A Fast Dot-Product Algorithm with Minimal Rounding Errors

Ein schneller Algorithmus zur Berechnung von Skalarprodukten mit minimalem Rundungsfehler

Leif Kobbelt

Institut für Betriebs- und Dialogsysteme
Abteilung geometrische Datenverarbeitung
Universität Karlsruhe
Germany

COMPUTING 52 (1994), pp. 355-369, Springer Verlag

Abstract

We present a new algorithm which computes dot-products of arbitrary length with minimal rounding errors, independent of the number of addends. The algorithm has an $O(n)$ time and $O(1)$ memory complexity and does not need extensions of the arithmetic kernel, i.e., usual floating-point operations. A slight modification yields an algorithm which computes the dot-product in machine precision. Due to its simplicity, the algorithm can easily be implemented in hardware.

Abstract

Es wird ein Algorithmus vorgestellt, der ein Skalarprodukt beliebiger Länge auswertet. Bei der Berechnung tritt nur ein minimaler Rundungsfehler auf, der nicht von der Anzahl der Summanden abhängt. Der Algorithmus hat einen konstanten Speicherbedarf, einen linearen Rechenaufwand und es ist keine Erweiterung der arithmetischen Grundoperationen notwendig. Durch eine kleine Änderung erhält man einen Algorithmus, der das Ergebnis in Maschinengenauigkeit berechnet. Wegen seiner einfachen Struktur kann der Algorithmus leicht in Hardware realisiert werden.

1 Introduction

A frequent task in numerical applications is to compute dot-products of the form

$$s_n = \sum_{i=0}^{n} a_i \, b_i .$$

For example dot-products occur in matrix multiplications. The special case of an ordinary sum where all $a_i = 1$ is in general the most unstable step in every numerical algorithm that obtains its result from adding a large number of partial results (subdivision strategies). The main problem in computing dot-products results from the instability of the subtraction in floating-point arithmetics since significant digits can get lost.

There are solutions to this problem which use some kind of sorting the addends [Kul81]. These solutions need a large amount of memory space and are not very efficient — especially in cases where the number of addends is too large to store all of them at the same time in the main storage. Another solution uses a special fixed-point arithmetic with large mantissae (about 4000 bits for IEEE-arithmetics) that can represent all intermediate results without rounding errors [CXSC92]. However, the implementation of such an arithmetic is both machine dependent and inefficient since a single floating-point addition is mapped onto several fixed-point additions. The third known strategy for adding floating-point numbers is based on an addition operation which gives the rounded result and the exact remainder. Thus the addition of n numbers yields a rounded sum $\tilde{s}$ and $n-1$ remainders r_i. The remainders are used to correct the value of $\tilde{s}$ [Kul81]. As in the sorting approach it is necessary to store all addends at the same time.

A detailed overview over the various addition algorithms is given in [Boh90]. However, all the algorithms mentioned there use specialized arithmetic operations with larger mantissae than those of the addends. In this paper a solution is given which only uses standard arithmetic (e.g. the operators "$+$" and "$*$" provided by the underlying system or compiler). Thus it can be implemented independent of a particular computer. Its time complexity is linear and in most practical applications much better than for the solutions mentioned above. The memory requirement is bounded by a small constant independent of the number of addends.

Although computer arithmetic can be defined with an arbitrary base b, in this paper we restrict ourselves to the binary case. This is done for practical reasons since floating-point arithmetics with other bases than 2 are very rare. Moreover, the algorithm can be formulated in a very neat and compact form for this special case. Hence we understand by machine independence that the algorithm is defined relative to an abstract computer arithmetic definition, e.g., the IEEE-floating-point format [IEEE85]. The independence concerns the architecture of the hardware platform.

The definition of a standard binary computer arithmetic includes the size r of the mantissa, the exponent range, and the basic operators "$+$", "$-$", "$*$", and "$/$" that compute a result with machine precision, i.e., with maximum relative rounding error 2^{-r}. The high performance of the algorithm presented here can be increased further if a coprocessor for these basic arithmetic operations is supported by the compiler.

