Accurate floating-point summation: a new approach

A. Eisinberg, G. Fedele *

Dip. Elettronica Informatica e Sistemistica, Università degli Studi della Calabria, 87036 Rende (Cs), Italy

Abstract

The aim of this paper is to find an accurate and efficient algorithm for evaluating the summation of large sets of floating-point numbers. We present a new representation of the floating-point number system in which a number is represented as a linear combination of integers and the coefficients are powers of the base of the floating-point system. The approach allows to build up an accurate floating-point summation algorithm based on the fact that no rounding error occurs whenever two integer numbers are summed or a floating-point number is multiplied by powers of the base of the floating-point system. The proposed algorithm seems to be competitive in terms of computational effort and, under some assumptions, the computed sum is greatly accurate. With such assumptions, less-conservative in the practical applications, we prove that the relative error of the computed sum is bounded by the unit roundoff.

© 2006 Elsevier Inc. All rights reserved.

Keywords: Floating-point system; Summation algorithms

1. Introduction

Many problems, such as numerical integration, numerical solution of differential equations, inner products, means, variances, norms, and all kinds of nonlinear functions, involve computing sums with many terms. Because of this ubiquity and since summation is a basic task in numerical analysis, there are numerous algorithms for that. Higham [1] devotes an entire chapter to summation. Excellent review of various applications in many different areas of numerical analysis can be found in [1–4]. Two recent applications can be found in [5,6]. In [5] Priest’s distillation algorithm [4] is used in the problem of surface interrogation and intersection that depends crucially on good root-finding algorithms, which in turn depends on accurate polynomial evaluation and indirectly on accurate summations. In [6] Demmel and Hida present several simple algorithms for accurately computing the sum of \( n \) floating-point numbers using a wider accumulator and apply their algorithms in computing a robust geometric predicate which determines whether a point \( D \) is to the left or right of the oriented plane defined by the points \( A, B \) and \( C \). Throughout the paper we assume a floating-point arithmetic adhering to IEEE 754 floating-point standard [7]. We will use only the working precision for floating-point computations. If this working precision is IEEE 754 double precision, this corresponds to 53 bits.
precision including an implicit bit. The set of working precision floating-point numbers is denoted by $\mathbb{F}$, that is a subset of the real numbers defined as follows:

$$\mathbb{F} = \{ y \in \mathbb{R} | y = \pm \beta^e \sum_{k=1}^{t} d_k \beta^{-k} \}.$$  \hfill (1)

The system $\mathbb{F}$ is characterized by four integer parameters:

- the base $\beta \geq 2$,
- the precision $t \geq 1$,
- the exponent range $e_{\text{min}} \leq e \leq e_{\text{max}}$.

Each digit $d_k$ satisfies $0 \leq d_k \leq \beta - 1$, and $d_1 \neq 0$ for normalized numbers. An excellent tutorial on many aspects of floating-point arithmetic is given in [8]. We denote by $fl(\cdot)$ the result of a floating-point computation, where all operations inside the parentheses are executed in working precision. Floating-point operations according to IEEE 754 satisfy the following standard model [1]:

$$fl(x \text{ op } y) = (x \text{ op } y)(1 + \delta), \quad |\delta| \leq u \text{ op } = +, -, \ast, /,$$

where $u = \frac{1}{2} \beta^{1+t}$ is called unit roundoff.

1.1. Previous summation algorithms

We collect some notes on previous work to put our results into suitable perspective; our remarks are by no means complete. It is well known that, unless precautions are taken, the summation of large sets of numbers can be very inaccurate due to the accumulation of rounding errors. There are many summation algorithms which are aimed at reaching a compromise between accuracy of the computed sum and computational effort. Here we briefly describe some of them with the pseudo-codes and error bounds.

Let $s$ be the sum of $n$ floating-point numbers to be evaluated:

$$s = \sum_{i=1}^{n} x_i,$$ \hfill (3)

and let $\hat{s}$ be the computed sum.

1.1.1. Recursive summation

The most used algorithm is:

$$\hat{s} = 0$$

for $i = 1 : n$

$$\hat{s} = \hat{s} + x_i$$

end

It is well known that the ordering of addends plays a crucial role in the performance of this algorithm [9].

1.1.2. Pairwise summation

The algorithm uses a queue as data structure. At every step, the partial sum obtained as sum of the first two elements extracted is inserted into the queue. These elements are initially $x_i$'s and later are the computed partial sums. The related pseudo-code is:

make queue of $x$
while queue is not empty do

$\text{addend}_i \leftarrow$ move front queue of $x$

\[ addend_2 \leftarrow \text{move front queue of } x \]
\[ \hat{s} \leftarrow addend_1 + addend_2 \]
\[ \text{insert tail queue } (x, \hat{s}) \]
end while

The sum is obtained in \( \log_2 n \) structurally independent stages and so can be executed in parallel.

