published in SIAM Journal on Scientific Computing (SISC), 26(6):1955-1988, 2005.

ACCURATE SUM AND DOT PRODUCT*

TAKESHI OGITA †, SIEGFRIED M. RUMP ‡, AND SHIN'ICHI OISHI §

Abstract. Algorithms for summation and dot product of floating point numbers are presented which are fast in terms of *measured computing time*. We show that the computed results are as accurate as if computed in twice or K-fold working precision, $K \geq 3$. For twice the working precision our algorithms for summation and dot product are some 40 % faster than the corresponding XBLAS routines while sharing similar error estimates. Our algorithms are widely applicable because they require only addition, subtraction and multiplication of floating point numbers in the same working precision as the given data. Higher precision is unnecessary, algorithms are straight loops without branch, and no access to mantissa or exponent is necessary.

Key words. accurate summation, accurate dot product, fast algorithms, verified error bounds, high precision

AMS subject classifications. 15-04, 65G99, 65-04

1. Introduction.

We present fast algorithms to compute approximations of high quality of the sum and the dot product of floating point numbers. We show that results share the same error estimates as if computed in twice or K-fold working precision and rounded back to working precision (for precise meaning, see Subsection 1.2).

We interpret fast not only in terms of floating point operations (flops) but in terms of *measured computing time*. Our algorithms do not require other than one working precision. Since summation and dot product are most basic tasks in numerical analysis, there are numerous algorithms for that, among them [6, 9, 11, 14, 17, 18, 19, 20, 22, 25, 26, 27, 28, 29, 31, 34, 35, 36, 37, 38, 41]. Higham [15] devotes an entire chapter to summation. Accurate summation or dot product algorithms have various applications in many different areas of numerical analysis. Excellent overviews can be found in [15, 26].

1.1. Previous work.

We collect some notes on previous work to put our results into suitable perspective; our remarks are by no means complete. Dot products can be transformed into sums, therefore much effort is concentrated on accurate summation of floating point numbers. In the following we assume the computer arithmetic to satisfy the IEEE 754 standard [2].

Many approaches to diminish rounding errors in floating point summation sort input data by absolute value. If all summands are of the same sign, increasing ordering is best. However, if summation is subject to cancellation, decreasing order may be superior [15]. A major drawback of such approaches is that optimization of code by today's compilers is substantially jeopardized by branches.

Floating point summation is frequently improved by *compensated summation*. Here, a cleverly designed correction term is used to improve the result. One of the first is the following one due to Dekker [10].

*This research was partially supported by 21st Century COE Program (Productive ICT Academia Program, Waseda University) from the Ministry of Education, Science, Sports and Culture of Japan.

†Graduate School of Science and Engineering, Waseda University, 3-4-1 Okubo Shinjuku-ku, Tokyo 169-8555, Japan (ogita@waseda.jp).

‡Institut für Informatik III, Technische Universität Hamburg-Harburg, Schwarzenbergstraße 95, Hamburg 21071, Germany (rump@tu-harburg.de).

§Department of Computer Science, School of Science and Engineering, Waseda University, 3-4-1 Okubo Shinjuku-ku, Tokyo 169-8555, Japan (oishi@waseda.jp).

ALGORITHM 1.1. *Compensated summation of two floating point numbers.*

$$\begin{array}{l}\text{function } [x, y] = \texttt{FastTwoSum}(a, b) \\ \quad x = \mathrm{fl}(a + b) \\ \quad y = \mathrm{fl}((a - x) + b)\end{array}$$

For floating point arithmetic with rounding to nearest and base 2, e.g. IEEE 754 arithmetic, Dekker [10] showed in 1971 that the correction is *exact* if the input is ordered by magnitude, that is

$$x + y = a + b \tag{1.1}$$

provided $|a| \geq |b|$. A similar correction is used in the Kahan-Babuška algorithm [3, 32, 31], and in a number of variants (e.g. [16, 31]). Those algorithms are almost ideally backward stable, that is for floating point numbers $p_i, 1 \leq i \leq n$, the computed sum $\tilde{s}$ satisfies

$$\tilde{s} = \sum_{i=1}^{n} p_i(1 + \varepsilon_i), \quad |\varepsilon_i| \leq 2\texttt{eps} + \mathcal{O}(\texttt{eps}^2). \tag{1.2}$$

An interesting variant of the Kahan-Babuška algorithm was given by Neumaier [31]. The result $\tilde{s}$ of Algorithm IV in his paper satisfies

$$|\tilde{s} - \sum_{i=1}^{n} p_i| \leq \texttt{eps}|\sum_{i=1}^{n} p_i| + (0.75n^2 + n)\texttt{eps}^2 \sum_{i=1}^{n} |p_i|, \quad \text{for } 3n\texttt{eps} \leq 1. \tag{1.3}$$

Without knowing, Neumaier uses Dekker's Algorithm 1.1 ensuring $|a| \geq |b|$ by comparison. Neumaier's result is of a quality *as if* computed in twice the working precision and then rounded into working precision. If input data is sorted decreasingly by absolute value, Priest showed that two extra applications of the compensation process produces a *forward stable* result of almost optimal relative accuracy [37].

However, branches may increase computing time due to lack of compiler optimization. Already in 1969, Knuth [23] presented a simple algorithm (that is Algorithm 3.1 (`TwoSum`) in this paper) to compute x and y satisfying (1.1) regardless of the magnitude of a and b. The algorithm requires 6 flops *without* branch. Counting absolute value and comparison as one flop, Dekker's Algorithm 1.1 with ensuring $|a| \geq |b|$ by comparison requires 6 flops as well; however, because of less optimized code, it is up to 50 % slower than Knuth's method. Therefore we will develop branch-free algorithms to be fast in terms of execution time. Combining Kahan-Babuška's and Knuth's algorithm is our first summation algorithm `Sum2`.

Knuth's algorithm, like Dekker's with sorting, transforms any pair of floating point numbers (a, b) into a new pair (x, y) with

$$x = \mathrm{fl}(a + b) \quad \text{and} \quad a + b = x + y.$$

We call an algorithm with this property *error-free transformation*. Such transformations are in the center of interest of our paper. Extending the principle of an error-free transformation of two summands to n summands is called "distillation algorithms" by Kahan [21]. Here input floating point numbers $p_i, 1 \leq i \leq n$, are transformed into $p_i^{(k)}$ with $\sum_{i=1}^{n} p_i = \sum_{i=1}^{n} p_i^{(k)}$ for $k \geq 1$. Some distillation algorithms

use sorting of input data by absolute value leading to a computing time $\mathcal{O}(n \log n)$ for adding n numbers. An interesting new distillation algorithm is presented by Anderson [1]. He uses a clever way of sorting and deflating positive and negative summands. However, branches slow down computations, see Section 6.

Another transformation is presented by Priest [36]. He transforms input numbers $p_i, 1 \leq i \leq n$ into a non-overlapping sequence q_i, such that the mantissa bit of lowest significance of q_i is greater than the one of highest significance of q_{i+1} (see Figure 1.1).

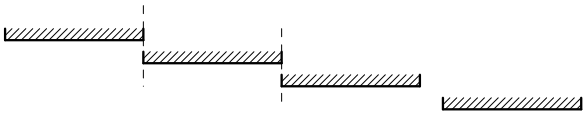

FIG. 1.1. *Non-overlapping sequence by Priest's scheme*

Shewchuk [41] weakens this into nonzero-overlapping sequences as shown in Figure 1.2. This means that mantissa bits of q_i and q_{i+1} may overlap, but only if the corresponding bits of q_i are zero. He shows that this simplifies normalization and improves performance.

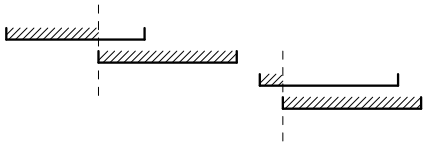

FIG. 1.2. *Nonzero-overlapping sequence by Shewchuk's scheme*

Quite different approaches use the fact that due to the limited exponent range, all IEEE 754 double precision floating point numbers can be added in a superlong accumulator (cf. Kulisch and Miranker [24]). Doubling the size of the accumulator creates algorithms for dot products as well. Splitting mantissas and adjustment according to exponents requires some bit manipulations. The result within the long accumulator represents the *exact* sum or dot product and can be rounded to the desired precision.

A simple heuristic to improve the accuracy of matrix products is implemented in INTLAB [39]. Matrices of floating point numbers $A = (a_{ij}), B = (b_{ij})$ are first split into $A = A_1 + A_0$ and $B = B_1 + B_0$, where the (i, j)-th entries of A_1 and B_1 comprise of the first M significant bits of a_{ij} and b_{ij}, respectively. For IEEE 754 double precision we choose $M = 17$. The matrix product $A \cdot B$ is then approximated by $A_1 \cdot B_1 + (A_1 \cdot B_0 + A_0 \cdot B)$. The heuristic is that for not too large dimension and for input a_{ij}, b_{ij} not differing too much in magnitude, the product $A_1 \cdot B_1$ is exact, so that the final approximation is of improved accuracy. This method is implemented in the routine `lssresidual` for residual correction of linear systems in INTLAB.

Another interesting and similar approach was recently presented by Zielke and Drygalla [44]. They split input numbers p_i according to the scheme in Figure 1.3.

The chunks of maximally M bits start at the most significant bit of all input data p_i. For proper choice of M depending on n and the input format, summation of the most significant chunks is exact. Cancellation can be monitored by the magnitude of the previous sum when adding the next chunks. Note that in the INTLAB method

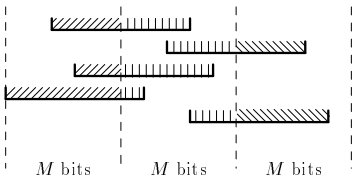

FIG. 1.3. *Zielke and Drygalla's scheme*

the splitting for all matrix entries is independent of the exponent of the entries and can be performed without branch, whereas here the splitting is individual for every matrix element. On the other hand, the method by Zielke and Drygalla allows to compute a forward stable result of the dot product. The advantage over the long accumulator approach is that the amount of work is determined by the condition of the problem.

Recently, Demmel and Hida presented algorithms using a wider accumulator [11]. Floating point numbers $p_i, 1 \le i \le n$, given in working precision with t bits in the mantissa are added in an extra-precise accumulator with T bits, $T > t$. Four algorithms are presented with and without sorting input data. The authors give a detailed analysis of the accuracy of the result depending on t, T and the number of summands. In contrast to our methods which use only working precision they need in any case some extra-precise floating point format.

Finally, XBLAS [26], the Extended and Mixed Precision BLAS, contains algorithms based on [4, 5] which are very similar to ours. However, XBLAS treats lower order terms of summation and dot product in a more complicated way than ours resulting in an increase of flop count of about 40 %. In the same computing time as XBLAS our algorithms can produce an accurate result for much more ill-conditioned dot products. For details, see Sections 4 and 5, and for timing and accuracy see Section 6.

With respect to the *result* of a summation or dot product algorithm we can grossly distinguish three different approaches:

i) heuristic/numerical evidence for improvement of result accuracy,
ii) result accuracy as if computed in higher precision,
iii) result with prescribed accuracy.

Algorithms of the second class are backward stable in the sense of (1.2) with respect to a certain `eps`, whereas algorithms of the third class are forward stable in a similar sense. Our methods belong to the second class and, with small modifications described below, to the third class. To our knowledge all known methods belonging to the second or third class except the new XBLAS approach [26] bear one or more of the following disadvantages:

- sorting of input data is necessary, either
  i) by absolute value or,
  ii) by exponent,
- the inner loop contains branches,
- besides working precision, some extra (higher) precision is necessary,
- access to mantissa and/or exponent is necessary.

Each of those properties may slow down the performance significantly and/or restrict application to specific computers or compilers. In contrast, our approach uses only floating point addition, subtraction and multiplication in working precision

without any of those disadvantages.

1.2. Our goal. We will present algorithms for summation and dot product without branch using only one working precision to compute a result s of the same quality *as if* computed in K-fold precision and rounded to working precision. This means the following.

Let floating point numbers $p_i, 1 \le i \le n$, with t-bit precision be given. We call that working precision. Then the result s shall satisfy

$$|s - \sum_{i=1}^{n} p_i| \le \texttt{eps}|s| + (\varphi\texttt{eps})^K \sum_{i=1}^{n} |p_i|, \tag{1.4}$$

with a moderate constant φ and $\texttt{eps} := 2^{-t}$. The second term in the right hand side of (1.4) reflects computation K-fold precision, and the first one rounding back into working precision.

1.3. Our approach. Our methods are based on iterative application of error-free transformations. This has been done before, Shewchuk for example uses such a scheme for sorted input data to produce nonzero-overlapping sequences [41].

We extend the error-free transformation of two floating point numbers to vectors of arbitrary length (Algorithm 4.3, `VecSum`) and use it to compute a result *as if* computed in twice the working precision. Then we show that iterative (K-1)-fold application of `VecSum` can be used to produce a result *as if* computed in K-fold working precision (Algorithm 4.8, `SumK`).

Furthermore, Dekker and Veltkamp [10] gave an error-free transformation of the product into the sum of two floating point numbers. We use this to create dot product algorithms with similar properties as our summation algorithms.

Moreover, we give simple criteria to check the *accuracy* of the result. This check is also performed in working precision and in rounding to nearest. Nevertheless it allows to compute valid error bounds for the result of a summation and of a dot product.

The structure of our algorithms is clear and simple. Moreover, the error-free transformations allow elegant error estimations diminishing involved battles with rounding error terms to a minimum. All error analysis is due to the second author.

1.4. Outline of the paper. The paper is organized as follows. After introducing notation we briefly summarize the algorithms for error-free summation and dot product of *two* floating point numbers and their properties. In Section 4 we present and analyze our algorithms for summation corresponding to 2-fold (i.e. doubled) and to K-fold working precision, and in the next section we do the same for the dot product. In Section 6 numerical examples for extremely ill-conditioned problems are presented, as well as timing, accuracy, comparison with other algorithms and some remarks on practical applications. In the concluding remarks we address possible implications for the design of the hardware of digital computers.

2. Notation. Throughout the paper we assume a floating point arithmetic adhering to IEEE 754 floating point standard [2]. We assume that no overflow occurs, but allow underflow. We will use only one working precision for floating point computations. If this working precision is IEEE 754 double precision, this corresponds to 53 bits precision including an implicit bit. The set of working precision floating point numbers is denoted by $\mathbb{F}$, the relative rounding error unit by `eps`, and the underflow unit by `eta`. For IEEE 754 double precision we have $\texttt{eps} = 2^{-53}$ and $\texttt{eta} = 2^{-1074}$.

We stress that IEEE 754 arithmetic is not necessary if the error-free transformations `TwoSum` and `TwoProduct` to be described in the next section are available.

