them and a change of binade is not possible. Otherwise the complement minus 1 on the most significant limb can lead to a change of binade; more precisely, this happens on the significand $01111\ldots111$, whose complement is $10000\ldots000$ and $\mathtt{com}(x) - 1$ is $01111\ldots111$. The change of binade is handled like in the positive case for `corr` < 0.

If `sq` $= 1$, the solution described above does not work when we need to add 2 to the significand, since 2 is not representable on 1 bit. And as this case `sq` $= 1$ is actually simpler, we prefer to consider it separately. First, we can force the only limb to `MPFR_LIMB_HIGHBIT`, which is the value 1 shifted `GMP_NUMB_BITS` $- 1$ bits to the left, i.e., the limb with the most significant bit being 1, the other bits being 0 (these are the trailing bits): this is the only possible sig-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2690632, IEEE Transactions on Computers

nificand in precision 1. Now we need to add the correction term, which corresponds to a modification of the exponent. In the positive case, we just add `corr` to the variable `e` (exponent). In the negative case, as forcing the only limb to `MPFR_LIMB_HIGHBIT` corresponds to the computation of $\texttt{com}(x)$, we just add $1 - \texttt{corr}$ to `e`, following the formula given in the case $\texttt{sq} \geqslant 2$.

6.4.4 Exponent Consideration

Finally, we set the (maybe out-of-range) exponent of the MPFR number to `e`, and check whether `e` is in the current exponent range with the `mpfr_check_range` function as usual; this function takes the necessary data to be able to handle a possible overflow or underflow: the current result (assumed to be correctly rounded with an unbounded exponent range), the current ternary value (giving the sign of the error), and the rounding mode.

6.5 A Simplified Example

To illustrate the high-level part of the algorithm, we provide an example, simplified for readability, focusing only on the main ideas and showing what is computed at each step. In particular, we will use small blocks, whose sizes have been fixed manually for the example (such sizes may be impossible in practice due to constraints on the accumulator size). Moreover, the numbers are ordered (in the algorithm, the order does not matter as it has loops over all the numbers); said otherwise, the value of `minexp` is chosen in some arbitrary way here.

We consider `MPFR_RNDD` (round toward $-\infty$), an output precision $\texttt{sq} = 2$, and $\texttt{rn} = 9$ regular input numbers, each with its own precision, corresponding to the number of digits of the fraction part, as written below:

$$\begin{aligned}
x_0 &= +0.10011101000010 \cdot 2^{0} \\
x_1 &= -0.100001 \cdot 2^{0} \\
x_2 &= -0.11000011 \cdot 2^{-3} \\
x_3 &= -0.11101 \cdot 2^{-9} \\
x_4 &= -0.1101000 \cdot 2^{-10} \\
x_5 &= +0.10111111011 \cdot 2^{-1000} \\
x_6 &= +0.110 \cdot 2^{-1009} \\
x_7 &= +0.10000 \cdot 2^{-1009} \\
x_8 &= -0.10000 \cdot 2^{-2000}
\end{aligned}$$

The splitting into blocks (determined after each iteration) of the main computation will occur as follows. A dot corresponds to a nonrepresented digit (0) in the block. A double bar corresponds to a zeroed accumulator (with a gap in the exponents for the second one).

+	100111010	00010.....			(x_0)
-	100001...				(x_1)
-	110000	11........			(x_2)
-		11101.....			(x_3)
-		1101000..			(x_4)
+			101111110	11	(x_5)

On this example, we have the following 3 iterations, where $\texttt{prec} = \texttt{sq} + 3 = 5$.

- First iteration: $[\![\texttt{minexp}, \texttt{maxexp}[\![\; = [\![-9, 0[\![$. Here, we have $\texttt{maxexp} = 0$ because the maximum exponent of the input numbers is 0. In this window, only 3 input numbers are concerned, and we have the following computation:

```
           ┌ minexp = -9
 + 100111010[00010] (x0)
 - 100001           (x1)
 -     110000[11]   (x2)
 ______________
...0000000010
           └ e = -7
```

