Comparison of Accurate Dot Product Algorithms

Jürgen Wolff V. Gudenberg

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Comparison of Accurate Dot Product Algorithms

Jürgen Wolff v. Gudenberg

Institut für Informatik

Universität Würzburg

N° 2413

Octobre 1994

PROGRAMME 6

ISSN 0249-6399

Comparison of Accurate Dot Product Algorithms

Jürgen Wolff v. Gudenberg*
Institut für Informatik
Universität Würzburg

Programme 6 — Calcul scientifique, modélisation et logiciel numérique
Projet Aladin

Rapport de recherche n° 2413 — Octobre 1994 — 25 pages

Abstract: A recently proposed algorithm for the accurate computation of the dot product [Kob 94] has been implemented and modified in order to obtain the best possible algorithm for the Aquarels toolbox. It is compared with the other well known algorithms with respect to runtime and intermediate storage space. A considerable improvement of the performance is obtained by a combination of this algorithm with an old one.

Key-words: computer arithmetic, dotproduct computation

(Résumé : tsvp)

*This work has been prepared during a stay at IRISA Rennes, supported by C.I.E.S

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 84 71 71

Comparaison d'algorithmes pour calculer précisément le produit scalaire

Résumé : Nous étudions ici un nouvel algorithme pour calculer le produit scalaire précis et nous le comparons aux algorithmes connus. Quelques modifications de cet algorithme sont mises en œuvre dans l'atelier Aquarels et en améliorent ainsi les performances.

Mots-clé : produit scalaire, arithmetique précise des ordinateurs

1 Introduction

The accurate computation of a dot product (or synonymously: scalar product) is essential for many numerical programs. In the next section we sketch several algorithms for the exact computation of these dot products

$$s := \square(\sum_{i=1}^{n} a_i \cdot b_i)$$

of floating-point numbers a_i, b_i with a single rounding $\square$, i.e. all intermediate results have to be computed *without rounding error*! It is obvious that the result will be different from the usually computed approximation

$$s := a_1 \boxtimes b_1 \boxplus a_2 \boxtimes b_2 \boxplus ... \boxplus a_n \boxtimes b_n$$

where $\boxplus$, $\boxtimes$ denote floating-point operations.

First of all the computation of double length products has to be performed without rounding error. If no double length floating-point format is available, this can be performed by splitting the factors and computing partial products of factors where at least half of all digits are zero. Thus no rounding error occurs. This means that we have to add 4 to 6 –if mantissa length is odd – numbers instead of one product but the problem then is reduced to the computation of a sum. We show that it suffices to compute 5 partial products for a binary system.

The algorithms which are introduced in the next chapter have been developped in the framework of the Karlsruhe accurate arithmetic approach [Kul 76, Kul 81], see also [Boh 90] for an overview.

We then compare these algorithms and a recently published method [Kob 94] with respect to time and space complexity. We also investigate the requirements for computing accumulated dotproducts, i.e. an addition of a value to an already computed dotproduct should be performable so that the updated value again is of optimal accuracy.

A detailed rounding error analysis of some newly proposed modifications follows in section 5. A description of the FORTRAN 77 implementation and some results of execution time measurements conclude this report.

2 Dotproduct Algorithms – an Overview

2.1 Long Accumulator

A very simple, but powerful algorithm for the computation of dot products makes use of a so-called dot precision variable which is basically a long fixed-point accumulator A permitting the addition of any product of floating-point numbers without rounding error ([Rump 80, Boh 83]): let u and v be two floating-point numbers with base β, l digits in the mantissa and exponent range $emin, \ldots, emax$. The product $x := u \cdot v$ has $2 \cdot l$ digits in the mantissa. x is converted into a fixed-point number by shifting its mantissa left or right according to its exponent which is in the range $2 \cdot emin, \ldots, 2 \cdot emax$. Finally the shifted value is added to the dot precision variable A *without rounding error*. $2 \cdot l + 2 \cdot emax + 2 \cdot |emin| + g$ digits are required for the representation of A, where g guard digits are added in order to prevent overflow. Therefore, even the square $L \cdot L$ of the largest floating-point number $L = 0.(\beta - 1) \cdots (\beta - 1) \cdot \beta^{emax}$ can be added β^g times without overflow.

Discussion: The long accumulator may be implemented as a vector of integers. After n additions the infinitely precise result is obtained, so the algorithm clearly is linear. Since we only add products instead of long accumulators , the constant factor before the n is not greater than 3. Accumulating scalar products are obtained just by addtion to the accumulator.

So this algorithm is a very efficient solution of the given problem. A portable version of it using integer arithmetic to a large extent has been implemented in the AQUARELS project [Pao 93].

We nevertheless introduce the other, in fact older, dotproduct algorithms in the following paragraphs.

