Nr.: FIN-002-2010

Engineering an Exact Sign of Sum Algorithm

Marc Mörig, Stefan Schirra

Arbeitsgruppe Algorithmische Geometrie

Technical report

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Nr.: FIN-002-2010

Engineering an Exact Sign of Sum Algorithm

Marc Mörig, Stefan Schirra

Arbeitsgruppe Algorithmische Geometrie

Technical report (Internet)
Elektronische Zeitschriftenreihe
der Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg
ISSN 1869-5078

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Impressum (§ 5 TMG)

Herausgeber:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Der Dekan

Verantwortlich für diese Ausgabe:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Marc Mörig
Postfach 4120
39016 Magdeburg
E-Mail: moerig@isg.cs.uni-magdeburg.de

http://www.cs.uni-magdeburg.de/Technical_reports.html

Technical report (Internet)
ISSN 1869-5078

Redaktionsschluss: 15.02.2010

Bezug: Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Dekanat

Engineering an Exact Sign of Sum Algorithm *

Marc Mörig Stefan Schirra

Department of Simulation and Graphics,
Otto-von-Guericke University Magdeburg, Germany,
moerig@isg.cs.uni-magdeburg.de

August 19, 2009

Abstract

Straightforward summation of floating-point values is subject to rounding errors and often produces an approximation $\tilde{s}$ whose sign differs from the sign of the actual value s, if s is close to zero. The Exact Sign of Sum Algorithm (ESSA) presented by Ratschek and Rokne allows one to compute the actual sign of a sum of floating-point values with floating-point arithmetic. We present and discuss variations of Ratschek and Rokne's original versions of ESSA. For one of our new variants we can prove an upper bound on the number of required iterations which is smaller than Ratschek and Rokne's bound on the number of iterations for the original ESSA. We compare several variants of ESSA experimentally. The paper contains all the relevant code. Thereby we ensure reproducibility of our experiments.

1 Introduction

About decade ago, Ratschek and Rokne [14] presented a nice algorithm to compute the sign of a sum of floating-point values exactly. They call their algorithm ESSA for Exact Sign of Sum Algorithm. Gavrilova et al. [4] propose some modifications to ESSA and evaluate them experimentally. Sample C-code for ESSA is presented in [14] and [16] and available on Rokne's web pages [9].
Ratschek, Rokne and Gavrilova also discuss applications of their methods in computational geometry [14, 16]. Suggested applications include testing for and computing line segment intersections [6, 7], 2D convex hull [15], and 2D Delaunay Triangulation and Voronoi diagram [5].
In this report we present several variants of ESSA and compare them experimentally. We compare our ESSA variants for randomly generated sums. In a recent paper [11] we compared the performance of ESSA and other exact sign of sum algorithms in the context of computing 2D Delaunay triangulations. For one of our new variants we can prove an upper bound on the number of required iterations which is smaller than Ratschek and Rokne's bound on the number of iterations for the original ESSA.
We use the literate programming paradigm [2, 8] in order to document the internals of our implementations. The source code is organized into small sections that are presented whenever most appropriate, together with a description of underlying algorithmic ideas, critical technical knowledge and unusual coding constructions, cf. [10].

2 Floating-point arithmetic preliminaries

Throughout the report we assume a binary floating-point arithmetic conforming to the IEEE 754 standard [1, 13]. We will first give a definition of floating-point numbers. Let $0 \neq x \in \mathbb{R}$, then

$$\mathrm{msb}(x) = 2^{\lfloor \log_2 |x| \rfloor}$$

*Supported by DFG grant SCHI 858/1-1

gives the value of the most significant non-zero bit in a binary representation of x. If $0 \neq x \in \mathbb{R}$ furthermore has a finite binary representation, then we define

$$\operatorname{lsb}(x) = \max\{\sigma \ : \ x \in \sigma\mathbb{Z},\ \sigma = 2^k,\ k \in \mathbb{Z}\}.$$

By definition, $\operatorname{lsb}(x)$ gives the value of the least significant non-zero bit in the binary representation of x. Floating-point numbers are a subset of $\mathbb{R}$ with a finite binary representation. We denote the smallest (largest) power of two that can be represented in a floating-point format by $f_{\min}$ ($f_{\max}$). Furthermore a floating-point format can store at most t consecutive bits. We denote by $\varepsilon_{\mathtt{m}}$ the quantity 2^{-t}.

Definition 1.
Denote the set of floating-point numbers by $\mathbb{F} = \mathbb{F}(t, f_{\min}, f_{\max})$. $\mathbb{F}$ contains 0. Let $0 \neq x \in \mathbb{R}$ have a finite binary representation, then $x \in \mathbb{F}$ if and only if

$$f_{\min} \leq \operatorname{lsb}(x), \qquad \operatorname{msb}(x) \leq \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1} \operatorname{lsb}(x), \qquad \operatorname{msb}(x) \leq f_{\max}. \qquad \square$$

