2007

TR-2007010: Error-Free Computations via Floating-Point Operations

V. Y. Pan

B. Murphy

G. Qian

R. E. Rosholt

Error-free Computations via Floating-Point Operations *

V. Y. Pan[1,3], B. Murphy[1], G. Qian[2], R. E. Rosholt[1]
[1] Department of Mathematics and Computer Science,
Lehman College, The City University of New York,
Bronx, NY 10468 USA
victor.pan/brian.murphy/rhys.rosholt@lehman.cuny.edu
[2] Ph.D. Program in Computer Science,
The City University of New York,
New York, NY 10036 USA
gqian@gc.cuny.edu
[3] http://comet.lehman.cuny.edu/vpan/

Abstract

Division-free arithmetic computations can be boiled down to summation due to Dekker/Veltkamp's algorithm of 1971. The known double-precision numerical algorithms for summation are highly effective but limited by rounding errors. Our new summation algorithms relaxe this limitation, although they still almost entirely amount to double-precision additions. The efficiency of the algorithms is confirmed by our analysis and extensive tests.

Key words: Error-free summation, floating-point additions.

1 Introduction

Floating-point summation is a basic operation in scientific computing. It has been extensively studied, e.g., in [H02], [LDB02], [DH03], [ORO05], [ORO05a], and the references therein. The paper [D71] expresses products via sums, and so division-free arithmetic computations boil down to summation. Approximation of reciprocals and error analysis extend this domain to approximate rational computations.

Numerical computations presently outperform symbolic computations but are limited by rounding errors. In some applications this is too restrictive. In particular our motivation came from the papers [Pa], [Pna], [PMQa], where we had to operate with

*Supported by PSC CUNY Awards 66437-0035 and 67297-0036

the sums $s_1 + \cdots + s_h$ that are nonzero but nearly vanish versus $\max_j |s_j|$. Moreover, in some cases we need to compute such sums error-free.

Our two new algorithms for error-free summation consist almost entirely of floating-point additions. Our experimental code for Algorithm 5.1 was proved effective in application to computing determinants in [PMQa]. This code is due to all coauthors. Otherwise the present paper (like [Pa], [Pna], [PMQa]), together with all its errors and typos, is the sole responsibility of the first author.

We organize our presentation as follows. After some preliminaries in the next section, we present a summation algorithm that combines the known floating-point summations and the symbolic technique of real modular reduction (see Section 3). In Section 4–6 we cover two numerical variations of this algorithm based on Dekker's splitting in [D71]. In Sections 7 we estimate the arithmetic cost of our summation and point out a heuristic recipe for saving the memory space.

2 Preliminaries and the background

Assume the IEEE standard representation of floating-point number as $\sigma 2^e f$. Here σ is equal to -1 or one and e is an integer in a fixed range $[1-r, r]$ for a fixed natural r. f is either zero or a binary number in the range $[0.5, 1)$ represented with $p+1$ bits. In particular $r = 127$ and $p = 23$ for the single precision IEEE standard floating-point numbers and $r = 1023$ and $p = 52$ for the double precision IEEE standard floating-point numbers. The leftmost bit is fixed to equal one.

We write $\epsilon = 1/2^{p+1}$, $s = s_1 + \cdots + s_h$, $s_+ = |s_1| + \cdots + |s_h|$, and $s^* = \mathrm{fl}(s_1 + \cdots + s_h)$ assuming that the sum s is computed by some customary floating-point summation algorithms [H02], for which

$$|s^* - s| \le \delta_s = c\epsilon s_+ \tag{2.1}$$

for a constant $c < 2$ (acctually for c close to one).

In the case where $|s| \ll s_+$ the latter error bound can be too large, but it is always sufficient to compute the sum s within the error bound $f|s|$ for $f < 1/2$ because (see Section 6) this can be readily extended to computing the sum s with anyfixed precision, which allows error-free output if s has finite precision. For this task we must generally allow other operations besides the floating-point additions, but we minimize and simplify their use.

3 Solution with real modular reduction

We first describe our algorithm by using the *real modular reduction* from [P92] (cf. also [EPY98]). This symbolic technique may be of independent value. Namely, for real s and $t \neq 0$, we write $s \mod t$ to denote a unique real q such that t divides $s - q$, $|q|$ is minimum, and $q \neq t/2$, so that $(t/2) \mod t = -t/2$. We have the following simple fact.

