

Correctly Rounded Arbitrary-Precision Floating-Point Summation

Vincent LEFÈVRE

AriC, Inria Grenoble – Rhône-Alpes / LIP, ENS-Lyon

RAIM 2016, Banyuls-sur-Mer, 2016-06-28

Introduction to GNU MPFR

Goal: complete rewrite of the `mpfr_sum` function for the future GNU MPFR 4.

GNU MPFR in a few words:

- An efficient *multiple-precision floating-point* library with *correct rounding* (and signed zeros, infinities, NaN, and exceptions, but no subnormals).
- Radix 2. Each number has its *own precision* ≥ 1 (or 2 before MPFR 4).
- 5 rounding modes: nearest-even; toward $-\infty$, $+\infty$, 0; away from zero. The functions return the sign of the error: *ternary value*.

About the GNU MPFR internals:

- Based on GNU MP, mainly the low-level *mpn* layer. A multiple-precision natural number: array of 32-bit or 64-bit integers, called *limbs*.
- Representation of a floating-point number with 3 fields: sign, significand (array of limbs, with value in $[1/2, 1[)$, exponent in $\llbracket 1 - 2^{62}, 2^{62} - 1 \rrbracket$. Special data represented with special values in the exponent field.

`mpfr_sum`: *correctly rounded sum* of N numbers ($N \geq 0$).

The Old `mpfr_sum` Implementation

Demmel and Hida's accurate summation algorithm + Ziv loop.

MPFR 3.1.3 [2015-06] and earlier: `mpfr_sum` was buggy with different precisions.
Reference here: trunk r8851 / MPFR 3.1.4 [2016-03] (latest release).

Performance issues:

- The working precision must be the same for all inputs and the output. → The maximum precision had to be chosen as the base precision (bug fix).
- The exact result may be very close to a *breakpoint*. Uncommon case, but... Large exponent range → *critical issue* (e.g., crashes due to lack of memory).
- High-level for MPFR (`mpfr_add` calls). → Prevents good optimization.

Specification (behavior) issues:

- The sign of an exact zero result is not specified.
- The *ternary value* is valid only when zero is returned: for some exact results, one knows that they are exact, otherwise one has no information.

The New `mpfr_sum` Algorithm and Implementation

Goals:

- As fast as possible. In particular, the exponent range should no longer matter.
→ Low level (*mpn*), based on the representation of the numbers.
- Completely specified. Exact result 0: same sign as a succession of binary +.

Basic ideas: [r10503, 2016-06-24]

- 1 Handle **special inputs** (NaN, infinities, only zeros, $N_{\text{regular}} \leq 2$). Otherwise:
- 2 **Single memory allocation** (stack or heap): accumulator, temporary area...
- 3 Fixed-point **accumulation** by blocks in some window $[[\text{minexp}, \text{maxexp}[[$ (re-iterate with a shifted window in case of cancellation): `sum_raw`.
Done in two's complement representation.
- 4 If the **Table Maker's Dilemma** (TMD) occurs, then compute the sign of the error term by using the same method (`sum_raw`) in a low precision.
- 5 **Copy/shift** the truncated result to the destination (**normalized**).
- 6 **Convert** to sign + magnitude, with **correction term** at the same time.

The New `mpfr_sum`: An Example

Just an example (not the common case), covering most issues (cancellations. . .).

Simplification for readability:

- Small blocks (may be impossible in practice: the accumulator size is a multiple of the limb size, i.e. 32 or 64).
- The numbers are ordered (in the algorithm, there are loops over all the numbers and the order does not matter).
- We will not show the accumulator, just what is computed at each step.

The New `mpfr_sum`: An Example [2]

`MPFR_RNDN` (roundTiesToEven), output precision `sq = 4`.

$N_{\text{regular}} = 10$ input numbers, each with its own precision:

$$\begin{array}{rcl} x_0 & = & +0.1011101000010 \cdot 2^0 & + & 1011101000010 \\ x_1 & = & -0.10001 \cdot 2^0 & - & 10001 \\ x_2 & = & -0.11000011 \cdot 2^{-2} & - & 11000011 \\ x_3 & = & -0.11101 \cdot 2^{-8} & - & 11101 \\ x_4 & = & -0.11010 \cdot 2^{-9} & - & 11010 \\ x_5 & = & +0.10101 \cdot 2^{-1000} & & \\ x_6 & = & +0.10001 \cdot 2^{-2000} & & \\ x_7 & = & -0.10001 \cdot 2^{-2000} & & \\ x_8 & = & -0.10000 \cdot 2^{-3000} & & \\ x_9 & = & +0.10000 \cdot 2^{-4000} & & \end{array}$$

The New `mpfr_sum`: An Example [3]

First iteration: $\llbracket \text{minexp}, \text{maxexp} \rrbracket = \llbracket -8, 0 \rrbracket$ (`maxexp`: chosen from the maximum exponent; `minexp`: chosen from various parameters, see details later).