An important example is IEEE *double precision* with $t = 53$, $f_{\min} = 2^{-1074}$, $f_{\max} = 2^{1024}$ and $\varepsilon_{\mathtt{m}} = 2^{-53}$. Let $0 \neq x \in \mathbb{R}$, have a finite binary representation, then the *exponent* E and *mantissa* m of x are given by $E = \lfloor \log_2 |x| \rfloor$ and $m = x/E$. For a floating-point number u this exponent and mantissa do not necessarily coincide with what is actually stored to represent u. Most floating-point numbers have a mantissa of t bits, but if $\operatorname{msb}(u)$ is smaller than $\frac{1}{2}\varepsilon_{\mathtt{m}}^{-1} f_{\min}$, less than t bits are available. Floating-point numbers with $\operatorname{msb}(u) < \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1} f_{\min}$ are called *denormalized*.
After having fixed the set of numbers we need to discuss mathematical operations. With $\oplus, \ominus, \odot$ we denote the floating-point operations corresponding to $+, -, \cdot$. IEEE 754 arithmetic is rounding *faithfully*, i.e., the result of a floating-point operation is the mathematically exact result if it is itself a floating-point number and one of the two neighboring floating-point numbers otherwise. When a number is not between two floating-point numbers, it may also be rounded to $\pm\infty$. Which of those numbers is taken is determined by the *rounding mode*. Let $\operatorname{fl}(.)$ denote the operation that rounds a real number to $\mathbb{F} \cup \{\pm\infty\}$. Then for $x, y \in \mathbb{R}$:

$$x \leq y \qquad \Rightarrow \qquad \operatorname{fl}(x) \leq \operatorname{fl}(y) \tag{1}$$

$$\operatorname{fl}(x) < \operatorname{fl}(y) \qquad \Rightarrow \qquad x < y \tag{2}$$

Here Property (2) follows from Property (1), which is clear for faithful rounding. *Overflow* is said to occur in $\mathbb{F}$, when x is rounded to a different number in the modified floating-point set $\mathbb{F}'$ with $f_{\max} > \operatorname{msb}(x)$. Overflow does not necessarily coincide with rounding x to $\pm\infty$, cf. Overton [13]. Assuming no overflow occurs we have

$$\operatorname{fl}(x) = x(1+\delta) + \eta \qquad \text{with } |\delta| < 2\varepsilon_{\mathtt{m}},\ |\eta| < f_{\min},\ \delta\eta = 0 \tag{3}$$

For this Property we refer to the book by Overton [13]. When rounding x to $\operatorname{fl}(x)$ we either have t bits available, then δ accounts for the rounding error, but it may also happen, that $\operatorname{msb}(x) < \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1} f_{\min}$ in which case $f_{\min}$ is the smallest bit that can be stored. Then η accounts for the rounding error. Unless the rounding mode is explicitly mentioned we only assume faithful rounding. When the rounding mode is *round-to-nearest*, we can replace $2\varepsilon_{\mathtt{m}}$ by $\varepsilon_{\mathtt{m}}$ in Property (3).
We collect some more properties. When we use a statement of the form "$x \in \sigma\mathbb{Z}$", σ is always a power of two. Such a statement tells us that $x = 0$ or $\sigma \leq \operatorname{lsb}(x)$.

$$\mathbb{F} \qquad \subset \qquad f_{\min}\mathbb{Z} \tag{4}$$

$$\sigma_1 \geq \sigma_2 \qquad \Rightarrow \qquad \sigma_1\mathbb{Z} \subseteq \sigma_2\mathbb{Z} \tag{5}$$

$$x, y \in \sigma\mathbb{Z} \qquad \Rightarrow \qquad x + y \in \sigma\mathbb{Z} \tag{6}$$

$$\left.\begin{array}{r} u, v \in \sigma\mathbb{Z} \cap \mathbb{F} \\ u \oplus v \text{ does not round to } \pm\infty \end{array}\right\} \qquad \Rightarrow \qquad u \oplus v \in \sigma\mathbb{Z} \tag{7}$$

Property (4) and Property (5) are obvious. Property (6) is simply a ring property of $\sigma\mathbb{Z}$ and we can derive Property (7) from Property (5) and Property (6) because in the process of rounding $u+v$ to $u \oplus v$, trailing bits are removed, i.e., either $u+v=0$ or $f_{\min} \leq \sigma \leq \operatorname{lsb}(u+v) \leq \operatorname{lsb}(u \oplus v)$. Property (7) allows to keep track of the lsb of floating-point numbers. To demonstrate the usefulness of this notation and for later use we will now show that addition and subtraction are exact in case the result falls in the range of denormalized numbers.

Lemma 2.
Let $u, v \in \mathbb{F}$ with $\operatorname{msb}(u+v) \leq \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1} f_{\min}$. Then $u+v \in \mathbb{F}$ and hence $u \oplus v = u+v$.

Proof. We know $u, v \in \mathbb{F}$ and hence from Property (4) and Property (6) that $u+v \in f_{\min}\mathbb{Z}$. If $u+v=0$ the claim holds. Otherwise we have $f_{\min} \leq \operatorname{lsb}(u+v)$ and

$$\operatorname{msb}(u+v) \;\leq\; \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1} f_{\min} \;\leq\; \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1} \operatorname{lsb}(u+v) \;\leq\; f_{\max}.$$

From Definition 1 then follows $u+v \in \mathbb{F}$. □

From this result directly follows an improved version of Property (3) for addition and subtraction. If no overflow occurs

$$u \oplus v = (u+v)(1+\delta) \qquad \text{with } |\delta| < 2\varepsilon_{\mathtt{m}} \tag{8}$$

where $2\varepsilon_{\mathtt{m}}$ can be replaced by $\varepsilon_{\mathtt{m}}$ in round-to-nearest. Furthermore Lemma 2 shows that no nonzero number is ever rounded to zero in an addition or subtraction. Together with Property (1) we have for $u, v \in \mathbb{F}$

$$\operatorname{sign}(u+v) = \operatorname{sign}(u \oplus v) \tag{9}$$

even in the case of overflow.
We want to compute the sign of a sum of floating-point numbers exactly, yet rounding error is inherent to floating-point computations. Sterbenz Lemma [13, 18] gives a sufficient condition when a floating-point subtraction is free from rounding error.