Fact 3.1. *For three real numbers a, b, and $t \neq 0$, we have*

$$(a \mod t)(\mathrm{rop})(b \mod t) \mod t = (a(\mathrm{rop})b) \mod t$$

where (rop) denotes any ring operation, that is addition, subtraction, or multiplication.

Now, suppose the floating-point summation of h summands $s_1, \dots, s_h$ has been performed with the precision of $p+1$ bits for $h \ll 2^{p+1}$ and has output an approximation s^* to the sum s within an error bound $\delta_s \geq |s^*|/2$. Clearly, this can only occur where $|s^*| \ll \max_j |s_j|$. Now we can reduce the summands modulo 2^d for

$$d = 3 + \lfloor \log_2 \tilde{s} \rfloor, \ \tilde{s} \geq |s^*| + \delta_s \tag{3.1}$$

and then repeat the computations. $1 + \lfloor \log_2 \tilde{s} \rfloor$ is the exponent of the floating-point representation of the number $\tilde{s}$, and so $|s| \leq |\tilde{s}| < 2^{d-2}$. Therefore we can readily recover the sum s from $s \mod 2^d$, that is we can reduce the original summation of the summands s_j to the summation of their residues modulo 2^d. The gain is the decrease of the values $\max_j |s_j|$, s_+, and δ_s.

We repeat the process recursively until we obtain the sum s within the error bound $\delta_s < |s^*|/2$. Then we can readily yield as many correct output bits as we wish or even all bits of s (see Section 6).

The resulting summation algorithm still performs only floating-point operations except that it uses the real modular reduction and periodically (and rarely) accesses the exponent d in the floating-point binary representation of an upper estimate $\tilde{s}$ for the absolute value of the current approximation to the sum s (cf. (3.1)). In the next sections we describe two distinct ways for replacing modular reduction by floating-point operations.

Let us comment on accessing the exponents of floating point numbers. This operation can actually be inexpensive. The IEEE floating point standard defines the function *logb(x)* to extract the significand and exponent of a floating point number. Floating point units (FPUs) in Intel's Pentium processor family provide hardware implementations of an instruction, FXTRACT, offering a superset of the *logb(x)* functionality [I01]. For double precision floating point numbers, the FPU of the Pentium 4 processor can execute the FXTRACT instruction in 12 cycles [F04] (almost three times as fast as the same FPU handles division). Because FXTRACT is a floating point instruction, the FPU can overlap the early cycles of FXTRACT with late cycles of various other floating point instructions when they immediately precede FXTRACT, thereby allowing further speed up [F04].

4 Dekker-like modular reduction

To implement the same approach semi-numerically, we can perform real modular reduction by adapting Dekker's basic algorithm [D71] for splitting a floating-point number into two parts. We use the Matlab-like notation [Matlab04] and assume that g is an integer, $0 < g \leq p$.

Algorithm 4.1. Splitting of a floating-point number into two parts *[D71].*

$$\begin{aligned} \text{function}[x, y] &= \text{Split}(a) \\ c &= \text{fl}(\text{factor} \cdot a) = \text{fl}(2^g a + a) \qquad \% \quad \text{factor} = 2^g + 1 \\ x &= \text{fl}(c - (c - a)) \\ y &= \text{fl}(a - x) \end{aligned}$$

The numbers x and y have shorter precision and satisfy the equation $a = x + y$. Under the common assumption that $0 \le \lceil p/2 \rceil - g \le 1$, these are the half-precision numbers.

Now we present two simple algorithms that both extend Dekker's algorithm to computing the residue $y = a \mod 2^g$ and the respective leading part $x = a - y$ for a given binary number a and an integer g. The first algorithm works where rounding is performed by chopping off the extraneous trailing bits of the number. In the second algorithm we assume rounding to the nearest value. Correctness of both algorithms is readily verified.

Algorithm 4.2. Binary mod (with chopping).

$$\begin{aligned} \text{function}[x, y] &= \text{Split}(a) \\ c &= \text{fl}(\text{factor} \cdot a) = \text{fl}(2^g a - a) \qquad \% \quad \text{factor} = 2^g - 1 \\ x &= \text{fl}((c + a) - c) \\ y &= \text{fl}(a - x) \end{aligned}$$

Algorithm 4.3. Binary mod (with rounding to the nearest value). *First replace the input value a with $a - \sigma 2^{e-g}$, and then apply Algorithm 4.1 or 4.2.*

In Algorithm 4.3 we must access σ, that is the sign of the input value a. Other than that, both of the latter algorithms only use the same operations as Algorithm 4.1, that is multiplication by 2^g and the floating-point additions/subtractions.

