Accurate Sum and Dot Product with Applications

Takeshi Ogita and Siegfried M. Rump and Shin'ichi Oishi

***Abstract*— In a recent paper the authors presented a new and very fast algorithm for accurate computation and inclusion of the sum and dot product of floating point numbers. In this paper we show that the algorithms can be used to compute a very accurate inclusion of the solution of systems of linear equations. As a basic building block, accurate solution of linear equations has applications in very many areas.**

I. Accurate summation

Let $\mathbb{F}$ denote the set of floating point numbers, and $\mathbb{F}^n$, $\mathbb{F}^{n\times n}$ the set of vectors, matrices over those, respectively. For $x \in \mathbb{F}^n$, the computation of an accurate approximation of $\sum x_i$ is a most basic task in numerical analysis. Accordingly, there is a huge number of papers dealing with this problem, and Higham [5] denotes an entire chapter to it. For bibliographical references see [5], [10].

Recently, the authors of this paper presented a new and very fast method for this problem and for the computation of dot products [10]. The approach uses only basic floating point operations, no special architecture and no special operations. The speed of the new approach stems from the facts that

- no sorting input data is necessary, neither
  i) by absolute value nor
  ii) by exponent,
- the algorithms contain not a single branch,
- no extra precision besides working precision is necessary,
- no access to mantissa or exponent is necessary.

(1)

That makes the algorithms not only fast but also widely applicable because they are executable on every standard hardware. To our knowledge all existing algorithms fail to satisfy one or more of the listed properties. However, each is important for a really fast algorithm.

The new methods are based on so-called error-free transformations. Denote by $\mathrm{fl}(\cdot)$ the result of an expression where every operation is executed in floating point in some given working precision. It is well known that floating point operations according to the IEEE 754 floating point standard [6] are of maximum accuracy, that is

$$\begin{aligned} |a \circ b - \mathrm{fl}(a \circ b)| &\leq \mathrm{eps}\,|a \circ b| \\ &\text{and} \\ |a \circ b - \mathrm{fl}(a \circ b)| &\leq \mathrm{eps}\,|\mathrm{fl}(a \circ b)| \end{aligned}$$

for all $a, b \in \mathbb{F}$ and for $\circ \in \{+, -, \cdot, /\}$ provided no underflow occurs. Here eps denotes the relative rounding error unit. For IEEE 754 double precision it is $\mathrm{eps} = 2^{-53} \sim 1.2 \cdot 10^{-16}$. In case of underflow, a tiny constant of size 2^{-1073} has to be added to the estimations.

It is known that for every pair of floating point numbers $a, b \in \mathbb{F}$, the error of the sum $x := \mathrm{fl}(a + b)$ is again a floating point number, that is for all $a, b \in \mathbb{F}$ there exist $x, y \in \mathbb{F}$ with

$$x := \mathrm{fl}(a + b) \quad \text{and} \quad x + y = a + b. \tag{2}$$

The transformation $(a, b) \to (x, y)$ can be regarded as an error-free transformation of the pair (a, b) into the best floating point approximation x of the sum and into the exact error y. Fortunately, there is a very fast algorithm to compute (x, y) due to Knuth [7]:

```
function [x, y] = TwoSum(a, b)
    x = fl(a + b)
    z = fl(x − a)
    y = fl((a − (x − z)) + (b + z))
```

Algorithm 1: Error-free transformation of the sum of two floating point numbers

Note that this algorithm has all properties listed in (1); it uses solely ordinary floating point addition and subtraction. An alternative approach is based on an algorithm by Dekker [3]:

```
function [x, y] = TwoSum1(a, b)
    x = fl(a + b)
    if |a| ≥ |b|
        y = fl(b − (x − a))
    else
        y = fl(a − (x − b))
```

Algorithm 2: Error-free transformation of sum with branch

If one counts the branch as well as every addition or subtraction as one flop, then the Algorithm 1 requires 6 flops compared to 4 flops for Algorithm 2. The results of both algorithms are always identical. One might expect a 50% slower performance of Algorithm 1. As we will see, due to the fact that branches have very negative effects on compiler optimization, the opposite is the case.