2.2 Addition with remainder

The first algorithm which could guarantee nearly full precision for the computation of a floating-point sum $s := \sum_{i=1}^{n} x_i$ was found by Pichat (see [Pich 72]). It is based on the observation that under certain assumptions about the rounding $\square$, the remainder of the rounded sum $a \boxplus b$ which is defined by $r := a + b - (a \boxplus b)$ can be represented as a floating-point number (provided that no overflow or underflow occurs). Starting with the values x_i, a sequence of values x_i' is computed as follows:

for $i := n - 1$ downto 1 do
  $s := x_i \boxplus x_{i+1}$
  $r := x_i + x_{i+1} - s$

$x'_i := s$
$x'_{i+1} := r$
end for i

This new sequence has the properties that $\sum x'_i = \sum x_i$, and x'_1 is an approximation of the sum $\sum x_i$, while $x'_2, \cdots, x'_n$ are remainders. Repeating the same process leads to values $x_i^{(k)}$; under certain conditions $x_1^{(k)}$ converges to a value $\tilde{s}$ with an error of less than two units of the last place of the mantissa. This algorithm was modified and applied to compute dot products in [Boh 77, Boh 78]: using an appropriate estimation of the sum of remainders $\sum_{i=2}^{n} x_i^{(k)}$, the iteration can be terminated as soon as the required accuracy is reached. Secondly, it could be proved that (if one disregards zeros) the operands $x_1^{(k)} \cdots x_{k+1}^{(k)}$ are ordered according to their exponents and if two operands overlap according to their exponents, the mantissa of the larger value contains only digits zero in the overlapping region. Therefore, the iteration may be stopped after at most $k = n - 1$ steps when all operands have been ordered. It can be proved that

$$\Box \sum_{i=1}^{n} a_i \cdot b_i = \Box \sum_{i=1}^{n} x_i = \Box x_1^{(k)}$$

for all relevant rounding modes $\Box$ and for doubled precision operands x_i and $x_1^{(k)}$.

Discussion : Worst case time complexity is $O(n^2)$, although we may assume that in general only two iterations are necessary. But even then the factor is relatively high since an addition with remainder costs at least three floating-point operations.

If the input vectors must be kept, a temporary vector of n components is necessary. This vector can be used as intermediate storage for the infinitely precise result, where after the summation usually many remainders are exactly 0 and may be dropped. When a kind of normalization is applied, so that the resulting vectors have exponent differences not less than the mantissa length l , $(2(emax - emin) + 1)/l$ components suffice (see 2.1). The addition of one value to that exact interpretation means an iteration for this number of components.

2.3 Order by Exponents

There exists another algorithm [Kul 76] which orders the summands according to their exponents and then starts to add from left to right, i.e. starting with the highest exponent, until the difference of the exponents of the sum and the next summand is larger than 2. This addition has to be performed without rounding error. Then the sum can be computed by addition from right to left.

Discussion: Because of the sorting the complexity of this algorithm can not be better than $O(n \cdot \log(n))$. During the sorting phase values with the same exponent are added. So a vector of mostly $2(emax - emin) + 1$ values has to be stored. Again the addition of one number means an insertion in that vector and an addition of all its components.

For detailed discussion of these dot product algorithms and related problems see e.g. [Boh 90].

2.4 Store Preprocessed Summands

Very recently, Kobbelt has published a new dotproduct algorithm [Kob 94] which uses standard IEEE floating-point operations to a large extent and intermediately stores the summands in a table with $4(emax - emin + 1)$ components.

This algorithm which has proven to be very fast is described in detail in the next section.

3 Kobbelt's Dotproduct Algorithm

The algorithm which is formulated for binary arithmetic mimics the procedure of the *Order by Exponents* algorithm and consists out of 3 phases.

3.1 Phase 1

The summands which are partial products represented in usual floating-point format are inserted in a table, so that the sum of all table entries is the exact dotproduct.

To limit the size of the table to a constant we distinguish between 'odd' and 'even' floating-point numbers according to their last bit. This property of a number is called its genus.

We then use the following propositions about the arithmetic.

- Addition of two numbers with the same exponent and the same genus is exact.
- Addition of two numbers with the same exponent and opposite sign is exact.

The summands are inserted in an array with $2(emax - emin + 1)$ rows, one for each exponent and 2 columns, one for each genus. If the location where the number is to be inserted is occupied with a nonzero value or its correspondant address for the different genus contains a value of opposite sign, an exact addition is performed and the resulting sum is inserted in the table.

At the end of this phase the sum of all table entries is exactly the wanted sum and the table can be processed in the order of the exponents, for each exponent there are at most 2 entries.

Discussion: Each addition in this phase reduces the number of table entries. Thus besides the insertion into the table no overhead is produced with respect to the rounded computation of the sum.

If dynamic memory management is available, the size of the table may be reduced either by its organization as a hash table, or by allocating storage only for the actually used range of values depending on the minimal and maximal exponents. Since this increases runtime and is also applicable to the long accumulator algorithm, we do not exploit this option further.

Before we come to the other phases, we want to describe the computation of double length products in more detail.

3.2 Double-length multiplication

To compute a double-length product in the usual floating-point format x and y are split into halves, if mantissa length is odd the second half of y is split again. This would result in inserting 6 partial products into the table.

1. split $x = x1 + x2; y = y1 + y2; y2 = y21 + y22$
2. insert $x1y1, x1y21, x1y22, x2y1, x2y21, x2y22$

Taking the genus of the numbers into account for the base $\beta = 2$ the number of partial products can be reduced to 4, if both factors are even, and to 5, if one or both are odd.