Lemma 3 (Sterbenz)**.**
Let $u, v \in \mathbb{F}$ with $\frac{1}{2} \leq \frac{u}{v} \leq 2$. Then $u-v \in \mathbb{F}$ and consequently $u \ominus v = u-v$.

Proof. Note that u and v have the same sign. Assumption and conclusion do not change when we switch the roles of u and v or when we replace u and v by $-u$ and $-v$, so we can assume $0 < \frac{1}{2}u \leq v \leq u$. By Properties (5, 6) $f_{\min} \leq \min\{\operatorname{lsb}(u), \operatorname{lsb}(v)\} \leq \operatorname{lsb}(u-v)$. Furthermore $u-v \leq u - \frac{1}{2}u = \frac{1}{2}u \leq v \leq u$ and thus

$$\operatorname{msb}(u-v) \leq \min\{\operatorname{msb}(u), \operatorname{msb}(v)\} \leq \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1} \min\{\operatorname{lsb}(u), \operatorname{lsb}(v)\} \leq \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1} \operatorname{lsb}(u-v).$$

Finally $\operatorname{msb}(u-v) \leq \operatorname{msb}(u) \leq f_{\max}$ and therefore $u-v \in \mathbb{F}$ by Definition 1. □

The name *error-free transformation* has been given to small algorithms, that let us rewrite expression of floating-point numbers into mathematically equivalent expression [12]. Sterbenz Lemma is a key tool in proving the correctness of error-free transformations.

3 Engineering ESSA

Let two sequences of positive floating-point numbers $a = a_1, \ldots, a_m$ and $b = b_1, \ldots, b_n$ be given. ESSA allows to compute the sign of

$$s = \sum_{i=1}^{m} a_i - \sum_{j=1}^{n} b_j.$$

We will call a the positive summands and b the negative summands. The total number of summands is denoted by $l = m + n$ and may not exceed $\frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}$. We further assume that at any time a_1 is the largest positive summand and b_1 the largest negative summand. Using an error-free transformation ESSA iteratively computes two new summands x and y with $a_1 - b_1 = x + y$, but $|x| + |y| < a_1 + b_1$. The old summands a_1 and b_1 are removed from a and b and $|x|$ and $|y|$ are inserted into a and b, according to the sign of x and y respectively. This step is iterated until the sum vanishes, or the positive summands dominate the negative ones, or vice versa. The overall number of summands never increases:

Lemma 4. *Let $l_0 \leq \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}$ be the sum of the initial values of m and n. At any stage of the algorithm we have $m + n = l \leq l_0$.* □

We will now present several variants of ESSA. They all follow the scheme above but use different different error-free transformations to compute x and y.

3.1 Revised ESSA

We first present our revised version of ESSA. Here is the overall structure of the implementation. Both a and b must provide enough space for up to l elements, cf. Lemma 4. The summands must be stored in positions $1, \ldots, m$ and $1, \ldots, n$. In the main loop, first the termination criteria is checked. The computation of x and y and the update of a and b occurs in the loop body. In Section 3.4 we will wrap this and other ESSA implementations into a `struct` so we may use it as a template parameter later.

⟨*revised essa 4.1*⟩≡
```
int revised_essa::
sign_of_sum(double *const a, double *const b, int m, int n)const{
  ⟨build heap 4.2⟩
  while (true){
    ⟨termination criteria 4.3⟩
    ⟨revised essa loop body 7.1⟩
  }
}
```

To assure that a_1 and b_1 are the largest positive and negative summand, we maintain a and b as a heap. See Section 3.6 for our implementation of the heap operations. This is in contrast to the sample implementation [9], where the summands are sorted initially and then maintained as a heap.

⟨*build heap 4.2*⟩≡
```
build_heap(a,m);
build_heap(b,n);
```

In the termination criteria it is checked whether one of the sets dominates the other one. While in the original version [14] the termination criteria of ESSA uses exponent extraction, the sample code [9] uses floating-point multiplication, as we do below. Note that the test if m is zero assures that a_1 is defined in the third line. If n is zero, then it does not matter that b_1 is not defined in the third line. If the algorithm proceeds to the last line, however, a_1 and b_1 are defined.

⟨*termination criteria 4.3*⟩≡
```
if ( ( m == 0 )&&( n == 0 ) ) return  0;
if ( m==0 )                   return -1;
if ( a[1] > n*b[1] )          return  1;
if ( b[1] > m*a[1] )          return -1;
```

Lemma 5.
Consider the termination criteria. If $l = m + n \leq 1$ the algorithm terminates immediately. If the algorithm terminates, the returned result is correct. Let $\alpha = \max\{a_1, b_1\}$ and $\beta = \min\{a_1, b_1\}$. If the algorithm does not return, then $\alpha < l \cdot \beta$.

