A Class of Fast and Accurate Summation Algorithms

Blanchard, Pierre and Higham, Nicholas J. and Mary, Theo

2019

MIMS EPrint: **2019.6**

Manchester Institute for Mathematical Sciences
School of Mathematics

The University of Manchester

Reports available from: http://eprints.maths.manchester.ac.uk/
And by contacting: The MIMS Secretary
School of Mathematics
The University of Manchester
Manchester, M13 9PL, UK

ISSN 1749-9097

A CLASS OF FAST AND ACCURATE SUMMATION ALGORITHMS*

PIERRE BLANCHARD†, NICHOLAS J. HIGHAM‡, AND THEO MARY§

Abstract. The need to sum floating-point numbers is ubiquitous in scientific computing. Standard recursive summation of n summands, often implemented in a blocked form, has an error bound proportional to nu, where u is the unit roundoff. With the growing interest in low precision floating-point arithmetic and ever increasing n in applications, computed sums are more likely to have insufficient accuracy. We propose a class of summation algorithms called FABsum (for "fast and accurate block summation") that applies a fast summation algorithm (such as recursive summation) blockwise and then sums the partial sums using an accurate summation algorithm (such as compensated summation, or recursive summation in higher precision). We give a rounding error analysis to show that FABsum with block size b has an error bound $(b+1)u + O(u^2)$, which is independent of n to first order. Our computational experiments show that FABsum can deliver substantially more accurate results than blocked recursive summation, with only a modest drop in performance. FABsum is especially attractive for low precisions, where it can provide correct digits for much larger n than recursive summation.

Key words. summation, recursive summation, compensated summation, rounding error analysis, floating-point arithmetic, numerical linear algebra.

AMS subject classifications. 65G50, 65Fxx

1. Introduction.

Summation is a key computational task at the heart of many numerical algorithms, most notably numerical linear algebra kernels involving inner products, such as matrix–vector or matrix–matrix multiplications, matrix factorizations, and the solution of linear systems. In floating-point arithmetic, summation incurs rounding errors. Rounding error bounds for the basic summation algorithms are proportional to nu, where n is the number of summands and u the unit roundoff, and thus they grow linearly with n.

While compensated summation algorithms achieving error bounds that do not grow (or grow slowy) with n are known [7, sec. 4.3], they are computationally expensive. Indeed running a basic summation algorithm in higher precision (e.g., using double precision rather than single precision) may provide a better performance/accuracy tradeoff. Compensated summation algorithms are thus only used when the highest precision available is not enough for the application at hand, and they are generally not available in optimized libraries such as the Intel Math Kernel Library or the NAG Library.

The rise of large-scale, low-precision computations presents new challenges. On the one hand, modern supercomputing allows larger and larger problems to be solved, and we must routinely evaluate sums of 10^8 or more terms. On the other hand, the use of low floating-point precisions (such as IEEE half precision, for which $u = 4.88\times10^{-4}$) is increasingly common [2], [6], [9]. For such large sizes and low precisions, error bounds of order nu exceed 1, and so not even a correct sign is guaranteed. This

*Version of April 22, 2019. **Funding**: This work was supported by Engineering and Physical Sciences Research Council grant EP/P020720/1, The MathWorks, and the Royal Society. The opinions and views expressed in this publication are those of the authors, and not necessarily those of the funding bodies.

†School of Mathematics, The University of Manchester, Manchester M13 9PL, UK (pierre.blanchard00@gmail.com)

‡School of Mathematics, The University of Manchester, Manchester, M13 9PL, UK (nick.higham@manchester.ac.uk, http://www.maths.manchester.ac.uk/~higham)

§School of Mathematics, The University of Manchester, Manchester M13 9PL, UK (theo.mary@manchester.ac.uk)

creates a dilemma, as we need to choose between benefiting from the speed, lower energy requirements, and reduced data movement of low precision arithmetic and being able to accurately solve large problems.

In this work we tackle this dilemma by introducing a new class of summation algorithms that excels in both performance and accuracy. These algorithms achieve error bounds that do not grow with n (that is, the error bounds are of the form cu where c is a moderate constant independent of n) and, at the same time, they can deliver a similar performance to optimized, standard algorithms by performing an arbitrarily low number of extra flops (e.g., less than a 1% overhead) and by allowing efficient implementation on modern computers.

We first review existing summation algorithms in section 2. Then, in section 3, we present the new class of algorithms, called `FABsum`, perform a rounding error analysis for algorithms in this class, and analyze their cost in a general framework. In section 4 we apply `FABsum` to several key numerical linear algebra algorithms. In section 5, we assess the practical performance and accuracy of `FABsum` through numerical experiments. We provide our concluding remarks in section 6.

2. Existing summation algorithms. We begin by briefly reviewing existing algorithms to compute the sum $s = \sum_{i=1}^{n} x_i$. We denote by $\widehat{s}$ the computed value of s in floating-point arithmetic. We use the standard model of floating-point arithmetic [7, sec. 2.2]

$$\mathrm{fl}(a \operatorname{op} b) = (a \operatorname{op} b)(1+\delta), \quad |\delta| \le u, \quad \operatorname{op} \in \{+, -, \times, /\}. \tag{2.1}$$

2.1. Recursive summation. The most basic algorithm to compute s is recursive summation, which starts with $s = x_1$ and computes

$$s \leftarrow s + x_i, \quad i = 2 : n.$$

The computed sum satisfies [7, sec. 4.2]

$$\widehat{s} = \sum_{i=1}^{n} x_i(1+\mu_i), \quad |\mu_i| \le \gamma_{n-1} = (n-1)u + O(u^2), \tag{2.2}$$

where $\gamma_n = nu/(1-nu)$ for $nu < 1$. This backward error bound grows linearly with n and is almost attainable [7, Prob. 4.2].

2.2. Blocked summation. In order to allow a sum to be computed in parallel, high–performance libraries divide the summands into n/b blocks of size b. Here, n is assumed to be a multiple of b for simplicity. First, the local sums

$$s_i = \sum_{j=(i-1)b+1}^{ib} x_j, \quad i = 1 : n/b, \tag{2.3}$$

are computed with recursive summation, and then $s = \sum_{i=1}^{n/b} s_i$ is also obtained with recursive summation.