1. if x and y are odd, split x into $msb(x)$ and $x1 = x - msb(x)$
   where $msb(x) = 0.5 \cdot 2^{exponent(x)}$
   insert $msb(x) * y$
2. else $x1 = x$
3. split $x1 = x11 + x12; y = y1 + y2$
4. insert $x11y1, x11y2, x12y1, x12y2$

The subtraction and multiplication in step 1 are exact, since both entries have the same exponent and one is a power of the base. If both x and y are odd, $x1$

is even, since one bit is cancelled, hence in step 3 at least one number is even and will therefore be split into 2 parts each of which has $k = \lfloor l/2 \rfloor$ digits.Note that in this case $2k = l - 1$. The products in step 4 therefore have at most l digits and are computed exactly.

Another alternative could be to collect the partial products in two floating-point numbers representing the approximate result and the remainder and insert these two.

1. split $x = x1 + x2; y = y1 + y2; y2 = y21 + y22$
2. $s := x \boxtimes y$
3. $r := ((((x1y1 \boxminus s) \boxplus x1y2) \boxplus x2y1) \boxplus x2y21) \boxplus x2y22$
4. insert s and r

This would reduce the number of table insertions, but since an insertion is a cheap operation, we did not implement this variant.

The following splitting procedure which only needs floating-point arithmetic produces a k-digit number $x1$ and a $l - k$ digit number $x2$, where $1 \leq k \leq l - 1$

split ($x, x1, x2$)

- $t := (\beta^{l-k} + 1) \boxtimes x$
- $x1 := t \boxminus (t \boxminus x)$
- $x2 := x \boxminus x1$

This algorithm is correct for an arbitrary monotone rounding, if arithmetical operations are optimally defined. It can also be used to determine the genus of x , when setting $k = l - 1$ and testing $x2$ for zero.

Proofs may be found in [Lin 81]

If the even part of a number is needed, which indeed is the case in the original algorithm [Kob 94], $x1$ has to be calculated as a truncation which in turn fixes the rounding mode:

- $t := (\beta^{l-k} + 1) \boxtimes \dagger x$
- $x1 := t \boxminus \dagger (t \boxminus \ddagger x)$

where operations with $\dagger$ are truncated and with $\ddagger$ are rounded away from zero.

If the layout of a floating-point number is fixed, as for the IEEE format,e.g., splitting of a number may more efficiently be implemented by exploiting bit field operations.

3.3 Phase 2

In phase 2 the entries of the table are transformed in such a way that all have the same sign. This can be achieved without changing the sum of the table entries by the exploitation of one of the following formulas

$$x + y := a2^{m+d} - b2^m = \sum_{i=0}^{d-1} a2^{m+i} + (a-b)2^m$$

$$x + y := a2^{m+d} - b2^m = \sum_{i=1}^{d-1} a2^{m+i} + (2a-b)2^m$$

where x and y are two adjacent table entries with different sign. Since either $a-b$ or $2a-b$ has the same sign as a we can modify the table by replacing x and y by the summands of the right hand side.

After the iterated application of this operation to all pairs of adjacent table entries with different sign from high to low exponents all values in the table have the same sign. Since the multiplications with a power of the base 2 as well as the additions which are actually performed are rounding error free, the value of the dotproduct still is the exact sum of all table entries which now in turn all have the same sign.

So the sign of a dotproduct may easily be read from the table. This operation is helpful for computation of bounds.

Note, however, that the number of summands may be considerably increased. For the subtraction of the square of the smallest number from the square of the largest number,e.g., half of the table, i.e. one entry for each exponent, are filled in. Therefore we investigated some alternatives which are presented in the following chapter.

3.4 Phase 3

In phase 3 the table entries are added from low to high exponents. The following algorithm guarantees a rounding error less than 2 ulps [Kob 94].

```
for exp := low to high
    if expo(sum) > exp then
      aux := make_even( table[exp,odd])
      sum := sum ⊞ (aux ⊞ table[exp,even])
    else
      sum := (sum ⊞ table[exp,odd]) ⊞ table[exp,even]
    end if
end for
```

3.5 Discussion

The runtime complexity of the whole algorithm is $O(n)$. All operations are standard floating-point operations, so the fast floating-point hardware may be used. Despite the fill-in in phase 2 the factor for the summation of the table usually is nearly 1. Due to the double-length calculation of products the execution time is between $8n$ and $10n$ floating-point operations (multiplications and additions) plus a constant overhead. If, however, the algorithm is to be compared with the usual rounded algorithm the index computations for the table insertion may not be neglected on modern computers, as we will see from our measurements presented in chapter 8.

The amount of storage to keep the infinitely precise result is relatively high, $4(emax - emin + 1)$ numbers, although in many applications a smaller vector will suffice.

The accumulation of an additional value means its insertion into the table (phase 1), the adjustment of signs, if necessary (phase 2) and finally the new computation of the sum of all table entries (phase 3).