  The digits in the square brackets are those outside the window, thus are ignored at this iteration.
  During the same loop over all the input numbers, we compute the next `maxexp` value: Let $\mathcal{T} = \{i : \texttt{Q}(x_i) < \texttt{minexp}\}$ be the set of the indices of nonempty tails, here all the indices except 1 (since x_1 has entirely been taken into account). Then

$$\texttt{maxexp2} = \sup_{i \in \mathcal{T}} e_i = \texttt{minexp} = -9$$

  since $e_0 = \texttt{minexp}$ (ditto for e_2).
  We have computed an approximation to the sum and we have an error bound $2^{\texttt{err}}$, where $\texttt{err} = \texttt{maxexp2} + \texttt{logn} = (-9) + 4 = -5$.
  We have $\texttt{e} - \texttt{err} = (-7) - (-5) = -2 < \texttt{prec}$, so that we need at least another iteration.
- Second iteration: $[\![\texttt{minexp}, \texttt{maxexp}[\![\; = [\![-19, -9[\![$. One gets:

```
     ┌ maxexp = -9
...0010          (from the previous sum)
  +    00010     (from x0)
  -    11        (from x2)
  -    11101     (x3)
  -     1101000  (x4)
 ______________
...00000000000000
```

  The truncated sum is 0: we have a full cancellation. And $\mathcal{T} = \{5, 6, 7, 8\}$, so that $\texttt{maxexp2} = -1000$ (from x_5): there is a big gap in the exponent values. The next iteration will be done with `maxexp` set to `maxexp2`, which is the maximum exponent of the remaining numbers (thus a bit like the first iteration).
- Third iteration: $[\![\texttt{minexp}, \texttt{maxexp}[\![= [\![-1009, -1000[\![$.
  The truncated sum is $0.101111110 \cdot 2^{-1000}$ (with the first 9 bits of x_5). We have $\texttt{e} - \texttt{err} = (-1000) - (-1009 + 4) = 5 \geqslant \texttt{prec}$, so that the truncated sum is accurate enough.

We now know a good approximation to the exact sum, but this exact sum is close to a machine number (the rounding bit 1 is followed by a long sequence of 1's), so that we need a TMD resolution. The accumulator will be set to the value -2^{-1008} (110 from the least significant part of the truncated sum, followed by 0's). The first iteration of the second call to `sum_raw` computes:

```
  ┌ maxexp = -1009
110            (from the previous sum)
 +   11        (from x5)
 +   110       (x6)
 +   10000     (x7)
 ____________
000000000000
```

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2690632, IEEE Transactions on Computers

We have a full cancellation. If we did not have x_8 in the array, then this would be the case D / 1 / 1 / 0 of Table 1, giving corr $= +1$ to get $0.11 \cdot 2^{-1000}$ and a null ternary value. With x_8, we are in the case D / 1 / 1 / $-$, giving corr $= 0$ to get $0.10 \cdot 2^{-1000}$ and a negative ternary value.

6.6 Worst-Case Complexity

We now seek to find an asymptotic upper bound on the time taken by this algorithm. We consider an abstract machine similar to an actual computer, but whose registers associated with some types (size, precision, exponent) are unbounded, and an operation on a register takes a unit of time; however, limbs have a fixed size.[10]

The parameters of our model are the length n of the array of MPFR data, the total bit size p_{in} of the significands of the inputs, the bit size p_{out} of the significand of the output (i.e., the target precision), and the bit size w of the exponent field of a MPFR number. The complexity here will not depend on w, but we include this parameter in our model to clearly express this fact: if w were regarded as a constant, there could be a huge constant hidden behind the $O()$ notation, but this is not the case here. For instance, the old algorithm had a 2^w in its worst-case complexity.

The part of the algorithm that takes most of the time in the worst case is in the first call to sum_raw, where wq $= O(p_{\mathrm{out}} + \log n)$; in the second call, we just have wq $= O(\log n)$. Here we have two nested loops as explained in Section 6.3:

- the $O(p_{\mathrm{in}})$ iterations computing the sum in some exponent window, done until the result is accurate enough (each iteration consumes at least one bit of the inputs);
- the $O(n)$ iterations over each summand.

The internal loop contains operations in $O(\mathtt{wq})$, thus in $O(p_{\mathrm{out}} + \log n)$. The loops in other parts of the algorithm are either in $O(n)$ or in $O(\mathtt{wq})$, thus do not increase the complexity obtained from the above nested loops. Therefore the worst-case complexity of this algorithm is in $O(p_{\mathrm{in}} \cdot n \cdot (p_{\mathrm{out}} + \log n))$.

This bound can be reached by using only input numbers with 1-bit precision chosen so that the following occurs at each iteration of the outer loop. From an accumulator containing zero:

1) Add $2^{\mathtt{maxexp}-1}$.
2) Add $-2^{\mathtt{minexp}}$ ($\rightarrow$ long carry propagation).
3) Add $2^{\mathtt{minexp}}$ ($\rightarrow$ long carry propagation).
4) Add $-2^{\mathtt{maxexp}-1}$.