1.1 Over- and Underflow Errors

If the algorithm is used only for the summation of arbitrarily many floating-point numbers and not for the summation of products, then underflow errors are impossible to occur. This is obvious, since in IEEE-arithmetics no normalization of floating-point numbers is performed if the exponent would fall below the lower limit of the exponent range. In contrast, overflow errors may occur and should be regarded.

Having products to add, the situation is more difficult to handle, since intermediate multiplications may produce exponent excesses in both directions. The simplest situation arises if one can bound the exponent values in the input of the algorithm to some "middle region". Consider for example the IEEE-floating-point standard: The allowed binary exponents for double precision numbers range from -1022 to 1023. Taking the mantissa length of 53 bit into account (to prevent underflows) and adding 27 guardian bits (for the overflow case), one can avoid over- and underflow errors during the computation by requiring that each number $x = a_i$ or $x = b_i$ in the given dot-product satisfies

$$\left\lceil \frac{-1022+53}{2} \right\rceil = -484 \leq \log_2 |x| \leq 484 = \left\lfloor \frac{1023}{2} \right\rfloor - 27$$

or equivalently

$$10^{-145} \leq |x| \leq 10^{145}.$$

If this is satisfied, all intermediate results during the computation of the dot-product are within the allowed range. However, for arbitrary input data this inequality is not guaranteed to hold. Thus, overflow handling has to be introduced into the algorithm. For the sake of simplicity we first describe the ideas of the algorithm without overflow handling. In section 3, we sketch how these exceptional cases can be handled.

2 The Algorithm

Since the sum s_n does not necessarily have a representation in the floating-point format with bounded mantissa, there is no algorithm which always computes dot-products exactly. Thus, one seeks to construct an algorithm which computes a floating-point number with minimal deviation from the exact result where the deviation should not depend on the number of addends.

The basic idea of our approach is to identify operations which can be performed exactly by the standard arithmetic kernel and to construct a summation strategy which uses only these operations.

2.1 Exact Standard Operations

The following operations can be done exactly in standard binary floating-point arithmetics. The proof of their exactness is straightforward [Kob92].

`OP1`: Summation of two numbers with same exponent, same sign and same genus. The *genus* of a floating-point number is the contents of the least significant bit in its mantissa. It is used to distinguish "odd" and "even" floating-point numbers.

`OP2`: Summation of two numbers with same exponent and opposite sign.

`OP3`: Multiplication of two numbers whose floating-point representations only use less than half the mantissa.

`OP4`: Multiplication of a floating-point number by a factor 2^n.

The most important observation that can be made is that two floating-point numbers with same exponent and same genus can always be added by standard arithmetics without rounding errors.

Remark: All the operations enumerated above can be generalized to floating-point arithmetics with an arbitrary base b instead of 2. Then in the first operator `OP1`, one has to distinguish more than two genera to avoid rounding errors, i.e. numbers x and y having the same sign and exponent can be added exactly if the last digit of x is l and the last digit of y is $b - l$. In `OP4` the factor will be b^n.

2.2 The Summation Strategy

It is fairly well-known how to reduce a floating-point multiplication to a sum of intermediate products which can be computed exactly

$$(a + b)\,(x + y) = a\,x + a\,y + b\,x + b\,y$$

where a, b, x and y are single precision numbers which can be multiplied exactly in double precision arithmetics. Thus, in the following it suffices to consider sums of arbitrarily many numbers. The proposed strategy works in three phases.

Phase I

In the first phase only the addition operators `OP1` and `OP2` are used. If a new addend x is given, the algorithm searches among the earlier addends and intermediate results for a "partner" y which has the same exponent and genus. If such a partner has been found, the addition can be performed without rounding error. The same procedure is repeated for the intermediate result $x + y$. If no partner can be found, the new addend is just stored in a table which holds a place for every exponent and both genera. Since in floating-point arithmetics the number of possible exponents is finite, the maximal memory requirement for the table is bounded. Using a hash table with the exponent as key to hold the addends yields constant time complexity for the search of a partner.