We denote by $\mathrm{fl}(\cdot)$ the result of a floating point computation, where all operations inside the parentheses are executed in working precision. If the order of execution is ambiguous *and* is crucial, we make it unique by using parentheses. Floating point operations according to IEEE 754 satisfy [15]

$$\begin{array}{rcll} \mathrm{fl}(a \circ b) & = & (a \circ b)(1+\varepsilon_1) & \\ & = & (a \circ b)/(1+\varepsilon_2) & \text{for } \circ \in \{+,-\} \text{ and } |\varepsilon_\nu| \le \mathtt{eps}, \end{array}$$

and

$$\begin{array}{rcll} \mathrm{fl}(a \circ b) & = & (a \circ b)(1+\varepsilon_1) + \eta_1 & \\ & = & (a \circ b)/(1+\varepsilon_2) + \eta_2 & \text{for } \circ \in \{\cdot,/\} \text{ and } |\varepsilon_\nu| \le \mathtt{eps}, |\eta_\nu| \le \mathtt{eta}. \end{array}$$

Addition and subtraction is exact in case of underflow [13], and $\varepsilon_1\eta_1 = \varepsilon_2\eta_2 = 0$ for multiplication and division. This implies

$$\begin{array}{rcll} |a \circ b - \mathrm{fl}(a \circ b)| & \le & \mathtt{eps}|a \circ b| & \\ |a \circ b - \mathrm{fl}(a \circ b)| & \le & \mathtt{eps}|\mathrm{fl}(a \circ b)| & \text{for } \circ \in \{+,-\}, \end{array} \tag{2.1}$$

and

$$\begin{array}{rcll} |a \circ b - \mathrm{fl}(a \circ b)| & \le & \mathtt{eps}|a \circ b| + \mathtt{eta} & \\ |a \circ b - \mathrm{fl}(a \circ b)| & \le & \mathtt{eps}|\mathrm{fl}(a \circ b)| + \mathtt{eta} & \text{for } \circ \in \{\cdot,/\} \end{array} \tag{2.2}$$

for all floating point numbers $a, b \in \mathbb{F}$, the latter because $\varepsilon_1\eta_1 = \varepsilon_2\eta_2 = 0$. Note that $a \circ b \in \mathbb{R}$ and $\mathrm{fl}(a \circ b) \in \mathbb{F}$, but in general $a \circ b \notin \mathbb{F}$. It is known [29, 10, 7, 8] that the approximation error of floating point operations is itself a floating point number:

$$\begin{array}{rcll} x = \mathrm{fl}(a \pm b) & \Rightarrow & a \pm b = x + y & \text{for} \quad y \in \mathbb{F}, \\ x = \mathrm{fl}(a \cdot b) & \Rightarrow & a \cdot b = x + y & \text{for} \quad y \in \mathbb{F}, \end{array} \tag{2.3}$$

where no underflow is assumed for multiplication. Similar facts hold for division and square root but are not needed in this paper. These are error-free transformations of the pair (a, b) into (x, y), where x is the result of the corresponding floating point operation. Note that no information is lost; the equalities in (2.3) are mathematical identities for all floating point numbers $a, b \in \mathbb{F}$.

Throughout the paper the number of flops of an algorithm denotes the number of floating point operations counting additions, subtractions, multiplications and absolute value separately.

We use standard notation and standard results for our error estimations. The quantities γ_n are defined as usual [15] by

$$\gamma_n := \frac{n\mathtt{eps}}{1 - n\mathtt{eps}} \qquad \text{for } n \in \mathbb{N}.$$

When using γ_n, we implicitly assume $n\mathtt{eps} < 1$. For example, for floating point numbers $a_i \in \mathbb{F}$, $1 \le i \le n$, this implies [15, Lemma 8.4]

$$a = \mathrm{fl}\left(\sum_{i=1}^{n} a_i\right) \quad \Rightarrow \quad |a - \sum_{i=1}^{n} a_i| \le \gamma_{n-1} \sum_{i=1}^{n} |a_i|. \tag{2.4}$$

Note that this result holds independent of the order of the floating point summation and also in the presence of underflow. We mention that we are sometimes a little generous with our estimates, for example, replacing γ_{4n-2} by γ_{4n}, or an assumption $2(n-1)\mathtt{eps} < 1$ by $2n\mathtt{eps} < 1$ to make formulas a little smoother.

3. Error-free transformations. Our goal is to extend the error-free transformations for the sum and product of *two* floating point numbers to vector sums and to dot products. Therefore, (2.3) will play a fundamental role in the following. Fortunately, the quantities x, y are effectively computable, without branches and only using ordinary floating point addition, subtraction and multiplication. For addition we will use the following algorithm by Knuth [23].

ALGORITHM 3.1. *Error-free transformation of the sum of two floating point numbers.*

$$\begin{aligned}&\text{function } [x, y] = \texttt{TwoSum}(a, b)\\ &\quad x = \mathrm{fl}(a + b)\\ &\quad z = \mathrm{fl}(x - a)\\ &\quad y = \mathrm{fl}((a - (x - z)) + (b - z))\end{aligned}$$

We use Matlab-like [43] notation. Algorithm 3.1 transforms two input floating point numbers a, b into two output floating point numbers x, y such that [23]

$$a + b = x + y \quad \text{and} \quad x = \mathrm{fl}(a + b). \tag{3.1}$$

The proof for that [23] is also valid in the presence of underflow since addition and subtraction is exact in this case.

The multiplication routine needs to split the input arguments into two parts. For the number t given by $\texttt{eps} = 2^{-t}$, we define $s := \lceil t/2 \rceil$; in IEEE 754 double precision we have $t = 53$ and $s = 27$. The following algorithm by Dekker [10] splits a floating point number $a \in \mathbb{F}$ into two parts x, y, where both parts have at most $s - 1$ nonzero bits. In a practical implementation, "`factor`" can be replaced by a constant.

ALGORITHM 3.2. *Error-free splitting of a floating point number into two parts.*

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

It seems absurd that a 53-bit number can be split into two 26-bit numbers. However, the trick is that Dekker uses one sign bit in the splitting. Note also that no access to mantissa or exponent of the input "a" is necessary, standard floating point operations suffice.

The multiplication to calculate "c" in Algorithm 3.2 cannot cause underflow except when the input "a" is deep in the gradual underflow range. Since addition and subtraction is exact in case of underflow, the analysis [10] of `Split` is still valid and we obtain

$$a = x + y \quad \text{and} \quad x \text{ and } y \text{ nonoverlapping with } |y| \le |x|.$$

With this the following multiplication routine by G.W. Veltkamp (see [10]) can be formulated.

ALGORITHM 3.3. *Error-free transformation of the product of two floating point numbers.*

$$\begin{aligned}&\text{function } [x, y] = \texttt{TwoProduct}(a, b)\\ &\quad x = \mathrm{fl}(a \cdot b)\\ &\quad [a_1, a_2] = \texttt{Split}(a)\\ &\quad [b_1, b_2] = \texttt{Split}(b)\\ &\quad y = \mathrm{fl}(a_2 \cdot b_2 - (((x - a_1 \cdot b_1) - a_2 \cdot b_1) - a_1 \cdot b_2))\end{aligned}$$

Note again that no branches, only basic and well optimizable sequences of floating point operations are necessary. In case no underflow occurs, we know [10] that

$$a \cdot b = x + y \quad \text{and} \quad x = \mathrm{fl}(a \cdot b).$$

In case of underflow of any of the five multiplications in Algorithm 3.3, suppose the algorithm is performed in a floating point arithmetic with the same length of mantissa but an exponent range so large that no underflow occurs. Denote the results by x' and y'. Then a rudimentary analysis yields $|y' - y| \le 5\texttt{eta}$. We do not aim to improve on that because underflow is rare and the quantities are almost always negligible.

We can summarize the properties of the algorithms `TwoSum` and `TwoProduct` as follows.

THEOREM 3.4. *Let $a, b \in \mathbb{F}$ and denote the results of Algorithm 3.1* (`TwoSum`) *by x, y. Then, also in the presence of underflow,*

$$a + b = x + y, \quad x = \mathrm{fl}(a + b), \quad |y| \le \texttt{eps}|x|, \quad |y| \le \texttt{eps}|a + b|. \tag{3.2}$$

The algorithm `TwoSum` *requires 6 flops.*

Let $a, b \in \mathbb{F}$ and denote the results of Algorithm 3.3 (`TwoProduct`) *by x, y. Then, if no underflow occurs,*

$$a \cdot b = x + y, \quad x = \mathrm{fl}(a \cdot b), \quad |y| \le \texttt{eps}|x|, \quad |y| \le \texttt{eps}|a \cdot b|, \tag{3.3}$$

and in the presence of underflow,

$$a \cdot b = x + y + 5\eta,\ x = \mathrm{fl}(a \cdot b),\ |y| \le \texttt{eps}|x| + 5\texttt{eta},\ |y| \le \texttt{eps}|a \cdot b| + 5\texttt{eta} \tag{3.4}$$

with $|\eta| \le \texttt{eta}$. The algorithm `TwoProduct` *requires 17 flops.*

Proof. The assertions follow by (2.1), (2.2), [23], [10] and the fact that addition and subtraction is exact in the presence of underflow. □

Note that the approximation error $y = a \cdot b - x$ is frequently available inside the processor, but inaccessible to the user. If this information were directly available, the computing time of `TwoProduct` would reduce to 1 flop. Alternatively, `TwoProduct` can be rewritten in a simple way if a Fused-Multiply-and-Add operation is available as in the Intel Itanium and many other processors (cf. [33]). This means that for $a, b, c \in \mathbb{F}$ the result of $\texttt{FMA}(a, b, c)$ is the rounded-to-nearest exact result $a \cdot b + c \in \mathbb{R}$. Then Algorithm 3.3 can be replaced by

ALGORITHM 3.5. *Error-free transformation of a product using Fused-Multiply-and-Add.*

function $[x, y] = \texttt{TwoProductFMA}(a, b)$
$x = \mathrm{fl}(a \cdot b)$
$y = \texttt{FMA}(a, b, -x)$

This reduces the computing time for the error-free transformation of the product to 2 flops. As `FMA` is a basic task and useful in many algorithms, we think a round-to-nearest sum of *three* floating point numbers is such. Suppose $\texttt{ADD3}(a, b, c)$ is a floating point number nearest to the exact sum $a + b + c \in \mathbb{R}$ for $a, b, c \in \mathbb{F}$. Then Algorithm 3.1 can be replaced by

ALGORITHM 3.6. *Error-free transformation of addition using* ADD3.

$$\begin{array}{l}\text{function } [x,y] = \texttt{TwoSumADD3}(a,b)\\ \quad x = \mathrm{fl}(a+b)\\ \quad y = \texttt{ADD3}(a,b,-x)\end{array}$$

This reduces the error-free transformation of the addition also to 2 flops. We will comment on this again in the last section.

4. Summation. Let floating point numbers $p_i \in \mathbb{F}$, $1 \le i \le n$, be given. In this section we aim to compute a good approximation of the sum $s = \sum p_i$. With Algorithm 3.1 (TwoSum) we have a possibility to add *two* floating point numbers with *exact* error term. So we may try to cascade Algorithm 3.1 and sum up the errors to improve the result of the ordinary floating point summation $\mathrm{fl}\,(\sum p_i)$.

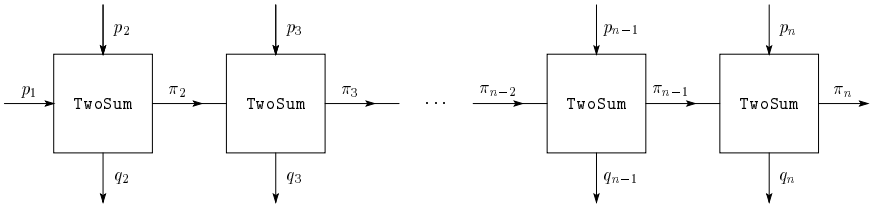

FIG. 4.1. *Cascaded error-free transformation*

Each of the boxes in Figure 4.1 represents Algorithm 3.1 (TwoSum), the error-free summation of two floating point numbers. A cascaded algorithm summing up error terms is as follows.

ALGORITHM 4.1. *Cascaded summation.*

$$\begin{array}{l}\text{function } \texttt{res} = \texttt{Sum2s}(p)\\ \quad \pi_1 = p_1;\ \sigma_1 = 0;\\ \quad \text{for } i = 2:n\\ \quad\quad [\pi_i, q_i] = \texttt{TwoSum}(\pi_{i-1}, p_i)\\ \quad\quad \sigma_i = \mathrm{fl}(\sigma_{i-1} + q_i)\\ \quad \texttt{res} = \mathrm{fl}(\pi_n + \sigma_n)\end{array}$$

Algorithm 4.1 is a compensated summation [15]. The well known Kahan-Babuška algorithm [32] is Algorithm 4.1 using FastTwoSum instead of TwoSum . Since the transformation $[\pi_i, q_i] = \texttt{FastTwoSum}(\pi_{i-1}, p_i)$ is only proved to be error-free if $|\pi_{i-1}| \ge |p_i|$, the result of the Kahan-Babuška algorithm is in general of less quality than Algorithm 4.1. Neumaier's Algorithm IV in [31] improves Kahan-Babuška by ordering $|\pi_{i-1}|$, $|p_i|$ before applying FastTwoSum . So it is mathematically identical to Algorithm 4.1 and only slowed down by sorting.

For the error analysis we first note

$$\pi_n = \mathrm{fl}\left(\sum_{i=1}^{n} p_i\right) \quad \text{and} \quad \sigma_n = \mathrm{fl}\left(\sum_{i=2}^{n} q_i\right) \tag{4.1}$$

which follows by successive application of (3.2). So π_n is the result of ordinary floating point summation in working precision, and σ_n is the (floating point) sum of the error

terms. The error-free transformation by each application of `TwoSum` allows an exact relation between the input p_i, the errors q_i and the floating point result π_n. We have

$$\pi_n = \pi_{n-1} + p_n - q_n = \pi_{n-2} + p_{n-1} - q_{n-1} + p_n - q_n = \pi_1 + \sum_{i=2}^{n}(p_i - q_i),$$

so that

$$s = \sum_{i=1}^{n} p_i = \pi_n + \sum_{i=2}^{n} q_i. \tag{4.2}$$

In other words, without the presence of rounding errors the result "`res`" of Algorithm 4.1 would be equal to the exact sum $s = \sum p_i$. We also note the well known fact that adding up the errors needs not necessarily improve the final result. Consider

$$p = [1 \;\; \theta \;\; \theta^2 \;\; -\theta \;\; -\theta^2 \;\; -1] \tag{4.3}$$

with a floating point number $\theta \in \mathbb{F}$ chosen small enough that $\mathrm{fl}(1+\theta) = \mathrm{fl}(1-\theta) = 1$ and $\mathrm{fl}(\theta + \theta^2) = \mathrm{fl}(\theta - \theta^2) = \theta$. Then Table 4.1 shows the intermediate results of Algorithm 4.1.