1.1.3. Insertion summation

The key idea of this approach is to encourage the addition between numbers of similar magnitude. It is based on an ordered tree summation:

build heap \((x)\)
while heap has more than 1 element do
\[ addend_1 \leftarrow \text{move minimum from heap } x \]
\[ addend_2 \leftarrow \text{move minimum from heap } x \]
\[ \hat{s} \leftarrow addend_1 + addend_2 \]
\[ \text{insertHeap} (x, \hat{s}) \]
rebuildHeap \((x)\)
end while

1.1.4. Error analysis

The three algorithms described in the previous subsections can be considered as a particular case of the following pseudo-code:

Let \( S = \{x_1, x_2, \ldots, x_n\} \)
repeat while \( S \) contains more than 1 element
\[ \text{remove two numbers } x \text{ and } y \text{ from } S \]
\[ \text{add their sum } x + y \text{ to } S \]
end
Assign the remaining element of \( S \) to \( \hat{s} \).

Note that there are \( n \) numbers to be added and consequently \( n - 1 \) additions to be performed. If \( T_i = x_{i_1} + y_{i_1} \) is the \( i \)-th execution of the repeat loop, then the computed sum is

\[ \hat{T}_i = \frac{x_{i_1} + y_{i_1}}{1 + \delta_i}, \quad |\delta_i| \leq u, \ i = 1, 2, \ldots, n - 1. \]  

The partial error is \( \delta_i \hat{T}_i \). The overall error is the sum of the partial errors:

\[ E_n = S_n - \hat{S}_n = \sum_{i=1}^{n-1} \delta_i \hat{T}_i. \]  

The error bound is therefore:

\[ |E_n| \leq u \sum_{i=1}^{n-1} |\hat{T}_i|. \]  

Since

\[ |\hat{T}_i| \leq \sum_{j=1}^{n} |x_j| + O(u), \quad \forall i, \]  

then (6) becomes

\[ |E_n| \leq u \sum_{i=1}^{n-1} \left( \sum_{j=1}^{n} |x_j| + O(u) \right), \]
that is,

$$|E_n| \leq (n - 1)u \sum_{i=1}^{n} |x_i| + O(u^2).$$

(9)

For pairwise summation, taking into account the particular structure and under the assumption that \( n = 2^r \), the bound for the relative error is improved:

$$|E_n| \leq \frac{\sum_{i=1}^{n} |x_i|}{\sum_{i=1}^{n} x_i} \frac{nu}{1 - nu}.$$  

(10)

1.1.5. Compensated summation

Floating-point summation is frequently improved by compensated summation. The compensated summation is a recursive summation with a correction term designed to reduce the rounding errors [10–15]. The compensated summation due to Kahan [12] uses a correction term \( e \) at every step of the recursive scheme:

\[
\begin{align*}
\hat{s} &= 0 \\
e &= 0 \\
&\text{for } i = 1 : n \\
&\quad \text{temp} = \hat{s} \\
&\quad y = x_i + e \\
&\quad \hat{s} = \text{temp} + y \\
&\quad e = (\text{temp} - \hat{s}) + y \\
&\end{align*}
\]

end

The error of the computed sum is as follows bounded:

$$|E_n| \leq (2u + O(nu^2)) \sum_{i=1}^{n} |x_i|. $$

(11)

Priest [4] derives an ingenious algorithm from Kahan by introducing two correction terms:

Sort the \( x_i \) so that \( |x_1| \geq |x_2| \geq \ldots \geq |x_n| \)

\[
\begin{align*}
s_1 &= x_1 \\
c_1 &= 0 \\
&\text{for } k = 2 : n \\
y_k &= c_{k-1} + x_k \\
u_k &= x_k - (y_k - c_{k-1}) \\
t_k &= y_k + s_{k-1} \\
v_k &= y_k - (t_k - s_{k-1}) \\
z_k &= u_k + v_k \\
s_k &= t_k + z_k \\
c_k &= z_k - (s_k - t_k) \\
&\end{align*}
\]

end

\[ \hat{s} = s_n \]

Priest shows that, under the assumption \( n \leq \beta^{l-3} \), the computed sum \( \hat{s} \) satisfies

$$|s - \hat{s}| \leq 2u|s|.$$  

(12)
1.2. Our goal

In this paper we present a fast and accurate algorithm for summation using only the working precision to compute the result. This means that all the operations are made without using wider accumulators and without extended precision. Moreover no ordering on the floating-point numbers to be summed is needed. Our method consists of two phases:

- Each number is mapped in an equivalent and new representation of the floating-point number system, namely $\mathbb{F}$. In $\mathbb{F}$ a number is represented as a linear combination of integer numbers belonged to $\mathbb{F}$, with coefficients that are powers of the base of the floating-point system.
- The numbers are summed taking into account that no error occurs whenever two integer numbers are summed or a number is multiplied by powers of the base of the floating-point system.