By applying (2.2) to s_i and s we find that the computed $\widehat{s}$ from blocked summation satisfies

$$\widehat{s} = \sum_{i=1}^{n} x_i(1+\mu_i), \quad |\mu_i| \le (b + n/b - 2)u + O(u^2), \tag{2.4}$$

and so the backward error bound depends on the block size b. The constant $b+n/b-2$ in the bound is minimized for $b = \sqrt{n}$, for which it equals $2\sqrt{n} - 2$. However, for performance reasons b is often chosen to be a moderate constant (such as 128 or 256), in which case the error bound for blocked summation still grows linearly with n.

2.3. Pairwise summation. Pairwise summation (also known as tree summation or binary cascade summation) generalizes blocked summation by recursively computing the local sums (2.3) with blocked summation, continuing until sums of two terms are obtained. There are thus at most $\lceil \log_2 n \rceil$ levels of recursion, which leads to the backward error result [7, sec. 4.2]

$$\widehat{s} = \sum_{i=1}^{n} x_i(1+\mu_i), \quad |\mu_i| \le \lceil \log_2 n \rceil u + O(u^2), \tag{2.5}$$

in which the bound grows with n at a much slower rate than for blocked summation. However, pairwise summation is not well suited for an efficient implementation on parallel computers, and it usually delivers poor performance compared with blocked summation [3].

2.4. Compensated summation. The principle of compensation is based on the fact that for any pair (a, b) of floating-point numbers and their computed sum $\widehat{s} = fl(a + b)$, there exists a floating-point number e such that

$$a + b = s + e. \tag{2.6}$$

In other words, the error in a floating-point addition is itself a floating-point number.

Compensated summation algorithms evaluate the error term e at each step of recursive summation in order to correct the computed sum. Many such algorithms have been designed and analyzed over the years; here we focus on Algorithm 2.1 [11], [15]. The backward error result for this algorithm is [5, Thm. 8], [12, Ex. 19, sec, 4.2.2]

$$\widehat{s} = \sum_{i=1}^{n} x_i(1+\mu_i), \quad |\mu_i| \le 2u + O(u^2), \tag{2.7}$$

which is an almost ideal result.

Algorithm 2.1 Compensated summation.

This algorithm takes as input n summands x_i and computes their sums by compensated summation.

```
1: s = 0, e = 0
2: for i = 1: n do
3:     z = s
4:     y = x_i + e
5:     s = z + y
6:     e = (z − s) + y
7: end for
```

Algorithm 2.1 requires 3 extra flops per loop iteration, which is typically more expensive than simply switching to a higher precision. For this reason, compensated summation is usually worth considering only when computations are already taking place at the highest precision available.

3. Fast and accurate blocked summation. Assume that we have at our disposal two summation algorithms: one that is fast, referred to as the `FastSum` algorithm, and one that is accurate, referred to as the `AccurateSum` algorithm. We describe in Algorithm 3.1 `FABsum`, a new blocked summation algorithm that exploits `FastSum` and `AccurateSum`. The key idea is to use `FastSum` to compute the local sums s_i and `AccurateSum` to sum the local sums. This idea is motivated by the observations that the bulk of the computation is in computing the local sums if the block size is chosen appropriately and that rounding error growth can be attenuated by accumulating the local sums more accurately.

We can therefore expect that the new algorithm will be almost as fast as `FastSum` and almost as accurate as `AccurateSum`. The rest of this section is devoted to proving this expectation by analyzing the rounding errors and the cost of Algorithm 3.1 and illustrating these analyses with some practical choices of `FastSum` and `AccurateSum`.

Algorithm 3.1 (`FABsum`) Fast and accurate blocked summation algorithms.

This algorithm takes as input n summands x_i, a block size b, and two summation algorithms `FastSum` and `AccurateSum`. It returns the sum $s = \sum_{i=1}^{n} x_i$.

1: **for** $i = 1 : n/b$ **do**
2: Compute $s_i = \sum_{j=(i-1)b+1}^{ib} x_j$ with `FastSum`.
3: **end for**
4: Compute $s = \sum_{i=1}^{n/b} s_i$ with `AccurateSum`.

We note the special cases $b = 1$, which is entirely `AccurateSum`, and $b = n$, which is entirely `FastSum`. We are interested in values of b between these two extremes: b should be large enough to give speed benefits but small enough to give error bounds independent of n. In the next two sections we give error and cost analyses that guide the choice of b.

Our general framework includes some previously proposed algorithms. For example, taking `AccurateSum` to be pairwise summation leads to the superblock algorithm from [3]. Similarly, [1] present a "selective compensation" algorithm that amounts to using compensated summation as `AccurateSum`. Our analysis below generalizes the analyses of these papers, considers other possible `AccurateSum` choices, and extends the analysis to numerical linear algebra kernels in Section 4.

3.1. Rounding error analysis. To carry out a rounding error analysis of Algorithm 3.1 we need to make some assumptions on the error bounds associated with algorithms `FastSum` and `AccurateSum`. For any sum of the form $s = \sum_{i=1}^{n} x_i$, we assume that the computed $\widehat{s}$ from `FastSum` satisfies

$$\widehat{s} = \sum_{i=1}^{n} x_i(1 + \mu_i^f), \quad |\mu_i^f| \le \varepsilon_f(n) \tag{3.1}$$

and the computed $\widehat{s}$ from `AccurateSum` satisfies

$$\widehat{s} = \sum_{i=1}^{n} x_i(1 + \mu_i^a), \quad |\mu_i^a| \le \varepsilon_a(n), \tag{3.2}$$

where $\varepsilon_f(n)$ and $\varepsilon_a(n)$ are $O(u)$ and depend on n and u. With these assumptions, we have the following backward error result.

THEOREM 3.1. *Let $s = \sum_{i=1}^{n} x_i$ be computed by Algorithm 3.1. The computed $\widehat{s}$ satisfies*

$$\widehat{s} = \sum_{i=1}^{n} x_i(1+\mu_i), \quad |\mu_i| \le \varepsilon(n,b) = \varepsilon_a(n/b) + \varepsilon_f(b) + \varepsilon_a(n/b)\varepsilon_f(b).$$

Proof. Each of the local sums s_i computed at line 2 of Algorithm 3.1 satisfies

$$\widehat{s}_i = \sum_{j=(i-1)b+1}^{ib} x_j\bigl(1+\mu_j^f\bigr), \quad \bigl|\mu_j^f\bigr| \le \varepsilon_f(b). \tag{3.3}$$

Let $t = \sum_{i=1}^{n/b} \widehat{s}_i$ be the sum of the computed local sums. Then the computed $\widehat{t}$, which is the overall computed sum $\widehat{s}$, satisfies

$$\widehat{t} = \widehat{s} = \sum_{i=1}^{n/b} \widehat{s}_i\bigl(1+\mu_i^a\bigr), \quad \bigl|\mu_i^a\bigr| \le \varepsilon_a(n/b). \tag{3.4}$$