TABLE 4.1
Intermediate results of Algorithm 4.1 for (4.3)

i	p_i	π_i	q_i	σ_i
1	1	1		0
2	θ	1	θ	θ
3	θ^2	1	θ^2	θ
4	$-\theta$	1	$-\theta$	0
5	$-\theta^2$	1	$-\theta^2$	$-\theta^2$
6	-1	0	0	$-\theta^2$

The table implies

$$\texttt{res} = \mathrm{fl}(\pi_6 + \sigma_6) = -\theta^2 \quad \text{whereas } \pi_6 = \mathrm{fl}\left(\sum p_i\right) = 0 = \sum p_i = s.$$

So ordinary summation accidentally produces the exact result zero, whereas Algorithm 4.1 yields $\texttt{res} = -\theta^2$. In other words, we cannot necessarily expect improvement of the *accuracy* of the result by using Algorithm 4.1 instead of ordinary summation. However, we can show that the *error bound* improves significantly. Our algorithms produce, as we will see, a result *as if* computed in higher *precision*. For the proof of the first corresponding theorem on that we need a technical lemma.

LEMMA 4.2. *Let floating point numbers $\beta_0 \in \mathbb{F}$ and $b_i \in \mathbb{F}$, $1 \le i \le n$, be given. Suppose the floating point numbers $c_i, \beta_i \in \mathbb{F}$, $1 \le i \le n$, are computed by the following loop.*

$$\begin{aligned} &\text{for } i = 1 : n \\ &\qquad [\beta_i, c_i] = \texttt{TwoSum}(\beta_{i-1}, b_i) \end{aligned}$$

Then

$$\sum_{i=1}^{n} |c_i| \le \gamma_{n-1} \sum_{i=1}^{n} |b_i| \quad \textit{for } \beta_0 = 0, \tag{4.4}$$

and

$$\sum_{i=1}^{n} |c_i| \le \gamma_n \left(|\beta_0| + \sum_{i=1}^{n} |b_i| \right) \quad \textit{for general } \beta_0 \in \mathbb{F}. \tag{4.5}$$

Proof. The second inequality (4.5) is an immediate consequence of the first one (4.4), so we assume $\beta_0 = 0$. For later use we state

$$\gamma_{n-1} + \mathtt{eps}(1+\gamma_n) \le \frac{(n-1)\mathtt{eps}}{1-n\mathtt{eps}} + \frac{\mathtt{eps}}{1-n\mathtt{eps}} = \gamma_n. \tag{4.6}$$

We proceed by induction and note that for $n = 1$ we have $\beta_1 = b_1$ and $c_1 = 0$. Suppose (4.4) is true for some $n \ge 1$. Then

$$\beta_{n+1} = \mathrm{fl}\left(\sum_{i=1}^{n+1} b_i \right),$$

and (3.2) and (2.4) imply

$$|c_{n+1}| \le \mathtt{eps}|\beta_{n+1}| \le \mathtt{eps}(1+\gamma_n) \sum_{i=1}^{n+1} |b_i|.$$

Using the induction hypothesis and (4.6) yields

$$\sum_{i=1}^{n+1} |c_i| \le \gamma_{n-1} \sum_{i=1}^{n} |b_i| + \mathtt{eps}(1+\gamma_n) \sum_{i=1}^{n+1} |b_i| \le \gamma_n \sum_{i=1}^{n+1} |b_i|. \quad \square$$

Before analyzing Algorithm 4.1 (`TwoSum`), we stress that it can be viewed as an *error-free vector transformation*: The n-vector p is transformed into the n-vector $[q_{2\ldots n}\, \pi_n]$ with properties similar to (3.1):

$$\begin{aligned} &i) \quad \sum_{i=1}^{n} p_i = \sum_{i=2}^{n} q_i + \pi_n = s \\ &ii) \quad \pi_n = \mathrm{fl}\left(\sum_{i=1}^{n} p_i \right). \end{aligned} \tag{4.7}$$

To underline this, we rewrite the first part of Algorithm 4.1 so that the vector entries are overwritten. This results in the following compact notation.

ALGORITHM 4.3. *Error-free vector transformation for summation.*

```
function p = VecSum(p)
    for i = 2 : n
        [p_i, p_{i-1}] = TwoSum(p_i, p_{i-1})
```

The vector p is transformed without changing the sum, and p_n is replaced by $\mathrm{fl}\,(\sum p_i)$. Kahan [21] calls this a "distillation algorithm". The following algorithm is equivalent to Algorithm 4.1, the results "`res`" are identical.

ALGORITHM 4.4. *Cascaded summation equivalent to Algorithm 4.1.*

```
function res = Sum2(p)
    for i = 2 : n
        [p_i, p_{i-1}] = TwoSum(p_i, p_{i-1})
    res = fl( (sum_{i=1}^{n-1} p_i) + p_n )
```

PROPOSITION 4.5. *Suppose Algorithm 4.4* (`Sum2`) *is applied to floating point numbers* $p_i \in \mathbb{F}$, $1 \le i \le n$, *set* $s := \sum p_i \in \mathbb{R}$ *and* $S := \sum |p_i|$ *and suppose* $n\mathtt{eps} < 1$. *Then, also in the presence of underflow,*

$$|\mathtt{res} - s| \le \mathtt{eps}|s| + \gamma_{n-1}^2 S. \tag{4.8}$$

Algorithm 4.4 requires $7(n-1)$ *flops. If Algorithm 3.6* (`TwoSumADD3`) *is used instead of Algorithm 3.1* (`TwoSum`)*, then Algorithm 4.4 requires* $3(n-1)$ *flops.*

Remark 1. The final result "`res`" is calculated as a floating point sum in the last line of Algorithm 4.4 or, equivalently, in Algorithm 4.1. The exact result "s" is, in general, not a floating point number. Therefore we cannot expect an error bound (4.8) better than $\mathtt{eps}|s|$. Besides that, the error bound (4.8) tells that the quality of the result "`res`" is *as if* computed in *doubled* working precision and rounded to working precision.

Remark 2. The important point in the analysis is the estimation of the error of the final summation $\mathrm{fl}(\pi_n + \sigma_n)$. A straightforward analysis includes a term $\mathtt{eps}|\pi_n|$, for which the only bound we know is $\mathtt{eps}S$. This, of course, would ruin the whole analysis. Instead, we can use the mathematical property (4.2) of the error-free transformation `TwoSum` implying that $\pi_n + \sum q_i - s$ is equal to zero.

Remark 3. It is very instructive to express and interpret (4.8) in terms of the condition number of summation. The latter is defined for $\sum p_i \ne 0$ by

$$\mathrm{cond}\left(\sum p_i\right) := \lim_{\varepsilon \to 0} \sup \left\{ \left| \frac{\sum \widetilde{p}_i - \sum p_i}{\varepsilon \sum p_i} \right| : \ |\widetilde{p}| \le \varepsilon |p| \right\},$$

where absolute value and comparison is to be understood componentwise. Obviously

$$\mathrm{cond}\left(\sum p_i\right) = \frac{\sum |p_i|}{|\sum p_i|} = \frac{S}{|s|}. \tag{4.9}$$

Inserting this into (4.8) yields

$$\frac{|\mathtt{res} - s|}{|s|} \le \mathtt{eps} + \gamma_{n-1}^2 \cdot \mathrm{cond}\left(\sum p_i\right).$$

In other words, the bound for the relative error of the result "`res`" is essentially $(n\mathtt{eps})^2$ times the condition number plus the inevitable summand `eps` for rounding the result to working precision.

Remark 4. Neumaier's [31] proves for his mathematically identical Algorithm IV the similar estimation

$$|\mathtt{res} - s| \le \mathtt{eps}|s| + (0.75n^2 + n)\mathtt{eps}^2 S \ ,$$

provided $3n\mathtt{eps} \le 1$. His proof is involved.

Proof of Proposition 4.5. For the analysis of Algorithm 4.4 we use the equivalent formulation Algorithm 4.1. In the notation of Algorithm 4.1 we know by (4.4)

$$\sum_{i=2}^{n} |q_i| \le \gamma_{n-1} \sum_{i=1}^{n} |p_i| = \gamma_{n-1} S. \tag{4.10}$$

Then $\sigma_n = \mathrm{fl}\,(\sum_{i=2}^{n} q_i)$ and (2.4) imply

$$|\sigma_n - \sum_{i=2}^{n} q_i| \le \gamma_{n-2} \sum_{i=2}^{n} |q_i| \le \gamma_{n-2}\gamma_{n-1} S. \tag{4.11}$$

Furthermore, $\mathtt{res} = \mathrm{fl}(\pi_n + \sigma_n)$ means $\mathtt{res} = (1+\varepsilon)(\pi_n + \sigma_n)$ with $|\varepsilon| \le \mathtt{eps}$, so that in view of (4.2),

$$\begin{aligned} |\mathtt{res} - s| &= |\mathrm{fl}(\pi_n + \sigma_n) - s| = |(1+\varepsilon)(\pi_n + \sigma_n - s) + \varepsilon s| \\ &= |(1+\varepsilon)(\pi_n + \sum_{2}^{n} q_i - s) + (1+\varepsilon)(\sigma_n - \sum_{i=2}^{n} q_i) + \varepsilon s| \\ &\le (1+\mathtt{eps})|\sigma_n - \sum_{i=2}^{n} q_i| + \mathtt{eps}|s| \\ &\le (1+\mathtt{eps})\gamma_{n-2}\gamma_{n-1} S + \mathtt{eps}|s|, \end{aligned} \tag{4.12}$$

and the result follows by $(1+\mathtt{eps})\gamma_{n-2} \le \gamma_{n-1}$. ∎

The high precision summation in XBLAS [26] is fairly similar to Algorithm 4.4 except that lower order terms are treated a different way. The corresponding algorithm `BLAS_dsum_x` in [26] is as follows:

ALGORITHM 4.6. *XBLAS quadruple precision summation.*

```
function s = SumXBLAS(p)
    s = 0;  t = 0;
    for i = 1 : n
        [t1, t2] = TwoSum(s, p_i)
        t2 = t2 + t;
        [s, t] = FastTwoSum(t1, t2)
```

Lower order terms are added in `SumXBLAS` using an extra addition and Algorithm 1.1 (`FastTwoSum`) to generate a pair $[s,t]$ with $s+t$ approximating $\sum p_i$ in quadruple precision. Omitting the last statement in Algorithm 4.1 (or equivalently in Algorithm 4.4, `Sum2`) it follows by (4.7, i)) and (4.11) that the pair $[\pi_n, \sigma_n]$ is of quadruple precision as well. Output of `SumXBLAS` is the higher order part s which satisfies a similar error estimate as `Sum2`. However, XBLAS summation requires $10n$ flops as compared to $7(n-1)$ flops for `Sum2`. For timing and accuracy see Section 6.

The error bound (4.8) for the result `res` of Algorithm 4.4 is not computable since it involves the exact value s of the sum. Next we show how to compute a valid error bound in pure floating point in round to nearest, which is also less pessimistic. We use again the notation of the equivalent Algorithm 4.1. Following (4.12) and (4.11) we have

$$\begin{aligned} |\mathtt{res} - s| &= |\mathrm{fl}(\pi_n + \sigma_n) - s| \le \mathtt{eps}|\mathtt{res}| + |\sigma_n + \pi_n - s| \\ &\le \mathtt{eps}|\mathtt{res}| + |\sum_{i=2}^{n} q_i + \pi_n - s| + \gamma_{n-2} \sum_{i=1}^{n} |q_i| \\ &\le \mathtt{eps}|\mathtt{res}| + (1+\mathtt{eps})^{n-2}\gamma_{n-2}\mathrm{fl}(\sum_{i=1}^{n} |q_i|) \\ &\le \mathtt{eps}|\mathtt{res}| + \gamma_{2n-4}\alpha\ , \end{aligned}$$

where $\alpha := \mathrm{fl}(\sum_{i=2}^{n} |q_i|)$. If $m\mathtt{eps} < 1$ for $m \in \mathbb{N}$, then $\mathrm{fl}(m\mathtt{eps}) = m\mathtt{eps} \in \mathbb{F}$ and $\mathrm{fl}(1 - m\mathtt{eps}) = 1 - m\mathtt{eps} \in \mathbb{F}$, so that only division may cause a rounding error in the

computation of $\gamma_m = m\mathtt{eps}/(1-m\mathtt{eps})$. Therefore

$$\gamma_m \le (1-\mathtt{eps})^{-1}\mathrm{fl}(m\mathtt{eps}/(1-m\mathtt{eps})).$$

The floating point multiplication $\mathrm{fl}(\mathtt{eps}|\mathtt{res}|)$ may cause underflow, so setting $\beta := \mathrm{fl}(2n\mathtt{eps}/(1-2n\mathtt{eps})\alpha)$ and being a little generous with the lower order terms shows

$$\begin{aligned}
|\mathtt{res}-s| &\le \mathrm{fl}(\mathtt{eps}|\mathtt{res}|) + (1-\mathtt{eps})^4\gamma_{2n}\alpha + \mathtt{eta} \\
&\le (1-\mathtt{eps})\mathrm{fl}(\mathtt{eps}|\mathtt{res}|) + (1-\mathtt{eps})^3\mathrm{fl}(2n\mathtt{eps}/(1-2n\mathtt{eps}))\alpha \\
&\quad + \mathtt{eps}\cdot\mathrm{fl}(\mathtt{eps}|\mathtt{res}|) + \mathtt{eta} \\
&\le (1-\mathtt{eps})\mathrm{fl}(\mathtt{eps}|\mathtt{res}|) + (1-\mathtt{eps})^2\beta + \mathrm{fl}(\mathtt{eps}^2|\mathtt{res}|) + 2\mathtt{eta} \\
&\le (1-\mathtt{eps})\mathrm{fl}(\mathtt{eps}|\mathtt{res}|) + (1-\mathtt{eps})^2\beta \\
&\quad + (1-\mathtt{eps})^3\mathrm{fl}(2\mathtt{eps}^2|\mathtt{res}|) + (1-\mathtt{eps})^3 3\mathtt{eta} \\
&\le (1-\mathtt{eps})\mathrm{fl}(\mathtt{eps}|\mathtt{res}|) + (1-\mathtt{eps})^2\beta \\
&\quad + (1-\mathtt{eps})^2\mathrm{fl}\left(2\mathtt{eps}^2|\mathtt{res}| + 3\mathtt{eta}\right) \\
&\le (1-\mathtt{eps})\mathrm{fl}(\mathtt{eps}|\mathtt{res}|) + (1-\mathtt{eps})\mathrm{fl}\left(\beta + (2\mathtt{eps}^2|\mathtt{res}| + 3\mathtt{eta})\right) \\
&\le \mathrm{fl}\left(\mathtt{eps}|\mathtt{res}| + \left(\beta + (2\mathtt{eps}^2|\mathtt{res}| + 3\mathtt{eta})\right)\right) \ .
\end{aligned}$$

The following corollary translates the result into the notation of Algorithm 4.4.