Simple floating point operations are of maximum accuracy in IEEE 754. However, composite operations may

published in Proceedings of 2004 IEEE International Symposium on Computer Aided Control Systems Design, Taipei, pages 152155, 2004

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, Japanoishi@waseda.jp

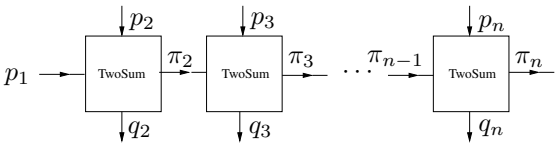

Fig. 1. Cascaded error-free transformation

bear an arbitrarily large relative error due to the limited computing precision. To compute the sum of n numbers p_i, the simple idea in [10] is to cascade Algorithm 1. This looks as follows.

Each box represents an application of Algorithm 1. So (2) implies $p_1 + p_2 = \pi_2 + q_2$, $\pi_2 + p_3 = \pi_3 + q_3$, etc., so that

$$\sum_{i=1}^{n} p_i = \pi_n + \sum_{i=2}^{n} q_i. \tag{3}$$

Also note that $\pi_n = \mathrm{fl}(\sum_{i=1}^{n} p_i)$, so that π_n is the result of ordinary floating point summation, and the q_i represent the exact error of that approximation.

In [10] it is shown that adding up the errors in ordinary floating point and adding this to π_n yields a result of the same quality as if computing in quadruple precision. The algorithm is as follows.

$$\begin{array}{l}\text{function res} = \mathrm{Sum1}(p)\\ \quad \text{for } i = 2:n\\ \quad\quad [p_i, p_{i-1}] = \mathrm{TwoSum}(p_i, p_{i-1})\\ \quad \text{res} = \mathrm{fl}\left(\left(\sum_{i=1}^{n-1} p_i\right) + p_n\right)\end{array}$$

Algorithm 3: Cascaded summation

Theorem 1: Suppose Algorithm 3 (Sum1) is applied to arbitrary floating point numbers $p_i \in \mathbb{R}$, $1 \le i \le n$. Define $s := \sum_{i=1}^{n} p_i$ and set $S := \sum_{i=1}^{n} |p_i|$. Denote by res the result of Algorithm 3. Then, also in the presence of underflow,

$$|\text{res} - s| \le \text{eps}\,|s| + c^2 S, \tag{4}$$

where eps denotes the relative rounding error unit and $c := n\text{eps}\,/(1 - n\text{eps}\,)$.

The proof is given in [10]. If the ordinary summation $\sum_{i=1}^{n} p_i$ would be executed in quadruple precision and the result rounded to double precision, then essentially the best achievable error estimate would be like (4).

We mention that the cascade presented in Fig. 1 can be staggered so that the accuracy of the final result is as if computed in k-fold precision. For details see [10].

Algorithm 3 can also be implemented using Algorithm 2 (TwoSum1) instead of Algorithm 1 (TwoSum). The latter costs 6 flops compared to 4 flops for the former counting the if-statement as one flop. Following are computing times in microseconds for the summation of randomly generated vectors of different lengths.

Instead of a decrease of 50% in performance when using TwoSum we observe an *increase* in performance of about

TABLE I
COMPUTING TIMES FOR ALGORITHM 3 USING TwoSum1 OR TwoSum

	Algorithm 3 with	
n	TwoSum1	TwoSum
100	0.90	0.62
10000	88	61
1000000	8844	6625

that size. It shows the effect of the lack of optimal code optimization.