Proof. Assume the algorithm returns in the third line: $a_1 > n \odot b_1$ implies by Property (2), that $a_1 > n \cdot b_1$. From there it follows that

$$s = \sum_{i=1}^{m} a_i - \sum_{j=1}^{n} b_j \geq a_1 - n \cdot b_1 > 0$$

and the correct sign is returned. Analogously for line four. If ESSA does not return we know from line three and by Property (3), ignoring overflow, that

$$a_1 \leq n \odot b_1 = (1+\delta)n \cdot b_1 + \eta \qquad |\delta| < 2\varepsilon_{\mathtt{m}}, \quad |\eta| < f_{\min}, \quad \delta\eta = 0.$$

We have $n \leq l - 1$ since $m > 0$ and $l \leq \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}$ by Lemma 4, so we can derive

$$\begin{array}{llllll} a_1 & < (1+2\varepsilon_{\mathtt{m}})n \cdot b_1 & \leq (1+2\varepsilon_{\mathtt{m}})(l-1)b_1 & < l \cdot b_1 & \text{for } \eta = 0 \\ a_1 & < n \cdot b_1 + f_{\min} & \leq (l-1) \cdot b_1 + b_1 & = l \cdot b_1 & \text{for } \delta = 0. \end{array}$$

If an overflow occured in the product $n \odot b_1$ we still have $a_1 < n \cdot b_1 < l \cdot b_1$, since $a_1 \geq n \cdot b_1$ implies that no overflow occurs. In any case $a_1 < l \cdot b_1$. From the fourth line we have analogously $b_1 < l \cdot a_1$ and combining these two inequalities we get $\alpha < l \cdot \beta$. □

The error-free transformation of $a_1 - b_1$ into $x + y$ is performed by an algorithm derived from the following theorem which is due to Dekker [3].

Theorem 6 (Dekker)**.**
Let $u, v \in \mathbb{F}$ with $|u| \geq |v|$ and compute x, y in round-to-nearest as $x = u \oplus v$ and $y = v \ominus (x \ominus u)$. Then, if no overflow occurs $x + y = a + b$. □

The theorem relies on the fact, that if $x = u \oplus v$ is computed in round-to-nearest, the exact error $u + v - x$ is a floating-point number. With the help of Sterbenz Lemma it can then be shown that $y = v \ominus (x \ominus u)$ actually computes this error. For faithful rounding it is not necessarily true that $u + v - x$ is a floating-point number, but Lemma 5 allows us to show that this is still true in our case. To compute y we need to distinguish the cases where $a_1 > b_1$ and $a_1 < b_1$. In our implementation we replace u with a_1 and v with $-b_1$ and then remove all unary $-$ operations.

⟨*revised essa compute x* 5.1⟩≡
```
const double x = a[1] - b[1];
```

⟨*revised essa compute y for $a_1 > b_1$* 5.2⟩≡
```
const double y = (a[1] - x) - b[1];
```

⟨*revised essa compute y for $a_1 < b_1$* 5.3⟩≡
```
const double y = a[1] - (b[1] + x);
```

Lemma 7. *Let x and y be computed by the code above, then $x + y = a_1 - b_1$. Let $\alpha = \max\{a_1, b_1\}$ and $\beta = \min\{a_1, b_1\}$. Then $|x| < \alpha$ and $|y| < 2\varepsilon_{\mathtt{m}} \cdot l \cdot \beta$. If the computation is performed in round-to-nearest, then $|y| < \varepsilon_{\mathtt{m}} \cdot l \cdot \beta$.*

This already shows that the algorithm will terminate because in each step a_1 and b_1 will be replaced by numbers with smaller absolute value. Since positive floating-point numbers can not be arbitrarily small, sooner or later zero values must be produced which will not be inserted into a or b again. The number of summands will drop and the algorithm will terminate.

Proof. By Lemma 5 we have $\alpha < l \cdot \beta$ so we can chose a small $\delta > 0$ such that still $\alpha \leq (l - \delta)\beta$. Let $s = a_1 - b_1$ be the exact difference. If $s = 0$ the code above will compute $x = 0$ and $y = 0$ and the

claim is true, so we assume $s \neq 0$ in the following. We have $-b_1 < s < a_1$ and hence $|s| < \alpha$. Then

$$\begin{aligned}
\operatorname{msb}(s) &\leq \operatorname{msb}(\alpha) \\
&\leq \operatorname{msb}((l-\delta)\beta) \\
&\leq 2\operatorname{msb}(l-\delta)\operatorname{msb}(\beta) \\
&= 2\operatorname{msb}(l-\delta)\min\{\operatorname{msb}(a_1), \operatorname{msb}(b_1)\} \\
&\leq 2\operatorname{msb}(l-\delta)\frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\min\{\operatorname{lsb}(a_1), \operatorname{lsb}(b_1)\} && \text{by Definition 1} \\
&\leq \varepsilon_{\mathtt{m}}^{-1}\operatorname{msb}(l-\delta)\operatorname{lsb}(s) && \text{by Properties (5,6)}
\end{aligned}$$