COROLLARY 4.7. *Let floating point numbers $p_i \in \mathbb{F}$, $1 \le i \le n$, be given. Append the statements*

$$\begin{aligned}
&\text{if } 2n\mathtt{eps} \ge 1, \ \mathtt{error}\text{('dimension too large')}, \text{ end} \\
&\beta = (2n\mathtt{eps}/(1-2n\mathtt{eps}))\cdot\left(\sum_{i=1}^{n-1}|p_i|\right) \\
&\mathtt{err} = \mathtt{eps}|\mathtt{res}| + (\ \beta + (2\mathtt{eps}^2|\mathtt{res}| + 3\mathtt{eta})\ )
\end{aligned}$$

(to be executed in working precision) to Algorithm 4.4 (`Sum2`)*. If the error message is not triggered,* `err` *satisfies*

$$\mathtt{res} - \mathtt{err} \le \sum p_i \le \mathtt{res} + \mathtt{err}.$$

This is also true in the presence of underflow. The computation of `err` *requires* $2n+8$ *flops.*

For later use we estimate the sum of the absolute values of the transformed vector. Using (4.2) and (4.10) we obtain

$$\sum_{i=2}^{n}|q_i| + |\pi_n| = \sum_{i=2}^{n}|q_i| + |s - \sum_{i=2}^{n}q_i| \le |s| + 2\sum_{i=2}^{n}|q_i| \le |s| + 2\gamma_{n-1}S. \tag{4.13}$$

Denoting the output vector of Algorithm 4.3 (`VecSum`) by p' this means

$$\sum_{i=1}^{n}|p'_i| \le \left|\sum_{i=1}^{n}p_i\right| + 2\gamma_{n-1}\sum_{i=1}^{n}|p_i|. \tag{4.14}$$

If summation of the p_i is not too ill-conditioned (up to condition number $\mathtt{eps}^{-1}$, say), the result of Algorithm 4.4 (`Sum2`) is almost maximally accurate and there is nothing more to do. Suppose the summation is extremely ill-conditioned with condition number beyond $\mathtt{eps}^{-1}$. Then the vector p is transformed by `Sum2` into a new vector p' with p'_n being the result of ordinary floating point summation in working precision. Because of ill-condition, p'_n is subject to heavy cancellation, so $|p'_n|$ is of the order $\mathtt{eps}\sum|p_i|$. Moreover, (4.14) implies that $\sum|p'_i|$ is so small as well.

By (4.9) this means that `Sum2` transforms a vector p_i being ill-conditioned with respect to summation into a new vector with identical sum but condition number improved by about a factor `eps`. This is the motivation to cascade the error-free vector transformation. Our algorithm is as follows.

ALGORITHM 4.8. *Summation as in K-fold precision by* $(K-1)$*-fold error-free vector transformation.*

$$\begin{array}{l} \text{function } \texttt{res} = \texttt{SumK}(p, K) \\ \quad \text{for } k = 1 : K-1 \\ \quad\quad p = \texttt{VecSum}(p) \\ \quad \texttt{res} = \mathrm{fl}\left(\left(\sum_{i=1}^{n-1} p_i \right) + p_n \right) \end{array}$$

Note that for $K = 2$, Algorithm 4.8 (`SumK`) and Algorithm 4.4 (`Sum2`) are identical. To analyze the behavior of Algorithm 4.8 denote the input vector p by $p^{(0)}$, and the vector p after finishing loop k by $p^{(k)}$.

LEMMA 4.9. *With the above notations set* $S^{(k)} := \sum_{i=1}^{n} |p_i^{(k)}|$ *for* $0 \le k \le K-1$. *Then the intermediate results of Algorithm 4.8* (`SumK`) *satisfy the following:*

i) $s := \sum_{i=1}^{n} p_i^{(0)} = \sum_{i=1}^{n} p_i^{(k)}$ *for* $1 \le k \le K-1$,

ii) $|\texttt{res} - s| \le \texttt{eps}|s| + \gamma_{n-1}^2 S^{(K-2)}$,

iii) $S^{(k)} \le 3|s| + \gamma_{2n-2}^k S^{(0)}$ *provided* $4(n-1)\texttt{eps} \le 1$ *and* $1 \le k \le K-1$.

Remark. Before we prove this result, consider the scheme in Figure 4.2 explaining the behavior of Algorithm 4.8 for $n = 5$ and $K = 4$. For simplicity we replace $p_i^{(0)}$ by p_i, and denote $p_i^{(k)}$ by p_i with k-fold prime.

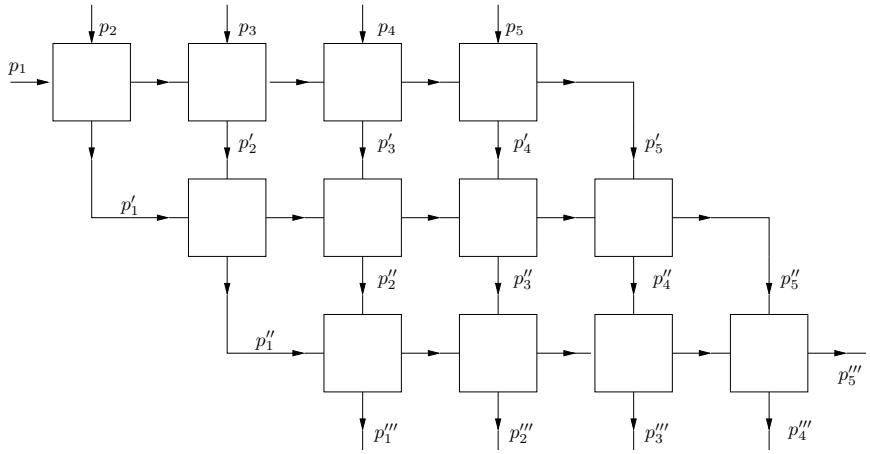

FIG. 4.2. *Outline of Algorithm 4.8 for* $n = 5$ *and* $K = 4$

Proof of Lemma 4.9. Figure 4.2 illustrates the $(K-1)$-fold application of the main loop of Algorithm 4.3 or, equivalently, the first three lines of Algorithm 4.1. Therefore, *i)* follows by successive application of (4.2). Using $s = \sum_{i=1}^{n} p_i^{(K-2)}$ and

applying Proposition 4.5 yields *ii*). To prove *iii*), successive application of (4.14) and using *i*) gives

$$S^{(2)} \le |s| + 2\gamma_{n-1}(|s| + 2\gamma_{n-1}S^{(0)})$$

and

$$S^{(k)} \le |s| \sum_{i=0}^{\infty} (2\gamma_{n-1})^i + (2\gamma_{n-1})^k S^{(0)}$$

for $1 \le k \le K-1$. We have

$$\sum_{i=0}^{\infty} (2\gamma_{n-1})^i = \frac{1}{1 - \dfrac{2(n-1)\mathtt{eps}}{1-(n-1)\mathtt{eps}}} = \frac{1-(n-1)\mathtt{eps}}{1-3(n-1)\mathtt{eps}} \ .$$

If $4(n-1)\mathtt{eps} \le 1$, then $1-(n-1)\mathtt{eps} \le 3(1-3(n-1)\mathtt{eps})$ and

$$\frac{1-(n-1)\mathtt{eps}}{1-3(n-1)\mathtt{eps}} \le 3 \ ,$$

so

$$S^{(k)} \le 3|s| + (2\gamma_{n-1})^k S^{(0)} \ .$$

Using $2\gamma_m \le \gamma_{2m}$ yields *iii*). □

Inserting *iii*) of Lemma 4.9 into *ii*) proves the following error estimate for Algorithm 4.8.

PROPOSITION 4.10. *Let floating point numbers $p_i \in \mathbb{F}$, $1 \le i \le n$, be given and assume $4n\mathtt{eps} \le 1$. Then, also in the presence of underflow, the result "*res*" of Algorithm 4.8* (SumK) *satisfies for $K \ge 3$*

$$|\mathtt{res} - s| \le (\mathtt{eps} + 3\gamma_{n-1}^2)|s| + \gamma_{2n-2}^K S,$$

where $s := \sum p_i$ and $S := \sum |p_i|$. Algorithm 4.8 requires $(6K-5)(n-1)$ flops. If Algorithm 3.6 (TwoSumADD3) *is used instead of Algorithm 3.1* (TwoSum)*, then Algorithm 4.8 requires $(2K-1)(n-1)$ flops.*

We mention that the factors γ_ν in Proposition 4.10 are conservative (see the computational results in Section 6). Moreover, the term $3\gamma_{n-1}^2$ is negligible compared to eps. So the result tells that with each new loop on k the error estimate drops by a factor of size eps, and the final result is of quality as if computed in K-fold precision. The extra term $\mathtt{eps}|s|$ reflects the rounding into working precision. This is the best we can expect.

COROLLARY 4.11. *Assume $4n\mathtt{eps} < 1$. The result "*res*" of Algorithm 4.8* (SumK)*, also in the presence of underflow, satisfies*

$$\frac{|\mathtt{res} - s|}{|s|} \le \mathtt{eps} + 3\gamma_{n-1}^2 + \gamma_{2n-2}^K \cdot \mathrm{cond}\left(\sum p_i\right).$$

*In other words, the bound for the relative error of the result "*res*" is essentially the relative rounding error unit* eps *plus $(\varphi\mathtt{eps})^K$ times the condition number for a moderate factor φ.*

Note that Corollary 4.11 illustrates the unusual phenomenon that for very ill-conditioned input the computed result is more accurate than cond $\times$ `eps`. This is solely due to the fact that all intermediate transformations `TwoSum` in `SumK` are error-free – although individual operations within `TwoSum` are afflicted with rounding errors. The usual heuristic applies only to the last line of Algorithm 4.8 (`SumK`).

We mention that Algorithm 4.8 can be used to achieve a certain *accuracy* using the rigorous error estimate in Corollary 4.7 to determine a suitable value of K.

Algorithm 4.8 (`SumK`) proceeds through the scheme in Figure 4.2 *horizontally*. This bears the advantage that one may continue if the desired accuracy is not yet achieved. However, a local vector of length n is necessary if the input vector shall not be overwritten. To overcome this problem, we may as well proceed *vertically* through the scheme in Figure 4.2. In this case the number of lines, i.e. $K-1$, has to be specified in advance. Note that K is usually very small, so the number of intermediate results is negligible, only some K values. The final result is exactly the same because all transformations are error-free!

ALGORITHM 4.12. *Equivalent formulation of Algorithm 4.8 – vertical mode.*

```
function res = SumKvert(p, K)
    K = min(K, n)
    for i = 1 : K − 1
        s = p_i
        for k = 1 : i − 1
            [q_k, s] = TwoSum(q_k, s)
        end
        q_i = s
    end
    for i = K : n
        α = p_i
        for k = 1 : K − 1
            [q_k, α] = TwoSum(q_k, α)
        end
        s = s + α
    end
    for j = 1 : K − 2
        α = q_j
        for k = j + 1 : K − 1
            [q_k, α] = TwoSum(q_k, α)
        end
        s = s + α
    end
    res = s + q_{K−1}
```

The algorithm divides into three parts, the initialization (leading upper triangle), the main part of the loop and the final lower triangle in Figure 4.2.

5. Dot product. With Algorithm 3.3 (or Algorithm 3.5) we already have an error-free transformation of the product of two floating point numbers into the sum of two floating point numbers. Combining this with our summation algorithms yields a first algorithm for the computation of the dot product of two n-vectors x, y.

ALGORITHM 5.1. *A first dot product algorithm.*

for $i = 1 : n$
    $[r_i, r_{n+i}] = \texttt{TwoProduct}(x_i, y_i)$
$\texttt{res} = \texttt{SumK}(r, K)$

For the moment we assume that no underflow occurs. Then, $\sum_{i=1}^{2n} r_i = x^T y$. With these preliminaries and noting that the vector r has $2n$ elements we can easily analyze Algorithm 5.1.

THEOREM 5.2. *Let $x_i, y_i \in \mathbb{F}$, $1 \le i \le n$, be given and denote by $\texttt{res} \in \mathbb{F}$ the result of Algorithm 5.1. Assume $4n\texttt{eps} < 1$. Then for $K \ge 2$ and without the presence of underflow,*

$$|\texttt{res} - x^T y| \le (\texttt{eps} + 3\gamma_{2n-1}^2)|x^T y| + \gamma_{4n}^K |x^T|\,|y|.$$

Algorithm 5.1 requires $6K(2n-1)+7n+5$ or less than $(12K+7)n$ flops. For $K = 2$ these are $31n - 7$ flops. If `TwoSumADD3` *and* `TwoProductFMA` *is used instead of* `TwoSum` *and* `TwoProduct`*, respectively, then Algorithm 5.1 requires $2K(2n-1)+1$ flops, which are $8n - 3$ flops for $K = 2$.*

Proof. In view of Proposition 4.10, (3.3) and $\sum_{i=1}^{2n} r_i = x^T y$ we only need to estimate $S := \sum_{i=1}^{2n} |r_i|$. By (3.3),

$$\sum_{i=1}^{2n} |r_i| \le \sum_{i=1}^{n} |\mathrm{fl}(x_i y_i)| + \texttt{eps} \sum_{i=1}^{n} |x_i y_i| \le (1 + 2\texttt{eps})|x^T|\,|y|,$$

so that Proposition 4.10 yields

$$\begin{aligned} |\texttt{res} - x^T y| &= \Big|\texttt{res} - \sum_{i=1}^{2n} r_i\Big| \\ &\le (\texttt{eps} + 3\gamma_{2n-1}^2)|x^T y| + \gamma_{4n-2}^K (1 + 2\texttt{eps})|x^T|\,|y| \\ &\le (\texttt{eps} + 3\gamma_{2n-1}^2)|x^T y| + \gamma_{4n}^K |x^T|\,|y|. \quad \square \end{aligned}$$

In the following we will improve Algorithm 5.1 first for $K = 2$, and then for general $K \ge 3$.

The most frequently used application of an improved dot product is the computation in doubled working precision, corresponding to $K = 2$ in Algorithm 5.1. In this case the remainders r_{n+i} are small compared to r_i, and we can improve the performance by adding those remainders in working precision instead of using `TwoSum`. This reduces the number of flops from $31n$ to $25n$. Furthermore we will obtain an improved error estimate in Proposition 5.5 and especially Corollary 5.7, most beautiful and the best we can expect. Our algorithm for $K = 2$ corresponding to a result as if computed in twice the working precision is as follows.