Only 3 input numbers are concerned:

$$\begin{array}{r} \quad \quad \quad \lrcorner \text{ minexp} = -8 \\ + 10111010 \text{ [00010]} \\ - 10001 \\ - \quad 110000 \text{ [11]} \\ \hline \dots 000000010 \quad \quad \quad \text{(If the signs were reversed: } \dots 111111110, e = -7) \\ \quad \quad \quad \lrcorner e = -6 \end{array}$$

During the same loop over all the input numbers, we compute the next `maxexp`: Let $\mathcal{T} = \{i : Q(x_i) < \text{minexp}\}$, where $Q(x)$ is the exponent of the last bit of x , be the indices of the inputs that have not been fully taken into account. Then

$$\text{maxexp2} = \sup_{i \in \mathcal{T}} \min(E(x_i), \text{minexp}) = \text{minexp} = -8.$$

The New `mpfr_sum`: An Example [4]

We have computed an approximation to the sum and we have an error bound:

$N_{\text{regular}} \cdot 2^{\text{maxexp}2}$, or 2^{err} with $\text{err} = \text{maxexp}2 + \lceil \log_2(N_{\text{regular}}) \rceil$.

The accuracy test is of the form: $e - \text{err} \geq \text{prec}$, where `prec` is (currently) $\text{sq} + 3 = 7$. Here, $e - \text{err} = (-6) - (-8) - \lceil \log_2(N_{\text{regular}}) \rceil \leq 0 < \text{prec}$.

→ We need at least another iteration.

Second iteration: $\llbracket \text{minexp}, \text{maxexp} \rrbracket = \llbracket -17, -8 \rrbracket$.

```
...0010          ← previous sum (shifted in the accumulator)
+   00010
-    11
-   11101
-   11010
-----
...0000000000000
```

Full cancellation (here with a big gap: $\text{maxexp}2 = -1000 \ll \text{minexp}$).

→ New iteration with $\text{maxexp} := \text{maxexp}2$ just like in the first iteration.

The New `mpfr_sum`: An Example [5]

The next and last 5 input numbers:

$$x_5 = +0.10101 \cdot 2^{-1000}$$

$$x_6 = +0.10001 \cdot 2^{-2000}$$

$$x_7 = -0.10001 \cdot 2^{-2000}$$

$$x_8 = -0.10000 \cdot 2^{-3000}$$

$$x_9 = +0.10000 \cdot 2^{-4000}$$

Third iteration: $[\text{minexp}, \text{maxexp}] = [-1008, -1000]$.

Truncated sum = $x_5 = +0.10101 \cdot 2^{-1000}$.

$e - \text{err} = (-1000) - (-2000) - 4 \geq 7 = \text{prec}$, so that the truncated sum is accurate enough, but it is close to a *breakpoint* (midpoint): TMD.

To solve the TMD:

- Do *not* increase the precision (as usually done for the elementary functions), due to potentially huge gaps (here between x_5 and x_6).
- Instead, determine the sign of the “error term” by computing this term to 1-bit target precision, using the same method ($\text{prec} = 1$).

The New `mpfr_sum`: An Example [6]

The input numbers used for the error term:

$$x_6 = +0.10001 \cdot 2^{-2000}$$

$$x_7 = -0.10001 \cdot 2^{-2000}$$

$$x_8 = -0.10000 \cdot 2^{-3000}$$

$$x_9 = +0.10000 \cdot 2^{-4000}$$

First iteration of the TMD resolution: full cancellation between x_6 and x_7 .

Second iteration of the TMD resolution: x_8 ; accurate enough \rightarrow negative.

Correctly rounded sum = $+0.1010 \cdot 2^{-1000}$.

Technical note: 2 cases to initiate the TMD resolution.

- Here, the gap between the breakpoint and the remaining bits is large enough. We start with a zeroed new accumulator.
- But a part of the error term may have already been computed in the lower part of the accumulator. In such a case, the new accumulator is initialized with some of these bits.

The New `mpfr_sum`: Accumulation (`sum_raw`)

To implement the steps presented in the example (before rounding)...

Function for accumulation: `sum_raw`

Computes a truncated sum in an accumulator such that if the exact sum is 0, return 0, otherwise satisfying $e - \text{err} \geq \text{prec}$, where e is the exponent of the truncated sum.

Calls of `sum_raw`:

- Main approximation: $\text{prec} = \text{sq} + 3$; zeroed accumulator in input.
- TMD resolution, if necessary: $\text{prec} = 1$ (only the sign of the result is needed); the accumulator may be zeroed or initialized with some of the lowest bits from the main approximation.