Moreover we give simple criteria to check the accuracy of the result and show that under assumption on the magnitude of the numbers involved, the computed sum is extremely accurate and its relative error is bounded by $u$. In our opinion this fact is extremely important because it improves the theoretical bound of Priest’s algorithm that has a bound of twice the unit roundoff. If the hypotheses are violated we show how to modify our approach.

1.3. Outline of the paper

The paper is organized as follows. In Section 2 we present the new representation of the floating-point system $\mathbb{F}$ and show the equivalence between $\mathbb{F}$ and $\mathbb{F}$. In next two sections we describe our summation algorithms and discuss their numerical properties, both by error analysis and by reporting numerical results, as well as timing, accuracy, comparison with other algorithms and some remarks on practical applications.

2. A new representation of the floating-point number system

The algorithms introduced in the previous section reach good results for the minimization of the relative error through arrangements and compensations that, however, do not modify the numbers to be summed. Every element is added, as extracted, to an accumulator or to a correction term. An alternative approach could be to modify in advance the elements in such a way to make the operations among the modified values as accurate as possible. The approach proposed in this paper, computes the sum of $n$ floating-point numbers as a linear combination among sums of integer values multiplied by coefficients that are powers of the base of the floating-point system. Every floating-point number $y \in \mathbb{F}$ can be represented as follows:

$$y = \text{sgn}(y) \sum_{j=1}^{q} y_j \beta^{-A(j)}$$

where:

$$y_j = \sum_{k=A(j-1)+1}^{A(j)} d_k \beta^{A(j)-k}, \quad j = 1, 2, \ldots, q, \quad (14)$$

$$A(j) = A(j-1) + x_j, \quad j = 1, 2, \ldots, q \quad (15)$$

$$x_j \quad \text{integer} \quad x_j \geq 1 \quad j = 1, 2, \ldots, q \quad (16)$$

$$\sum_{j=1}^{q} x_j = t$$

$$\text{sgn}(y) = \begin{cases} +1, & y > 0 \\ 0, & y = 0 \\ -1, & y < 0 \end{cases} \quad (17)$$

then $\mathbb{F}$ is the set of floating numbers defined by (13).
Formula (13) can be easily proved by inspection. It should be noted that \( y_j \) is integer, as it can be seen by (14). Next proposition shows how to compute the integer quantities \( y_j \).

**Proposition 1.** Let

\[
y = \pm \beta^e \sum_{k=1}^{t} d_k \beta^{-k}
\]

be a floating-point number, then \( y_j \) can be expressed as:

\[
y_j = \left[ \beta^{A(j)} y_n - \sum_{i=1}^{j-1} \beta^{A(j) - A(i)} y_i \right]
\]

where:

\[
y_n = \frac{y}{\pm \beta^e} = \sum_{k=1}^{t} d_k \beta^{-k}
\]

and \( \lfloor x \rfloor \) is the greatest integer less than or equal to \( x \).

**Proof.** It is sufficient to show that (19) and (14) are equivalent. By substituting (20) and (14) in (19) we have:

\[
y_j = \left[ \sum_{k=1}^{t} d_k \beta^{A(j) - k} - \sum_{i=1}^{j-1} \beta^{A(j) - A(i)} \sum_{k=A(i)+1}^{A(j)} d_k \beta^{-k} \right].
\]

By simple algebraic manipulations [17] we obtain:

\[
y_j = \left[ \sum_{k=1}^{t} d_k \beta^{j - k} - \sum_{k=1}^{A(j-1)} d_k \beta^{-k} \right] = \sum_{k=A(j)+1}^{t} d_k \beta^{A(j) - k} = \sum_{k=A(j)+1}^{t} d_k \beta^{A(j) - k}. \quad \Box
\]

Note that the value of \( e \) in (20) can be computed by using the result in the next proposition.