4 Modifications of Kobbelt's Dotproduct Algorithm

In this section we describe some modifications of Kobbelt's Dotproduct Algorithm which have been implemented in order to

- increase accuracy
- reduce runtime
- reduce storage requirements
- facilitate accumulation

4.1 Addition with Remainder in Phase 3

The maximal rounding error committed in phase 3 is 2 ulps. If we replace the addition of the table entries by the following algorithm, this error is computed in *rem* and an update of the result *sum* is possible to obtain 1 ulp accuracy.

```
for exp := low to high
    if expo(sum) > exp then
      aux := make_even( table[exp,odd])
      (aux,r1) := aux + table[exp,even]
      (sum,r2) := sum + aux
    else
      (aux,r1) := sum + table[exp,odd]
      (sum,r2) := aux + table[exp,even]
    end if
    rem := r1 ⊞ r2
end for
```

After the for loop sum and rem can be added and subtracted to receive a one ulp approximation or a sharp inclusion of the result.

Each adition with remainder, however, costs 3 simple additions and therefore this variant is debatable. It is too slow to beat the long accumulator implementation.

4.2 Replace Phase 2

In phase 2 the number of table entries is increased in general. It may be replaced by a cheaper procedure which computes a rough approximation of the sum of the table entries, so that total cancellation is avoided in phase 3. This procedure is the addition of the table entries from high to low exponents until the exponent difference between the sum and the next entry is greater than 3. This corresponds to the addition from left to right in the algorithm which sorts the summands. In [Kul 76] it is shown that the remaining summands may not totally cancel the result, if the exponent difference is greater than 2. Since we may have 2 entries for each exponent we have to replace this by 3. The addition from left to right, however, has to be performed without rounding error, it determines the order of magnitude of the sum and gives a rough approximation. Therefore we apply addition with remainder and insert the remainders into the table. Finally the sum is inserted in the table,

which still represents the exact value of the dotproduct and may now be added from low to high exponents to obtain a good approximation.

If no cancellation occurs the number of additions with remainder is at most 5 and the number of table entries remains constant. If cancellation of more than two digits occurs in a single addition which can only be the case, if the exponent difference is 1, the exact result can be represented as a floating-point number and thus the remainders are 0. Hence the number of table entries is decreased. Cancellation of one digit, however, may start an addition procedure which runs through the whole table.

In the cases of large fill-in which means large exponent differences one or zero additions from left suffice and this procedure clearly is superior to the adjustment of signs.

Replacement of phase 2 by addition from left to right reduces the number of additions in the average. Its rounding error behavior is analyzed in section 6.

4.3 Rough Approximation

The idea of initial addition from high to low exponents can be extended, so that the relative error of the remaining terms diminishes. If we compute until the difference of exponents between sum and first table entry is d, an approximation of the sum with relative error 2^{4-d} is obtained, even if we discard all remaining entries. This follows from the fact that the table entries are ordered by exponents with at most 2 values for each exponent. See section 6 for details.

Because all the remainders have to be inserted in the table and some of them have to be reconsidered for higher accuracy, this procedure saves time only for very low accuracy requirements.

4.4 Combination with Long Accumulator

We perform phase 1 and then add the table entries using a long accumulator. Hence optimal accuracy is obtained, the storage for the infinitely precise result is minimized, and accumulation of scalar products consisting out of several parts is facilitated.

5 Summarized Comparison of the Algorithms

In the following table 1 we depict for each algorithm its maximal and minimal execution time measured in floating-point additions A, and Comparisons C where the average case will be close to the minimal time very often.

We assume that double length products are computed by the splitting method described in section 3.2 except for the long accumulator where integer arithmetic is used. Hence in all other algorithms a minimum effort of 4n and a maximum of 5n multiplications M have to be added to the table entries.

Storage requirements are stored in the last 2 rows. MT the maximally needed space during the computation disregarding integers and stack space. MI the memory used to represent the infinitely precise result. The values are given in bits where V denotes the number of bits used for a floating-point number and e = emax-emin+1.

It should not be concealed that on modern computers, in particular RISC architectures, the integer time should actually not be neglected. We further do not consider scaling efforts in this comparison which are necessary for all the algorithms which store the result as a vector of floating-point numbers, i.e. all without the long accumulator. The algorithms are denoted by the numbers of the preceeding paragraphs in which they were described.

	2.1	2.2	2.3	
max	nP+R	$3(25n^2$ -5n)A	cn log(n) C +5nA'	
min	nP+R	24nA	cn log(n) C + 4nA'	
MT	2e	5nV	5nV	
MI	2e	2e	2eV	

	3	4.1	4.2	4.4
max	(5n+4e)A+4eM + 6nC	(15n+3e)A + 6nC	(5n+3e)A + 6nC	(5n-4e)A + 4eS + 6nC
min	4nA+3nC	4nA+3nC	4nA+3nC	4nA+3nC+S
MT	4eV	4eV	4eV	4eV
MI	4eV	4eV	4eV	2e

Table 1: Comparison by operation count and storage requirements

All table entries are obtained by counting the operations assuming the following properties.

For the long accumulator (2.1) P denotes an addition of a product and S that of a value, R the extraction of the final result. Roughly estimated the time for P equals that for 4M+2.5A , that for S = 1.5A, and that for R = e/V A, but these figures highly depend on the hard- and software of the system.