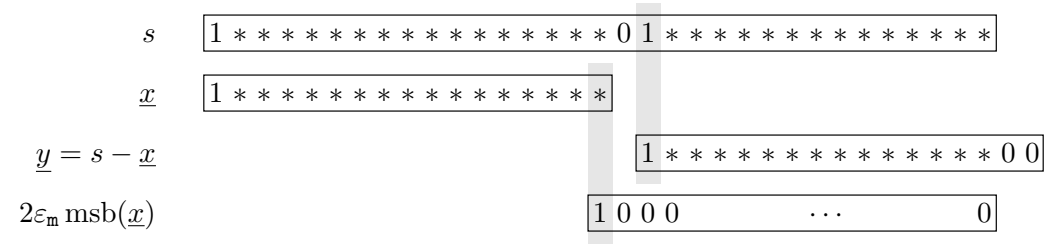
and therefore

$$\log_2\left(\frac{\operatorname{msb}(s)}{\operatorname{lsb}(s)}\right) \leq t + \lfloor \log_2(l-\delta) \rfloor.$$

This shows that s has a mantissa of at most $t + \lfloor \log_2(l-\delta) \rfloor + 1 \leq 2t-1$ bits, since $l - \delta < \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}$ by Lemma 4. We also know that

$$f_{\min} \leq \min\{\operatorname{lsb}(a_1), \operatorname{lsb}(b_1)\} \leq \operatorname{lsb}(s) \qquad \text{and} \qquad \operatorname{msb}(s) \leq f_{\max}. \tag{10}$$

Assume s is not itself a floating-point number. With faithful rounding there are two directions where s may be rounded to, towards zero or away from zero. Let $\underline{x}$ be the floating-point number next to s closer to zero. Then $\underline{x}$ can be obtained by truncating the mantissa of s after t bits. Consequently the rounding error $\underline{y} = s - \underline{x}$ consists of the at most $t-1$ trailing bits of s. Since the bounds from Property (10) immediately apply to $\underline{y}$, it is $\underline{y} \in \mathbb{F}$ by Definition 1.



Now let $\overline{x}$ be the floating-point number next to s that is further away from zero. Then for $s > 0$ there is $\overline{x} = \underline{x} + 2\varepsilon_{\mathtt{m}}\operatorname{msb}(\underline{x})$ and $\overline{y} = s - \overline{x} = \underline{y} - 2\varepsilon_{\mathtt{m}}\operatorname{msb}(\underline{x})$. Thus also $\overline{y} \in \mathbb{F}$, since $\underline{y}$ has no non-zero bits with a value lower than $2\varepsilon_{\mathtt{m}} \cdot 2\varepsilon_{\mathtt{m}}\operatorname{msb}(\underline{x})$. No matter where s is rounded to, the rounding error is a floating-point number.

We now show that this error is computed by the code above. We consider the case $a_1 > b_1$. Then we compute $x = a_1 \ominus b_1$ and $y = (a_1 \ominus x) \ominus b_1$. In case $b_1 \geq \frac{1}{2}a_1$, by Sterbenz Lemma we have $x = a_1 - b_1$ and consequently $y = 0$. In the other case $b_1 < \frac{1}{2}a_1$, we have $\frac{1}{2}a_1 = a_1 - \frac{1}{2}a_1 < a_1 - b_1$ and from Property (1) follows $\frac{1}{2}a_1 \leq x$. Furthermore we have $x \leq a_1$ and can apply Sterbenz Lemma to the computation $b_v = a_1 \ominus x$, so $b_v = a_1 - x$. Finally we already know, that $b_v - b_1 = a_1 - x - b_1$ is a floating-point number, so in the last step $y = b_v \ominus b_1 = b_v - b_1$. For the case $a_1 < b_1$ the proof proceeds analogously, after observing that $x < 0$ and $b_1 \oplus x = b_1 \ominus (-x)$. Thus we have shown $a_1 - b_1 = x + y$. Finally we turn to the bounds on $|x|$ and $|y|$. From $-b_1 < a_1 - b_1 < a_1$ follows $|a_1 - b_1| < \alpha$ and by Property (1) $|x| \leq \alpha$. For $|y|$ we have:

$$\begin{aligned}
|y| &= |a_1 - b_1 - x| \\
&= |\delta||a_1 - b_1| && \text{by Property (8)} \\
&< 2\varepsilon_{\mathtt{m}} \cdot \alpha \\
&< 2\varepsilon_{\mathtt{m}} \cdot l \cdot \beta && \text{by Lemma 5}
\end{aligned}$$

When computing in round-to-nearest we can replace $2\varepsilon_{\mathtt{m}}$ by $\varepsilon_{\mathtt{m}}$. Now assume $|x| = \alpha$, then $|y| = \beta$ which is by Lemma 4 a contradiction to the bound on $|y|$. □

In the loop body, we compute x and y and update a and b. From Property (9) we know that $\operatorname{sign}(x) = \operatorname{sign}(a_1 - b_1)$, so we can branch on $\operatorname{sign}(x)$ to decide which way to compute y. When $x = 0$ then also $y = 0$, so only the top elements of the two sequences are replaced by the last elements in this case. Any case leaves us with a new top element in a and b, so we restore the heap property for those.