**Proposition 2.** Let \( y = \pm \beta^e \sum_{k=1}^{t} d_k \beta^{-k}, \ d_1 \neq 0 \) be a floating-point number, then

\[
e = \lfloor \log_{\beta}|y| \rfloor + 1.
\]

**Proof**

\[
\log_{\beta}|y| = e + \log_{\beta} \left| \sum_{k=1}^{t} d_k \beta^{-k} \right|.
\]

Since

\[
\log_{\beta} \frac{1}{\beta} \leq \log_{\beta} \left| \sum_{k=1}^{t} d_k \beta^{-k} \right| \leq \log_{\beta} (1 - \beta^{-t})
\]

and \( 1 - \beta^{-t} < 1 \), then

\[
-1 \leq \log_{\beta} \left| \sum_{k=1}^{t} d_k \beta^{-k} \right| < 0.
\]

By (24) one has \( \log_{\beta}|y| = e + \gamma \) with \(-1 \leq \gamma < 0\), therefore Eq. (23) holds. \( \Box \)

**Example 1.** Assuming the IEEE double standard \((t = 53, \beta = 2)\) we transform the floating-point number \( y = fl(\sqrt{2}) = 1.4142135623730951 \) in the new representation with \( q = 2 \). Without loss of generality we choose \( z_1 = 26 \) and \( z_2 = 27 \) and accordingly \( A(0) = 0, A(1) = 26 \) and \( A(2) = 53 \).
By Eq. (23) it follows that
\[ e = \lfloor \log_2 |y| \rfloor + 1 = 1, \]
therefore
\[ y_n = \frac{y}{2} = 0.7071067811865476. \]

Finally we compute integer quantities by Eq. (19):
\[ y_1 = 2^{4\{1\}} y_n = 47453132, \]
\[ y_2 = 2^{4\{2\}} y_n - 2^{4\{2\} - 4\{1\}} y_1 = 109001677. \]

Therefore, by Eq. (13):
\[ y = 47453132 \cdot 2^{-25} + 109001677 \cdot 2^{-52}. \]

From Eq. (19), it is easy to derive a recurrent property of \( y_j, j = 1, 2, \ldots, q \). Let \( s_j \) be the quantity defined as:
\[ s_j = \beta A\{j\} y_n - \sum_{i=1}^{j-1} \beta A\{i\} y_i, \quad j = 1, 2, \ldots, q, \quad (27) \]
then \( y_j = [s_j], j = 1, 2, \ldots, q \) and by simple manipulations
\[ s_{j+1} = \beta A\{j\+1\} (s_j - y_j), \quad j = 1, 2, \ldots, q - 1. \quad (28) \]

The following pseudo-code maps the generic floating-point number into the new representation (13), by using the IEEE double standard (\( t = 53, \beta = 2 \)):

```
function fp2nfp
Input:
  x = floating-point number
  q \geq 2 = number of splitting
Output:
  y = integer vector containing the \( y_j \) in (14)
  esp = integer vector containing the quantities \( esp(i) = e - A(i) \)
  \( x(i) = \lfloor x \rfloor, \ i = 1, 2, \ldots, q - 1 \)
  \( x(q) = t - \sum_{j=1}^{q-1} x(i) \)
  e = \lfloor \log_2(|x|) \rfloor + 1
  f = \frac{|x|}{2^e}
  aux = f
for \ i = 1 : q
  aux = aux \cdot 2^{x(i)}
  y(i) = \lfloor aux \rfloor
  aux = aux - y(i)
end
esp(i) = e - \sum_{k=1}^{i} x(k)
```

3. The proposed summation algorithm

The representation proposed in the previous section can be used to calculate the quantity:
\[ s = \sum_{i=1}^{n} x_i, \quad (29) \]
where each element is expressed as

\[ x_i = \text{sgn}(x_i)\beta^{e(i)} \sum_{j=1}^{A(i)} y_j(i)\beta^{-A(j)} \]  

(30)

with

\[ y_j(i) = \sum_{k=A(j-1)+1}^{A(j)} a_k^{(i)}\beta^{A(j)-k}. \]

(31)

Let

\[ \text{esp}_\text{min} = \min\{e(i) | i = 1, 2, \ldots, n\}, \]
\[ \text{esp}_\text{max} = \max\{e(i) | i = 1, 2, \ldots, n\}, \]

(32)

be the minimum and the maximum, respectively, of the addend exponents, then each exponent can be expressed as:

\[ e(i) = \text{esp}_\text{min} + \gamma(i), \quad i = 1, 2, \ldots, n \]

(33)

where

\[ \gamma(i) \geq 0, \quad i = 1, 2, \ldots, n. \]

(34)