The hash technique also allows to reduce the maximal memory requirement in most practical applications. This is done by using

$$k(x) := \text{EXPONENT}(x) \bmod S$$

as hashing key value for the floating-point addend x, where S is the size of the table. As long as no two intermediate results differ in their exponent by more than S, no conflicts occur. Otherwise the hash table has to be enlarged dynamically.

A sketch of an implementation of the first phase is given by the following pseudo-code which inserts the number x into the table.

```
while    TABLE [k(x), genus of x] not empty
        aux := TABLE [k(x), genus of x]
        delete TABLE [k(x), genus of x]
        x := x + aux
TABLE [k(x), genus of x] := x
```

Phase II

After all addends have been added or stored, the table still contains numbers of different signs. This can be changed by means of OP4. For example let $a\,2^{k+n}$ and $-b\,2^k$ be two addends (a and b may be both positive or both negative). Since

$$a\,2^{k+n} - b\,2^k = \sum_{i=0}^{n-1} a\,2^{k+i} + (a\,2^k - b\,2^k),$$

one can change $-b\,2^k$ into $(a-b)\,2^k$ without rounding error (OP2). This can be repeated if necessary — either $(a-b)\,2^k$ or $(2\,a-b)\,2^k$ must have the same sign as a if both numbers are represented as *normalized* floating-point numbers. Notice, the second addition $a\,2^k + (a-b)\,2^k$ is computed exactly even though $(a-b)\,2^k$ has an exponent smaller than k. This is obvious since the zeros in the least significant part of the mantissa of $(a-b)\,2^k$ are correct (a and b have no rounding errors) and are pushed out of the mantissa again before adding a because of the exponent adaption which is done implicitly by the "$+$"-operator of the underlying arithmetic kernel.

Using this idea, it is possible to transform the table such that the mantissae of the remaining entries all have the same sign (positive or negative). Thus, all the addends having a sign opposite to the result are eliminated. To guarantee that the right sign is eliminated, one starts at the top of the table where the large exponents are stored and successively eliminates each change of sign between addends. In every step, one change of sign is vanished thus termination of the second phase is ensured.

While computing the second phase, it is possible that intermediate results with smaller and smaller exponents arise. However, since in IEEE-arithmetics the normalization of floating-point numbers is omitted whenever the minimum exponent occurs, the floating-point addition of phase II in this case actually is a fixed-point addition. Thus, the iterated elimination of sign changes always terminates at the bottom of the table and it is guaranteed that during the second phase, no underflows occur.

Until this phase, no rounding error has occurred, so the whole information of the addends is still contained in the table entries. Notice, the first two phases transform the problem of adding arbitrary many numbers of different signs and exponents into the much easier problem of adding a small set of numbers all having the same sign which are ordered by their exponents.

Phase III

In the last phase all the remaining numbers have the same sign, but different exponents and genera. In this phase the numbers are added up in the order of their size, beginning with the two smallest ones. These operations may cause some rounding errors, but since the exponents of the intermediate results increase at each addition, no significant digits get lost. A detailed description with rounding error analysis is given in Section 2.3.

The operations during the last phase can not be performed exactly. Notice however, that if no table entries are modified in this phase, i.e., the additions are done by using a supplementary accumulator, then no information gets lost. Thus, the sum s_n can be computed without affecting the exactness of s_{n+m}.

2.3 Analysis of the Third Phase

We use the notation $1\Box x_a$ for a normalized floating-point number with exponent a and genus $x \in \{0, 1\}$. The field $\Box$ represents an arbitrary sequence of $r-2$ binary digits. We assume a to be the exponent of the least significant bit, e.g.,

$$1101_3 = 1101\,000, \qquad r = 4.$$

The basic arithmetic operations provided by the system cause relative rounding errors which are bounded by the *machine precision* 2^{-r} where r is the width of the mantissa. By the notation

$$1\Box x_a + 1\Box x_b = 1\Box x_c + E(c, b), \qquad a \geq b$$

we mean that adding the two numbers on the left side in computer arithmetics with constant mantissa size approximates the exact result with an absolute error being half as big as the value of the result's least significant bit in the worst case, more precisely

$$-2^{c-1} \leq E(c, b) \leq 2^{c-1} - 2^b.$$

With a mantissa size of r bits this limits the maximal relative rounding error to 2^{-r} if normalized representation is used.