Combining (3.3) and (3.4), we obtain

$$\widehat{s} = \sum_{i=1}^{n} x_i\bigl(1+\widetilde{\mu}_i^f\bigr)\bigl(1+\mu_i^a\bigr), \quad \bigl|\widetilde{\mu}_i^f\bigr| \le \varepsilon_f(b), \quad \bigl|\mu_i^a\bigr| \le \varepsilon_a(n/b).$$

Defining $\mu_i = \widetilde{\mu}_i^f + \mu_i^a + \widetilde{\mu}_{i_1}^f \mu_i^a$, the result follows. ∎

We can interpret Theorem 3.1 as follows. If ε_a is a term of order $O(u^2)$ (such as when `AccurateSum` uses extended precision), then $\varepsilon(n,b) = \varepsilon_f(b) + O(u^2)$ is independent of n to first order and thus does not grow with n. In fact, even if ε_a is a term of order $O(u)$, as long as it does not grow with n to first order (such as when `AccurateSum` uses compensation), $\varepsilon(n,b)$ does not grow with n to first order either.

To explore this point more precisely we provide an expression for $\varepsilon(n,b)$ in the following cases. Assume `FastSum` corresponds to recursive summation, so that $\varepsilon_f(b) = (b-1)u + O(u^2)$.

- If `AccurateSum` uses recursive summation in extended precision with unit roundoff u^2, with a final rounding back to the working precision, then we have $\varepsilon_a(n/b) = u + O(u^2)$ and an overall error bound

$$\varepsilon(n,b) = bu + O(u^2). \tag{3.5}$$

- If `AccurateSum` uses compensated summation, we have $\varepsilon_a(n/b) = 2u + O(u^2)$ by (2.7) and an overall bound

$$\varepsilon(n,b) = (b+1)u + O(u^2). \tag{3.6}$$

- If `AccurateSum` uses pairwise summation, we have $\varepsilon_a(n/b) = \lceil \log_2(n/b) \rceil u + O(u^2)$ by (2.5) and an overall bound

$$\varepsilon(n,b) = (b - 1 + \lceil \log_2(n/b) \rceil)u + O(u^2). \tag{3.7}$$

In all cases we thus obtain an overall error bound that does not grow (or, in the case of pairwise summation, grows very slowly) with n.

3.1.1. Accuracy versus stability. Accurate summation means achieving a small forward error, but our error analysis bounds the backward error. We will show that these concepts are closely related for summation. The backward error of an approximation $\widehat{s}$ to $s = \sum_{i=1}^n x_i$ is

$$\eta(\widehat{s}) = \min\left\{ \max|\delta_i| : \widehat{s} = \sum_{i=1}^n x_i(1+\delta_i) \right\}. \tag{3.8}$$

A formula for the backward error can be obtained as a special case of the Oettli–Prager theorem [7, Thm. 7.3], [16]:

$$\eta(\widehat{s}) = \frac{\left|\widehat{s} - \sum_{i=1}^n x_i\right|}{\sum_{i=1}^n |x_i|}. \tag{3.9}$$

The numerator is the absolute forward error in the computed sum. So for summation, backward error and absolute forward error differ by a constant factor. The relative forward error is a factor $\sum_{i=1}^n |x_i| / |\sum_{i=1}^n x_i|$ larger than $\eta(\widehat{s})$—a factor that depends on the data but not the algorithm. This factor is precisely the condition number

$$\operatorname{cond}(x) = \sup_{\varepsilon\to 0}\left\{ \frac{\left|\sum_{i=1}^n (x_i+\Delta x_i) - \sum_{i=1}^n x_i\right|}{\varepsilon\left|\sum_{i=1}^n x_i\right|} : |\Delta x_i| \le \varepsilon|x_i|,\ i = 1\colon n \right\}.$$

We conclude that achieving a small backward error and a small absolute forward error are essentially equivalent, while the relative forward error is a factor $\operatorname{cond}(x)$ larger, this factor depending only on the data and hence being out of our control.

3.1.2. On second order contributions. Up to now we have considered expansions of the error bounds up to first order in u. However, when low precision arithmetic is being used it is likely that the $O(u^2)$ terms will become significant even for moderate n. We will now obtain the second order terms explicitly in order to assess their potential influence on the error bounds of `FABsum`.

Let us start with the error bounds given in section 2. In the case of recursive summation, if $(n-1)u < 1$ a second order bound is found by using the expansion of $\gamma_{n-1}(u)$ in (2.2):

$$\widehat{s} = \sum_{i=1}^n x_i(1+\mu_i), \quad |\mu_i| \le (n-1)u + (n-1)^2u^2 + O(u^3).$$

Here, the second order term becomes significant when $(n-1)u$ approaches 1, i.e., when the bound becomes invalid and hence of no use. Similarly, for recursive summation in extended precision (with unit roundoff u^2), we have

$$\widehat{s} = \sum_{i=1}^n x_i(1+\mu_i), \quad |\mu_i| \le u + (n-1)u^2 + O(u^3).$$

Finally, an explicit second order backward error bound for compensated summation can be found in [5, Thm. 8]:

$$\widehat{s} = \sum_{i=1}^n x_i(1+\mu_i), \quad |\mu_i| \le 2u + 2(2n+1)u^2 + O(u^3).$$

By Theorem 3.1, the second order terms of `FABsum` can be obtained by adding the second order terms of $\varepsilon_a(n/b)$ and $\varepsilon_f(b)$ to the product of their first order terms. We assume that `FastSum` uses recursive summation at the working precision, so that $\varepsilon_f(b) = (b-1)u + (b-1)^2u^2 + O(u^3)$.

- If `AccurateSum` uses recursive summation in extended precision, we have $\varepsilon_a(n/b) = u + (n/b-1)u^2 + O(u^3)$ and an overall error bound

$$\varepsilon(n,b) = bu + \left[n/b - 1 + (b-1)^2 + (b-1)\right] u^2 + O(u^3).$$

- If `AccurateSum` uses compensated summation, we have $\varepsilon_a(n/b) = 2u + 2(2n/b+1)u^2 + O(u^3)$ and an overall bound

$$\varepsilon(n,b) = (b+1)u + \left[4n/b + 2 + (b-1)^2 + 2(b-1)\right] u^2 + O(u^3).$$

We will not derive a higher–order bound for pairwise summation as the bound (3.7) depends on n to first order already. For large n, n/b is in practice much larger than b and so we have, roughly,

$$\varepsilon(n,b) = bu + (n/b)u^2 + O(u^3). \tag{3.10}$$

for extended precision and

$$\varepsilon(n,b) = bu + (4n/b)u^2 + O(u^3). \tag{3.11}$$

for compensated summation. The order u^2 terms becomes significant when n exceeds the critical values b^2/u for extended precision and $b^2/(4u)$ for compensated summation. For instance, in half precision arithmetic and for a block size $b = 128$, the second order terms in the bounds are significant for n larger than 3.3×10^7 with extended precision and 8.3×10^6 with compensation.