And the accumulator is back to zero for the next iteration. Due to the long carry propagations, the $p_{\mathrm{out}} + \log n$ bound (with a constant factor) is reached in the inner loop. Only 4 bits are consumed at each iteration, so that there will be $p_{\mathrm{in}}/4$ iterations. Thus the time taken is at least some constant times $p_{\mathrm{in}} \cdot n \cdot (p_{\mathrm{out}} + \log n)$.

However, since the parameters are not independent in the above example ($n \geqslant p_{\mathrm{in}}$ and $w \geqslant \log_2(p_{\mathrm{in}}) + \text{constant}$), this does not mean that this bound cannot be improved.

10. The reason is that we express the precision in bits. Alternatively, we could have chosen to express the precision in limbs and let the limb size vary; but this makes less sense due to the use of bit operations such as count_leading_zeros (CLZ), which may be implemented with a loop on some machines and would not take a unit of time.

Note: It is possible to obtain $O(n \cdot \log n + p_{\mathrm{in}} \cdot (p_{\mathrm{out}} + \log n))$ if the inputs are initially sorted by decreasing magnitude and are removed from the list (in constant time) once all their bits have been consumed.

7 Testing

Different kinds of tests are done. First, there are usual generic random tests, with limited precisions and exponent range: the exact sum is computed with basic additions (mpfr_add) with enough precision, then rounded to the target precision, allowing us to check the result of mpfr_sum. Note that this test could be able to detect bugs in either mpfr_add or mpfr_sum; it is very unlikely to get a same wrong result for both computations, because completely different algorithms are used (when the array has at least 3 regular numbers).

As usual, cases involving singular values are also tested. In particular, tests are done with an array of 6 values and every combination of values among NaN, $+\infty$, $-\infty$, $+0$, -0, $+1$ and -1.

We have some specific tests to trigger particular cases in the implementation, the goal being to have a high code coverage. For instance, the sum of 4 numbers $i \cdot 2^{46} + j \cdot 2^{45} + k \cdot 2^{44} + f \cdot 2^{-2}$ with $-1 \leqslant i, j, k \leqslant 1$, $i \neq 0$ and $-3 \leqslant f \leqslant 3$ is tested with the target precision chosen to have the ulp of the exact sum equal to 2^0 or to 2^{44} (all the cases satisfying these conditions are tested).

Code (not enabled by default) has been introduced in the mpfr_sum implementation to be able to check some combined parameter value coverage in the TMD cases, allowing us to make sure that all allowed combinations of rounding mode, tmd value (1 or 2), rbit value, sign of the secondary term and sign of the sum are tested.

We have generic random tests with cancellations. This is done by starting with some array of random numbers, then computing a correctly rounded sum with mpfr_sum, and appending the opposite value to the array, so that the next mpfr_sum call will have cancellations. We reiterate several times.

Finally, we also have tests with underflows and overflows.

We have also done timings on pseudo-random inputs with various sets of parameters: size n $= 10^1$, 10^3 or 10^5; small or large input precision (all the inputs have the same precision precx in these tests); small or large output precision precy; inputs uniformly distributed in $[-1, 1]$, or with scaling by a uniform distribution of the exponents in $[\![0, 10^8[\![$; test of partial cancellation. Comparison has been done with the old implementation and with a basic sum implementation using mpfr_add (thus inaccurate and possibly completely wrong in case of cancellation). Timings can vary a lot between one invocation to another on the same data: factors larger than 3 have sometimes been observed! However, this can be regarded as acceptable since the implementations can differ by larger factors, and we are mostly interested in such big differences. This shows that the new implementation performs incredibly well, being much faster than the old implementation in most cases, except in the pathological cases where precy $\ll$ precx with an important cancellation, where it is much slower due to the reiterations always done in a small precision (this might be solved in the future). In some cases, the new `mpfr_sum` is even much faster than the (inaccurate) basic sum implementation. Sources and timing results are available in the MPFR repository: https://gforge.inria.fr/scm/viewvc.php/mpfr/misc/sum-timings/.

8 Conclusion

We have designed and implemented a new algorithm to compute the correctly rounded sum of several floating-point numbers in radix 2 in arbitrary precision for GNU MPFR, where each number (the inputs and the output) has its own precision. Together with the sum, the sign of the error is returned too.

The description in the paper gives a proof of the algorithm and implementation at some level of details. Since it is almost impossible to guarantee that a proof like that covers everything, the quality of the test suite is important. Various kinds of tests are included in MPFR, and good coverage, in particular combined parameter value coverage in some cases, is checked. Since not all C implementations and not all value combinations can be tested, a formal proof would be useful, but it would have to be expressed in a very low level.