A rough but intuitive estimation of the maximal rounding error can be made by the following idea. Consider in the third phase the addition of the table entries at the exponent stage a. The maximal exponent that is possible for the accumulator at this time is bounded by $a+1$ since adding all entries with smaller exponent yields

$$\underbrace{\sum_{i=-\infty}^{a-1}(2^r-1)\,2^i}_{\text{“odd”}}+\underbrace{\sum_{i=-\infty}^{a-1}(2^r-2)\,2^i}_{\text{“even”}}<(2^r-1)\,2^{a+1}.$$

In the worst case the elimination on this stage a needs two addition steps (both genera) with results having an exponent of $a+2$. Thus, using the standard operations, a maximal rounding error of 2^{a+2} is possible. Such an error can occur at each stage and hence we have a global error E_{max} of

$$E_{max}<\sum_{i=-\infty}^{q-2}2^{i+2}=2^{q+1}$$

where q is the exponent of the result s_n. Hence, E_{max} is bounded by four times the machine precision, independent of the number of addends.

A smaller maximal error, however, is achieved if the additions in the third phase are done more carefully.

To precisely analyse the maximal rounding error in the third phase of the algorithm (the other two phases cause no errors), we have to distinguish the situations where one or two table entries in an exponent stage a are occupied and the cases where the accumulator has a larger exponent than the smallest table entries or where the accumulator is smaller. We list all possible cases and indicate the maximal rounding errors. Only the rightmost tail of the sum is shown where the last addend stands for the accumulator whose exponent is bounded by $a+1$.

1.) $\quad \ldots + 1\square x_a + 1\square x_b, \qquad b \le a$

   (a) $\quad = \ldots + 1\square x_a + E(a,b), \qquad b < a$

   (b) $\quad = \ldots + 1\square x_{a+1} + E(a+1,b)$

2.) $\quad \ldots + 1\square 0_a + 1\square x_{a+1}$

   (a) $\quad = \ldots + 1\square x_{a+1}$

   (b) $\quad = \ldots + 1\square x_{a+2} + E(a+2,a+1)$

3.) $\quad \ldots + 1\square 1_a + 1\square x_{a+1}$
   $= \ldots + 1\square 0_a + 1\square x_{a+1} + 2^a$

   (a) $\quad = \ldots + 1\square x_{a+1} + 2^a$

   (b) $\quad = \ldots + 1\square x_{a+2} + E(a+2,a+1) + 2^a$

4.) $\quad \ldots + 1\square 0_a + 1\square 1_a + 1\square x_b, \qquad b \le a$

(a) $= \ldots + 1\square 0_a + 1\square x_a + E(a,b), \qquad b < a$
$= \ldots + 1\square x_{a+1} + E(a,b) + E(a+1,a)$

(b) $= \ldots + 1\square 0_a + 1\square x_{a+1} + E(a+1,b)$

i.) $= \ldots + 1\square x_{a+1} + E(a+1,b)$

ii.) $= \ldots + 1\square x_{a+2} + E(a+1,b) + E(a+2,a+1)$

5.) $\ldots + 1\square 0_a + 1\square 1_a + 1\square x_{a+1}$
$= \ldots + 1\square 0_a + 1\square 0_a + 1\square x_{a+1} + 2^a$
$= \ldots + 1\square x_{a+1} + 1\square x_{a+1} + 2^a$
$= \ldots + 1\square x_{a+2} + E(a+2,a+1) + 2^a$

If two table entries have the same exponent (but different genera), the order in which they are added to the accumulator is relevant (case (4)). In case (4.b.i) the maximal rounding error will be $E(a+1,b) + E(a+1,a)$ if the even summand is added first.

