Correctly Rounded Arbitrary-Precision Floating-Point Summation

Vincent Lefèvre
INRIA, LIP / CNRS / ENS Lyon / Université de Lyon
Lyon, France
Email: vincent@vinc17.net

***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 describe how the implementation is tested.**

***Keywords*-summation; floating point; arbitrary precision; multiple precision; correct rounding;**

I. 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 provides almost no specification. 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 (in a way that is incompatible with IEEE 754).

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 a bit 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`. Due to these constraints, our algorithm is not based on any previous work, even though one can find similar ideas used in a different context such as in [10], which also uses blocks (but differently).

[1]http://www.mpfr.org/ and http://www.mpfr.org/mpfr-3.1.4/mpfr.html

We first give some notation (Section II). In Section III, we present a brief overview of GMP and GNU MPFR. In Section IV, we describe the old `mpfr_sum` implementation and explain why a new one was needed. In Section V, we give the complete specification of the summation operation in MPFR. In Section VI, 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. In Section VII, we explain how `mpfr_sum` is tested. Additional details with figures, the complete code, and timings are given in annex[2].

II. Notation

In addition to ISO 80000-2 notation, we will use $[\![$ and $]\!]$ for the bounds of integer intervals, e.g. $[\![0, 3]\!] = \{0, 1, 2, 3\}$ and $[\![0, 3[\![= \{0, 1, 2\}$.

A minor typography difference will be used between variable names (e.g. `minexp`) and their values (e.g. **`minexp`**).

III. 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.

Each MPFR floating-point number has its own precision in bits, starting at 2. 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,

[2]https://hal.inria.fr/hal-01242127
[3]https://gmplib.org/

with the even rounding rule), MPFR_RNDD (toward $-$Inf), MPFR_RNDU (toward $+$Inf), MPFR_RNDZ (toward zero), MPFR_RNDA (away from zero).

The MPFR numbers are represented with 3 fields: 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, which contains either the exponent or a special value for *singular* datums: zero, infinity, and NaN (the significand contains garbage in such cases); contrary to IEEE 754, MPFR has only a single kind of NaN and does not have subnormals. An important point is that the exponent range can be very large: up to $[\![1 - 2^{62}, 2^{62} - 1]\!]$ on 64-bit machines.

Moreover, most arithmetic operations return a ternary value, giving the sign of the rounding error.

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 implies a huge gap between the minimum value MPFR_EXP_MIN of the type and the minimum valid exponent MPFR_EMIN_MIN.

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 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.

IV. The Old MPFR_SUM Implementation

The implementation of mpfr_sum from the current MPFR releases (up to version 3.1.x) 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.

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).