ALGORITHM 5.3. *Dot product in twice the working precision.*

function $\texttt{res} = \texttt{Dot2}(x, y)$
    $[p, s] = \texttt{TwoProduct}(x_1, y_1)$
    for $i = 2 : n$
        $[h, r] = \texttt{TwoProduct}(x_i, y_i)$
        $[p, q] = \texttt{TwoSum}(p, h)$
        $s = \mathrm{fl}(s + (q + r))$
    $\texttt{res} = \mathrm{fl}(p + s)$

An inspection yields

$$p = \mathrm{fl}(x^T y),$$

so that the final result "res" of Algorithm 5.3 is the ordinary floating point result p of the dot product plus the summed up error terms s. If there is not much cancellation in $x^T y$, then the approximation p is dominant, whereas the error term s takes over for an ill-conditioned dot product. For the analysis we rewrite Algorithm 5.3 into the following equivalent one.

ALGORITHM 5.4. *Equivalent formulation of Algorithm 5.3.*

```
function res = Dot2s(x, y)
    [p_1, s_1] = TwoProduct(x_1, y_1)
    for i = 2 : n
        [h_i, r_i] = TwoProduct(x_i, y_i)
        [p_i, q_i] = TwoSum(p_{i-1}, h_i)
        s_i = fl(s_{i-1} + (q_i + r_i))
    res = fl(p_n + s_n)
```

For the analysis of Algorithm 5.4 (and therefore of Algorithm 5.3), for the moment still assuming no underflow occurred, we collect some facts about the intermediate results. By (3.3) we have for $i \geq 2$,

$$q_i + r_i = (p_{i-1} + h_i - p_i) + (x_i y_i - h_i) = x_i y_i + p_{i-1} - p_i,$$

and therefore

$$s_1 + \sum_{i=2}^{n} (q_i + r_i) = (x_1 y_1 - p_1) + \left(\sum_{i=2}^{n} x_i y_i + p_1 - p_n \right) = x^T y - p_n. \tag{5.1}$$

Applying (3.3), Lemma 4.2 and again (3.3) yields

$$\begin{array}{rcl} |s_1| & \leq & \mathbf{eps}|x_1 y_1|, \\ \sum_{i=2}^{n} |q_i| & \leq & \gamma_{n-1}(|p_1| + \sum_{i=2}^{n} |h_i|) = \gamma_{n-1} \sum_{i=1}^{n} |\mathrm{fl}(x_i y_i)| \leq (1 + \mathbf{eps})\gamma_{n-1}|x^T|\,|y|, \\ \sum_{i=2}^{n} |r_i| & \leq & \mathbf{eps} \sum_{i=2}^{n} |x_i y_i|, \end{array}$$

and with

$$\mathbf{eps} + (1 + \mathbf{eps})\gamma_{n-1} = \mathbf{eps} + \frac{(1 + \mathbf{eps})(n-1)\mathbf{eps}}{1 - (n-1)\mathbf{eps}} = \frac{n\mathbf{eps}}{1 - (n-1)\mathbf{eps}}$$

we obtain

$$\begin{array}{rcl} |s_1| + \sum_{i=2}^{n} |q_i| + \sum_{i=2}^{n} |r_i| & \leq & \mathbf{eps}|x^T|\,|y| + (1 + \mathbf{eps})\gamma_{n-1}|x^T|\,|y| \\ & = & \dfrac{n\mathbf{eps}}{1 - (n-1)\mathbf{eps}}|x^T|\,|y|. \end{array} \tag{5.2}$$

For later use we apply (5.1) to obtain

$$|p_n| = \left| x^T y - s_1 - \sum_{i=2}^{n} (q_i + r_i) \right| \leq |x^T y| + |s_1| + \sum_{i=2}^{n} |q_i| + \sum_{i=2}^{n} |r_i|$$

and using (5.2),

$$\begin{aligned} |s_1| + \sum_{i=2}^{n} |r_i| + \sum_{i=2}^{n} |q_i| + |p_n| &\leq |x^T y| + 2\left(|s_1| + \sum_{i=2}^{n} |q_i| + \sum_{i=2}^{n} |r_i| \right) \\ &\leq |x^T y| + \gamma_{2n} |x^T|\,|y|. \end{aligned} \tag{5.3}$$

It follows by (5.1), Algorithm 5.4 and (5.2),

$$\begin{aligned} |(x^T y - p_n) - s_n| &= \left| s_1 + \sum_{i=2}^{n} (q_i + r_i) - \mathrm{fl}\left(s_1 + \sum_{i=2}^{n} (q_i + r_i) \right) \right| \\ &\leq \gamma_{n-1} \left(|s_1| + \sum_{i=2}^{n} |\mathrm{fl}(q_i + r_i)| \right) \\ &\leq \gamma_n \left(|s_1| + \sum_{i=2}^{n} |q_i + r_i| \right) \\ &\leq \gamma_n \frac{n\mathtt{eps}}{1 - (n-1)\mathtt{eps}} |x^T|\,|y|, \end{aligned} \tag{5.4}$$

and finally

$$\frac{1 + \mathtt{eps}}{1 - (n-1)\mathtt{eps}} \leq \frac{1}{1 - n\mathtt{eps}} \quad \Rightarrow \quad (1 + \mathtt{eps}) \frac{n\mathtt{eps}}{1 - (n-1)\mathtt{eps}} \leq \gamma_n$$

yields for some $|\varepsilon| \leq \mathtt{eps}$,

$$\begin{aligned} |\mathtt{res} - x^T y| &= |(1 + \varepsilon)(p_n + s_n) - x^T y| \\ &= |\varepsilon x^T y + (1 + \varepsilon)(p_n + s_n - x^T y)| \\ &\leq \mathtt{eps} |x^T y| + (1 + \mathtt{eps}) \gamma_n \frac{n\mathtt{eps}}{1 - (n-1)\mathtt{eps}} |x^T|\,|y| \\ &\leq \mathtt{eps} |x^T y| + \gamma_n^2 |x^T|\,|y|. \end{aligned} \tag{5.5}$$

Our analysis is easily adapted to the presence of underflow. The main point is that, due to Theorem 3.4 and (5.1), the transformation of $x^T y$ into $s_1 + \sum_{i=2}^{n} (q_i + r_i) + p_n$ is error-free if no underflow occurs. But our analysis is only based on the latter sum, so that in the presence of underflow Theorem 3.4 tells that the difference between $x^T y$ and $s_1 + \sum_{i=2}^{n} (q_i + r_i) + p_n$ is at most $5n\mathtt{eta}$. We proved the following result.

PROPOSITION 5.5. *Let $x_i, y_i \in \mathbb{F}$, $1 \leq i \leq n$, be given and denote by $\mathtt{res} \in \mathbb{F}$ the result computed by Algorithm 5.3* (Dot2)*. Assume $n\mathtt{eps} < 1$. Then, if no underflow occurs,*

$$|\mathtt{res} - x^T y| \leq \mathtt{eps} |x^T y| + \gamma_n^2 |x^T|\,|y|,$$

and, in the presence of underflow,

$$|\mathtt{res} - x^T y| \leq \mathtt{eps} |x^T y| + \gamma_n^2 |x^T|\,|y| + 5n\mathtt{eta}. \tag{5.6}$$

Algorithm 5.3 requires $25n - 7$ flops. If TwoSumADD3 *and* TwoProductFMA *is used instead of* TwoSum *and* TwoProduct*, respectively, then Algorithm 5.3 requires $6n - 3$ flops.*

We mention that if TwoSumADD3 is available, then the second last line of Algorithm 5.3 can be replaced by $s = \mathtt{ADD3}(s, q, r)$, thus improving the error estimate slightly.

The first part of our algorithm is similar to `BLAS_ddot_x` in XBLAS [26], which in turn is based on Bailey's double-double precision arithmetic [5, 4]. As for summation, the low order parts are treated differently from `Dot2` :

ALGORITHM 5.6. *XBLAS quadruple precision dot product.*

function $s = \texttt{DotXBLAS}(x, y)$
  $s = 0;\ t = 0$
  for $i = 1 : n$
    $[h, r] = \texttt{TwoProduct}(x_i, y_i)$
    $[s_1, s_2] = \texttt{TwoSum}(s, h)$
    $[t_1, t_2] = \texttt{TwoSum}(t, r)$
    $s_2 = s_2 + t_1$
    $[t_1, s_2] = \texttt{FastTwoSum}(s_1, s_2)$
    $t_2 = t_2 + s_2$
    $[s, t] = \texttt{FastTwoSum}(t_1, t_2)$

Algorithm `DotXBLAS` includes the pair $[s, t]$ with $s+t$ approximating $x^T y$ in quadruple precision. Omitting the last statement in Algorithm 5.3 (`Dot2`) it follows by (5.4) that the final pair $[p, s]$ in `Dot2` is of quadruple precision as well. The output of `DotXBLAS` is the higher order part s. The error analysis in [26, (10)] relates $|s - x^T y|$ to some internal precision and to the computed approximation s rather than to $x^T y$ as in (5.6). An error estimation for `DotXBLAS` similar to (5.6) might contain something of the order $n\texttt{eps}^2$ instead of γ_n^2 due to the different treatment of the lower order terms. If this is true, the error estimate would improve over (5.6) for condition numbers exceeding $\texttt{eps}^{-1}$. We did not attempt to prove such an error estimate, but numerical results seem to indicate this behavior. However, XBLAS dot product requires $37n$ flops as compared to $25n$ flops for (`Dot2`). This is confirmed by the measured computing times (see Section 6).

As we will see in a moment, our Algorithm 5.10 (`DotK`) for K=3 requires the same computing time $37n$ flops as the XBLAS dot product but delivers a result as if computed in *tripled* working precision and rounded back into working precision. For IEEE 754 double precision this means `DotK` calculates accurate results for condition numbers up to some 10^{48} rather than 10^{32}.

Again it is very instructive to express and interpret our result in terms of the condition number of the dot product. One defines for $x^T y \neq 0$,

$$\operatorname{cond}(x^T y) := \lim_{\varepsilon \to 0} \sup \left\{ \left| \frac{(x + \Delta x)^T (y + \Delta y) - x^T y}{\varepsilon x^T y} \right| : |\Delta x| \leq \varepsilon |x|, |\Delta y| \leq \varepsilon |y| \right\},$$

where absolute value and comparison is to be understood componentwise. A standard computation yields

$$\operatorname{cond}(x^T y) = 2 \frac{|x^T|\,|y|}{|x^T y|}\,. \tag{5.7}$$

Combining this with the estimation in Proposition 5.5 gives the following result.

COROLLARY 5.7. *Let $x_i, y_i \in \mathbb{F}$, $1 \leq i \leq n$, be given and denote by $\texttt{res} \in \mathbb{F}$ the result computed by Algorithm 5.3* (`Dot2`). *Assume $n\texttt{eps} < 1$. Then, if no underflow occurs,*

$$\left| \frac{\texttt{res} - x^T y}{x^T y} \right| \leq \texttt{eps} + \frac{1}{2} \gamma_n^2 \operatorname{cond}(x^T y)\,. \tag{5.8}$$

We think this is a most beautiful result. It tells that the *relative error* of the result of Algorithm 5.3 is not more than a moderate factor times $\texttt{eps}^2$ times the condition number plus the relative rounding error unit eps. This is the best we can expect for a result computed in twice the working precision and rounded back into working precision. The factor γ_n^2 is mainly due to (2.4) which is known to be pessimistic. This will be confirmed in our computational results in Section 6.

The following algorithm computes an error bound for the dot product in pure floating point. We will prove the error bound to be rigorous, also in the presence of underflow.

ALGORITHM 5.8. *Dot product in twice the working precision with error bound including underflow.*

function $[\texttt{res}, \texttt{err}] = \texttt{Dot2Err}(x, y)$
  if $2n\texttt{eps} \geq 1$, error('inclusion failed'), end
  $[p, s] = \texttt{TwoProduct}(x_1, y_1)$
  $e = |s|$
  for $i = 2 : n$
    $[h, r] = \texttt{TwoProduct}(x_i, y_i)$
    $[p, q] = \texttt{TwoSum}(p, h)$
    $t = \mathrm{fl}(q + r)$
    $s = \mathrm{fl}(s + t)$
    $e = \mathrm{fl}(e + |t|)$
  $\texttt{res} = \mathrm{fl}(p + s)$
  $\delta = \mathrm{fl}((n\texttt{eps})/(1 - 2n\texttt{eps}))$
  $\alpha = \mathrm{fl}(\texttt{eps}|\texttt{res}| + (\delta e + 3\texttt{eta}/\texttt{eps}))$
  $\texttt{err} = \mathrm{fl}(\alpha/(1 - 2\texttt{eps}))$

For the proof of validity we use the computed quantities $e, \texttt{res}, \delta, \alpha$ and err of Algorithm 5.8 after execution of the last statement. First note that the quantity in (5.4) is estimated by

$$\gamma_{n-1}\left(|s_1| + \sum_{i=2}^{n} |\mathrm{fl}(q_i + r_i)|\right) \leq (1 + \texttt{eps})^{n-1}\gamma_{n-1}\mathrm{fl}\left(|s_1| + \sum_{i=2}^{n} |q_i + r_i|\right) \leq \frac{(n-1)\texttt{eps}}{1 - (2n-2)\texttt{eps}}\, e\ .$$

Taking underflow into account and using $\texttt{res} = \mathrm{fl}(p + s)$ it follows

$$\begin{aligned} |x^T y - \texttt{res}| &\leq \texttt{eps}|\texttt{res}| + |x^T y - p - s| + 5n\texttt{eta} \\ &\leq \mathrm{fl}(\texttt{eps}|\texttt{res}|) + \frac{(n-1)\texttt{eps}}{1 - (2n-2)\texttt{eps}}\, e + (5n+1)\texttt{eta} \\ &=: \mathrm{fl}(\texttt{eps}|\texttt{res}|) + \Delta. \end{aligned} \tag{5.9}$$

Furthermore,

$$\frac{(n-1)\texttt{eps}}{1 - (2n-2)\texttt{eps}} \leq (1 + \texttt{eps})^{-3}\frac{n\texttt{eps}}{1 - 2n\texttt{eps}} \leq (1 + \texttt{eps})^{-2}\delta$$

and, using $2n\texttt{eps} < 1$ and regarding possible underflow,

$$\begin{aligned}\Delta &\le (1+\texttt{eps})^{-2}\delta e + (5n+1)\texttt{eta} \\ &\le (1+\texttt{eps})^{-1}\mathrm{fl}(\delta e) + (5n+2)\texttt{eta} \\ &\le (1+\texttt{eps})^{-1}\left(\mathrm{fl}(\delta e) + 3\texttt{eta}/\texttt{eps}\right) \\ &\le \mathrm{fl}(\delta e + 3\texttt{eta}/\texttt{eps}).\end{aligned}$$

Finally, (5.9) yields

$$\begin{aligned}|x^T y - \texttt{res}| &\le \mathrm{fl}(\texttt{eps}|\texttt{res}|) + \mathrm{fl}(\delta e + 3\texttt{eta}/\texttt{eps}) \le (1-\texttt{eps})^{-1}\alpha \\ &\le (1-\texttt{eps})\alpha/(1-2\texttt{eps}) \le \mathrm{fl}(\alpha/(1-2\texttt{eps})) = \texttt{err},\end{aligned}$$

and proves the following result.