One of the main goals was to make sure that this algorithm is efficient in any corner case. This is particularly important to avoid denial of service in a client-server system. Contrary to the initial algorithm, the worst-case complexity is now polynomial: $O(p_{\text{in}} \cdot n \cdot (p_{\text{out}} + \log n))$ in the model defined in Section 6.6 (similar to word complexity), where p_{in} is the total bit size of the significands of the inputs, n is the length of the array of MPFR data, p_{out} is the bit size of the significand of the output (i.e., the target precision), and the bit size of the exponent field is allowed to vary (the complexity of this algorithm does not depend on it); this bound can be reached on some class of instances.

In future work, one may try to say more about the worst-case complexity. For instance, can the above bound be improved for this algorithm? What other bounds could one get if the parameters are changed (e.g., considering the maximum precision of the input numbers instead of the sum p_{in} of their precisions)?

Future work will also consist in finding real applications to check whether we may want to modify some parameters. For instance, the precision of the accumulator may be increased if need be.

Acknowledgments

The author would like to thank the anonymous reviewers for their valuable comments and suggestions.

References

[1] IEEE, "IEEE standard for floating-point arithmetic," *IEEE Std 754-2008*, 2008. [Online]. Available: http://dx.doi.org/10.1109/IEEESTD.2008.4610935

[2] ——, "IEEE standard for interval arithmetic," *IEEE Std 1788-2015*, 2015. [Online]. Available: http://dx.doi.org/10.1109/IEEESTD.2015.7140721

[3] J. Demmel and Y. Hida, "Fast and accurate floating point summation with application to computational geometry," *Numerical Algorithms*, vol. 37, no. 1–4, pp. 101–112, Dec. 2004. [Online]. Available: http://dx.doi.org/10.1023/B:NUMA.0000049458.99541.38

[4] S. M. Rump, "Ultimately fast accurate summation," *SIAM Journal on Scientific Computing*, vol. 31, no. 5, pp. 3466–3502, 2009. [Online]. Available: http://dx.doi.org/10.1137/080738490

[5] P. Kornerup, V. Lefèvre, N. Louvet, and J.-M. Muller, "On the computation of correctly rounded sums," *IEEE Transactions on Computers*, vol. 61, no. 3, pp. 289–298, Mar. 2012. [Online]. Available: http://dx.doi.org/10.1109/TC.2011.27

[6] S. M. Rump, T. Ogita, and S. Oishi, "Accurate floating-point summation Part II: Sign, K-fold faithful and rounding to nearest," *SIAM Journal on Scientific Computing*, vol. 31, no. 2, pp. 1269–1302, 2008. [Online]. Available: http://dx.doi.org/10.1137/07068816X

[7] U. W. Kulisch, *Advanced Arithmetic for the Digital Computer: Design of Arithmetic Units*. Springer-Verlag, Berlin, 2002.

[8] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torrès, *Handbook of Floating-Point Arithmetic*, 1st ed. Birkhäuser Boston, 2010. [Online]. Available: http://www.springer.com/birkhauser/mathematics/book/978-0-8176-4704-9

[9] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A multiple-precision binary floating-point library with correct rounding," *ACM Transactions on Mathematical Software*, vol. 33, no. 2, Jun. 2007. [Online]. Available: http://doi.acm.org/10.1145/1236463.1236468

[10] V. Lefèvre, "Correctly rounded arbitrary-precision floating-point summation," in *Proceedings of the 23rd IEEE Symposium on Computer Arithmetic*. IEEE, Sep. 2016. [Online]. Available: https://www.vinc17.net/research/papers/arith23.pdf

[11] J. Demmel and H. D. Nguyen, "Fast reproducible floating-point summation," in *Proceedings of the 21st IEEE Symposium on Computer Arithmetic*, A. Nannarelli, P.-M. Seidel, and P. T. P. Tang, Eds. Austin, TX, USA: IEEE Computer Society Press, Los Alamitos, CA, Apr. 2013, pp. 163–172. [Online]. Available: http://dx.doi.org/10.1109/ARITH.2013.9

Vincent Lefèvre received the MSc and PhD degrees in computer science from the École Normale Supérieure de Lyon, France, in 1996 and 2000, respectively. He has been an INRIA researcher at the LORIA, France, from 2000 to 2006, and at the LIP, ENS-Lyon, France, since 2006. His research interests include computer arithmetic. He participated in the revision of the IEEE 754 standard for 2008 and in the standardization of interval arithmetic. He is one of the main GNU MPFR developers.