V. Specification of the Summation Operation

The prototype of the mpfr_sum function is:

```
int mpfr_sum (mpfr_ptr sum,
              mpfr_ptr *const 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. It is currently assumed that no input pointers point to sum, i.e., that the output number sum is not shared with an input.[4]

If $\mathtt{n} = 0$, 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 $\mathtt{n} = 1$: a copy with rounding (mpfr_set);
- if $\mathtt{n} > 1$: 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 $+$Inf and a $-$Inf, 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.

[4]This will be fixed later. This can be handled by using a temporary area for the output when sharing occurs, or in some optimized way.

VI. 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).

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 $[\![\texttt{minexp}, \texttt{maxexp}[\![$, 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.

A. Preliminary Steps

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

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

Now, $\texttt{n} \geqslant 3$. We iterate on 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 will contain the sign of an infinity result. 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 $\texttt{sign\_inf} \neq 0$, then the result is an infinity of this sign, and we return it.
2) If $\texttt{rn} = 0$, then all the inputs are zeros, so that we return the result zero whose sign is given by `sign_zero`.
3) If $\texttt{rn} \leqslant 2$, then one can use `mpfr_set` or `mpfr_add` as an optimization, similarly to what was done for $\texttt{n} \leqslant 2$. 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.

B. Introduction to the Generic Case

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

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 a bit 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 $\mathtt{rn} \cdot 2^{\mathtt{maxexp}}$, taking up to logn extra bits, and one needs one more bit to be able to determine the sign due to two's complement, so that 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.

The precision of the accumulator is defined as:

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

and minexp will always be the exponent of the least significant bit (LSB) of the accumulator. 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, $\mathtt{sum}(2^E, 2^F)$ 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 low precision here. Thus we will define a function sum_raw, used for both computations; it is described in the next section.

C. 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 holes between inputs in case of full cancellation, e.g. $1 + (-1) + r$ where r has a tiny exponent.
- 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 hole 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. First, it takes the following arguments:

- wp: pointer to the accumulator (least significant limb first).
- ws: size of the accumulator (in limbs).
- wq: precision of the accumulator ($\mathtt{ws} \times$ GMP_NUMB_BITS).
- 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).
- tp: pointer to a temporary area (pre-allocated).
- ts: size of this temporary area.
- 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 0), otherwise the number of cancelled bits, defined as the number of identical bits on the most significant part of the accumulator. In the latter case, it also returns the following data in variables passed by reference (via pointers):

- for ep: the exponent e of the computed result;
- for minexpp: the last value of the variable minexp;
- for maxexpp: the new value of the variable maxexp (the one for a potential new iteration).

The temporary area must be large enough to hold a shifted input block, and the value of ts is used only when the full assertions are checked, in order to make sure that a buffer overflow does not occur.

Some notation used below:

- $\mathtt{E}(\mathtt{v})$: the exponent of a MPFR number v.
- $\mathtt{P}(\mathtt{v})$: the precision 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, $\mathtt{maxexp2} = \mathtt{MPFR\_EXP\_MIN}$ 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 $\mathtt{E}(\mathtt{x[i]}) \leqslant \mathtt{minexp}$, then no bits of x[i] need to be considered here. We set the maxexp2 variable to $\max(\mathtt{maxexp2}, \mathtt{E}(\mathtt{x[i]}))$, then go to the next input.
3) Now, we have: $\mathtt{E}(\mathtt{x[i]}) > \mathtt{minexp}$. If the tail of x[i] is not empty, i.e., if $\mathtt{E}(\mathtt{x[i]}) - \mathtt{P}(\mathtt{x[i]}) < \mathtt{minexp}$, then we set the maxexp2 variable to minexp.
4) We prepare the input for the accumulation. First, this means that 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), storing the result to the temporary area at address tp.
5) If x[i] is positive: an addition with carry out is done with mpn_add_n; if the most significant limb needs to be masked, then it is not taken into account in the addition, but the masked part is just added to the carry; carry propagation is done with mpn_add_1 if the size of the destination is larger than the size of the block.
   Note: 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 for the subtraction and mpn_sub_1 to propagate borrows.