COROLLARY 5.9. *Let $x_i, y_i \in \mathbb{F}$, $1 \le i \le n$, be given and denote by $\texttt{res}, \texttt{err} \in \mathbb{F}$ the results computed by Algorithm 5.8* (`Dot2Err`)*. If Algorithm 5.8 runs to completion, then, also in the presence of underflow,*

$$\texttt{res} - \texttt{err} \le x^T y \le \texttt{res} + \texttt{err}.$$

Algorithm 5.8 requires $27n+4$ flops, where taking the absolute value is counted as one flop. If instructions `ADD3` *and* `FMA` *are available, then Algorithm 5.8 can be executed in $8n+8$ flops.*

The trick is to move all underflow related constants into `eta/eps`, the smallest positive normalized floating point number, which vanishes when added to δe except in pathological situations.

Our final step is to extend the idea of the previous algorithm for doubled working precision to higher precision by applying our summation Algorithm 4.8 (`SumK`).

ALGORITHM 5.10. *Dot product algorithm in K-fold working precision, $K \ge 3$.*

```
function res = DotK(x, y, K)
    [p, r_1] = TwoProduct(x_1, y_1)
    for i = 2 : n
        [h, r_i] = TwoProduct(x_i, y_i)
        [p, r_{n+i-1}] = TwoSum(p, h)
    r_{2n} = p
    res = SumK(r, K - 1)
```

The call of Algorithm 4.8 (`SumK`) in the last line of Algorithm 5.10 may be replaced by the equivalent Algorithm 4.12 (`SumKvert`, vertical mode). With our previous results the analysis is not difficult. The error-free transformations `TwoProduct` (without underflow) and `TwoSum` yield

$$s := \sum_{i=1}^{2n} r_i = x^T y.$$

To apply Proposition 4.10 we need to estimate $S := \sum_{i=1}^{2n} |r_i|$. But this is nothing else than the quantity on the left hand side of (5.3). Inserting this into Proposition 4.10, using $2\gamma_m \le \gamma_{2m}$ and noting that the vector r is of length $2n$ yields

$$\begin{aligned}|\texttt{res} - x^T y| &\le (\texttt{eps} + 3\gamma_{2n-1}^2)|x^T y| + \gamma_{4n-2}^{K-1}(|x^T y| + \gamma_{2n}|x^T|\,|y|) \\ &= (\texttt{eps} + 3\gamma_{2n-1}^2 + \gamma_{4n-2}^{K-1})|x^T y| + \gamma_{2n}\gamma_{4n-2}^{K-1}|x^T|\,|y| \\ &\le (\texttt{eps} + \frac{3}{4}\gamma_{4n-2}^2 + \gamma_{4n-2}^{K-1})|x^T y| + \gamma_{4n-2}^{K}|x^T|\,|y|.\end{aligned}$$

If $4(2n-1)\mathtt{eps} \le 1$, then $\gamma_{4n-2} \le 1$ and

$$|\mathtt{res} - x^T y| \le (\mathtt{eps} + 2\gamma_{4n-2}^2)|x^T y| + \gamma_{4n-2}^K |x^T|\,|y|. \tag{5.10}$$

The analysis in the presence of underflow is again based on the fact that the only additional error can occur in the initial transformation of the dot product $x^T y$ by the n calls of `TwoProduct`. So (5.10) and Theorem 3.4 prove the following result.

PROPOSITION 5.11. *Let $x_i, y_i \in \mathbb{F}$, $1 \le i \le n$, be given and assume $8n\mathtt{eps} \le 1$. Denote by $\mathtt{res} \in \mathbb{F}$ the result of Algorithm 5.10* (`DotK`)*. Then, in case no underflow occurs,*

$$|\mathtt{res} - x^T y| \le (\mathtt{eps} + 2\gamma_{4n-2}^2)|x^T y| + \gamma_{4n-2}^K |x^T|\,|y|$$

and

$$\left|\frac{\mathtt{res} - x^T y}{x^T y}\right| \le \mathtt{eps} + 2\gamma_{4n-2}^2 + \frac{1}{2}\gamma_{4n-2}^K \mathrm{cond}\,(x^T y). \tag{5.11}$$

In the presence of underflow,

$$|\mathtt{res} - x^T y| \le (\mathtt{eps} + 2\gamma_{4n-2}^2)|x^T y| + \gamma_{4n-2}^K |x^T|\,|y| + 5n\mathtt{eta}.$$

Algorithm 5.10 requires $6K(2n-1) + n + 5$ or less than $(12K+1)n$ flops. If Algorithms 3.6 (`TwoSumADD3`) *and 3.5* (`TwoProductFMA`) *are used instead of Algorithms 3.1* (`TwoSum`) *and 3.3* (`TwoProduct`)*, respectively, then Algorithm 5.10 requires $(2K+1)(2n-1)$ flops.*

The term $2\gamma_{4n-2}^2$ is negligible against `eps`. So, as before, this means that the relative error of the result is essentially the relative rounding error unit `eps` plus a moderate factor times `eps` to the K-th power times the condition number. This is again the best we can expect of a computation in K-fold working precision.

Note that Algorithm 5.10 improves the computing time of Algorithm 5.1 from about $(12K+7)n$ to $(12K+1)n$ flops, whilst satisfying the same error estimate.

6. Numerical results. In this section we present timing and accuracy results. We compare our summation algorithms to ordinary summation, Kahan-Babuška, XBLAS and Anderson's algorithm, and our dot product algorithms to BLAS and XBLAS. All timing was done in Fortran by the first author; some accuracy measurements were done in Matlab. All calculations in this section are performed in IEEE 754 double precision as working precision corresponding to about 16 decimal digits.

We will not say much about the many practical applications of our algorithms because *i*) this is widely known, and *ii*) there are excellent treatments in the recent literature [15, 26]. In the latter, for example, the second chapter consists of sections on "Iterative Refinement of Linear Systems and Least Squares Problems", on "Avoiding Pivoting in Sparse Gaussian Elimination", on "Accelerating Iterative Methods for $Ax = b$, like GMRES", on "Using Normal Equations Instead of QR for Least Squares", on "Solving Ill-Conditioned Triangular Systems", on "Eigenvalues and Eigenvectors of Symmetric Matrices" with detailed discussion of the numerical properties and effects and advantages of the use of more accurate dot products (in this case doubled working precision). Everything said over there applies to *any* algorithm calculating the value of a dot product in doubled working precision, regardless of the method in use, so it applies to our Algorithm 5.3 (`Dot2`) and, *mutatis mutandis*, to the higher precision Algorithm 5.10.

To test our algorithms we need *extremely* ill-conditioned dot products, and a problem is to generate vectors x, y with *floating point* entries causing hundreds of bits cancellation. Ill-conditioned sums of length $2n$ are generated from dot products of length n using Algorithm 3.3 (`TwoProduct`) and randomly permuting the summands.

For ill-conditioned dot products we designed the following Algorithm 6.1 (`GenDot`) to generate vectors x, y with anticipated condition number c. We put some effort into this routine to make sure that the vectors are general, not following obvious patterns.

ALGORITHM 6.1. *Generation of extremely ill-conditioned dot products.*

```
function [x,y,d,C] = GenDot(n,c)
%
%input   n    dimension of vectors x,y, n>=6
%        c    anticipated condition number of x'*y
%output  x,y  generated vectors
%        d    dot product x'*y rounded to nearest
%        C    actual condition number of x'*y
%
%uses    r=DotExact(x,y)  calculating a floating point number r nearest to x'*y
%

  n2 = round(n/2);                      % initialization
  x = zeros(n,1);
  y = x;

  b = log2(c);
  e = round(rand(n2,1)*b/2);            % e vector of exponents between 0 and b/2
  e(1) = round(b/2)+1;                  % make sure exponents b/2 and
  e(end) = 0;                           %   0 actually occur in e
  x(1:n2) = (2*rand(n2,1)-1).*(2.^e);   % generate first half of vectors x,y
  y(1:n2) = (2*rand(n2,1)-1).*(2.^e);

  % for i=n2+1:n and v=1:i, generate x_i, y_i such that  (*) x(v)'*y(v) ~ 2^e(i-n2)
  e = round(linspace(b/2,0,n-n2));      % generate exponents for second half
  for i=n2+1:n
    x(i) = (2*rand-1)*2^e(i-n2);        % x_i random with generated exponent
    y(i) = ((2*rand-1)*2^e(i-n2)-DotExact(x',y))/x(i);   % y_i according to (*)
  end

  index = randperm(n);                  % generate random permutation for x,y
  x = x(index);                         % permute x and y
  y = y(index);
  d = DotExact(x',y);                   % the true dot product rounded to nearest
  C = 2*(abs(x')*abs(y))/abs(d);        % the actual condition number
```

Algorithm 6.1 is executable Matlab [43] code. The only assumption is availability of a routine `DotExact` such that for floating point vectors $x, y \in \mathbb{F}^n$ the call `r=DotExact(x,y)` produces a floating point number `r` $\in \mathbb{F}$ near to the exact value of the dot product $x^T y \in \mathbb{R}$. Since the exponent range of double precision floating point numbers is limited, such a routine is easily written using some high accuracy floating point arithmetic (cf., for example, [12], Section 2.1.1). Algorithm 5.10 (`DotK`) for suitably chosen K can be used as well.

Algorithm 6.1 (`GenDot`) works as follows. The condition number (5.7) of the dot product $x^T y$ is proportional to the degree of cancellation. In order to achieve a prescribed cancellation, we generate the first half of the vectors x and y randomly within a large exponent range. This range is chosen according to the anticipated condition number. The second half of x and y is then constructed choosing x_i randomly with decreasing exponent, and calculating y_i such that some cancellation occurs. Finally, we permute the vectors x, y randomly and calculate the achieved condition number.

We first show the performance of our "random" generator for ill-conditioned dot

products. For the following Table 6.1 we generated 50000 test vectors of lengths varying between 100 and 500 and with condition number varying between 1 and 10^{100}. We display the minimum, maximum, average (arithmetic mean) and median of the achieved divided by the anticipated condition number. Together with the median we give the median absolute deviation in parenthesis. The median absolute deviation of a vector x is the median of the vector $|x - \mathrm{median}(x)|$. In [30] this measure is called robust and the 'best of an inferior lot'.

TABLE 6.1
Ratio of achieved and anticipated condition number by `GenDot`

minimum	maximum	average	median
$1.5 \cdot 10^{-3}$	$1.9 \cdot 10^{6}$	99.4	10.0 (7.1)

So in general the actual condition number is a little larger than the anticipated one with quite some variation. However, this is unimportant because in the following figures we plot the relative error against the *actually achieved* condition number.

Next we display timing and accuracy of the summation algorithms `DSum` (ordinary recursive summation), `Kahan-Babuška` algorithm [32], `Sum2` (Algorithm 4.1), `SumXBLAS` (XBLAS Algorithm 4.6, `BLAS_dsum_x`), `Sum3` (Algorithm 4.8, `SumK` for K=3) and `Anderson` algorithm [1]. First we display the accuracy in Table 6.2 for sums of lengths n from 100 to 5000 in steps of 100 with condition numbers ranging from 10^7 to 10^{28}. For each dimension and condition number we ran some 1000 test cases. We compute the number of correct decimal digits, that is $-\log_2(|x - \tilde{x}|/|x|)$, where x depicts the true result and $\tilde{x}$ its approximation. A negative number of digits is set to 0.0. The displayed numbers per row are the median (with median absolute deviation in parenthesis) over all test cases corresponding to that condition number.

TABLE 6.2
Number of correct digits of summation routines

cond	`DSum`	`K-B`	`Sum2`	`SumXBLAS`	`Sum3`	`Anderson`
10^7	8.9 (0.3)	9.8 (0.2)	16.0 (0.0)	16.0 (0.0)	16.0 (0.0)	16.0 (0.0)
10^{14}	2.0 (0.3)	2.9 (0.2)	16.0 (0.0)	16.0 (0.0)	16.0 (0.0)	15.9 (0.1)
10^{21}	0.0 (0.0)	0.0 (0.0)	10.3 (0.5)	11.1 (0.3)	16.0 (0.0)	15.5 (0.5)
10^{28}	0.0 (0.0)	0.0 (0.0)	3.8 (0.4)	4.3 (0.4)	16.0 (0.0)	15.9 (0.1)

The numbers confirm the expected behavior. For `DSum` the number of correct digits is basically $16 - \log_{10} \mathrm{cond}$, and `Kahan-Babuška` improves a little on that. `Sum2` and `SumXBLAS` are maximally accurate until about condition number 10^{16}. For sums with extreme condition number well over 10^{16} the results of `SumXBLAS` are up to one digit better than `Sum2`. Note that such extreme condition numbers can occur in numerical computations only for exactly given data. Such extreme cases can be handled by `Sum3` with maximally accurate result for condition numbers up to about 10^{32}. Anderson's algorithm is designed to proceed until very high accuracy is achieved, and this is confirmed by the data. On a processor with built-in extended precision compiler optimization would improve the results of `DSum` . We compared results achieved solely in working precision.

We tested timing of the summation algorithms in the two environments listed in Table 6.3. All algorithms were tested in Fortran. For a fair comparison we rewrote the XBLAS routine `BLAS_dsum_x` [26] into a plain summation routine in Fortran

TABLE 6.3
Testing environments

env. I)	Pentium 3 800 MHz, Fortran: GNU Compiler Collection (g77-3.3.3)
	C: GNU Compiler Collection (gcc-3.3.3), BLAS: ATLAS 3.6.0
env. II)	Pentium 4 2.53 GHz, Compaq Visual Fortran 6.6C
	C: Microsoft Visual C++ 6.0, BLAS: Intel Math Kernel Library 7.0

omitting extra functionalities like increment etc. The computing time in environment I) improved slightly by that, and for environment II) rewriting in Fortran improved the time by more than 40 % over the C-code. For the same values 100 to 5000 in steps of 100 for n and the same condition numbers we performed 1000 tests each.