3.2. Cost analysis. We now analyze the cost $C(n,b)$ in flops of Algorithm 3.1. Let C_f and C_a denote the costs of algorithms `FastSum` and `AccurateSum`, respectively. Then

$$C(n,b) = \frac{n}{b} C_f(b) + C_a\Big(\frac{n}{b}\Big).$$

In particular, if the costs C_f and C_a are linear functions of the number of summands, as is often the case, $C(n,b)$ simplifies to

$$C(n,b) = C_f(n) + \frac{1}{b} C_a(n).$$

Therefore the cost of Algorithm 3.1 can be made arbitrarily close to that of `FastSum` by increasing the block size b.

For example, assume `FastSum` is recursive summation, whose cost is $C_f(n) = n-1$, and `AccurateSum` is compensated summation, whose cost is $C_a(n) = 3n-2$. Then $C(n,b)$ is equal to $n - 1 + (3n-2)/b)$, that is, it requires only $(3n-2)/b$ extra flops compared with recursive summation, which is a very small overhead for typical choices of block sizes. For instance, for $b = 300$, this represents an overhead of only 1%. Similar results hold when `AccurateSum` uses extended precision (which increases the cost by a constant factor, usually 2) or pairwise summation (which does not change the number of required flops, but may decrease the speed due to poor efficiency on parallel computers).

4. Application to numerical linear algebra. Now we apply our new `FABsum` algorithm (Algorithm 3.1) within some important summation-based kernels in numerical linear algebra, namely inner products, matrix–vector and matrix–matrix products, matrix factorizations, and the solution of linear systems. The core computation is an inner product, and to be precise we specify in Algorithm 4.1 how `FABsum` is used.

Algorithm 4.1 Inner product via `FABsum`.

This algorithm takes as input vectors $x, y \in \mathbb{R}^n$ and a block size b and returns the inner product $x^T y$.

1: Apply Algorithm 3.1 to the summands $x_i y_i$, $i = 1\colon n$.

We now derive rounding error bounds for the resulting algorithms. Recall that $\varepsilon_f(n)$ and $\varepsilon_a(n)$ are the error constants in (3.1) and (3.2) for `FastSum` and `AccurateSum`, respectively.

THEOREM 4.1 (inner products). *Let $x, y \in \mathbb{R}^n$ and let $s = x^T y$ be computed by Algorithm 4.1. The computed $\widehat{s}$ satisfies*

$$\widehat{s} = \sum_{i=1}^{n} x_i y_i (1+\mu_i), \quad |\mu_i| \le u + \varepsilon_a(n/b) + \varepsilon_f(b) + O(u^2).$$

Proof. The initial products $z_i = x_i y_i$ each incur one rounding error. The rest of the proof follows from the application of Theorem 3.1 to the computation of $\sum_{i=1}^n z_i$. ∎

In the next result we consider matrix–vector and matrix–matrix products computed by the usual inner product formulas.

THEOREM 4.2 (matrix–vector and matrix–matrix products). *Let $A \in \mathbb{R}^{m\times n}$, $B \in \mathbb{R}^{n\times p}$, and $x \in \mathbb{R}^n$. If $y = Ax$ is computed via inner products using Algorithm 4.1 then*

$$\widehat{y} = (A + \Delta A)x, \quad |\Delta A| \le \left(u + \varepsilon_a(n/b) + \varepsilon_f(b) + O(u^2)\right)|A|.$$

If $C = AB$ is computed via inner products using Algorithm 4.1 then

$$|\widehat{C} - C| \le \left(u + \varepsilon_a(n/b) + \varepsilon_f(b) + O(u^2)\right)|A||B|.$$

Proof. The proof follows directly from Theorem 4.1. ∎

In order to analyze the solution of linear systems we need the next lemma. For convenience in stating the bounds, we will assume that $\varepsilon_a(n) \ge u$ and $\varepsilon_f(n) \ge u$, which is true for all the methods under consideration here.

LEMMA 4.3. *If $y = (c - \sum_{i=1}^{k-1} a_i b_i)/b_k$ is computed by applying Algorithm) 4.1 to the summation term then the computed $\widehat{y}$ satisfies*

$$b_k \widehat{y}(1+\mu_0) = c - \sum_{i=1}^{k-1} a_i b_i (1+\mu_i), \tag{4.1}$$

where $|\mu_i| \le u + \varepsilon_a((k-1)/b) + \varepsilon_f(b) + O(u^2)$ for all i.

Proof. By Theorem 3.1 and (2.1) we have

$$\widehat{y} = b_k^{-1}\left(c - \sum_{i=1}^{k-1} a_i b_i (1+\delta_i)(1+\mu_i')\right)(1+\delta_k)(1+\delta_{k+1}),$$

where

$$|\delta_i| \le u, \quad i = 1\colon k+1, \qquad |\mu_i'| \le \varepsilon_a((k-1)/b) + \varepsilon_f(b) + \varepsilon_a((k-1)/b)\varepsilon_f(b).$$

Therefore (4.1) holds with $1+\mu_0 = \big((1+\delta_k)(1+\delta_{k+1})\big)^{-1}$, which implies $|\mu_0| \le 2u + O(u^2)$, and

$$1 + \mu_i = 1 + \delta_i + \mu_i' + \delta_i\mu_i',$$

which implies

$$|\mu_i| \le u + \varepsilon_a((k-1)/b) + \varepsilon_f(b) + O(u^2). \qquad \square$$

We can now derive results for the solution of linear systems by LU factorization. The next result is immediate from Lemma 4.3.

THEOREM 4.4. *Let the triangular system $Tx = b$, where $T \in \mathbb{R}^{n\times n}$, be solved by substitution using Algorithm 4.1. Then the computed solution $\widehat{x}$ satisfies*

$$(T + \Delta T)\widehat{x} = b, \qquad |\Delta T| \le \big(u + \varepsilon_a((n-1)/b) + \varepsilon_f(b) + O(u^2)\big)|T|.$$

For the next two results we assume that the Doolittle form of LU factorization is used (see, e.g., [7, Alg. 9.2]) and that Algorithm 4.1 is used in the inner products therein.