So

\[ x_i = \text{sgn}(x_i)\beta^{\text{esp}_\text{min} + \gamma(i)} \sum_{j=1}^{q} y_j(i)\beta^{-A(j)} \]

(35)

and

\[ s = \sum_{j=1}^{q} f_j \]

(36)

where

\[ f_j = \beta^{\text{esp}_\text{min} - A(j)} \sum_{i=1}^{n} \text{sgn}(x_i)y_j(i)\beta^{\gamma(i)}. \]

(37)

The Matlab-code in Table 1 implements the summation algorithm in IEEE double standard \((t = 53, \beta = 2)\):

3.1. Error analysis

Note that the rounding error involved in (36) is bounded by

\[ \frac{\sum_{j=1}^{q} |f_j|}{\sum_{j=1}^{q} f_j} (q-1)\epsilon, \]

(38)

because the sum of \(n\) addends is reduced to the sum of \(q\) floating-point numbers. Obviously this theoretical bound is guaranteed only if in (36) the sum of the integer quantities

\[ \sum_{i=1}^{n} \text{sgn}(x_i)y_j(i)\beta^{\gamma(i)} \]

(39)

does not exceed the maximum integer number which can be expressed in a \(t\)-digit register:

\[ \text{int}_\text{max} = \beta^t - 1. \]

(40)

This assumption limits the number of elements to be summed up. In fact \(y_j\) is bounded by

\[ |y_j| \text{max} = \beta^{\gamma} - 1, \]

(41)
while $\gamma(i)$, in the worst case, takes the value:

$$\gamma_{\max} = \esp_{\max} - \esp_{\min}. \quad (42)$$

The following inequality must hold:

$$nf^{\esp_{\max} - \esp_{\min}}[y_j]_{\max} \leq \text{int}_{\max}, \quad j = 1, 2, \ldots, q, \quad (43)$$

therefore:

$$n \leq \frac{1}{\beta^{\esp_{\max} - \esp_{\min}}} \min \left\{ \frac{\beta' - 1}{\beta'^2 - 1}, \frac{\beta' - 1}{\beta'^2 - 1}, \ldots, \frac{\beta' - 1}{\beta'^2 - 1} \right\}. \quad (44)$$

### 3.2. Optimal choice $q = 2$

The choice of $q = 2$ in (36) allow us to build a summation algorithm with an optimal relative error bound. In this case the (36) becomes:

$$s = \beta^{\esp_{\min} - A(1)} \sum_{i=1}^{n} \text{sgn}(x_i)y_1(i)\beta'^{(i)} + \beta^{\esp_{\min} - A(2)} \sum_{i=1}^{n} \text{sgn}(x_i)y_2(i)\beta'^{(i)}. \quad (45)$$

Such an expression contains the sum of two floating-point numbers. If these numbers are correctly formed, then by Eq. (2) their sum is bounded by $u$.

Formula (45) is error-free only if

$$n \leq \frac{\beta' - 1}{\beta^{\esp_{\max} - \esp_{\min}}(\beta'^2 - 1)}. \quad (46)$$

In IEEE double standard ($t = 53, \beta = 2$), by choosing for example $x_1 = 26$ and $x_2 = 27$, the (46) is satisfied if

$$n \leq \frac{\beta' - 1}{\beta^{\esp_{\max} - \esp_{\min}}(\beta'^2 - 1)}. \quad (47)$$
therefore, in order to have \( n \geq 1 \) it must be
\[
\text{esp}_{\text{max}} - \text{esp}_{\text{min}} \leq 26. \tag{48}
\]
If the (48) holds, but the (47) is not satisfied, it follows that the integer quantities involved in (45):
\[
f_1(i) = \text{sgn}(x_i)y_1(i)b^{(i)}, \quad i = 1, 2, \ldots, n \tag{49}
\]
and
\[
f_2(i) = \text{sgn}(x_i)y_2(i)b^{(i)}, \quad i = 1, 2, \ldots, n \tag{50}
\]
are computable exactly, but their sum may still exceed the maximum representable integer number (40).
As \( f_1 \) and \( f_2 \) are integer quantities, their sum can be expressed in terms of modulo \( 2^{53} \) and remainder:
\[
\sum_{i=1}^{n} f_1(i) = A_1 2^{53} + R_1, \tag{51}
\]
\[
\sum_{i=1}^{n} f_2(i) = A_2 2^{53} + R_2. \tag{52}
\]
Therefore the (45) becomes:
\[
s = A_1 2^{\text{esp}_{\text{min}}-A(1)+53} + R_1 2^{\text{esp}_{\text{min}}-A(1)} + A_2 2^{\text{esp}_{\text{min}}-A(2)+53} + R_2 2^{\text{esp}_{\text{min}}-A(2)}. \tag{53}
\]
The four elements in (53) are generated without errors because the (48) holds. Their sum can be done by using Priest’s algorithm which guarantees a theoretical bound of \( 2n \) and, in this case, there is a drastic reduction of time used due to the fact that it must sum only four floating-point numbers. The algorithm is summarized by the following pseudo-code:

function cutSum
Input:
\( v \) = vector of dimension \( n \geq n_{\text{max}} \) with \( n_{\text{max}} \) equals to the r.h.s. of (47)
Output:
computed sum
Calculate \( f_1 \) and \( f_2 \) as in (49) and (50), respectively.
Build \( k_1 \) sub-vectors of \( f_1 \), \( \{w_{1,1}, w_{1,2}, \ldots, w_{1,k_1}\} \) such as \( \dim(w_{1,j}) \leq n_{\text{max}}, j = 1, 2, \ldots, k_1 \).
Build \( k_2 \) sub-vectors of \( f_2 \), \( \{w_{2,1}, w_{2,2}, \ldots, w_{2,k_2}\} \) such as \( \dim(w_{2,j}) \leq n_{\text{max}}, j = 1, 2, \ldots, k_2 \).
Build a vector \( p_1 \in \mathbb{N}^{k_1} \) such as \( p_1(i) = \sum_{j=1}^{\dim(w_{1,i})} w_{1,j}(j), i = 1, 2, \ldots, k_1 \).
Build a vector \( p_2 \in \mathbb{N}^{k_2} \) such as \( p_2(i) = \sum_{j=1}^{\dim(w_{2,i})} w_{2,j}(j), i = 1, 2, \ldots, k_2 \).
Calculate \( A_1, A_2, R_1 \) and \( R_2 \) such as \( \sum_{i=1}^{k_1} p_1(i) = A_1 2^{53} + R_1 \sum_{i=1}^{k_2} p_2(i) = A_2 2^{53} + R_2 \).
Calculate the (53) by using Priest’s algorithm.

Moreover if the (49) is not satisfied, then it is possible to split the original vector in a set of vectors in which the (49) is true. In the worst case, not common in practice, the number of vectors is equal to the number of elements to be summed. Then, for each vector the partial sum can be computed by the algorithm in Table 1, and the whole sum is computed by summing up the partial sums by a compensated algorithm. The Matlab-code in Table 2 implements such idea. Note that the algorithm can be improved by sorting the elements such that \( |x_i| \leq |x_{i+1}|, \quad i = 1, 2, \ldots, n - 1 \).

3.3. Numerical experiments and computational aspects

In this section we present timing and accuracy results. We compare our summation algorithms (EF1, EF2) to recursive summation (RI), Kahan (KA) and Priest’s algorithm (PR). All the above algorithms were per-
formed on a Pentium-based computer in which the unit roundoff is $u = 2^{-53} \approx 1.11 \times 10^{-16}$ and they have been implemented in Matlab [18] and Mathematica [19] which allows arbitrary precision numbers, in order to compare the computed sums with the “exact” ones by using 1024 digit precision numbers and to investigate the relative error. We considered the algorithms RI, KA, PR and EF1 (with $q = 2$) from a computational point of view by computing the mean time necessary to perform a fixed number of iterations. The mean time was computed by using two Matlab built-in functions: tic-toc. The first reset the clock, the latter returns the elapsed time since the latest reset. The obtained results are presented in Table 3. If the assumptions (47) and (48) are satisfied then the relative error bound of our algorithm is $u$. In such case we perform an experiment in order to estimate numerically the probability density function of the relative error (PDFRE) for the algorithm EF1 (with $q = 2$): we generate 1500000 vectors of 10 numbers uniformly distributed in the interval $[0.5; 1]$. Fig. 1 shows the PDFRE normalized to the unit roundoff. We note that this distribution is a bimodal one. In the other cases we compare summation methods through statistical estimates of error, which may be more representative of the average case [15]. We consider the following sets of data for the algorithm EF1:

- Set 1: the $x_i$ are random numbers uniformly distributed in the interval $[-1, 1]$, for $n = 4096$.
- Set 2: the $x_i$ are random numbers uniformly distributed in the interval $[0, 1]$, for $n = 4096$.