In case (5) one first adds the odd and the even entry of equal exponent and thus produces a known error of 2^a. This error has to be considered in the addition to the accumulator. A little trick can be employed to deal with this situation. Since the sum of the three addends in (5) surely has an exponent of $a+2$, the digit which is relevant for the correct rounding has the value 2^{a+1}. The last digit of $1\square 1_a$ therefore should not affect the rounding process. This is achieved by deleting this bit before adding.

Case (3) can be handled similar to case (5), i.e., directed rounding can be performed by deleting the least significant bit in the mantissa of the odd addend $1\square 1_a$. For the case (3.b) the deletion does not affect the result since the concluding rounding step does not depend on this bit.

We collect the remarks on the summation strategies in the third phase by giving the following pseudo-code:

```
for  exp := smallest exponent  to  largest exponent
      if  exponent of accu > exp  then
            aux := TABLE [exp,odd]
            delete least significant bit in aux
            accu := accu + (aux + TABLE [exp,even])
      else
            accu := (accu + TABLE [exp,odd]) + TABLE [exp,even]
```

Notice that empty table entries equal zero and thus adding to the accumulator or deleting a bit in the mantissa causes no changings.

Theorem 1 *Let $1 \square x_q$ be the result of the summation algorithm with the above mentioned handling of the cases (3), (4) and (5). Then the maximum absolute rounding error is bounded by 2^q. With normalized floating-point representation this means a relative precision of two times the machine precision ($2^{1-r} = 2 \cdot 2^{-r}$).*

Proof We prove the theorem by induction on the number of addition steps. The base case for the induction is trivial since the accumulator is zero in the beginning and adding the first table entry causes no rounding errors.

For the induction step we show a stronger result. If case (3.a) does not occur, the resulting error E is bounded by $-2^e \leq E \leq 2^{e-1}$ where e is the exponent of the accumulator after a single addition step. In case (1), since $b < e \in \{a, a+1\}$, we have

$$-2^e \leq E(e,b) - 2^b \leq E(e,b) + 2^{b-1} \leq 2^{e-1}.$$

In case (2.a) no error occurs and in (2.b) we have

$$-2^{a+2} \leq E(a+2, a+1) - 2^{a+1} \leq E(a+2, a+1) + 2^a \leq 2^a.$$

Case (3.b) is

$$-\frac{3}{4}\,2^{a+2} \leq E(a+2, a+1) + 2^a - 2^{a+1} \leq E(a+2, a+1) + 2^a + 2^a \leq 2^{a+1}.$$

The first addition in case (4) is covered by case (1) and the second addition either by case (1) or by case (2). Finally in case (5) we have

$$-\frac{3}{4}\,2^{a+2} \leq E(a+2, a+1) + 2^a - 2^{a+1} \leq E(a+2, a+1) + 2^a + 2^a \leq 2^{a+1}.$$

Only case (3.a) disturbs the stronger error estimation

$$-2^a \leq 2^a - 2^{a+1} \leq 2^a + 2^a \leq 2^{a+1},$$

but the statement of the theorem still holds. If situation (3.a) has occurred, the next addition step will be of kind (1) or (4) since the exponent of the accumulator has not increased. Hence we have after the next addition step

$$-2^e \leq E(e,b) - 2^b \leq E(e,b) + 2^b \leq 2^{e-1}$$

and thus the tighter estimation is reestablished. This concludes the induction. □

The estimation above considers the worst case which is rather unlikely to arise. In the average case the rounding error will be smaller.

2.4 An Example

A simple example is given to demonstrate the algorithm. We emphasize the occurrence of all operators described in Section 2.2 and do not give an example where cumulative addition would produce large rounding errors by cancellation of significant digits. Such examples are easy to establish but do not give much insight into the execution of the algorithm.