II. ACCURATE DOT PRODUCTS

For vectors $x, y \in \mathbb{F}^n$, the dot product $x^T y$ is $\sum_{i=1}^{n} x_i y_i$. In order to apply the accurate summation algorithms presented in the last section, all we need is a representation of the products $x_i y_i$ as the sum of two floating point numbers. A first idea is to split x_i and y_i in two halves and use all cross products. However, the usual splitting of a 53-bit double precision number would result in a 26-bit and a 27-bit part, so that one cross product could be 54 bits long and not representable in double precision.

Amazingly, there is an algorithm by Dekker [3] splitting a 53-bit double precision number without error into two 26-bit numbers.

$$\begin{array}{l}\text{function } [x, y] = \mathrm{Split}(a)\\ \quad \text{factor} = 2^{27} + 1\\ \quad c = \mathrm{fl}(\text{factor} \cdot a)\\ \quad x = \mathrm{fl}(c - (c - a))\\ \quad y = \mathrm{fl}(a - x)\end{array}$$

Algorithm 4: Error-free split of a floating point number into two parts

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

Algorithm 4 satisfies $x + y = a$ for all $a \in \mathbb{F}$ provided no underflow occurs. With this we can formulate an error-free transformation of a product $a \cdot b$ of two floating point numbers $a, b \in \mathbb{F}$ into the sum of two floating point numbers $x + y = a \cdot b$ with $x, y \in \mathbb{F}$ and $x = \mathrm{fl}(a \cdot b)$. The algorithm due to Veltkamp [3] is as follows.

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

Algorithm 5: Error-free transformation of the product of two floating point numbers

With this we have all ingredients to present a fast and accurate algorithm for the dot product $x^T y$.

function res = Dot1(x, y)
  $[p, s]$ = TwoProduct(x_1, y_1)
  for $i = 2 : n$
    $[h, r]$ = TwoProduct(x_i, y_i)
    $[p, q]$ = TwoSum(p, h)
    $s = \mathrm{fl}(s + (q + r))$
  res = $\mathrm{fl}(p + s)$

Algorithm 6: Dot product in doubled working precision

The result of Algorithm 6 satisfies an error estimate as if computed in quadruple precision. Note however that all presented algorithms satisfy all properties listed in (1). In [10] the following is proved.

Theorem 2: Let $x_i, y_i \in \mathbb{F}$, $1 \le i \le n$, be given and denote by res the result of Algorithm 6 (Dot1). Then, if no underflow occurs,

$$|\mathrm{res} - x^T y| \le \mathrm{eps}\,|x^T y| + c^2 |x^T|\,|y|,$$

where eps denotes the relative rounding error unit and $c := n\mathrm{eps}\,/(1 - n\mathrm{eps}\,)$.

We mention that as in the case of summation a cascaded algorithm can be used to calculate an approximation to $x^T y$ as if computed in k-fold precision. We also note that there is a beautiful relation to the condition number of the dot product, namely

$$\left| \frac{\mathrm{res} - x^T y}{x^T y} \right| \le \mathrm{eps}\, + \frac{1}{2} c^2 \cdot \mathrm{cond}\,(x^T y).$$

This shows the maximum size of the relative error of res [10].

III. Applications

Several applications of our algorithm are now in order.

A. Application to linear systems

There are a number of so-called self-validating algorithms to compute an inclusion of the solution of a system of linear equations $Ax = b$ [9], [13], [1], [11]. One example uses the so-called Krawczyk operator [8]. Let $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$ be given as well as $\tilde{x} \in \mathbb{R}^n$, $R \in \mathbb{R}^{n \times n}$ and $X \in \mathbb{IR}^n$. Here $\mathbb{IR}^n$ denotes the set of n-dimensional interval vectors. For details concerning intervals and interval operations cf. [9]. Assume

$$R \cdot (b - A\tilde{x}) + (I - RA) \cdot X \subseteq \mathrm{int}(X), \qquad (5)$$

where all operations are interval operations, I denotes the identity matrix and $\mathrm{int}(X)$ denotes the interior of X. Then it follows [13] that

- A and R are nonsingular, and
- $A^{-1}b \in \tilde{x} + X$.