### Table 2
Matlab algorithm sumEF2 for accurate summation

```matlab
function s = sumEF2(v)
    n = length(v);
    [f, e] = log2(v);
    eMax = max(e);
    eMin = min(e);
    if (eMax - eMin) < 26
        s = sumEF1(v, 2);
    else
        aux = v;
        k = 1;
        while length(aux) > 0
            pos = find(abs(e - e(1)) < 26);
            sp(k) = sumEF1(aux(pos), 2);
            aux(pos) = [ ];
            e(pos) = [ ];
            k = k + 1;
        end
        s = priest(sp);
    end
end
```

### Table 3
Mean times in computing 50000 sums

<table>
<thead>
<tr>
<th>$n$</th>
<th>RI</th>
<th>KA</th>
<th>PR</th>
<th>EF1</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>2.70–04</td>
<td>4.27–04</td>
<td>1.44–03</td>
<td>4.29–04</td>
</tr>
<tr>
<td>20</td>
<td>3.50–04</td>
<td>6.68–04</td>
<td>2.56–03</td>
<td>5.06–04</td>
</tr>
<tr>
<td>30</td>
<td>4.09–04</td>
<td>8.64–04</td>
<td>3.64–03</td>
<td>5.34–04</td>
</tr>
<tr>
<td>40</td>
<td>4.76–04</td>
<td>1.06–03</td>
<td>4.72–03</td>
<td>5.69–04</td>
</tr>
<tr>
<td>50</td>
<td>5.81–04</td>
<td>1.28–03</td>
<td>5.80–03</td>
<td>5.94–04</td>
</tr>
<tr>
<td>60</td>
<td>6.27–04</td>
<td>1.48–03</td>
<td>7.11–03</td>
<td>7.23–04</td>
</tr>
<tr>
<td>70</td>
<td>7.17–04</td>
<td>1.70–03</td>
<td>7.98–03</td>
<td>7.52–04</td>
</tr>
<tr>
<td>80</td>
<td>7.82–04</td>
<td>1.89–03</td>
<td>9.08–03</td>
<td>7.79–04</td>
</tr>
<tr>
<td>90</td>
<td>8.44–04</td>
<td>2.11–03</td>
<td>1.02–02</td>
<td>8.20–04</td>
</tr>
<tr>
<td>100</td>
<td>9.50–04</td>
<td>2.37–02</td>
<td>1.13–02</td>
<td>8.54–04</td>
</tr>
</tbody>
</table>
and

• Set 3: The $x_i = \pm 10^p$, where the $p_i$ are uniformly distributed in the interval $[0, 35]$.
• Set 4: $x_1 = x_2 = \cdots = x_{2047} = 1.0, x_{2048} = 1.0e - 18, x_{2050} = x_{2051} = \cdots = x_{4096} = -1.0$.
• Set 5: $x_i = 1/i^2$, for $n = 4096$.

for the algorithm EF2.

The summations were performed 10000 times each (or 1 time for data sets 4–5). The results for data sets 1–3 are shown in Tables 4–6, respectively. For each method we report the maximum and mean value of the relative errors in terms of unit roundoff. We also report the fraction of trials in which each algorithm gives equal or more accurate result than the others. The matrix in the bottom of Tables 4–6 must be read as follows: in each line is reported the probability of success of the corresponding algorithm in comparison to the other ones. In Table 7 we report the relative errors in terms of unit roundoff for data sets 4–5.

### Table 4
**Set 1**

<table>
<thead>
<tr>
<th></th>
<th>RI</th>
<th>KA</th>
<th>PR</th>
<th>EF1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max</td>
<td>140023</td>
<td>937.42</td>
<td>0.9909</td>
<td>1.0923</td>
</tr>
<tr>
<td>Mean</td>
<td>72.0336</td>
<td>1.65733</td>
<td>0.360019</td>
<td>0.360156</td>
</tr>
<tr>
<td>RI</td>
<td>0</td>
<td>0.0350</td>
<td>0.0257</td>
<td>0.0257</td>
</tr>
<tr>
<td>KA</td>
<td>0.9912</td>
<td>0</td>
<td>0.6644</td>
<td>0.6645</td>
</tr>
<tr>
<td>PR</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>EF1</td>
<td>1</td>
<td>0.9998</td>
<td>0.9998</td>
<td>0</td>
</tr>
</tbody>
</table>