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, defined as the number of consecutive identical bits starting with the most significant one in 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 $\mathtt{maxexp2} = \mathtt{MPFR\_EXP\_MIN}$ (meaning no more tails), then we return 0, otherwise we reiterate at the beginning of sum_raw with minexp set to $\mathtt{cq} + \mathtt{maxexp2} - \mathtt{wq}$ 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. 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: $\mathtt{e} = \mathtt{maxexp} + \mathtt{cq} - \mathtt{cancel} = \mathtt{minexp} + \mathtt{wq} - \mathtt{cancel}$ and $\mathtt{err} = \mathtt{maxexp2} + \mathtt{logn}$.

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 $\mathtt{maxexp2} = \mathtt{MPFR\_EXP\_MIN}$ (meaning no more tails), then the error is 0.

We need $\mathtt{prec} \geqslant 1$ to be at least able to determine the sign of the result, hence this precondition.

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.

We now need to determine the value of shiftq. We prefer it to be as large as possible: 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., $\mathtt{minexp} - \mathtt{shiftq} < \mathtt{maxexp2}$, i.e., $\mathtt{shiftq} > \mathtt{minexp} - \mathtt{maxexp2}$;
(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 the double inequality above is a weak form of what is actually required in (A) and (B).

Since we prefer shiftq to be maximum, we focus on (B) first. The absolute value of the accumulator at the end of the next iteration will be strictly bounded by: $2^{\mathtt{e}} + 2^{\mathtt{err}} \leqslant 2^{\mathtt{e}+1+\max(0,\mathtt{err}-\mathtt{e})}$. This means that if we do not shift the value in the accumulator, then at the end of the next iteration, the accumulator will contain at least $\mathtt{cancel}-1-\max(0,\mathtt{err}-\mathtt{e})$ identical bits on its most significant part. Only the last of these bits is needed (which gives the sign) and the other ones are redundant. Therefore, in order to satisfy (B), we can choose:

$$\mathtt{shiftq} = \mathtt{cancel} - 2 - \max(0, \mathtt{err} - \mathtt{e}).$$

Now, let us prove that for this value, (A) is satisfied, using the fact that $\mathtt{wq} \geqslant \mathtt{logn} + \mathtt{prec} + 2$ on input.

- If $\mathtt{err} - \mathtt{e} \geqslant 0$, then by using $\mathtt{err} = \mathtt{maxexp2} + \mathtt{logn}$ and $\mathtt{e} = \mathtt{minexp} + \mathtt{wq} - \mathtt{cancel}$, we obtain:

$$\begin{aligned} \mathtt{shiftq} &= \mathtt{cancel} - 2 - (\mathtt{err} - \mathtt{e}) \\ &= \mathtt{minexp} - \mathtt{maxexp2} + (\mathtt{wq} - \mathtt{logn} - 2) \\ &> \mathtt{minexp} - \mathtt{maxexp2} \end{aligned}$$

- If $\mathtt{err} - \mathtt{e} < 0$, then this is the case where the error can be potentially small: to be able to prove the inequality, we need to use the fact that the stop condition was not satisfied, i.e., $\mathtt{e} - \mathtt{err} < \mathtt{prec}$. Thus $(\mathtt{minexp} + \mathtt{wq} - \mathtt{cancel}) - (\mathtt{maxexp2} + \mathtt{logn}) < \mathtt{prec}$, and as a consequence:

$$\begin{aligned} &\mathtt{shiftq} - (\mathtt{minexp} - \mathtt{maxexp2}) \\ &= \mathtt{cancel} - 2 - (\mathtt{minexp} - \mathtt{maxexp2}) \\ &> \mathtt{wq} - \mathtt{logn} - \mathtt{prec} - 2 \geqslant 0 \end{aligned}$$

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.

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

Memory is allocated both for the accumulator and for the temporary area needed by sum_raw. For performance reasons, the allocation is done in the stack if the size is small enough. No other memory allocation will be needed (except for automatic variables).

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.

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

When maxexp (value returned by sum_raw) is different from 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. We compute $\mathtt{d} = \mathtt{u} - \mathtt{err}$, which is larger or equal to 3 (see above) and can be very large if maxexp is very small; nevertheless, it can be proved that d is representable in a mpfr_exp_t. The TMD occurs when the sum is close enough to a breakpoint, which 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

(in the second case, 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 to the destination (significand of sum);
- 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 (6.1.0).

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.

We start by doing the first two operations at the same time: the bits of exponents in $[\![\max(\mathtt{u}, \mathtt{minexp}), \mathtt{e}[\![$ are copied with a possible shift to the most significant part of the destination, and the least significant limbs (if any) are zeroed.

By definition of `e`, the most significant bit that is copied is the complement of the value of the cancelled bits. A variable `pos` will contain its value, i.e., $\mathtt{pos} = 1$ for a positive number, $\mathtt{pos} = 0$ for a negative number.

The values of three variables are also determined at about the same time:

- `inex`: 0 if the final sum is *known* to be exact, else 1.
- `rbit`: the rounding bit (0 or 1) of the truncated sum, corrected to 0 for halfway cases that round downward if rounding is to nearest (so that this bit gives the rounding direction).
- `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.

All this is done by considering two cases: $\mathtt{u} > \mathtt{minexp}$ and $\mathtt{u} \leqslant \mathtt{minexp}$. Details are not given in the paper.