The New `mpfr_sum`: Accumulation (`sum_raw`) [2]

The **accumulator**, for the first iteration:

- $cq = \lceil \log_2(N_{\text{regular}}) \rceil + 1$ bits for the sign bit and to avoid overflows.
- sq bits: output precision.
- $dq \geq \lceil \log_2(N_{\text{regular}}) \rceil + 2$ bits to take into account truncation errors.

Example of first iteration and after a partial cancellation (\rightarrow shift):

```

      [-----]-----]
           cq  |----- maxexp      sq + dq      minexp |-----]
Before shift: 000000000000000000000000000000001-----]
<--- identical bits (0) --->
                               <----- 26 zeros ----->
After shift:  001-----]00000000000000000000000000000000
This iteration:           minexp |-----| maxexp2
Next iteration:           |-----| maxexp      minexp |-----]
```

`maxexp2`: maximum exponent of the *tails* (`MPFR_EXP_MIN` if no tails).

The New `mpfr_sum`: Correction (in short)

We now have 3 terms: the `sq`-bit truncated significand S , a trailing term t in the accumulator such that $0 \leq t < 1 \text{ ulp}$, and an error on the trailing term.

→ The error ε on S is of the form: $-2^{-3} \text{ ulp} \leq \varepsilon < (1 + 2^{-3}) \text{ ulp}$.

4 correction cases, depending on ε (from t and possibly a TMD resolution), the sign of the significand, the rounding bit, and the rounding mode:

$$\text{corr} = \begin{cases} -1 & : \text{ equivalent to nextDown} \\ 0 & : \text{ no correction} \\ +1 & : \text{ equivalent to nextUp} \\ +2 & : \text{ equivalent to 2 consecutive nextUp} \end{cases}$$

This is done *efficiently* with:

- For $\text{sq} \geq 2$, one-pass operation on the two's complement significand:
 - ▶ For positive results: $x + \text{corr}$.
 - ▶ For negative results: $\bar{x} + (1 - \text{corr})$.

In case of change of binade, just set the MSB to 1 and correct the exponent.

- For $\text{sq} = 1$, specific code (but trivial).

Tests

Tests needed to detect various possible issues:

- unnoticed error in the pen-and-paper proof (complex due to many cases);
- coding error, such as typos (without a full formal proof of MPFR);
- bug in MPFR, such as internal utility macros (this did happen: r9295);
- bug in compilers;

and to check that some bounds in the pen-and-paper proof are optimal.

Different kinds of tests, including:

- Special values (e.g., with combinations of NaN, infinities and zeroes).
- Specific tests to trigger particular cases in the implementation. Comparison with the sum computed exactly with `mpfr_add` then rounded.
- Generic random tests with cancellations (no full check, though).
- Tests with underflows and overflows.
- Check for value coverage in the TMD cases to make sure that the various combinations have occurred in the tests (this could be improved).

Timings

Comparison of 3 algorithms:

- **sum_old**: `mpfr_sum` from MPFR 3.1.4 (old algo).
- **sum_new**: `mpfr_sum` from the trunk patched for MPFR 3.1.4 (new algo).
- **sum_add**: basic sum implementation with `mpfr_add` (inaccurate and sensitive to the order of the inputs).

Random inputs with various sets of parameters:

- array size $n = 10^1, 10^3$ or 10^5 ;
- small or large input precision `precx` (the same one);
- small or large output precision `precy`;
- inputs uniformly distributed in $[-1, 1]$, or with scaling by a uniform distribution of the exponents in $[[0, 10^8]]$;
- partial cancellation or not.

Timings [2]

Inaccurate timings (up to a factor 3 between two calls), but we focus on much larger factors (theoretically unbounded).

Conclusion:

- `sum_new` vs `sum_add`:
 - ▶ sometimes slower, due to the accuracy requirements;
 - ▶ sometimes faster, as low level and low significant bits may be ignored.
- `sum_new` vs `sum_old`:
 - ▶ much faster in most cases;
 - ▶ much slower in some pathological cases: $\text{prec}_y \ll \text{prec}_x$ and there is a cancellation, due to the fact that the reiterations are always done in a low precision (assuming that a reiteration would stop with a large probability).
Change in the future?

Sources and results are provided in the MPFR repository:

<https://gforge.inria.fr/scm/viewvc.php/mpfr/misc/sum-timings/>

Conclusion and Future Work

Major improvements over the old algorithm and implementation:

- Much faster in most tested cases (application dependent, though).
- Much less memory in some cases (no more crashes in simple cases).
- Fully specified, with ternary value (as usual).

Temporary memory: twice the output precision + a few limbs.

For the next MPFR release: GNU MPFR 4.0.

Possible future work:

- Determine a worst-case time complexity (could be pessimistic).
- Bad cases could be improved, but this could slow down the average case.
- What is the average case? Too much context dependent.
→ Based on real-world applications?