`Phase I`			
`add(`1011_1`)`		1011_1	
`add(`-1100_1`)`	-1100_1	1011_1	
`add(`-1110_1`)`		-1101_2	$(-1100_1 - 1110_1 = -1101_2)$
		1011_1	
`add(`-1000_1`)`		-1101_2	
	-1000_1	1011_1	
`add(`1111_3`)`		1111_3	
		-1101_2	
	-1000_1	1011_1	
`add(`1011_1`)`		1111_3	
		-1101_2	$+1011_2$ $(1011_1 + 1011_1 = 1011_2)$
	-1000_1		
		1111_3	
			$(-1101_2 + 1011_2 = -1000_0)$
	-1000_1		
	-1000_0		
`Phase II`		1111_2	
	-1000_1	1111_1	$+1111_1$ $(1111_2 + 1111_1 + 1111_1 = 1111_3)$
	-1000_0		
		1111_2	
		1111_1	$+1110_0$ $(-1000_1 + 1111_1 = 1110_0)$
	-1000_0		
		1111_2	
		1111_1	
			$(-1000_0 + 1110_0 = 1100_{-1})$
	1100_{-1}		
`Phase III`		1111_2	
		1111_1	$+1100_{-1}$
		1111_2	$+1001_2$ $(1100_{-1} + 1111_1 = 1001_2)$
`Result`	1100_3		$(1111_2 + 1001_2 = 1100_3)$

3 Over- and Underflow Error Handling

As already mentioned in the introduction, there is still a lack of robustness in the algorithm if products of very large or very small numbers occur. In order to extend the algorithm to correctly handle these cases, it is necessary to check before multiplying floating-point numbers whether the exponent of the result will exceed the allowed range. This is simply done by considering the sum of the exponents of the factors. In the sequel, we confine ourselves to describe this extension for an IEEE-arithmetic based system. Transferring the ideas to other arithmetic standards is a straightforward task.

In addition to table $\mathcal{T}$ which holds the addends during phase I, we now need two more tables $\mathcal{T}_o$ and $\mathcal{T}_u$. In the overflow case, the larger factor of the product to be added is scaled down by 2^{-1024}. As already stated in the introduction, this is always possible since this factor must be larger than 2^{484}. The result of the multiplication is then stored in $\mathcal{T}_o$ as if it was an ordinary addend (cf. phase I). In the opposite case, i.e., in case of an underflow, one scales up the smaller factor by 2^{1024} which likewise is always possible since this factor has to be smaller than 2^{-484}. The result is then stored in $\mathcal{T}_u$. All other products are stored in $\mathcal{T}$ as before. The scaling of every critical multiplication is an operation covered by the case `OP4` and therefore can be performed without rounding errors. Obviously, this ensures that no exponent excess can happen.

Another source of overflow errors during the computation are additions of very large numbers at the top of tables $\mathcal{T}$ or $\mathcal{T}_u$. These can be avoided by means of the same scaling operation as described above. In this case *both* addends are scaled by 2^{-1024} and the result is stored in table $\mathcal{T}_o$ or $\mathcal{T}$, respectively. Underflow errors due to the subtraction of very small numbers can never occur since the normalization is omitted for floating-point numbers with minimum exponent.

The second and third phase of the algorithm are realized very similar to the simpler case without over- and underflow handling, despite the fact that elimination (phase II) and accumulation (phase III) now run over all the three tables. Notice, that switching from one table to another always requires a multiplication by the factor 2^{1024} or 2^{-1024}, respectively. To prevent rounding errors, a positive number that is scaled by 2^{-1024} has to be $\geq 4 = 2^2$ and a positive number that is scaled by 2^{1024} has to be $< 1 = 2^0$.

Since the values for the exponents in the different tables $\mathcal{T}$, $\mathcal{T}_o$ and $\mathcal{T}_u$ may overlap, one has to eliminate doubly occupied stages by simply adding up the entries after suitably scaling. Ordering the three tables $\mathcal{T}_o \succeq \mathcal{T} \succeq \mathcal{T}_u$ by their significance, there is no difference whether the entry from the more significant table is scaled up or the entry from the less significant table is scaled down. The exception to this is, when the less significant entry is smaller than 4 and thus scaling it down is no longer an exact operation. If one wishes to avoid this overlapping, one has to scale both factors in phase I by a suitable amount each, instead of scaling only one by a fixed amount. Nevertheless, this more involved scaling in the first phase does not essentially simplify the following phases.