Then we seek to determine how the value will be rounded, more precisely, what correction will be done to the significand that has been copied just above. We currently have a significand, a trailing term t in the accumulator (bits whose exponent is in $[\![\mathtt{minexp}, \mathtt{u}[\![$) such that $0 \leqslant t < 1\,\mathrm{ulp}$ (nonnegative thanks to the two's complement representation), and an error on the trailing term bounded by $t' \leqslant 2^{\mathtt{u}-3} = 2^{-3}\,\mathrm{ulp}$ in absolute value, so that the error ε on the significand satisfies $-t' \leqslant \varepsilon < 1\,\mathrm{ulp} + t'$. 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: same as nextDown;	$+1$: same as nextUp;
0: no correction;	$+2$: same as two 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 (this check is common to almost all MPFR functions).

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

- $\mathtt{tmd} = 0$. The TMD does not occur, so that the error has no influence on the rounding and the ternary value (one can assume $t' = 0$). In the directed rounding modes, one currently has: $\mathtt{inex} = 0$ if and only if $t = 0$ (from the various cases not detailed in the paper). Therefore `inex` is the absolute value of the ternary value, and we set `corr` as follows:
  - for `MPFR_RNDD`, $\mathtt{corr} = 0$;
  - for `MPFR_RNDU`, $\mathtt{corr} = \mathtt{inex}$;
  - for `MPFR_RNDZ`, $\mathtt{corr} = \mathtt{inex}$ && !`pos`;
  - for `MPFR_RNDA`, $\mathtt{corr} = \mathtt{inex}$ && `pos`;
  - for `MPFR_RNDN`, $\mathtt{corr} = \mathtt{rbit}$.

  We now correct the sign of the ternary value: if $\mathtt{inex} \neq 0$ (i.e., $\mathtt{inex} = 1$) and $\mathtt{corr} = 0$, we set `inex` to -1.
- $\mathtt{tmd} \neq 0$. The TMD occurs and will be resolved by determining the sign (-1, 0 or $+1$) of a secondary term thanks to a second `sum_raw` invocation with a low-precision accumulator. Since the uncorrected significand has already been copied from the accumulator to the destination, we can reuse the memory of the accumulator, but its precision is now set to $\mathtt{cq} + \mathtt{dq}$ rounded up to the next multiple of the limb size (`GMP_NUMB_BITS`). There may remain some bits that have not been copied, but they will be taken into account as described below.

  Let us recall that the $\mathtt{d} - 1$ bits from exponent $\mathtt{u} - 2$ to $\mathtt{u} - \mathtt{d}$ (= `err`) are identical. We distinguish two subcases:
  - $\mathtt{err} \geqslant \mathtt{minexp}$. The last two over the $\mathtt{d} - 1$ identical bits and the following bits, i.e., the bits from $\mathtt{err} + 1$ to `minexp`, are shifted to the most significant part of the accumulator.
  - $\mathtt{err} < \mathtt{minexp}$. 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 `maxexp`, with `cq` bits reserved to avoid overflows, just like in the main sum.

  Once the accumulator is set up, `sum_raw` is called with $\mathtt{prec} = 1$, satisfying the first `sum_raw` precondition ($\mathtt{prec} \geqslant 1$). And we have:

  $$\mathtt{wq} \geqslant \mathtt{cq} + \mathtt{dq} \geqslant \mathtt{logn} + 3 = \mathtt{logn} + \mathtt{prec} + 2,$$

  corresponding to the second `sum_raw` precondition. This allows us to get the correction case `corr` and the ternary value `inex` (details are not given in the paper).

We now distinguish the two cases $\mathtt{pos} = 1$ (positive sum) and $\mathtt{pos} = 0$ (negative sum), to set the sign of the MPFR number here and update the significand field to its final contents: rounding based on the correction case `corr`, change of representation at the same time in the negative case, and clear the trailing bits. One can show that in the positive case, this corresponds to an operation of the form $x + \mathtt{corr}$ on the significand field, and in the negative case, $\overline{x} + (1 - \mathtt{corr})$ where $\overline{x}$ is the complement; GMP does not provide such a

composed complement with addition/subtraction of a limb (or similar operation), but we can do this efficiently. The only correction to do in case of the change of binade[5] is to set the MSB of the significand to 1 and correct the variable `e`.

Finally, we set the exponent of the MPFR number to `e`, and check the range with `mpfr_check_range`.

VII. 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, $+$Inf, $-$Inf, $+0$, -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 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). 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.

[5]This can be detected with a carry/borrow out of a GMP operation.

VIII. 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 part of a proof. Since it is almost impossible to check 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. More about this will be said in future work (there are various ways to express the complexity here).

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.

References

[1] IEEE, "IEEE standard for floating-point arithmetic," *IEEE Std 754-2008*, 2008.

[2] ——, "IEEE standard for interval arithmetic," *IEEE Std 1788-2015*, 2015.

[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.

[4] S. M. Rump, "Ultimately fast accurate summation," *SIAM Journal on Scientific Computing*, vol. 31, no. 5, pp. 3466–3502, 2009.

[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.

[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.

[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.

[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.

[10] 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.