Addition with remainder used in (2.2) and (4.1) costs 3 additions.

The maximal performance of (2.2) is quadratic for 5n summands whereas the minimal is obtained by 2 iterations of 4n summands.

The time for the order by exponent algorithm (2.3) is dominated by the sorting time. A' means the specific addition with longer mantissa.

The maximal fill-in for Kobbelt's algorithm (3) is obtained in pathologically construed examples only, the same holds for (4.2).

Addition with remainder can be combined with sign adjustment or addition from left to right, in (4.1) the latter is listed.

The minimal execution time for all algorithms using the large table is obtained, if all entries are added during phase 1.

In (4.4), if 5n > 4e the maximum time is obtained, if the table is filled in phase 1, hence a maximal number of additions to the long accumulator has to be performed.

A table insertion costs between 3 and 6 comparisons with 0.

6 Rounding Error Analysis

We recall that $R(\beta, l, emin, emax)$ denotes the floating-point system with base β , l mantissa digits, and exponent range $emin..emax$. The normalization is such that $1/\beta \leq m < 1$ for each mantissa m . As in the previous section $\boxplus$ denotes addition with rounding to the nearest floating-point number. $\tilde{s}$ is the computed approximation to s .

Lemma 1: Let $a_i = m_i \beta^{e_i} \epsilon R(\beta, l, emin, emax), i = 1(1)n$ with

$$e_1 \geq e_2 + d$$

$$e_i \geq e_{i+1} \wedge e_i > e_{i+2}, i \geq 2$$

Let further $s = \sum_{i=2}^{n} a_i$ and $\tilde{s}$ be its approximate value computed by the following algorithm:

$\tilde{s} := a_n$
for $i = n - 1$ downto 2
  $\tilde{s} := \tilde{s} \boxplus a_i$
end for

Then the relative error ε of the sum of all a_i is bounded by
$\varepsilon \leq \varepsilon 1 + (1 + \varepsilon 1)\frac{1}{2}\beta^{1-l}$
with

$$\varepsilon 1 \leq 16 \frac{\beta^{-d}}{\beta^{-1} - 4\beta^{-d}} \frac{1}{2} \beta^{-l}$$

Proof

$$\varepsilon = \left|\frac{a_1 + s - (a_1 \boxplus \tilde{s})}{a_1 + s}\right| \leq \left|\frac{a_1 + s - (a_1 + \tilde{s})}{a_1 + s}\right| + \left|\frac{a_1 + \tilde{s} - (a_1 \boxplus \tilde{s})}{a_1 + \tilde{s}}\right| \left|\frac{a_1 + \tilde{s}}{a_1 + s}\right|$$

$$\leq \left|\frac{s - \tilde{s}}{a_1 + s}\right| + \frac{1}{2}\beta^{1-l} \left|\frac{a_1 + \tilde{s}}{a_1 + s}\right|$$

We first derive a lower bound for the denominator $a_1 + s$, then compute the absolute error of the sum. If we then have a bound $\varepsilon 1$ for the first error term the fraction in the second term is bounded by $1 + \varepsilon 1$

$$|a_1 + s| = \left|a_1 + \sum_{i=2}^{n} a_i\right|$$

$$\geq |a_1| - \sum_{i=2}^{n} |a_i| \geq |a_1| - \sum_{i=2}^{n} (1 - \beta^{-l})\beta^{e_i}$$

$$\geq |a_1| - \sum_{i=2}^{n} \beta^{e_i} = |a_1| - \beta^{e_2} \sum_{i=2}^{n} \beta^{e_i - e_2}$$

$$\geq |a_1| - \beta^{e_2} \cdot 2 \sum_{i=0}^{(n-2)/2} \beta^{-i} \geq |a_1| - \beta^{e_2} \cdot 2 \sum_{i=0}^{\infty} \beta^{-i}$$

$$= |a_1| - \beta^{e_2} \frac{1}{1 - \beta^{-1}} \geq |a_1| - 4\beta^{e_2}$$

$$\geq \beta^{e_1 - 1} - 4\beta^{e_2} \geq \beta^{e_1 - 1} - 4\beta^{e_1 - d}$$

We may assume w.l.o.g. $s \geq \tilde{s}$, when we now derive a bound for the absolute error of the sum.

$$s - \tilde{s} = \sum_{i=2}^{n} a_i - (...(a_n \boxplus a_{n-1}) \boxplus ... \boxplus a_2) = \sum_{i=2}^{n} a_i - \sum_{i=2}^{n} a_i + \delta_i = \sum_{i=2}^{n} \delta_i$$

where δ_i denotes the absolute error in the $i-$th summation step which is bounded by
$\delta_i \leq |s_i| \frac{1}{2} \beta^{-l}$
Here we apply rounding to the nearest otherwise the factor $\frac{1}{2}$ has to be dropped.