⟨*revised essa loop body* 7.1⟩≡
```
⟨revised essa compute x 5.1⟩
if(x > 0.0){
  ⟨revised essa compute y for a_1 > b_1 5.2⟩
  ⟨revised essa update a and b for a_1 > b_1 7.2⟩
}else if(x < 0.0){
  ⟨revised essa compute y for a_1 < b_1 5.3⟩
  ⟨revised essa update a and b for a_1 < b_1 7.3⟩
}else{
  a[1] = a[m-];
  b[1] = b[n-];
}
restore_heap_from_top(a,m);
restore_heap_from_top(b,n);
```

In case $x > 0$ we insert x into a, replacing the top element and branch further on the sign of y. If $y > 0$ we insert y at the bottom of a and immediately restore the heap property for y. Although the top element has already been replaced by x and may violate the heap property, this works correctly since y is smaller than x. The other cases are straightforward. At the end of the code chunk both a and b have new top elements which may violate the heap property.

⟨*revised essa update a and b for* $a_1 > b_1$ 7.2⟩≡
```
a[1] = x;

if(y > 0.0){
  b[1] = b[n-];
  a[++m] = y;
  restore_heap_from_bottom(a,m);
}else if(y < 0.0){
  b[1] = -y;
}else{
  b[1] = b[n-];
}
```

The case $x < 0$ is symmetric to $x > 0$.

⟨*revised essa update a and b for* $a_1 < b_1$ 7.3⟩≡
```
b[1] = -x;

if(y < 0.0){
  a[1] = a[m-];
  b[++n] = -y;
  restore_heap_from_bottom(b,n);
}else if(y > 0.0){
  a[1] = y;
}else{
  a[1] = a[m-];
}
```

We will now give an upper bound on the number of iterations revised ESSA performs when run in round-to-nearest.

Theorem 8.
Run revised ESSA *with initially* $l \leq \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}$ *summands in round-to-nearest. Then the algorithm termi-*

nates after at most

$$\chi(l) \cdot l^2 \qquad \textit{where} \qquad \chi(l) = \frac{1}{2}\left\lceil \frac{t + \log_2 l}{t - \log_2 l} \right\rceil$$

iterations of the main loop.

Ratschek and Rokne [14] show that original ESSA terminates after at most $t \cdot l^2$ iterations, cf. Theorem 9. We remark that $\chi(l) < t$ for $l \leq \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}$ so our bound is an improvement. Furthermore for IEEE 754 double precision arithmetic we have $\chi(l) \leq 1$ for $l \leq 2^{17}$. This should include nearly all applications. There are however examples, where original ESSA needs less iterations, cf. Section 4.1. Our proof of the bound closely follows the proof of the upper bound for the original ESSA by Ratschek and Rokne. Instead of working with the exponents of numbers we look at their actual values and exploit the better reduction of a single value in one iteration.

Proof. For $l = 1$ the algorithm does not perform an iteration, so we can assume $l \geq 2$. We consider the input numbers $a_1, \ldots, a_m$ and $b_1, \ldots, b_n$ as a single sequence of variables $c_1, c_2, \ldots, c_l$. Let σ be the smallest lsb in the sequence and let it be attained by c_η:

$$\sigma = \min\{\operatorname{lsb}(c_i) \ : \ i = 1, \ldots, l\} = \operatorname{lsb}(c_\eta)$$

By Property (5) it follows that $c_i \in \sigma\mathbb{Z}$ for $i = 1, \ldots, l$. When we compute $x + y = c_i - c_j$ in an iteration, we replace $\alpha = \max\{c_i, c_j\}$ with $|x|$ and $\beta = \min\{c_i, c_j\}$ with $|y|$ in the sequence. Since x and y are computed using the operations $\oplus, \ominus$ only, by Property (7) after such an operation still $c_i \in \sigma\mathbb{Z}$ for $i = 1, \ldots, l$. By replacing α and β with $|x|$ and $|y|$, the size of the elements in the sequence is gradually reduced. We denote by c_i^r the value of c_i after it has played r times the role of β. As soon as all elements are smaller than $\varepsilon_{\mathtt{m}}^{-1}\sigma$, all computations $x + y = c_i - c_j$ will result in $y = 0$. This holds in the case $c_i - c_j = 0$. Otherwise we have

$$\begin{aligned}
\operatorname{msb}(c_i - c_j) &\leq \operatorname{msb}(\max\{c_i, c_j\}) && \text{since } c_i, c_j > 0 \\
&\leq \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\sigma && \text{since } \max\{c_i, c_j\} < \varepsilon_{\mathtt{m}}^{-1}\sigma \\
&\leq \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1} \operatorname{lsb}(c_i - c_j) && \text{by Property (6)}
\end{aligned}$$

and $f_{\min} \leq \sigma \leq \operatorname{lsb}(c_i - c_j)$, again by Property (6), and $\operatorname{msb}(c_i - c_j) \leq \operatorname{msb}(\max\{c_i, c_j\}) < f_{\max}$, so by Definition 1 we have $c_i - c_j \in \mathbb{F}$, $x = c_i \ominus c_j = c_i - c_j$ and $y = 0$.
When an element plays the role of β in a computation, its absolute value is reduced by a factor of $\varepsilon_{\mathtt{m}} \cdot l$ by Lemma 7. We will now count how often an element must play the role of β to be reduced to $\varepsilon_{\mathtt{m}}^{-1}\sigma$. We do not count when an element plays the role of α, since then another element plays the role of β. Also, when an element plays the role of α it is not increased so this does not invalidate the analysis.
Assume that at the start of the algorithm the sequence is ordered, i.e., $c_1 \geq c_2 \geq \ldots \geq c_l$. We can further assume, that the quotient of two consecutive numbers is not too large, precisely we assume