Note that there are no additional assumptions on R, $\tilde{x}$ or X; the assertion is true in any case. In order to satisfy assumption (5), R should be chosen to be an approximate inverse of A and $\tilde{x}$ to be an approximate solution of $Ax = b$.

TABLE II
Results without and with residual iteration

condition number	10^5	10^9	10^{13}
max. rel. err. $\tilde{x}$	4.7e-12	2.8e-08	1.9e-04
max. rel. err. x^k	1.8e-16	1.8e-16	1.8e-16
number of iterations	3	3	5
ratio computing time	1.38	1.38	1.65

We stress again that no explicit assumption on that need to be verified.

There is much to say haw to determine an appropriate X and how to build an algorithm for computing an inclusion of $A^{-1}b$. The interested reader is referred to [12]. An easy-to-use and public domain Matlab toolbox for interval computations is INTLAB [14]. Included in there is also a self-validating algorithm `verifylss` for linear systems. It also covers over- and underdetermined as well as sparse linear systems.

The accuracy of an approximate solution $\tilde{x}$ of $Ax = b$ can be improved by a residual iteration. This requires an accurate calculation of the residual $A\tilde{x} - b$. It is known that an approximate solution accurate to the last bit can be calculated if the residual $A\tilde{x} - b$ can be calculated in quadruple precision (with result rounded to double) [5]. But exactly this is done by Algorithm 6.

Assume an approximate solution $\tilde{x} \approx A^{-1}b$ has been computed using an LU-decomposition with partial pivoting. For $y \in \mathbb{R}^n$ denote by solve(y) the approximate solution of the linear system $Ax = y$ using the previously computed LU-decomposition. This solution process costs $\mathcal{O}(n^2)$ operations. Then

$$\begin{array}{lcll} x^0 & = & \tilde{x}; & \\ x^{k+1} & = & x^k + \mathrm{solve}(b - Ax^k) & \text{for } k = 1, 2, \ldots \end{array}$$

represents a residual iteration. Suppose the residual $b - Ax^k$ is computed using Algorithm 6. Then the following table shows the number of iterations and achieved maximum relative error for 1000×1000 linear systems of different condition numbers with averaged results of 100 randomly generated linear systems. We display the maximum relative error of the initial approximative $\tilde{x}$ without residual correction. The additional computing time is determined by the number of residual iterations, whereas one iteration costs only $\mathcal{O}(n^2)$ operations. Therefore the additional costs to achieve high accuracy are moderate. In the last row of Table II we display the ratio of computing time with residual refinement divided by the one without residual refinement. The additional cost depend on the condition number and is between 30 and 70 per cent. For larger dimensions the additional cost decreases.

The same principle can be applied to the verified inclusion of a linear system $Ax = b$. For that purpose we need an inclusion, i.e. lower and upper bounds for the residual $A\tilde{x} - b$. This is performed by the following modification of

TABLE III

ACCURACY OF INCLUSION AND RATIO COMPUTING TIME TO PURE FLOATING- POINT

condition number	10^5	10^9	10^{13}
max. rel. err. of Y	2.5e-09	1.5e-05	1.2e-01
ratio computing time	6.96	6.96	7.26

Algorithm 6.

function $[\text{res}, \text{err}] = \text{Dot1Err}(x, y)$
  $[p, s] = \text{TwoProduct}(x_1, y_1)$
  $\text{err} = |s|$
  for $i = 2 : n$
    $[h, r] = \text{TwoProduct}(x_i, y_i)$
    $[p, q] = \text{TwoSum}(p, h)$
    $s = \text{fl}(s + (q + r))$
    $\text{err} = \text{fl}(\text{err} + (|q| + |r|))$
  $\text{res} = \text{fl}(p + s)$
  $\text{err} = \text{fl}(\text{err}/(1 - (n + 2)\text{eps}))$

Algorithm 7: Dot product in doubled working precision with error bound

It follows [10] that the interval res$\pm$err is a valid inclusion of the exact value of the dot product $x^T y$. Note that again only pure floating point operations are used and that the algorithm satisfies all properties listed in (1).