If the algorithm is used only for the summation of floating-point numbers (instead of computing dot-products) then underflow can never occur and overflow can be handled much easier by allowing more than one entry at the top of table $\mathcal{T}$, which corresponds to

the maximum exponent. On this stage only subtractions can be carried out correctly and thus addends with equal sign may not be further processed. If there is more than one entry on this stage after the second phase of the algorithm then the result can not be represented as a regular floating-point number; otherwise phase III can be performed without any changes.

4 Towards a Hardware Implementation

If the algorithm is to be implemented as a hardware unit, machine independence is no longer required. A slight modification of the third phase of the algorithm makes it possible to get the most exact result of the dot-product, i.e., machine precision is achieved. The additional requirements for this modification are the possibility of directed rounding and an accumulator with *one* supplementary bit, i.e., a mantissa size of $r + 1$ bits.

Let us assume an accumulator with an arbitrary mantissa length and fixed exponent. If we use the floating-point notation of Section 2.3 (mantissa size r bits), the addition of a table entry with exponent a in the third phase only affects the bits of the accumulator with a value $\geq 2^a$ and leaves the less significant bits unchanged. Since the accumulator mantissa is not bounded, no rounding is needed. Let 2^{q+r-1} be the value of the leftmost 1-digit of the accumulator after the summation of all table entries. Then the correct rounding of the accumulator contents to a mantissa length of r bits is determined by the digit with the value 2^{q-1}. Thus, only the upper $r + 1$ digits of the accumulator are needed to compute the result with minimal deviation.

Now suppose an accumulator with mantissa length $r + 1$. Then, after adding all table entries from exponent stages $< a$, the exponent of the accumulator is $\leq a$. Notice, following the notation of Section 2.3, the exponent of a floating-point number represents the value of its least significant bit. Hence, due to its extended mantissa, the accumulator with exponent a can represent normal floating-point numbers with exponent $a + 1$.

If both table entries for an exponent stage a are occupied the odd number is added first. This guarantees that at every time the value of the lowest bit in the extended mantissa of the accumulator is not larger than the value of the lowest 1-digit in the remaining addends. Hence the window of $r + 1$ bits in the accumulator always covers the region of the "infinite" accumulator where digits actually are modified.

If every addition step with the modified accumulator in the third phase is concluded by *truncating* instead of *rounding* to reduce the mantissa size of the intermediate result, then the $r + 1$ bits of the accumulator always represent the most significant part of the above mentioned "infinite" accumulator. This is particularly true in the end where one final rounding step yields the optimal result s_n.

Using this modification of the algorithm, the advantage of machine independence is diminished. Nevertheless, high efficiency remains. This fact is primarily interesting for a hardware implementation of the algorithm. There are arithmetic-processors which internally use a supplementary bit. The basic operations are computed by truncating behind the supplementary bit and rounding the result corresponding to the contents of this bit.

During the three phases of the algorithm, this hardware can be used in different modi: in phases I and II one computes with r bit mantissa and in phase III one computes with $r + 1$ bit mantissa but needs no rounding. Thus, for a hardware implementation, existing arithmetic units can be reused and only little control logic has to be supplemented.

5 Conclusion

The algorithm given here is $O(n)$ in time and $O(1)$ in space complexity since the second and the third phase do not directly depend on the number of addends but rather on the interval which contains the addends and all intermediate results. The use of standard arithmetics makes the algorithm machine independent. Practical experiences have shown that the memory requirement for the addend table in most applications is much smaller than the theoretical upper bound. A table which covers about $S = 200$ exponent stages ($= 3$ Kbyte) seems to be sufficient in most cases (cf. 2.2, phase I). The $O(n)$ time complexity contains only a small constant factor since the number of addition operations in the first phase is not greater than the number of addends. In the implementation of the large mantissa approach [Boh90], [CXSC92], linear time complexity is also achieved, but with a constant factor of about 3. Since the exponent interval which contains all intermediate results increases with the number of addends only as $O(\log(n))$, the second and third phase do not significantly affect the execution time of the algorithm.