TABLE 6.4
Measured computing time for environments in Table 6.3, recursive summation `DSum` *normed to 1*

	cond	`K-B`	`Sum2`	`SumXBLAS`	`Sum3`	`Anderson`
env. I)	10^7	4.3 (0.2)	7.1 (0.2)	15.5 (0.1)	16.7 (0.7)	52.5 (3.5)
	10^{14}	4.3 (0.2)	7.0 (0.3)	15.4 (0.2)	16.6 (0.7)	53.8 (2.6)
	10^{21}	4.4 (0.1)	7.1 (0.1)	15.5 (0.1)	16.9 (0.4)	74.7 (3.7)
	10^{28}	4.3 (0.2)	7.0 (0.3)	15.4 (0.3)	16.5 (0.8)	73.0 (2.8)
env. II)	10^7	3.4 (0.1)	5.1 (0.1)	8.3 (0.1)	11.3 (0.2)	51.8 (1.0)
	10^{14}	3.4 (0.0)	5.1 (0.1)	8.3 (0.1)	11.3 (0.1)	51.6 (0.8)
	10^{21}	3.4 (0.0)	5.1 (0.1)	8.4 (0.1)	11.3 (0.1)	73.7 (2.3)
	10^{28}	3.4 (0.0)	5.1 (0.1)	8.3 (0.1)	11.3 (0.1)	73.5 (1.4)
theor.		4	7	10	13	*

The timing in Table 6.4 is relative to `DSum`. We display the median over all test cases (with median absolute deviation in parenthesis). The measured timing of the 'simple algorithms' `Kahan-Babuška` and `Sum2` corresponds in their ratio almost exactly to the flop count, with numbers even better than theory in environment II). The 'more complicated' algorithms `SumXBLAS` and `Sum3` are significantly slower than theoretically predicted in environment I), and a little better in environment II). In all cases the performance in Mflops drops for smaller values of n for all algorithms such that the ratio stay the same.

The comparison to Anderson's algorithm is not entirely fair because it is designed to compute a result of high accuracy, independent of the condition number. This makes the numbers look worse than they are. On the other hand, as Anderson notes in his paper, special care is necessary to avoid infinite loops. We did not do that but filtered those cases. The theoretical computing time depends on the condition number and on the distribution of the data.

Next we test Algorithms 5.3 (`Dot2`) and 5.10 (`DotK`) using ill-conditioned dot products generated by Algorithm 6.1 (`GenDot`). We display the *relative error* of the computed result `res`, i.e. $|\texttt{res} - x^T y|/|x^T y|$, where the difference in the numerator is calculated in one dot product of length $n+1$ by `DotExact`. We set relative errors greater than 2, which means almost no useful information is left, to the value 2. Figure 6.1 shows the results of Algorithm 5.3 (`Dot2`) for 1000 sample dot products of length 100 for condition numbers between 1 and 10^{120}.

Figure 6.1 corresponds to the error estimate (5.8): for condition numbers up to $\texttt{eps}^{-1} \sim 10^{16}$, the result is of maximum accuracy, and for condition numbers between

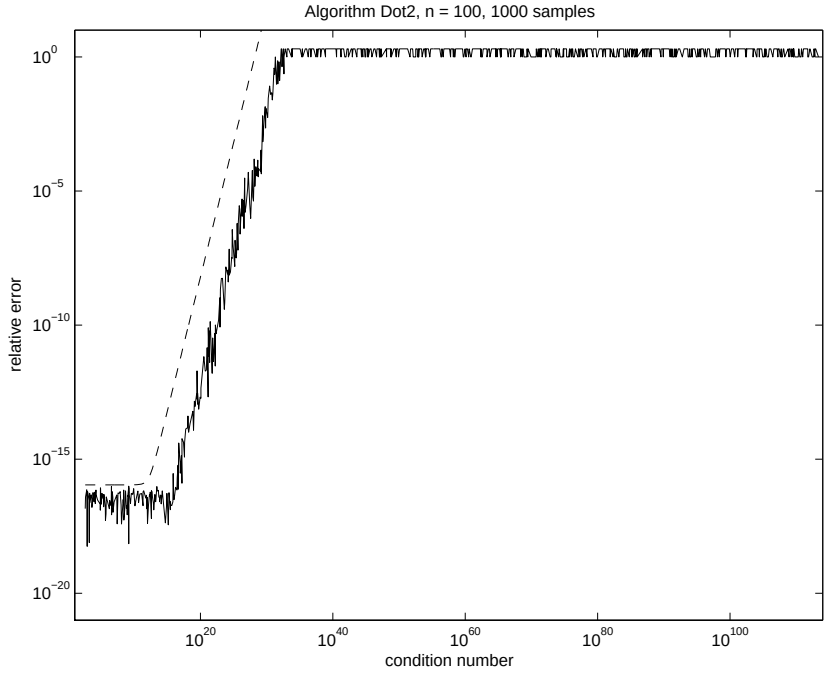

FIG. 6.1. *Test results for Algorithm 5.3* (`Dot2`)*,* $n = 100$*,* 1000 *samples*

$\texttt{eps}^{-1}$ and $\texttt{eps}^{-2} \sim 10^{32}$ the relative error degrades to no precision at all (relative error 1). The dashed line is the error estimate (5.8), showing that it is (of course) valid, but beyond condition numbers eps^{-1} pessimistic by 2 or 3 orders of magnitude.

The next Figures 6.2 and 6.3 display the same test for Algorithm 5.10 (`DotK`) for $K = 3$ and $K = 4$.

The estimation (5.11) for the relative error of the result is again displayed by the dotted line. With increasing value of K, this error estimate becomes more and more pessimistic. For condition numbers beyond $\texttt{eps}^{-4}$ the error estimate (5.11) is already pessimistic by some 10 orders of magnitude. The figures show that the results of Algorithm 5.10 (`DotK`) are of the same quality *as if* calculated in K-fold precision.

Finally, we performed similar tests for 1000 samples of vectors of length 2000 and condition numbers between 1 and 10^{120}. To save space, we display all results, that is that of Algorithm 5.3 (`Dot2`) and Algorithm 5.10 (`DotK`) for $K = 3, 4, 5, 6, 7$ in one figure.

The behavior is quite similar to the previous examples, so that there seems to be not much dependency on the length of the vectors. For larger values of K and n the error estimate (5.11) becomes very conservative. For example, Figure 6.4 shows that for condition numbers up to 10^{93} the results of Algorithm 5.10 (`DotK`) for $K = 7$ are still of maximum accuracy. However, the factor $\frac{1}{2}\gamma_{4n-2}^{K} \sim 2 \cdot 10^{-85}$ in the error estimate (5.11) indicates that maximum accuracy is *assured* only until condition numbers up to about 10^{69}, thus conservative by 24 orders of magnitude.

Next we tested the timing of Algorithm 5.3 (`Dot2`, corresponding to quadruple precision), and XBLAS Algorithm `BLAS_ddot_x` (Algorithm 5.6). As for summation we rewrote `BLAS_ddot_x` in Fortran according to Algorithm 5.6, omitting extra functionalities like increment etc. The computing time improved similarly to summation.

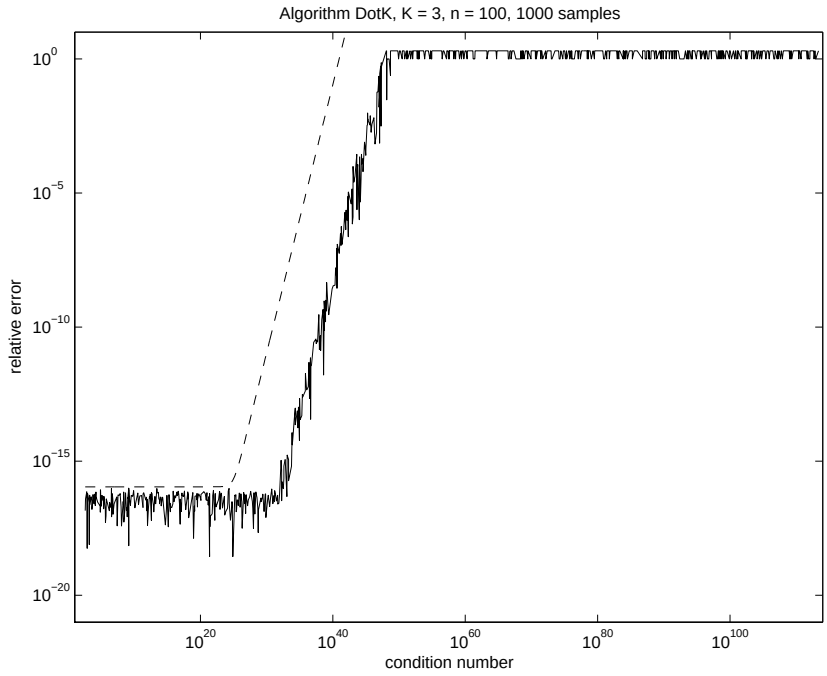

FIG. 6.2. *Test results for Algorithm 5.10* (`DotK`)*,* $K = 3$*,* $n = 100$*,* 1000 *samples*

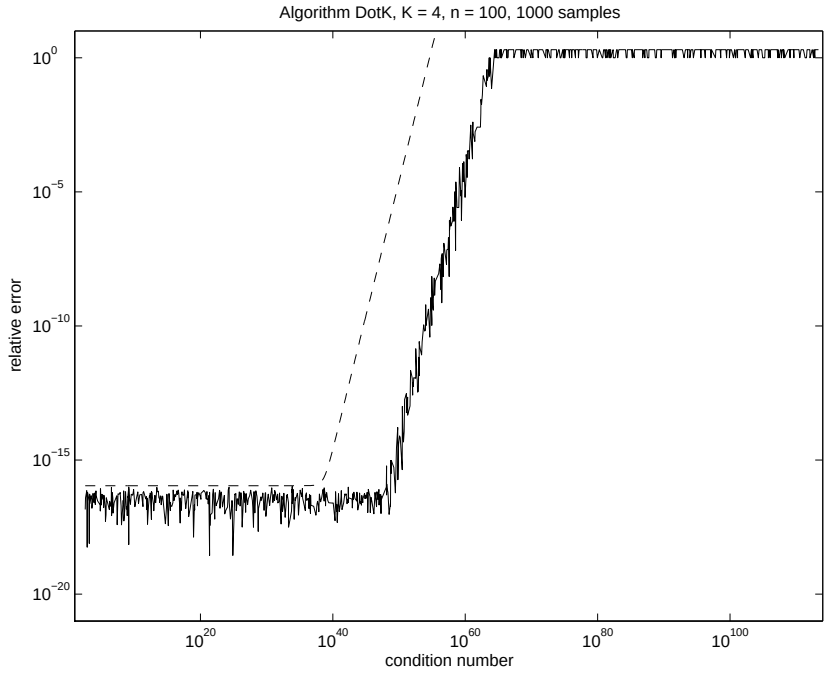

FIG. 6.3. *Test results for Algorithm 5.10* (`DotK`)*,* $K = 4$*,* $n = 100$*,* 1000 *samples*

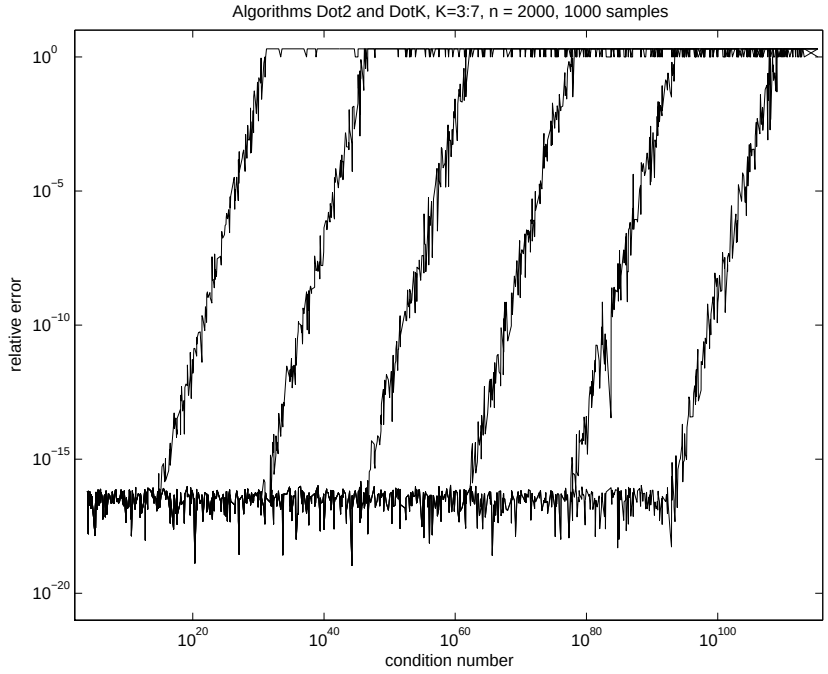

FIG. 6.4. *Test results for* `Dot2` *and* `DotK`*,* $K = 3 : 7$*,* $n = 2000$*,* 1000 *samples*

TABLE 6.5
Measured computing time for environments in Table 6.3 with BLAS (`DDOT`) *normed to 1*

	`Dot2`	`DotXBLAS`	`DotK` (K=3)	`DotK` (K=4)	`DotK` (K=5)
env. I)	13.5 (0.00)	18.5 (0.00)	21.3 (0.67)	28.2 (0.83)	35.1 (1.00)
env. II)	22.7 (0.33)	38.3 (0.33)	38.0 (0.33)	48.7 (0.33)	59.0 (0.33)
theor.	12.5	18.5	18.5	24.5	30.5

Furthermore we display the timing of Algorithm 5.10 (`DotK`) for the dot product in K-fold working precision. For vector lengths of 100 to 5000 in steps of 100 we tested some 1000 samples each. We compute the measured computing times of the listed routines relative to that of the BLAS routine `DDOT`; in Table 6.5 we display the median over all test cases together with the median absolute deviation in parenthesis. In the last row we display the theoretical ratio (number of flops for the listed routines divided by $2n - 1$ flops for `DDOT`).

The numbers confirm the expected ratio 37 : 25 between XBLAS and `Dot2` . The flop counts given in Propositions 5.5 and 5.11 for `DotK` relative to `DDOT` are confirmed as well.

For the first environment the theoretical ratio is approximately achieved, where in the second environment the measured computing time is about 2 times slower than theoretically expected. Regarding that the multiply-and-add instruction is likely to be used, the flop count of `DDOT` drops to n, and the measured computing time for first environment is much better than theoretically expected. This may be due to the fact that more operations are performed in the inner loop thus avoiding possible cache misses.