THEOREM 4.5. *If LU factorization applied to $A \in \mathbb{R}^{n\times n}$ runs to completion then the computed LU factors $\widehat{L}$ and $\widehat{U}$ satisfy*

$$A + \Delta A = \widehat{L}\widehat{U}, \quad |\Delta A| \le \big(u + \varepsilon_a(n/b) + \varepsilon_f(b) + O(u^2)\big)|\widehat{L}||\widehat{U}|.$$

Proof. The result is obtained by applying Lemma 4.3 to the equations that determine L and U, and is analogous to the proof of [7, Thm. 9.3]. $\square$

THEOREM 4.6. *Let $A \in \mathbb{R}^{n\times n}$ and suppose LU factorization produces computed LU factors $\widehat{L}$, $\widehat{U}$, and a computed solution $\widehat{x}$ to $Ax = b$. Then*

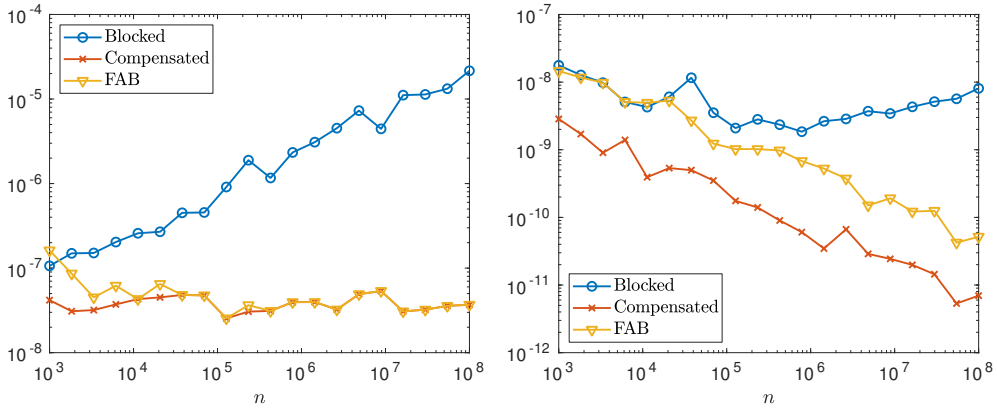
$$(A + \Delta A)\widehat{x} = b, \quad |\Delta A| \le 3\big(u + \varepsilon_a(n/b) + \varepsilon_f(b) + O(u^2)\big)|\widehat{L}||\widehat{U}|. \tag{4.2}$$

Proof. The result is obtained by combining Theorems 4.5 and 4.6, and is analogous to the proof of [7, Thm. 9.4]. $\square$

When recursive summation is used (which corresponds to $b = n$ in Algorithm 3.1, with `FastSum` equal to recursive summation), we have $\varepsilon_a = 0$ and $\varepsilon_f(b) = \varepsilon_f(n) = (n-1)u + O(u^2)$, so the error constants in Theorems 4.4–4.6 have first order terms proportional to n. But as shown in section 3.1, if `FABsum` is used with `FastSum` equal to recursive summation and `AccurateSum` equal to either compensated summation or recursive summation in extended precision then the first order terms in Theorems 4.4–4.6 are of order bu and hence independent of n. `FABsum` therefore offers error constants roughly n/b times smaller than those for recursive summation.

5. Numerical experiments. We have carried out an extensive set of numerical experiments to assess the performance and accuracy of our new `FABsum` algorithm (Algorithm 3.1) with different `AccurateSum` choices and to compare it with existing algorithms.

In section 5.1 we experiment with inner products using MATLAB R2018b, whereas in section 5.2 we present performance and accuracy results with PLASMA [4], a state-of-the-art numerical linear algebra library that we have modified by integrating `FABsum`.



(a) Random uniform $[0, 1]$ vectors.

(b) Random uniform $[-1, 1]$ vectors.

Fig. 5.1: Backward error in single precision for the summation $s = \sum_{i=1}^{n} x_i$, for a vector x with random uniform entries and block size $b = 128$.

In all the experiments, the working precision is single and the "exact" quantities appearing in the error bound formulas for inner products and matrix–matrix products are computed in double precision.

5.1. Summation. We first experiment with summation with MATLAB R2018b. We compute $s = \sum_{i=1}^{n} x_i$, where x is a randomly generated vector from one of the distributions

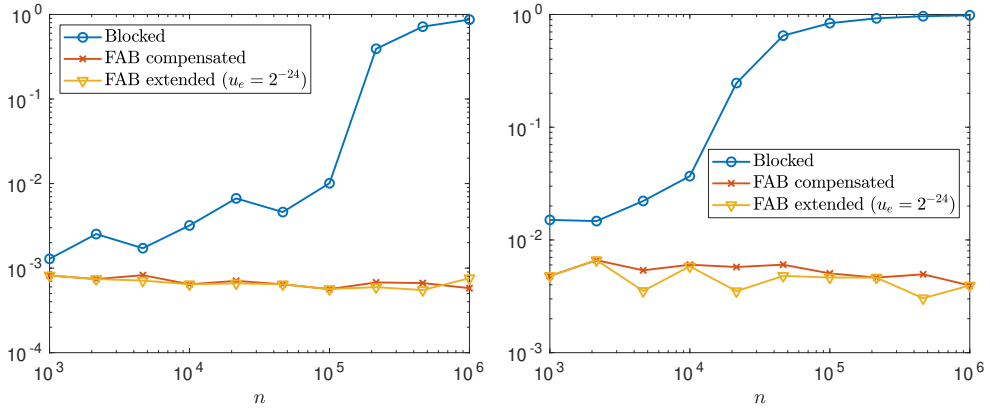
- random uniform $[0, 1]$: `x = rand(m,n)`,
- random uniform $[-1, 1]$: `x = (rand(m,n)-0.5)*2`.

In order to make the experiments reproducible, we seed the random number generator at the beginning of each script generating a figure of this section. We have made these scripts available online[1]. For each size of problem n, we run the same experiment $N_{test} = 10$ times and plot the value of the backward error (3.8).