5 Alternative computation of the leading bit of the sum

In this section we approximate the sum of h summands s_j, $j = 1, \ldots, h$, strictly within the relative error $1/2$. In the next section we readily extend this to error-free computation of the sum. We still periodically (and rarely) access the exponent d of equation (3.1), but now we avoid using modular reduction.

Algorithm 5.1. Computation of the leading bit of a sum.

INPUT: *A positive integer h, a sufficiently large precision $1 + p$ such that*

$$1 + p > \log_2(h + v) \tag{5.1}$$

for a positive parameter v specified in Remark 5.1, floating-point binary numbers $s_i = \sigma_i 2^{e_i} f_i$ for $i = 1, \ldots, h$, and a black box subroutine FLOAT·SUM, which uses only floating-point operations to compute the sum of i numbers $t_1, \ldots, t_i$ and an output error bound $\delta_t = c\epsilon t_+$ for $t_+ = |t_1| + \cdots + |t_i|$, $\epsilon = 2^{p+1}$, and a constant $c < 2$.

OUTPUT: *a floating-point number s and a positive δ_s such that*

$$|s - (s_1 + \cdots + s_h)| \le \delta_s, \tag{5.2}$$

$$\delta_s \le |s|/2. \tag{5.3}$$

COMPUTATIONS:

1. *Apply the Subroutine FLOAT·SUM to compute the values s^* and s^*_+ where s^* approximates the sum $s = s_1 + \cdots + s_h$ and s^*_+ closely approximates the sum $|s_1| + \cdots + |s_h|$ from above. Also compute the values $\delta_s = c\epsilon s^*_+$, satisfying (5.2), and $\tilde{s}$ closely approximating the value $|s| + \delta_s$ from above.*
2. *If bound (5.3) holds, output the values s and δ_s and stop. Otherwise compute the integer d defined by equation (3.1).*
3. *Apply Dekker's Algorithm 4.1 for $a = s_j$ and $g = p + 1 - (e_j - d)$ to obtain the leading part x_j of the summand s_j and its trailing part y_j such that $s_j = x_j + y_j$ for $j = 1, \ldots, h$.*
4. *Apply the Subroutine FLOAT·SUM to compute a binary number x approximating the sum $\sum_{j=1}^h x_j$. Write $s_j \leftarrow y_j$, $j = 1, \ldots, h$, $s_{h+1} \leftarrow x$, and $h \leftarrow h + 1$.*
5. *Go to Stage 1.*

To prove correctness of the algorithm, we first show that the sum $x_1 + \cdots + x_h$ is computed error-free at Stage 4. The rounding error $\delta_x = |x - (x_1 + \cdots + x_h)|$ is zero if it is less than 2^{d-1} because the floating-point sum x and the summands x_j have no bits for representing the powers 2^j for $j < d$. We have $\delta_x \le c\epsilon x_+$, $x_+ = |x_1| + \ldots + |x_h| \le s_+ + |y_1| + \ldots + |y_h| \le s_+ + h2^d$, $c\epsilon s^*_+ = \delta_s \le \tilde{s} \le 2^{d-2}$ (cf. (3.1)), so that $c\epsilon s_+ < 1.5 * 2^{d-2}$ because $|s^*_+/s_+ - 1| \le c\epsilon \ll 1/2$. Furthermore, $c\epsilon h 2^d < 2^{d-3}$ (substitute $\epsilon = 2^{p+1}$ and $c < 2$ into bound (5.1)). Therefore $\delta_x = c\epsilon x_+ < 2^{d-1}$.

Next we estimate the sum $s_+(1) = |x| + \sum_{j=1}^h |y_j|$ and the summation error $\delta_{s(1)} = c\epsilon s_+(1)$ at Stage 4 and compare them with the respective values s_+ and $\delta_s = c\epsilon s_+$ at Stage 1. Recall that $|y_j| < 2^{d-1}$ provided at Stage 3 we apply rounding to the nearest value, whereas $|y_j| \le 2^d$ if rounding is by chopping off the trailing bits below the level 2^d. In both cases we have $|\sum_{j=1}^h y_j| < h2^d$, whereas $|x + \sum_{j=1}^h y_j| < 2^d$. Therefore, $|x| < (h+1)2^d$, $s^{(1)}_+ = |x| + \sum_{j=1}^h |y_j| < (2h+1)2^d$. The summation error at Stage 4 is less than $(2h+1)c\epsilon 2^d$.