$$c_i \leq l \cdot \varepsilon_{\mathtt{m}}^{-1} \cdot c_{i+1} \tag{11}$$

To justify this assumption, let μ be the lowest index such that the assumption is not fulfilled:

$$c_\mu > l \cdot \varepsilon_{\mathtt{m}}^{-1} \cdot c_{\mu+1}. \tag{12}$$

Let $\hat{\sigma} = \min\{\operatorname{lsb}(c_i) \ : \ i = 1, \ldots, \mu\} = \operatorname{lsb}(c_{\hat{\eta}})$. The algorithm will now reduce the c_i values, however as long as an operation $x + y = c_i - c_j$ only involves elements with $i, j \leq \mu$, we have $c_i \in \hat{\sigma}\mathbb{Z}$ for

$i = 1, \ldots, \mu$, none of the numbers will ever be smaller than $c_{\mu+1}$:

$$\begin{aligned} c_i^r &\geq \hat{\sigma} = \mathrm{lsb}(c_{\hat{\eta}}) && \text{by } 0 \neq c_i^r \in \hat{\sigma}\mathbb{Z} \\ &\geq 2\varepsilon_{\mathtt{m}} \cdot \mathrm{msb}(c_{\hat{\eta}}) && \text{by Definition 1} \\ &> \varepsilon_{\mathtt{m}} \cdot c_{\hat{\eta}} \\ &\geq \varepsilon_{\mathtt{m}} \cdot c_{\mu} \\ &> l \cdot c_{\mu+1} && \text{by Property (12)} \end{aligned}$$

There are now two possibilities:

- The algorithm only considers elements c_i, c_j with $i, j \leq \mu$ until all these elements are zero and then proceeds with elements with larger index. In this case we can divide all c_i, $i > \mu$ by an appropriate power of two, such that $c_\mu \leq l \cdot \varepsilon_{\mathtt{m}}^{-1} \cdot c_{\mu+1}$ but still $c_\mu > \varepsilon_{\mathtt{m}}^{-1} \cdot c_{\mu+1}$. This will not change the course of the algorithm.
- At some point the algorithm considers for the first time two elements c_i^r with $i \leq \mu$ and $c_j = c_j^0$ with $j > \mu$. Then the algorithm will terminate, because $c_i^r > l \cdot c_{\mu+1} \geq l \cdot c_j$ but Lemma 5 guarantees $c_i^r < l \cdot c_j$. This is not a worst case, the algorithm would perform more steps if the gap was smaller.

Having justified the assumption we proceed with counting how often c_i must play the role of β to be smaller than $\varepsilon_{\mathtt{m}}^{-1}\sigma$. We have $c_l \leq c_\eta < \varepsilon_{\mathtt{m}}^{-1}\sigma$ and therefore

$$\begin{aligned} c_i^{k_i} &< (\varepsilon_{\mathtt{m}} \cdot l)^{k_i} c_i && \text{by Lemma 7} \\ &\leq (\varepsilon_{\mathtt{m}} \cdot l)^{k_i} (l \cdot \varepsilon_{\mathtt{m}}^{-1})^{l-i} c_l && \text{by Property (11)} \\ &< (\varepsilon_{\mathtt{m}} \cdot l)^{k_i} (l \cdot \varepsilon_{\mathtt{m}}^{-1})^{l-i} \varepsilon_{\mathtt{m}}^{-1}\sigma \end{aligned}$$

so $c_i^{k_i} < \varepsilon_{\mathtt{m}}^{-1}\sigma$ if $(\varepsilon_{\mathtt{m}} \cdot l)^{k_i}(l \cdot \varepsilon_{\mathtt{m}}^{-1})^{l-i} \leq 1$. This is the case if

$$k_i \geq (l-i) \left\lceil \frac{t + \log_2 l}{t - \log_2 l} \right\rceil = (l-i) \cdot 2\chi(l).$$

The right hand side must be rounded up, because only an integral number of iterations can be performed. When all summands are smaller than $\varepsilon_{\mathtt{m}}^{-1}\sigma$, at most l additional iterations are required, so finally the total number of iterations is bounded by

$$\begin{aligned} l + \sum_{i=1}^{l} k_i &= l + 2\chi(l) \cdot \sum_{i=1}^{l} (l-i) \\ &= l + 2\chi(l) \cdot \frac{l \cdot (l-1)}{2} \\ &\leq \chi(l) \cdot l + \chi(l) \cdot (l^2 - l) && \text{assuming } l \geq 2 \\ &= \chi(l) \cdot l^2 \end{aligned}$$

which finishes the proof. □

3.2 Original ESSA

In this section we provide an implementation of the original ESSA. For a detailed discussion of the algorithm, we refer the interested reader to the paper by Ratschek and Rokne [14]. Our implementation basically follows the paper and the sample implementation [9]. We mention some differences.

- We do not sort the summands initially but put them in heap order only.