Since the table entries represent the exact result (no rounding errors are made in the first two phases and the third phase does not modify the table), the precision of the sum can be increased by subtracting the first result. Let s be the exact sum of the given addends and $s' = s + \varepsilon$ the result obtained by the algorithm where the error ε is smaller than the value of the last binary digit of s. If we add $-s'$ to the table entries, we have an exact representation of $-\varepsilon$. Applying the addition algorithm once more gives a value $-\varepsilon' = -\varepsilon + \eta$ where again the error affects only the last bit of ε. Putting s' and $-\varepsilon'$ together yields the improved result $s'' = s + \eta$. This correction can be iterated.

In the analysis of the third phase we used the fact that the rounding error which occurs when adding two numbers of same exponent but different genus is fixed (Section 2.3, case (5)). This can be exploited in the first phase too and allows to reduce the memory requirements of the algorithm to its half. If a new addend is given one searches for a partner with the same exponent as described in Section 2.2. If the genera coincide, the addition can be performed without rounding error. If they differ, the rounding error amounts $\pm 2^a$ where a is the exponent of the two addends. Only a flag for this rounding error has to be maintained on each exponent stage and has to be taken into consideration the next time a new addend with this exponent is stored.

Besides reduction of the memory requirements this modification also simplifies the third phase since less special cases have to be addressed. However, the first phase requiring most of the computing time is more involved. Hence, in order to obtain a maximum time efficiency the first phase should be implemented such that the addends are inserted into the table as fast as possible.

The algorithm can be implemented in a very compact form. In order to make the algorithm available to a wide domain of users, a short and efficient implementation in C [Ker83] is offered by the author. If you like a copy of the source file send your request via e-mail to `kobbelt@ira.uka.de`.

The underlying arithmetic is the IEEE standard double format [IEEE85]. If other arithmetics are used, only the few defines in the preamble of the code have to be changed which access the various data fields of the IEEE floating point numbers.

The data structure holding the addends which have not yet found their partner for the addition without rounding error are stored in a dynamic hash table `AddTabTyp`. The initial size of this table is passed as parameter `n` to the initialization procedure `Addouble_Init()`. If the table overflows, it is enlarged automatically by procedure `Addouble_Push()`. The procedure `Addouble_Add()` performs the second and third phase of the summation algorithm.

The functionality of the procedures is obvious. Other operations like adding two `AddTabTyp` or multiplying an `AddTabTyp` by a scalar factor are straightforward to implement.

The over- and underflow error handling described in section 3 is not included but for most of the application this implementation works very well, i.e., if the bounds given in the introduction hold.

References

[Boh90] G. Bohlender, *What do we need beyond IEEE-Arithmetic?*, Computer Arithmetic and Self-validating numerical Methods, pp. 1-32, Academic Press 1990

[CXSC92] R. Klatte et al., *C-XSC: A C++ Class Library for Scientific Computing*, Institut für angewandte Mathematik der Universität Karlsruhe (Preprint)

[IEEE85] American National Standards Institute / Institute of Electrical and Electronical Engineers, *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std. 754-1985, New York, 1985

[Ker83] B. W. Kernighan / D. M. Ritchie, *Programmieren in C*, Carl Hanser Verlag 1983

[Kob92] L. Kobbelt, *Approximative Berechnung metrischer Eigenschaften*, Dipl.-Arb. 1992, Universität Karlsruhe, Fakultät für Informatik

[Kul81] U. Kulisch / W.L. Miranker, *Computer Arithmetic in Theory and Practice*, Academic Press, New York 1981

[Sto83] Josef Stoer, *Einführung in die Numerische Mathematik I*, Springer-Verlag 1983