TABLE 6.6
Measured ratio of computing time for matrix-vector residual for environments in Table 6.3

	`Dot2`	`DotXBLAS` (F)	`DotXBLAS` (C)
environment I)	3.8 (0.07)	5.0 (0.10)	6.5 (0.84)
environment II)	4.8 (0.13)	9.3 (0.23)	11.3 (0.29)
theoretical	12.5	18.5	18.5

A very typical and more practical application is using a precise dot product for the iterative refinement of the solution of a linear system.

We wrote a straightforward (unblocked) code for the computation of $Ax - b$ using `Dot2` and XBLAS routine `BLAS_ddot_x`, where the $n \times n$ matrix A and n-vectors x and b are randomly generated. We tested dimensions $100, 200, \ldots, 3000$ and computed the ratio of computing times to the corresponding BLAS routine `DGEMV`. In Table 6.6 we display the median of the ratios over all test cases together with the median absolute deviation in parenthesis. The table gives timing for the Fortran- and for the C-code of XBLAS.

Table 6.6 shows that `Dot2` takes on the average only about 5 times as long for a result as if computed in quadruple precision compared to BLAS `DGEMV` in double precision. This is significantly faster than theoretically expected. The reason may be again that many operations in the inner loop can be performed in registers. The numbers also confirm the expected ratio $37 : 25$ between XBLAS and `Dot2`.

Let us interpret Table 6.6 again. Algorithm `Dot2` delivers a result as if computed in quadruple precision and rounded to double. Simulating quadruple precision in double, we expect a factor 4 in computing time. But this is about the factor we measure between `DGEMV` and n-fold application of `Dot2`. And this although the BLAS-routine uses optimized code whereas we implemented straightforward code. So our algorithm shows practically optimal performance !

Frequently few residual iterations in quadruple (i.e. twice the working) precision suffice to produce an accurate approximation to the solution of a system of linear equations. To test this we generate linear systems with $n = 3000$ unknowns and condition numbers between 1 and 10^{15}. For the generation of an ill-conditioned matrix A of specified condition number we use `randsvd` (cf. [15], Section 28.3) and compute the right hand side by $b = A * \texttt{ones}(n, 1)$. So the approximate solution is the vector e with all components equal to 1.

However, the multiplication $A * \texttt{ones}(n, 1)$ to generate the right hand side b is contaminated by rounding errors, so that the difference of the *true* solution of the *generated* linear system and e is of the order of the condition number. To display the *true* error of the computed solution we need a reference. The computation of the true solution of the generated linear system in multiple precision arithmetic is too time consuming for a system of that size.

Fortunately, we can use so-called verification or self-validating methods (cf. [40]) which compute rigorous error bounds. We used the implementation `verifylss` in INTLAB (cf. [39]), which takes about six times the computing time of the Matlab built-in linear system solver. For high accuracy error bounds we used Algorithm 5.3 (`Dot2`) for iterative refinement. This sounds like a vicious circle: We test the accuracy of `Dot2` by using a routine where `Dot2` is involved. However, iterative refinement is only used to improve an *approximate* solution; the error of this approximation is *rigorously estimated* by the self-validating routine `verifylss` (for details, cf. [40] and

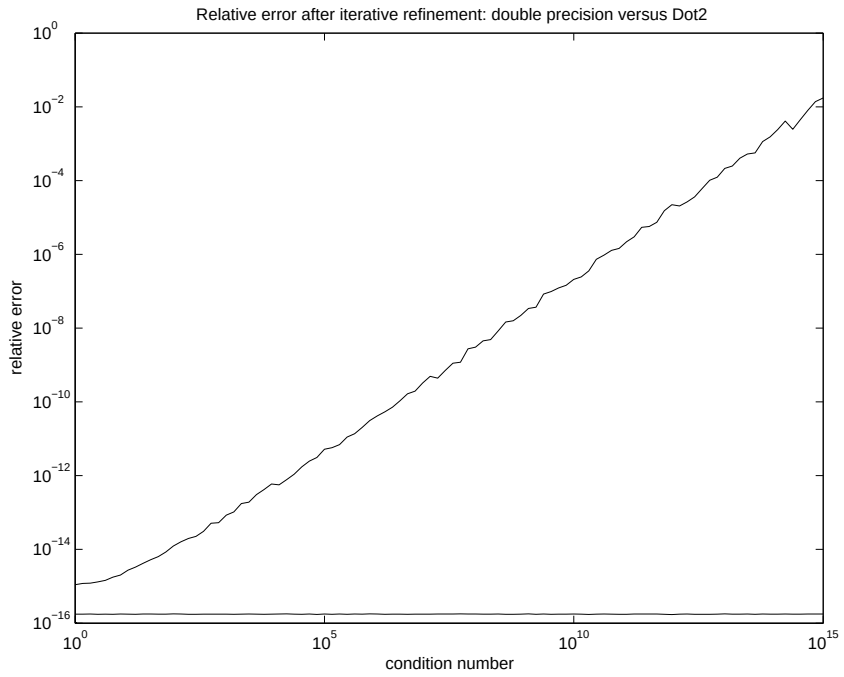

FIG. 6.5. *Iterative refinement using double and* `Dot2` *for 3000 unknowns*

the literature cited there). All error bounds had a width of two or three units in the last place for all solution components, so they are of almost maximum accuracy.

Figure 6.5 displays the maximum relative error of all solution components for performing residual iterations *i)* in working precision and *ii)* using `Dot2`. The two curves confirm the expected behavior; it was the same for both testing environments listed in Table 6.3. The relative error for iterative refinement in working precision generates a backward stable solution (cf. [42]), but the forward error (i.e. the relative error to the true solution) is of the order $\texttt{eps} \cdot \mathrm{cond}\,(A)$. For iterative refinement using Algorithm 5.3 (`Dot2`) we achieve a maximum relative error 10^{-16} independent of the condition number. The number of residual iterations was detected by a standard stopping criterion, in fact exactly the same as used in `verifylss` in INTLAB (cf. [39]), ranging from 2 to maximally 8 iterations depending on the condition number.

The additional (measured) computing time to be invested to achieve maximum accuracy when using `Dot2` for iterative refinement was in the worst case 25 %. For moderate condition numbers (up to 10^{12}) the additional cost was less than 15 %.

7. Concluding remarks. We presented accurate and fast algorithms for summation and dot product. The code of the algorithms can be highly optimized, so they are not only fast in terms of flop count but also in terms of measured computing time.

Our algorithms use only basic floating point operations addition, subtraction and multiplication, and they use only the same working precision as the data. This offers the possibility to put them into numerical library algorithms since no special computer architecture is required. This may especially be interesting for iterative refinement since highly accurate results can be computed at reasonable additional cost.

We stress again that our algorithms are based on the error-free transformations

`TwoSum` and `TwoProduct` (Algorithms 3.1 and 3.3). If those would be available directly from the processor, precise dot product evaluation in doubled working precision by `Dot2` would cost only twice as much as the ordinary dot product. An alternative would be routines `TwoSumADD3` and `TwoProductFMA` (Algorithms 3.6 and 3.5) with the advantage of having only one output argument.

The error-free transformations allow a nice and simple analysis; we frequently made use of the mathematical equalities in (2.3). We hope that this article can be a step towards hardware implementations and standardization of error-free transformations to further improve the accuracy of numerical algorithms.

Acknowledgement. The authors wish to express their hearty thanks to the anonymous referees. Their most useful comments and constructive suggestions helped to improve the paper significantly.

REFERENCES

[1] I. J. Anderson, *A distillation algorithm for floating-point summation*, SIAM J. Sci. Comput., 20 (1999), pp. 1797–1806.

[2] ANSI/IEEE, *IEEE Standard for Binary Floating Point Arithmetic*, IEEE, New York, Std 754–1985 ed., 1985.

[3] I. Babuška, *Numerical stability in mathematical analysis*, in Proc. IFIP Congress, vol. 68 of Information Processing, Amsterdam, 1969, North-Holland, pp. 11–23.

[4] D. H. Bailey, *A Fortran-90 double-double precision library.* `http://crd.lbl.gov/~dhbailey/mpdist/mpdist.html`.

[5] ———, *A Fortran-90 based multiprecision system*, ACM Trans. Math. Softw., 21 (1995), pp. 379–387.

[6] G. Bohlender, *Floating-point computation of functions with maximum accuracy*, IEEE Trans. Comput., C-26 (1977), pp. 621–632.

[7] G. Bohlender, W. Walter, P. Kornerup, and D. W. Matula, *Semantics for exact floating point operations*, in Proceedings of the 10th IEEE Symposium on Computer Arithmetic, P. Kornerup and D. W. Matula, eds., Grenoble, France, 1991, IEEE Computer Society Press, pp. 22–26.

[8] S. Boldo and M. Daumas, *Representable correcting terms for possibly underflowing floating point operations*, in Proceedings of the 16th IEEE Symposium on Computer Arithmetic, J.-C. Bajard and M. Schulte, eds., Santiago de Compostela, Spain, 2003, IEEE Computer Society Press, pp. 79–86.

[9] K. L. Clarkson, *Safe and effective determinant evaluation*, in Proceedings of the 33rd Annual Symposium on Foundations of Computer Science, Pittsburgh, Pennsylvania, 1992, IEEE Computer Society Press, pp. 387–395.

[10] T. J. Dekker, *A floating-point technique for extending the available precision*, Numer. Math., 18 (1971), pp. 224–242.

[11] J. Demmel and Y. Hida, *Accurate and efficient floating point summation*, SIAM J. Sci. Comput., 25 (2003), pp. 1214–1248.

[12] J. Grabmeier, E. Kaltofen, and V. Weispfennig, *Computer Algebra Handbook*, Springer, 2003.

[13] J. R. Hauser, *Handling floating-point exceptions in numeric programs*, ACM Trans. Program. Lang. Syst., 18 (1996), pp. 139–174.

[14] N. J. Higham, *The accuracy of floating point summation*, SIAM J. Sci. Comput., 14 (1993), pp. 783–799.

[15] ———, *Accuracy and Stability of Numerical Algorithms*, SIAM Publications, Philadelphia, 2nd ed., 2002.

[16] H. Hlavacs and C. W. Ueberhuber, *Improving the accuracy of numerical integration*, Technical Report TR 2001–06, AURORA, 2001.

[17] M. Jankowski, A. Smoktunowicz, and H. Woźniakowski, *A note on floating-point summation of very many terms*, J. Information Processing and Cybernetics-EIK, 19 (1983), pp. 435–440.

[18] M. Jankowski and H. Woźniakowski, *The accurate solution of certain continuous problems using only single precision arithmetic*, BIT, 25 (1985), pp. 635–651.

[19] W. Kahan, *A survey of error analysis*, in Proc. IFIP Congress, vol. 71 of Information Processing, Amsterdam, 1972, North-Holland, pp. 1214–1239.
[20] ———, *Implementation of algorithms (lecture notes by W. S. Haugeland and D. Hough)*, Tech. Rep. 20, Department of Computer Science, University of California, Berkeley, CA, USA, 1973.
[21] ———, *Doubled-precision IEEE standard 754 floating point arithmetic.* manuscript, 1987.
[22] A. Kiełbaszyński, *Summation algorithm with corrections and some of its applications*, Math. Stos., 1 (1973), pp. 22–41.
[23] D. E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms*, vol. 2, Addison-Wesley, Reading, Massachusetts, 1969.
[24] U. Kulisch and W. L. Miranker, *The arithmetic of the digital computer: A new approach*, SIAM Review, 28 (1986), pp. 1–40.
[25] H. Leuprecht and W. Oberaigner, *Parallel algorithms for the rounding exact summation of floating point numbers*, Computing, 28 (1982), pp. 89–104.
[26] X. Li, J. Demmel, D. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Kang, A. Kapur, M. Martin, B. Thompson, T. Tung, and D. Yoo, *Design, implementation and testing of extended and mixed precision BLAS*, ACM Trans. Math. Softw., 28 (2002), pp. 152–205. `http://crd.lbl.gov/~xiaoye/XBLAS/`.
[27] S. Linnainmaa, *Software for doubled-precision floating point computations*, ACM Trans. Math. Softw., 7 (1981), pp. 272–283.
[28] M. Malcolm, *On accurate floating-point summation*, Comm. ACM, 14 (1971), pp. 731–736.
[29] O. Møller, *Quasi double precision in floating-point arithmetic*, BIT, 5 (1965), pp. 37–50.
[30] F. Mosteller and J. W. Tukey, *Data Analysis and Linear Regression*, Addison-Wesley, Reading, MA, 1977.
[31] A. Neumaier, *Rundungsfehleranalyse einiger Verfahren zur Summation endlicher Summen*, ZAMM, 54 (1974), pp. 39–51.
[32] K. Nickel, *Das Kahan-Babuškasche Summierungsverfahren in Triplex-ALGOL 60*, ZAMM, 50 (1970), pp. 369–373.
[33] Y. Nievergelt, *Scalar fused multiply-add instructions produce floating-point matrix arithmetic provably accurate to the penultimate digit*, ACM Trans. Math. Softw., 29 (2003), pp. 27–48.
[34] J. Peña, *Computing the distance to infeasibility: Theoretical and practical issues*, tech. rep., Center for Applied Mathematics, Cornell University, 1998.
[35] M. Pichat, *Correction d'une somme en arithmétique à virgule flottante*, Numer. Math., 19 (1972), pp. 400–406.
[36] D. M. Priest, *Algorithms for arbitrary precision floating point arithmetic*, in Proceedings of the 10th IEEE Symposium on Computer Arithmetic, P. Kornerup and D. W. Matula, eds., Grenoble, France, 1991, IEEE Computer Society Press, pp. 132–145.
[37] ———, *On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*, PhD thesis, University of California at Berkeley, CA, USA, 1992.
[38] D. R. Ross, *Reducing truncation errors using cascading accumulators*, Comm. ACM, 8 (1965), pp. 32–33.
[39] S. M. Rump, *INTLAB – INTerval LABoratory*, in Developments in Reliable Computing, T. Csendes, ed., Kluwer Academic Publishers, Dordrecht, 1999, pp. 77–104. `http://www.ti3.tu-harburg.de/~rump/intlab/`.
[40] ———, *Self-validating methods*, Linear. Alg. Appl., 324 (2001), pp. 3–13.
[41] J. R. Shewchuk, *Adaptive precision floating-point arithmetic and fast robust geometric predicates*, Discrete Comput. Geom., 18 (1997), pp. 305–363.
[42] R. D. Skeel, *Iterative refinement implies numerical stability for Gaussian elimination*, Math. Comp., 35 (1980), pp. 817–832.
[43] The MathWorks Inc., *Matlab user's guide, version 7*, 2004.
[44] G. Zielke and V. Drygalla, *Genaue Lösung Linearer Gleichungssysteme*, GAMM Mitteilungen, 26 (2003), pp. 7–108.