Results using single precision and a block size $b = 128$ are displayed in Figure 5.1, where we compare blocked summation (one of the fastest summation algorithms), compensated summation (Algorithm 2.1, one of the most accurate summation algorithms), and `FABsum` (Algorithm 3.1), which in this case uses recursive summation for `FastSum` and compensated summation for `AccurateSum`. Clearly, `FABsum` is much more stable than blocked summation, by up to three orders of magnitude for large n. It yields a backward error that does not grow with n and is almost ideally small for $[0, 1]$ vectors, matching the compensated summation error. For $[-1, 1]$ vectors, compensated summation remains more accurate, but only by a constant factor always less than 10.

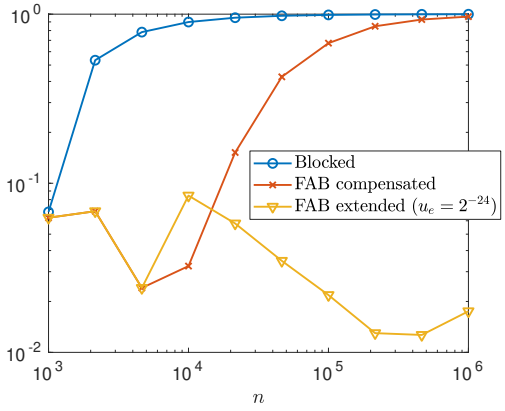
The backward error bound for `FABsum` is $(b + 1)u + O(u^2)$ (see (3.6)) versus $2u + O(u^2)$ for compensated summation, but we see a ratio in the actual backward errors roughly $\sqrt{b}/2$ rather than $b/2$. This is not surprising because the probabilistic error analysis in [8] shows that if the rounding errors are independent random variables of zero mean then an error bound of order the square root of the worst-case bound

[1] https://gitlab.com/nla-grp/FABsum.



(a) Half precision (fp16).

(b) Half precision (bfloat16).



(c) Quarter precision (fp8).

Fig. 5.2: Backward error of the summation $s = \sum_{i=1}^{n} x_i$, for a vector x with random uniform $[0, 1]$ entries and for a block size $b = 32$. The methods are blocked summation and `FABsum` with compensated summation or recursive summation in extended precision (either u^2 or u^4) for `AccurateSum`.

holds with high probability.

The next experiment uses IEEE half precision (fp16, $u = 2^{-11}$), bfloat16 ($u = 2^{-8}$), and quarter precision ($u = 2^{-4}$)—the latter is not standard, but this choice of precision for an 8-bit words is suggested in [14]. These lower precisions are simulated with the rounding function `chop.m` from [10]. Since our objective is to assess the effect of rounding errors, in order to avoid overflow interfering with the results we only simulate the significand of these low precision formats; for the exponent, we keep the same number of bits as double precision. Figure 5.2 compares blocked summation with `FABsum` with recursive summation for `FastSum` and two choices for `AccurateSum`: compensated summation and recursive summation in extended precision $u_e = 2^{-24}$ (that is, single precision). As can be seen in the figure, for such low precisions blocked summation leads to a backward error that quickly reaches 1, at which point the computed sum is meaningless from a backward error point of view. Since in this

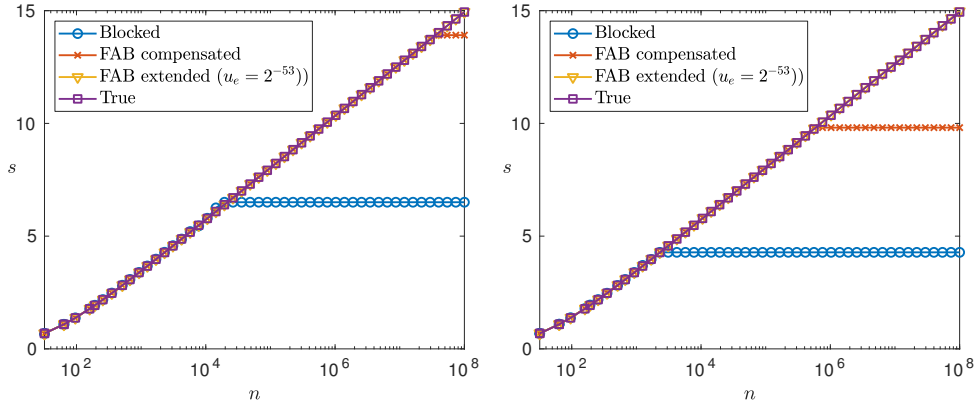
experiment x is a nonnegative vector, one important difficulty is that the sum s keeps increasing as it is computed. For large n, the sum may become so large that the following increments do not increase its value in floating-point arithmetic. This phenomenon, which we call stagnation, leads to a dramatic increase of the error. The reason is that the rounding errors incurred during stagnation are all negative, which makes the worst-case error bound close to being sharp [8, Sec. 4.2.1] and also invalidates the model upon which the probabilistic analysis in [8] is based. Since the entries of x are bounded by 1, we know that s (in line 4 of Algorithm 3.1) must necessarily stagnate after its value exceeds b/u; note that stagnation can, however, occur for smaller values of s when the increments x_i are small. For random uniform summands in $[0, 1]$, the value of s should be about $n/2$. This explains the sharp increase of the error for blocked summation observed in Figure 5.2 when n becomes larger than $2b/u$, that is, 131072, 16384, and 1024 in fp16, bfloat16, and fp8 arithmetic, respectively.

We now discuss the behavior of `FABsum` with respect to stagnation for different choices of `AccurateSum`. For fp8 arithmetic, stagnation occurs for `FABsum` and eventually leads to a sharp increase of the error in the case of compensated summation. However, when extended (single) precision is used, s does not stagnate because it never exceeds $2b/u_s \approx 1.1 \times 10^9$. For larger n, one could use an even higher extended precision, such as double precision. In fact, provided that stagnation does not occur within local sums, `FABsum` with extended precision summation is able to avoid stagnation as long as standard blocked summation in precision u_e, at a fraction of the cost. For the fp16 and bfloat16 precisions, even though the range of sizes n are too small to show the full effect of stagnation, the variant with extended precision is again noticeably and consistently more accurate than the variant with compensated summation. In contrast to the fp8 case, it is not clear, though, whether this is due to some mild beginning of stagnation or to the effect of the second-order terms analyzed in section 3.1.2.

We conclude with a classical example affected by stagnation: the evaluation of the harmonic series $s = \sum_{i=1}^{n} 1/i$. In exact arithmetic, s diverges as $n \to \infty$, whereas in floating-point arithmetic, s is known to "converge" due to stagnation [10], [13]. In Figure 5.3, we compare the value to which s converges for different precisions and summation algorithms. These experiments demonstrate again that `FABsum` with extended precision summation (in this case, double precision) is able to delay stagnation much longer than the other algorithms.