Now we have to bound s_i . This again is done by exploiting the order of exponents.

$$s_i \leq \sum_{j=i}^{n} |a_j| \leq 4\beta^{e_i}$$

Therefore

$$s - \tilde{s} = \sum_{i=2}^{n} \delta_i \leq 4\frac{1}{2}\beta^{-l} \sum_{i=2}^{n} \beta^{e_i} \leq \frac{1}{2}\beta^{-l} 16\beta^{e2}$$

and

$$\varepsilon 1 \leq 16\frac{\beta^{-d}}{\beta^{-1} - 4\beta^{-d}}\frac{1}{2}\beta^{-l}$$

•

Lemma 1 may now be applied for different bases β and which is the more interesting case to determine the exponent difference d necessary to guarantee a specified accuracy of the approximation.

Note that due to the final addition the maximal roundoff error can not be bounded by $\frac{1}{2}ulp = \frac{1}{2}\beta^{1-l}$ without further consideration of the specific situation.

In the following part we assume $l > 2$.

Corollary 2: Let the algorithm to determine the sum be given as in Lemma 1. For $\beta = 2$ and $d = 5$ we obtain

$$\varepsilon \leq 2^{1-l}$$

which means a rounding error of one ulp .

Proof

$$\varepsilon 1 \leq 16\frac{2^{-5}}{2^{-1} - 4 \cdot 2^{-5}}\frac{1}{2}2^{-l} = \frac{2}{3}2^{-l} \Rightarrow \varepsilon 1 + (1 + \varepsilon 1)2^{-l} < 2 \cdot 2^{-l}$$

•

For $d = 4$ we obtain $\varepsilon 1 \leq 2^{1-l}$ which implies $\varepsilon < 3.1 \cdot 2^{-l}$, but if we change our addition algorithm by summing up the two entries with equal exponent before adding them to the final sum, we can improve our estimation of s_i .

Corollary 3: Let the summation algorithm be given as follows

$\tilde{s} := 0$
$i := n$
while $i > 2$ do
    if $e_i = e_{i-1}$ then
        $\tilde{s} := \tilde{s} \boxplus (a_i \boxplus a_{i-1})$
        $i := i - 2$

else
$\tilde{s} := \tilde{s} \boxplus a_i$
$i := i - 1$
end if
end while

For $\beta = 2$ and $d = 4$ we then obtain

$$\varepsilon \leq 2.6 \cdot 2^{-l}$$

Proof
The worst case for bounding the absolute errror of the sum again occurs, if there are two entries for each exponent, otherwise a factor of 2 is easily gained. But with the new summation algorithm half of the local errors δ_i are bounded by $2\beta^{e_i}$ rather than $4\beta^{e_i}$. This leads to the estimation

$$s - \tilde{s} = \sum_{i=2}^{n} \delta_i \leq \frac{1}{2}(4\sum_{i=2}^{n} \beta^{e_i} + 2\sum_{i=2}^{n} \beta^{e_i})\frac{1}{2}\beta^{-l} \leq \frac{1}{2}\beta^{-l}12\beta^{e2}$$

Inserting $d = 4$ and $\beta = 2$ this yields

$$\varepsilon 1 \leq \frac{3}{2}2^{-l}$$

which implies the assertion.

•

We can do even better. Kobbelt shows in [Kob 94] that for the summation algorithm given in section 3.4 the absolute error of $s - \tilde{s}$ is bounded by $4\beta^{e_2}\beta^{-l}$. The assumption of equally signed summands fits well in our considerations, since this is the worst case in order to obtain large cancellation of the previously calculated approximation a_1.

Corollary 4: Let the algorithm to determine the sum be given as in section 3.4
For $\beta = 2$ and $d = 4$ then holds

$$\varepsilon \leq 2.1 \cdot 2^{-l}$$

Proof
Using Kobbelt's bound we obtain

$$\varepsilon 1 \leq 2^{-l}$$

and the result is obvious.

•

Finally we want to compute the rounding error for the approximation a_1

Corollary 5: In the notation of Lemma 1 let the exponent difference

$$d = e_1 - e_2 \geq 4$$

and $\beta = 2$

Then a_1 approximates the sum of all a_i with a maximal relative error

$$\varepsilon 0 \leq \frac{2^{2-d}}{2^{-1} - 2^{2-d}} \leq 2^{4-d}$$

Proof

$$\varepsilon 0 = \frac{|s|}{|a_1 + s|} \leq 4\frac{\beta^{-d}}{\beta^{-1} - 4\beta^{-d}} = \frac{2^{2-d}}{2^{-1} - 2^{2-d}} \leq 2^{4-d}$$

7 Implementation Description

Kobbelt's dotproduct algorithm as well as the before mentioned variants and modifications have been implemented in FORTRAN 77 and compared with the long accumulator implementation of Fortran Aquarels [Pao 93]. The algorithms are callable as subroutines with interfaces identical or at least similar to those for the AQUARELS implementation. Indeed, the program `dottest` features an interactive test suite for all dotproduct algorithms.

The following remarks characterize the implementation

- The programs are written in standard FORTRAN 77
- The low level operations like splitting a floating-point number, extracting the exponent or genus use the IEEE representation of `double precision` numbers.
- Scaling is not performed, hence the products have to have exponents in the double precision range.
- Error handling is not implemented, the parameter `err` should not be used. It is kept to have the same interfaces as the long accumulator routines.