It follows that upper estimates for both the sum of the absolute values of the summands and the rounding error of the summation decrease by the factor of 2^k, for $k \ge p - \log_2 h - O(1)$, versus the respective estimates at Stage 1.

As soon as the maximum absolute value of the summands decreases below the level of $|s|/(2c\epsilon)$, the summation produces the sum within the error bound δ_s satisfying (5.3). Therefore, in at most $v = (e_{\max} - e_{\rm sum})/k$ loops, the algorithm satisfies bound (5.3). Here $e_{\max}$ is the maximum exponent of the input summands s_j and $e_{\rm sum}$ is the exponent of the sum $s_1 + \cdots + s_h$.

Remark 5.1. *It is realistic to assume bound (5.1) for the parameter v defined above, but in fact we only need the weaker bound with the value $h + v$ replaced by the maximum number of terms in the auxiliary sums $x + \sum_j y_j$ computed by the Subroutine FLOAT·SUM. $h + v$ is an upper estimate for this number and tends to be an overestimate because many leading bits of $\max_i |s_i|$ tend to be cancelled in the summation of the terms $s_1, \ldots, s_h$.*

Remark 5.2. *We have devised and analyzed Algorithm 5.1 that works under both policies of rounding to the nearest value and of chopping of the extraneous bits beyond the allowed precision. If we specialize the algorithm to the policy of chopping, then we have $e_x = 0$ wherever $e_x < 2^d$ (rather than wherever $e_x < 2^{d-1}$) and furthermore $x_+ = s_+$ (rather than $x_+ \le s_+ + h2^d$). Therefore, in this case we can decrease the exponent d by two. If we adopt the policy of rounding to the nearest value, we cannot decrease the exponent d like that, but we have $|y_j| < 2^{d-1}$, and so $s_+^{(1)} < (h+2)2^{d-1}$ rather than $s_+^{(1)} < (2h+1)2^d$.*

6 Extension to the error-free summation

Suppose we know an approximation s^* to the sum s with a relative error less than $1/2$. Then with a rather routine techniques we can readily extend our algorithms to computing the sum s with any desired precision, still staying essentially with floating-point operations. For completeness, let us specify this for Algorithm 5.1.

First compute the exponent d of (3.1) for $\tilde{s} = 1.5|s^*|$, apply Stages 3 and 4 of the algorithm once again, and obtain the summands x and y_j such that $|x| < (h+1)2^d$ and $|y_j| < 2^d$ for all j.

Now floating-point summation of these updated summands outputs a new approximation s^* to the original sum s within a relative error bound β_0 and the absolute error bound $\tilde{\beta}_0 = \beta_0|s|$. Due to the error bound $\delta_t = c\epsilon t_+$ for the Subroutine FLOAT·SUM for $j = h$, we obtain that $\log_2 \beta_0 = -p + \log_2 h + O(1)$.

If the values $\tilde{\beta}_0$ and β_0 are not small enough, we write $s_{h+1} \leftarrow -s^*$ and $h \leftarrow h+1$ for the current approximation s^* to the sum $s_1 + \ldots + s_h$ and the current summands $s_1, \ldots, s_h$ and reapply our algorithm to approximate the value $\tilde{\beta}_0 = s_1 + \ldots + s_h + s_{h+1}$ of the error of this approximation.

By continuing this process recursively, in u loops we obtain approximations $s + \tilde{\beta}_0 + \tilde{\beta}_1 + \cdots + \tilde{\beta}_u$ to the sum $s_1 + \ldots + s_h$ within relative errors of less than $2^{(-p+\log_2(h+u)+O(1))u}$ for $u = 0, 1, \ldots$. Here $h + u$ is an upper estimate for the number of summands in the sum in u loops of the algorithm. Then again, we expect this to be an overestimate. Although each loop can increase the number of the summands by one, this number decreases when the trailing terms y_j vanish in the summation

process. If we seek the sum with b correct bits, we need $u + v + O(1)$ loops for v defined at the end of Section 5 and $u = b/p + O(1)$. This means error-free sum if it is represented with at most b bits.