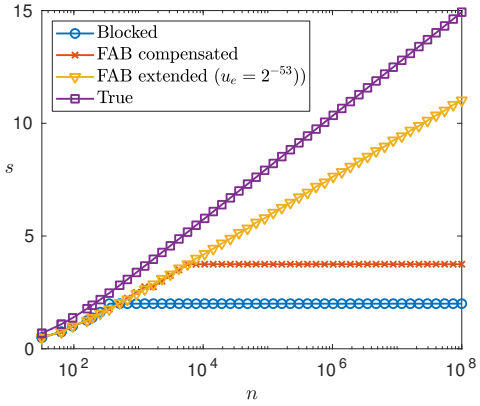
5.2. Matrix multiplication. We now experiment with the state-of-the-art parallel numerical linear algebra library PLASMA [4]. PLASMA includes much of the functionality of the BLAS and LAPACK, but partitions matrices into $t \times t$ blocks called tiles. Independent operations on different tiles are performed concurrently using OpenMP task-based parallelism in order to achieve high performance on shared-memory multicore machines. The tile size is tuned for performance: it must be large enough to allow for computations of high granularity, but small enough to expose enough parallelism to feed all available cores.

By design, PLASMA therefore lends itself naturally to using blocked summation. Note that it is not easy to determine what specific summation algorithm PLASMA uses, because it relies on third-party libraries (in our experiments, Intel MKL) to compute the local operations on tiles. Nevertheless, we expect the default implementation of the library to suffer from growth of the error with the matrix size. In order to assess how `FABsum` can overcome this issue and at what performance cost, we have



(a) Half precision (fp16).

(b) Half precision (bfloat16).



(c) Quarter precision (fp8).

Fig. 5.3: Value of the sum $s = \sum_{i=1}^{n} 1/i$ for different precisions and summation algorithms, using a block size $b = 32$.

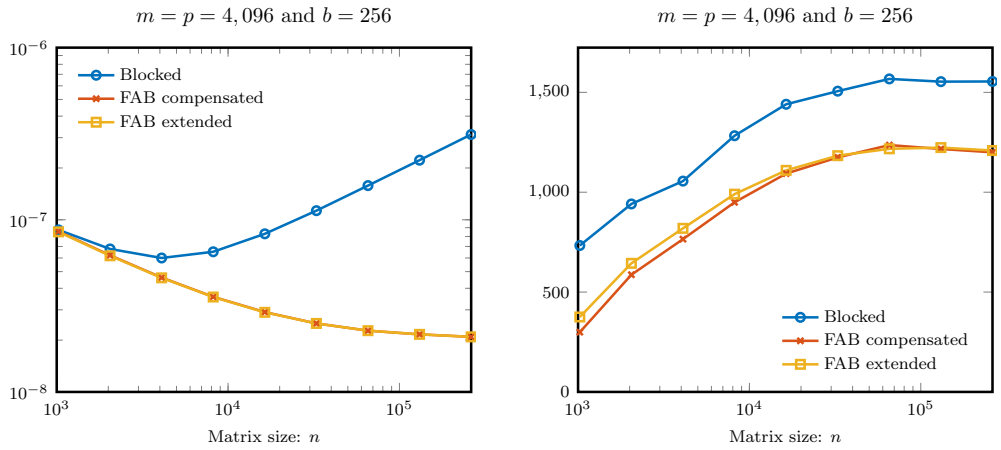
implemented `FABsum` in some of the PLASMA routines[2].

We experiment with single precison matrix–matrix multiplications $C = AB$, where $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times p}$ are randomly generated matrices with uniform entries in $[0, 1]$. We use two 14-core Intel Broadwell processors on the Kebnekaise supercomputer (Umeå, Sweden). We perform $N_{test} = 10$ consecutive runs and take the maximum performance and the maximum backward error, measured as

$$\frac{\|\widehat{C} - C\|}{\|A\|\|B\|},$$

where $\|\cdot\|$ denotes the Frobenius norm and where the "exact" C is computed by a double precision multiplication. We wish to study the effect of large n on the backward error, but the matrices would not fit in memory if taken to be square. For this reason, we fix $m = p = 4096$ and vary n from 10^3 to 2×10^5.

[2]The code is available online at https://gitlab.com/nla-grp/plasma-17.1-xp.



(a) Backward error.

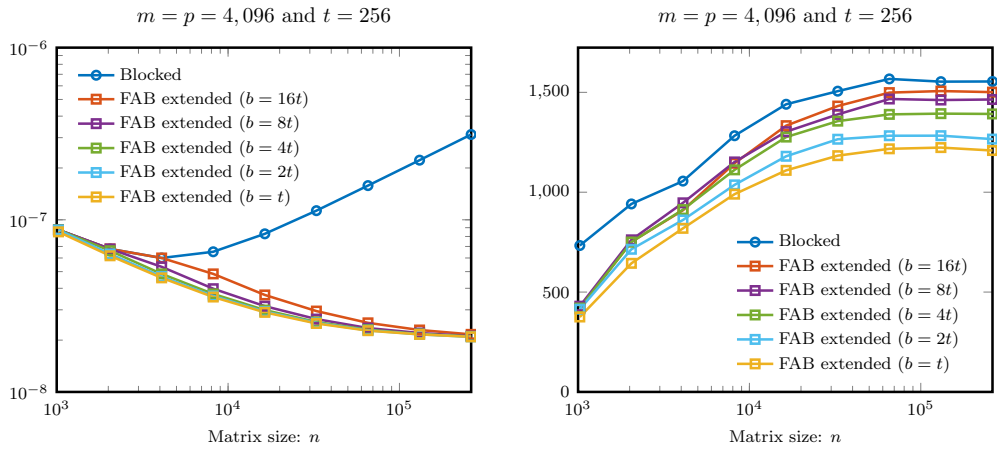
(b) Performance in Gflops (using 28 cores).

Fig. 5.4: Performance and accuracy of $m \times n$ by $n \times p$ matrix–matrix multiplication in single precision with $m = p = 4096$, block size $b = 256$, tile size $t = 256$, and varying n. We compare the standard implementation based on blocked summation with two versions based on the new `FABsum` algorithm.