### Table 5
**Set 2**

<table>
<thead>
<tr>
<th></th>
<th>RI</th>
<th>KA</th>
<th>PR</th>
<th>EF1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max</td>
<td>56.0788</td>
<td>1.00823</td>
<td>0.998056</td>
<td>1.99072</td>
</tr>
<tr>
<td>Mean</td>
<td>11.1005</td>
<td>0.372495</td>
<td>0.372467</td>
<td>0.381584</td>
</tr>
<tr>
<td>RI</td>
<td>0</td>
<td>0.0445</td>
<td>0.0444</td>
<td>0.0449</td>
</tr>
<tr>
<td>KA</td>
<td>0.9999</td>
<td>0</td>
<td>0.9968</td>
<td>0.9969</td>
</tr>
<tr>
<td>PR</td>
<td>1</td>
<td>0.9999</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>EF1</td>
<td>0.9997</td>
<td>0.9874</td>
<td>0.9874</td>
<td>0</td>
</tr>
</tbody>
</table>
4. Accurate interpolation on equidistant nodes in $[0, 1]$

We will not say much about the many practical applications of our algorithm because the applications of accurate summation are widely known and there are excellent treatments in the recent literature. We want only to present an application of the new representation of the floating-point system related to the problem of the polynomial interpolation on equidistant nodes in $[0, 1]$ [16]. It can be shown that the inverse of the Vandermonde matrix on equidistant nodes in $[0, 1]$ can be factorized as a product of four matrices with integer entries that can be stored without rounding errors:

$$ W_n = \frac{1}{(n - 1)!} D_1 U D_2 L, $$  \hfill (54)

where:

$$ D_1 = \text{diag}\{(n - 1)^{i-1}\}_{i=1,2,\ldots,n}, $$

$$ U(i, j) = (-1)^{j+1} \binom{j-1}{i-1}, \quad i = 1, 2, \ldots, n, \quad j = i, i + 1, \ldots, n, $$

$$ D_2 = \text{diag}\{\frac{(n - 1)!}{(i - 1)!}\}_{i=1,2,\ldots,n}, $$

$$ L(i, j) = (-1)^{j+1} \binom{i-1}{j-1}, \quad i = 1, 2, \ldots, n, \quad j = 1, 2, \ldots, i. $$

Therefore the polynomial which interpolates the function $f(x)$ on the set of nodes $X_n = \{\frac{k-1}{n-1}, \ k = 1, 2, \ldots, n\}$ is:

$$ p(x) = \sum_{k=1}^{n} c_k x^{k-1}, $$ \hfill (55)

where the coefficient vector $c$ can be calculated as:

$$ c = \frac{1}{(n - 1)!} D_1 U D_2 L \cdot f $$ \hfill (56)

and $f_k = f\left(\frac{k-1}{n-1}\right), \ k = 1, 2, \ldots, n.$

By using the integer properties of the matrices involved in (54) and the fact that each entry of the vector $f$ can be represented as a linear combination of integers where the coefficients are powers of the base, we build an accurate algorithm for the interpolation problem on equidistant nodes in $[0, 1]$. We consider the quantity
and, by using (13) we write the vector as

\[
\begin{bmatrix}
  f_1 \\
  f_2 \\
  f_3 \\
  \vdots \\
  f_n
\end{bmatrix} = Y \cdot e,
\]

where

\[
Y = \begin{bmatrix}
  y_1(1) & y_2(1) & y_3(1) \\
  y_1(2) & y_2(2) & y_3(2) \\
  y_1(3) & y_2(3) & y_3(3) \\
  \vdots & \vdots & \vdots \\
  y_1(n) & y_2(n) & y_3(n)
\end{bmatrix}
\]

and

\[
e = \begin{bmatrix}
  2^{\text{esp}(1)} \\
  2^{\text{esp}(2)} \\
  2^{\text{esp}(3)}
\end{bmatrix}.
\]

Note that \( y_q(k), q = 1, 2, 3; k = 1, 2, \ldots, n \) are integer quantities multiplied by power of the base of the floating-point number system. Then

\[
c = \frac{1}{(n - 1)!} D_1 U D_2 L f e.
\]

The numerical experiments are aimed in showing the high accuracy of formula (57). We have used package Mathematica to compute the approximate solutions \( \hat{c} \), the exact ones (using extended precision of 1024 significant digits) and the error

\[
\epsilon_c = \max_{1 \leq i \leq n} \frac{|\hat{c}_i - c_i|}{|c_i|},
\]

100000 experiments have been run for \( n = 2, 3, \ldots, 10 \). Table 8 reports the maximum and mean value of \( \epsilon_c \) in terms of unit roundoff \( u = 2^{-53} \).

5. Conclusions

We have compared our method with Priest’s algorithm. In our opinion integer based approach is conceptually more simple because it avoids the error compensation logic and uses only some elementary properties of the floating-point system [8]. This implies a drastic reduction in the number of floating-point operations.
Moreover we have shown that, under less-conservative hypothesis, the theoretical relative error is bounded by the unit-roundoff.

References