- The large table is organized as a one-dimensional array where the odd and even values for the same exponent are adjacent. This table is created in the top level subroutines and passed as parameter to the others. Its occupied range are all entries between the minimum and maximum index which in turn are also passed to and accordingly updated by the called subroutines.

The software is organized in the following files

filename	contents
dottest	interactive testprogram
kobsub	dotproduct subroutines
phase1	subroutines for table insertion
phase2	adjustment of signs and computation of leading term
phase3	sum of table entries
port1	primitives for splitting etc.

Table 2 : Organisation of souce files

7.1 Dotproduct Subroutines

We list the interfaces of the dotproduct subroutines together with a short description of their purpose. For more detailed documentation we refer to the source texts.

`x` and `y` are vectors with `dim` components which are accessed using the strides `incx` and `incy`, respectively. `err` is an unused integer error flag.

```
DOUBLE PRECISION FUNCTION KDP23( x, incx, y, incy, dim, err)
```

Kobbelt's original algorithm.

```
DOUBLE PRECISION FUNCTION KDPL3( x, incx, y, incy, dim, err)
```

The adjustment of signs is replaced by addition from left to right

```
DOUBLE PRECISION FUNCTION KDPLR( x, incx, y, incy, dim, err)
```

The adjustment of signs is replaced by addition from left to right Addition from right to left updates the error term by adding up the remainders.

```
DOUBLE PRECISION FUNCTION KDPLRN( x, incx, y, incy, dim,n, err)
```

compute rough approximation. relative error 2^{-n}

```
DOUBLE PRECISION FUNCTION KDPLRI(x,incx, y,incy, dim,left,right,err)
```

compute inclusion, collect the error term by addition with remainder, add with directed rounding. This routine uses directed rounding via Sun's interface to IEEE arithmetic.

```
DOUBLE PRECISION FUNCTION KDPACC( x, incx, y, incy, dim, err)
```

Insert products into table, and then add table entries with long accumulator.

This routine delivers dotproducts with optimal accuracy which are also suitable to be updated by accumulation and can be used as `dot precision` variables in Fortran aquarels programs.

7.2 Table Insertion

The insertion of products into the large table is the essential new idea of Kobbelt's algorithm. It is accomplished by a call to

```
SUBROUTINE PINSERT (XX,YY, T, MINIX, MAXIX)
```

which splits the operands and inserts the 4 or 5 partial products by calling

```
SUBROUTINE INSERT (X, T, MINIX, MAXIX)
```

In both subroutines T denotes the table the sum of whose entries represents the exact dotproduct. The occupied range in this table is between the indices `minix` and `maxix`. A formal proof that the invariant is kept may easily be deduced from the source code.

7.3 Normalizing the Table

The table has to be normalized before its entries can be summed up from right to left. The normalization is either an adjustment of the signs or a computation of the leading term of the sum by addition from left to right. It is important that the sum of the table entries does not change.

```
SUBROUTINE PHASE2 ( T, MINIX,MAXIX)
```

equalizes the signs.

```
SUBROUTINE ALR ( T, MINIX,MAXIX)
```

adds leading terms until difference of exponents is larger than 3. The remainders and the final sum are inserted in the table, hence its sum keeps invariant.

`DOUBLE PRECISION FUNCTION ALRN (N, T, MINIX,MAXIX)`

adds leading terms until difference of exponents is larger than n + 3

7.4 Sum of the Table

In the file `phase3.f` the different routines to sum up the table entries are comprised. In particular there are

`DOUBLE PRECISION FUNCTION ARL ( T, MINIX, MAXIX)`

Kobbelt's original procedure exploiting as much of the table structure as possible.

`DOUBLE PRECISION FUNCTION KARL ( T, MINIX, MAXIX)`

Simple sum from right to left.

`DOUBLE PRECISION FUNCTION EARL ( T, MINIX, MAXIX)`

Add entries with equal exponent before adding to the sum. (not yet implemented)

`SUBROUTINE ARL2 ( SUM, REM, T, MINIX, MAXIX)`

Sum of the table with remainder to obtain highest accuracy.

7.5 Primitives for elemetary operations

These routines all exploit the specific lay-out of an IEEE double precision floating point number.

`SUBROUTINE AWR (S,R,X,Y)`

computes addition with remainder, i. e. (S,R) = X+Y Is valid for binary arithmetic with rounding to the nearest.

`INTEGER FUNCTION SIGNUM ( X )`

determines the sign and is written in standard Fortran.

`INTEGER FUNCTION GENUS ( X )`

determines the genus of a number by testing the last bit of the representation using 32 bit integer arithmetic. A call of an explicit bit test function may improve the performance. A similar comment holds for the following procedures.

Another anomaly with this procedure is that currently an 'odd' number with the last bit set has the genus 0. This may be changed in a future version but requires rewriting of the calling routines.

`DOUBLE PRECISION FUNCTION GENEVE ( X )`

generates the 'even' version of a number by clearing the last bit.