7 Estimating arithmetic cost and saving memory space

Each loop of Algorithm 5.1 uses $h' - 1$ additions to compute each of the sums s^*, s^*_+, and x at Stages 1 and 4 and uses $3h'$ or so additions to split h' terms s_j at Stage 3, where h' is at most $h + u + v$ for u and v defined in the previous section. If we ignore the cost of computing $u + v + O(1)$ exponents d, the overall cost of the summation based on Algorithm 5.1 is $6(h+u+v+O(1))(u+v+O(1))$. We can avoid computing the sum s^*_+ if we stop the computations where the computed sum s^* stabilizes to a nonzero value. By shifting to the algorithm in Section 4 we can also avoid computing the sum x. Overall we would need just $4(h + u + v + O(1))(u + v + O(1))$ additions.

We can yield further acceleration if, instead of customary floating-point summation, we apply the more advanced Algorithm 4.4 in [ORO05] at Stages 1 and 4. This algorithm yields the bound $|s^* - s| = \delta_s \le 2^{-p-1}s^* + \gamma^2_{h-1}s^*_+$ instead of bound (2.1). As the result, the maximum absolute value of a summand decreases by the factor of $2^{k(+)}$ per loop of the algorithm where $k(+) \ge 2p - \log_2 h - O(1)$. This ensures twice as rapid gain in the precision of the approximation and thus saves about 50% of the operations involved.

In some cases, particularly if we reduce a sequence of additions and multiplications to summation, the summands can become too numerous to store. Then one can perform summation in stages. If the overall sum nearly vanishes, our heuristic recipe is to sum at first all terms s_j whose absolute values exceed θs^*_+ where $s^*_+ = \mathrm{fl}(\sum_{j=1}^{h} |s_j|)$ and $\theta < 1$ is a fixed fraction, e.g., 1/2. We can expect (although with no insurance) that the sum of these selected terms also nearly vanishes, in which case its computation is simplified.

References

[D71] T. J. Dekker, A Floating-point Technique for Extending the Available Precision, *Numerische Mathematik*, **18**, 224–242, 1971.

[DH03] J. Demmel, Y. Hida, Accurate and Efficient Floating Point Summation, *SIAM J. on Scientific Computing*, **25**, 1214–1248, 2003.

[EPY98] I. Z. Emiris, V. Y. Pan, Y. Yu, Modular Arithmetic for Linear Algebra Computations in the Real Field, *J. of Symbolic Computation*, **21**, 1–17, 1998.

[F04] Agner Fog, *How to optimize for the Pentium family of microprocessors*, www.agner.org, 1996–2004, last updated 2004-04-16.

[H02] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd edition, SIAM Publications, Philadelphia, 2002.

[I01] *IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*, (Order Number 245470) Intel Corporation, Mt. Prospect, Illinois, 2001.

[LDB02] X. Li, J. Demmel, D. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Kang, A. Kapur, M. Martin, B. Thompson, T. Tung, D. Yoo, Design, Implementation and Testing of Extended and Mixed Precision BLAS, *ACM Transactions on Mathematical Software*, **28**, 152–205, 2002. http //crd.lbl.gov/x̃iaoye/XBLAS/.

[Matlab04] The MathWorks Inc., *Matlab User's Guide*, Version 7, 2004.

[ORO05] T. Ogita, S. M. Rump, S. Oishi, Accurate Sum and Dot Product, *SIAM Journal on Scientific Computing*, **26**, **6**, 1955–1988, 2005.

[ORO05a] S. M. Rump, T. Ogita, S. Oishi, Accurate Floating-Point Summation, Tech. Report 05.12, *Faculty for Information and Communication Sciences*, Hamburg University of Technology, November, 2005.

[P92] V. Y. Pan, Can We Utilize the Cancelation of the Most Significant Digits? Tech. Report TR 92 061, *The International Computer Science Institute*, Berkeley, California, 1992.

[Pa] V. Y. Pan, The Schur Aggregation and Extended Iterative Refinement, Technical Report TR 2007, *CUNY Ph.D. Program in Computer Science, Graduate Center, City University of New York*, April 2007.

[PMQa] V. Y. Pan, B. Murphy, G. Qian, R. E. Rosholt, I. Taj-Eddin, Numerical Computation of Determinants with Additive Preconditioning and the Schur Aggregation, Technical Report TR 2007, *CUNY Ph.D. Program in Computer Science, Graduate Center, City University of New York*, April 2007.

[Pna] V. Y. Pan, Computations in the Null Spaces with Additive Preconditioning, Technical Report TR 2007, *CUNY Ph.D. Program in Computer Science, Graduate Center, City University of New York*, April 2007.