Using Algorithm 7 we can provide an inclusion $Y := \tilde{x} + X$ based on (5) of high accuracy. It is mainly based on the previously mentioned residual iteration for $\tilde{x}$ and on the accurate calculation of bounds for the residual $b - A\tilde{x}$ by Algorithm 7. The following table shows the accuracy of the achieved inclusion and the ratio in computing time T/t for several condition numbers. Here T denotes the computing time to compute a validated inclusion Y, and t denotes the time to compute an approximation by DGESV, the BLAS [4] routine for solving a linear system. Both algorithms are executed with residual refinement. Again we take the average over 100 randomly generated samples.

There is a factor in computing time of about 7 we have to pay. However, the result by the self-validating algorithm is verified to be correct. That means, provided the hardware and the architecture works correctly, the linear system is *proved* to be solvable (i.e. the system matrix A is not singular) and the computed interval vector Y is *proved* to contain the exact solution $A^{-1}b$ of the linear system. Using the newly developed algorithms for fast and accurate computation of dot products the achieved inclusion is almost of maximum accuracy.

B. Linear inequalities in control theory

Many design problems of control systems reduce to a problem of solving the following linear inequality

$$\alpha_1 A_1 + \alpha_2 A_2 + \cdots + \alpha_n A_n + A_0 < 0. \qquad (6)$$

Here, A_i's are $n \times n$ symmetric matrices and α_i's are scalars [2]. For a given set of α_i's, the left hand side of inequality (6) is reducing to nothing but the problem of calculating dot product.

IV. CONCLUSION

We presented fast algorithms for the accurate computation of sums and dot products. In combination with self-validating methods they allow to compute verified inclusions of the solution of linear systems of high quality. The proof of correctness as well as the high accuracy of the results may be important in sensitive areas such as certain control problems.

REFERENCES

[1] G. Alefeld and G. Mayer. Interval Analysis: Theory and Applications. *J. Comput. Appl. Math.*, 121(1-2):421–464, 2000.

[2] S. Boyd, L. El GThaoui, E. Feron, and V. Balakrishnan. *Linear Matrix Inequalities in System and Control Theory*, volume 15. SIAM, Philadelphia, PA, 1994.

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

[4] J.J. Dongarra, J.J. Du Croz, I.S. Duff, and S.J. Hammarling. A set of level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 16:1–17, 1990.

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

[6] *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*, 1985.

[7] D.E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison Wesley, Reading, Massachusetts, second edition, 1981.

[8] R. Krawczyk. Newton-Algorithmen zur Bestimmung von Nullstellen mit Fehlerschranken. *Computing*, 4:187–201, 1969.

[9] A. Neumaier. *Interval Methods for Systems of Equations*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1990.

[10] T. Ogita, S.M. Rump, and S. Oishi. Fast and Accurate Computation of Scalar Product. to appear, 2004.

[11] S. Oishi and S.M. Rump. Fast verification of solutions of matrix equations. *Numer. Math.*, 90(4):755–773, 2002.

[12] S.M. Rump. Verification Methods for Dense and Sparse Systems of Equations. In J. Herzberger, editor, *Topics in Validated Computations — Studies in Computational Mathematics*, pages 63–136, Elsevier, Amsterdam, 1994.

[13] S.M. Rump. Self-validating methods. *Linear Algebra and its Applications (LAA)*, 324:3–13, 2001.

[14] S.M. Rump. INTLAB - Interval Laboratory, a Matlab toolbox for verified computations, Version 3.1, 2002. `http://www.ti3.tu-harburg.de/rump/intlab/index.html`.