`INTEGER FUNCTION INDEXP ( X )`

determines the table index of a number according to its exponent. Actually the table position of the value is `indexp(x) + genus(x)` where `indexp(x) = 2(exponent(x) + 1024)`

`DOUBLE PRECISION FUNCTION MSB ( X )`

computes the floating-point number dertermined by the most significand matissa bit, i.e. `msb(x) =` $2^{exponent(x)}$.

`DOUBLE PRECISION FUNCTION HALF1 ( X )`

computes the floating-point number dertermined by the first 26 matissa bits,the remaining bits are set to 0.

8 Runtime Comparison

Runtime measurements have been performed on a Sun SPARC station using the `time` command of Sun's Fortran. Randomly generated vectors with varying dimension and varying exponent difference have been generated. The dotproduct algorithms are repeated in a loop which is obviously not optimized by the compiler. Before we list some sample times, we summarize our observations.

For short vectors (dimension 10) the long accumulator algorithm was sometimes faster than the new one, sometimes the times were about equal. The overhead due to the initialization of the table is too large, it sometimes took more time than the

computation itself. This overhead may be decreased by using a common block for the large table. We also could observe the expected superiority of the addition from left to right in contrast to the adjustment of signs.

But for longer vectors these differences nearly vanished, all variants of the new algorithm perform equally and considerably faster than the long accumulator. The longer a vector the more work is done in phase 1 and this speeds up the algorithm.

We further observed that rounded computation is faster by a factor of about 40. This contradicts our theoretical treatment in section 5 and supports the statement that integer arithmetic may not be neglected totally.

In table 3 some sample times are listed where ediff denotes the differnce of exponents of the input vectors and rep the repetition factor. The times are given in seconds. such a way that the execution of a rounded scalar product is 1.

dim	10	100	1000	10000
ediff	100	15	600	500
rep	2000	2000	200	20
DO-loop	0	1	1	1
KDP23	25	51	36	32
KDPL3	19	47	37	32
KDPACC	60	79	63	34
LA	14	124	125	116

Table 3 : Execution times

9 Conclusion

Several new variants of scalar product algorithms have been implemented which in general are faster than those known before. Their accuracy is very high and guaranteed, so each might be used for verified accurate computation. But only the algorithm KDPACC which uses the large table for preprocessing and then adds the values into the long accumulator provides optimal accuracy for all relevant roundings. It also facilitates accumulated calculation of dotproducts and supports the data type `dot precision`. Its runtime for large vectors is about a factor of 3 faster than the long accumulator algorithm and nearly as fast as the less accurate variants. We therefore suggest to use this algorithm for vectors with many components (more than 50) and prefer the long accumulator algorithm for fewer components.

References

[Boh 77] Bohlender, G.: *Floating-Point Computation of Functions with Maximum Accuracy.* IEEE Transactions on Computers, vol. C-26, no. 7, July 1977.

[Boh 78] Bohlender, G.: *Genaue Berechnung mehrfacher Summen, Produkte und Wurzeln von Gleitkommazahlen und allgemeine Arithmetik in höheren Programmiersprachen.* Ph.D. thesis, Universität Karlsruhe, 1978.

[Boh 83] Bohlender, G., Grüner, K.: *Realization of an Optimal Computer Arithmetic.* In [Kul 83].

[Boh 90] Bohlender, G.: *What Do We Need Beyond the IEEE Standard.* In: Ullrich (ed.): Computer Arithmetic and Self-Validating Numerical Methods. Academic Press, 1990.

[Dek 71] Dekker, T.J.: *A Floating-Point Technique for Extending the Available Precision*, Num. Math.18,224-242, 1971

[Kob 94] Kobbelt,L. *A Fast Dot-Product Algorithm with Minimal Rounding Errors* Computing 52,355–369,1994

[Knu 69] Knuth, D.: *The Art of Computer Programming, Vol.2 Seminumerical Algorithms.* Addison-Wesley, Reading, 1969.

[Kul 76] Kulisch, U. : *Grundlagen des numerischen Rechnens.* BI, Mannheim, 1976

[KuBo 76] Kulisch, U., Bohlender, G.: *Formalization and Implementation of Floating-Point Matrix Operations.* Computing 16, 239–261, 1976.

[Kul 81] Kulisch, U., Miranker, W. L. : *Computer Arithmetic in Theory and Practice* Academic Press, Orlando, 1981.

[Kul 83] Kulisch, U., Miranker, W. L. (eds.) : *A New Approach to Scientific Computation* Academic Press, Orlando, 1983.

[Lin 81] Linnainmaa,S : *Software for Doubled-Precision Floating-Point Computations*, ACM ToMS, Vol 7 , No. 3 272-283, 1981

[Pao 93] Paoletti, J.-C. *Fortran Aquarels*, Rapport Technique RAP.TS.93.SEP.34, V-1.0, SIMULOG, 1993

[Pich 72] Pichat,M.: *Correction d'une somme en arithmétique à virgule flottante.* Num. Math. 19, 400–406, 1972.

[Rump 80] Rump,S.M. : *Kleine Fehlerschranken bei Matrixproblemen* Ph.D. thesis, Universität Karlsruhe, 1980.

[WvG90] Wolff v. Gudenberg,J: *Arithmetic for Vector and Parallel Computers* Decision Support Systems 7, 1991

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399