In Figure 5.4, we compare the performance and accuracy of PLASMA using `FABsum` to the default implementation of the PLASMA library. In this first experiment, we have taken the summation block size b to be equal to the default tile size $t = 256$, because this makes for the most convenient implementation of `FABsum`. Figure 5.4a shows that for the largest matrices, the error is reduced by more than an order of magnitude. In theory, the performance loss should follow the flop overhead, which is about 1% since $b = 256$. However, Figure 5.4b reports a much higher performance loss: while the default implementation achieves a performance peak of over 1500 Gflops for the largest matrices, `FABsum` only reaches about 1200 Gflops, which represents a 20% loss. This is probably explained by the fact that the extra work required by `FABsum` cannot achieve a high speed, being of BLAS-2 type rather than BLAS-3.

Therefore, even though it is convenient to take the summation block size b and the tile size t to be the same, doing so does not allow for an entirely satisfying performance/accuracy tradeoff. While dissociating b and t slightly complicates the implementation, it allows us to find the best possible compromise between performance and accuracy, as shown in Figure 5.5. Figure 5.5a shows that increasing b from $t = 256$ to $16t = 4096$ barely impacts the error, even though the bound is 16 times larger (this is partly explained by the probabilistic arguments discussed in the previous section, which suggest the error should increase by only factor around $\sqrt{16}$—but even this prediction turns out to be pessimistic for large n). In turn, taking a larger b significantly improves the performance of `FABsum`: Figure 5.5b shows that with $b = 16t$ the performance loss reduces significantly to about 3%.

6. Conclusion. As problem sizes n in scientific computing continue to grow, rounding error bounds proportional to nu, where u is the unit roundoff, become less satisfactory—especially in the context of the low precision arithmetics that are in-



(a) Backward error.

(b) Performance in Gflops (using 28 cores).

Fig. 5.5: Performance and accuracy of $m \times n$ by $n \times p$ matrix–matrix multiplication in single precision with $m = p = 4096$, tile size $t = 256$, varying n. We compare the standard implementation based on blocked summation with the `FABsum` algorithm for different choices of block size b varying from t to $16t = 4096$.

creasingly available in hardware. Error bounds for (blocked recursive) summation have the form $nu + O(u^2)$ and so can readily exceed 1 in practice. We have proposed a class of blocked summation algorithms, `FABsum`, that uses a fast algorithm to sum blocks of numbers and an accurate algorithm to sum the resulting local sums. When the fast algorithm is recursive summation and the accurate algorithm is either compensated summation or recursive summation in extended precision, `FABsum` has a backward error bound $(b+1)u+O(u^2)$; this bound is about a factor n/b smaller than that for standard summation and, for constant b, does not grow with n to first order.

When `FABsum` is used in the core linear algebra operations of matrix multiplication and solution of linear systems by LU factorization, error bounds with constant first order term are again obtained, as shown in section 4. Our numerical experiments show that `FABsum` does indeed produce substantial reductions to the actual backward errors in practice, and can do so with only a small reduction in performance in the context of the high performance state-of-the-art linear algebra library PLASMA.

`FABsum` provides an attractive way to obtain more accurate sums and more accurate linear algebra kernel evaluations, especially for low precisions.

Acknowledgments. This research made use of the resources of High Performance Computing Center North (HPC2N).

REFERENCES

[1] M. Badin, L. Bic, M. Dillencourt, and A. Nicolau, *Improving accuracy for matrix multiplications on GPUs*, Scientific Programming, 19 (2011), pp. 3–11, https://doi.org/10.3233/SPR-2011-0315, https://www.hindawi.com/journals/sp/2011/417569/abs/.

[2] E. Carson and N. J. Higham, *Accelerating the solution of linear systems by iterative refinement in three precisions*, SIAM J. Sci. Comput., 40 (2018), pp. A817–A847, https://doi.org/10.1137/17M1140819.

[3] A. M. Castaldo, R. C. Whaley, and A. T. Chronopoulos, *Reducing floating point error in dot product using the superblock family of algorithms*, SIAM J. Sci. Comput., 31 (2008), pp. 1156–1174, https://doi.org/10.1137/070679946.

[4] J. Dongarra, M. Gates, J. Kurzak, P. Luszczek, P. Wu, I. Yamazaki, A. YarKhan, M. Abalenkovs, N. Bagherpour, S. Hammarling, and J. Sistek, *PLASMA: Parallel linear algebra software for multicore using OpenMP*, ACM Trans. Math. Software, (2019). To appear.

[5] D. Goldberg, *What every computer scientist should know about floating-point arithmetic*, ACM Computing Surveys, 23 (1991), pp. 5–48, https://doi.org/10.1145/103162.103163.

[6] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham, *Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC '18 (Dallas, TX), Piscataway, NJ, USA, 2018, IEEE Press, pp. 47:1–47:11, http://dl.acm.org/citation.cfm?id=3291656.3291719.

[7] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second ed., 2002, https://doi.org/10.1137/1.9780898718027.

[8] N. J. Higham and T. Mary, *A new approach to probabilistic rounding error analysis*, MIMS EPrint 2018.33, Manchester Institute for Mathematical Sciences, The University of Manchester, UK, Nov. 2018, http://eprints.maths.manchester.ac.uk/2673/.

[9] N. J. Higham and T. Mary, *A new preconditioner that exploits low-rank approximations to factorization error*, SIAM J. Sci. Comput., 41 (2019), pp. A59–A82, https://doi.org/10.1137/18M1182802.

[10] N. J. Higham and S. Pranesh, *Simulating low precision floating-point arithmetic*, MIMS EPrint 2019.xx, Manchester Institute for Mathematical Sciences, The University of Manchester, UK, 2019. In preparation.

[11] W. Kahan, *Further remarks on reducing truncation errors*, Comm. ACM, 8 (1965), p. 40, https://doi.org/10.1145/363707.363723.

[12] D. E. Knuth, *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*, Addison-Wesley, Reading, MA, USA, third ed., 1998.

[13] D. Malone, *To what does the harmonic series converge?*, Irish Math. Soc. Bulletin, 71 (2013), pp. 59–66, https://www.maths.tcd.ie/pub/ims/bull71/.

[14] C. B. Moler, *"Half precision" 16-bit floating point arithmetic*. http://blogs.mathworks.com/cleve/2017/05/08/half-precision-16-bit-floating-point-arithmetic/, May 2017.

[15] O. Møller, *Quasi double-precision in floating point addition*, BIT, 5 (1965), pp. 37–50, https://doi.org/10.1007/BF01975722, http://link.springer.com/10.1007/BF01975722.

[16] W. Oettli and W. Prager, *Compatibility of approximate solution of linear equations with given error bounds for coefficients and right-hand sides*, Numer. Math., 6 (1964), pp. 405–409, https://doi.org/10.1007/BF01386090.