- Unlike the paper [14] but following the sample implementation [9] we do not use exponent extraction for the termination criteria.
- We integrate the code for updating a and b into the code for the error-free transformation. This saves some comparisons and allows to eliminate some variables, by storing results in their final position immediately.

Again we start with the overall structure of the implementation. The same requirements apply: a and b must provide enough space for up to l elements. The summands must be stored in positions $1, \ldots, m$ and $1, \ldots, n$. Note that we reuse some code chunks, especially the termination criteria, so Lemma 5 is valid. In the main loop original ESSA modifies the two sequences a and b by an error-free transformation.

⟨*original essa* 10.1⟩≡
```
int original_essa::
sign_of_sum(double *const a, double *const b, int m, int n)const{
  ⟨build heap 4.2⟩
  while (true){
    ⟨termination criteria 4.3⟩
    ⟨original essa loop body 10.3⟩
  }
}
```

The error-free transformation used in the original ESSA is based on knowing the exponents of a_1 and b_1. We use `double frexp( double x, int* e)` to extract the exponent. Note that `frexp` returns the mantissa in the range $[\frac{1}{2}, 1)$ and the corresponding exponent. Thus we have $E = \lfloor \log_2(a_1) \rfloor + 1$ and $F = \lfloor \log_2(b_1) \rfloor + 1$. We do not decrement the exponents immediately, since this does not affect exponent comparison, but decrement only on demand later on.

⟨*exponent extraction* 10.2⟩≡
```
int E,F;
frexp(a[1],&E);
frexp(b[1],&F);
```

We distinguish cases based on the exponents.

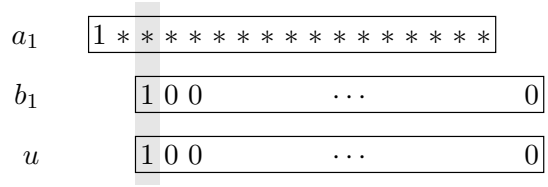
⟨*original essa loop body* 10.3⟩≡
```
⟨exponent extraction 10.2⟩
if (E == F){
  ⟨E = F 10.4⟩
}else if (E > F){
  ⟨E > F 11.1⟩
}else{ // (E < F)
  ⟨E < F 12.2⟩
}
```

If both leading summands have the same exponent, we can compute their difference exactly, i.e. $x = a_1 \ominus b_1 = a_1 - b_1$. By Property (9) we can compare a_1 and b_1 to determine the sign of x. We get at most one new summand, reducing the total number of summands.

⟨$E = F$ 10.4⟩≡
```
if ( a[1] > b[1] ){
  a[1] -= b[1];
  b[1] = b[n-];
}else if( a[1] < b[1] ){
  b[1] -= a[1];
  a[1] = a[m-];
}else{
  a[1] = a[m-];
  b[1] = b[n-];
}
```

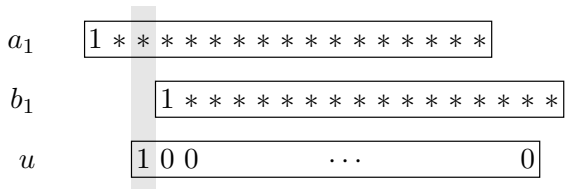
```
restore_heap_from_top(a,m);
restore_heap_from_top(b,n);
```

In the next case we have $E > F$ and consequently $a_1 > b_1$. We compute $x = (a_1 \ominus u)$ and $y = (u \ominus b_1)$ where $u = 2^{\lceil \log_2 b_1 \rceil}$. The function `double ldexp(double x, int e)` returns $x2^e$. We use it to compute $u = 2^{F-1}$. If $b_1 = u$, we subtract u from a_1 and get rid of one summand.



⟨$E > F$ 11.1⟩≡
```
double u = ldexp(1.0,F-1);
if( b[1] == u ){
  a[1] -= u;
  restore_heap_from_top(a,m);
}
```

If $b_1 \neq u$, we set $u = 2^F$ and compute $a_1 - u$ and $u - b_1$. In this case $u > 2^{F-1}$ and hence the exponent of $u - b_1$ is smaller than the exponent of b_1.



There is a caveat. We might have $a_1 = u$ and thus get a zero value, so we check for this case explicitly.

The subtraction $a_1 - u$ is error-free, if the leading bit of u overlaps the mantissa of a_1, i.e., if $E - F < t$. By Lemma 5 we have $a_1 < l \cdot b_1$ and hence $E - F = \lfloor \log_2(a_1) \rfloor - \lfloor \log_2(b_1) \rfloor < \log_2(l) + 1 \leq t$. So there is no rounding error.

⟨$E > F$ 11.1⟩+≡
```
else {
  u *= 2.0;
  if ( a[1] != u ){
    a[1] -= u;
    restore_heap_from_top(a,m);
    a[++m] = u - b[1];
    restore_heap_from_bottom(a,m);
  }else{
    a[1] = u - b[1];
    restore_heap_from_top(a,m);
  }
}
```

We inserted all new summands into a, the sequence b contains now one summand less. Note that we can not delay restoring the heap property for the new top elements as in revised ESSA, since x may be smaller than y.

⟨$E > F$ 11.1⟩+≡
```
b[1]=b[n-];
restore_heap_from_top(b,n);
```

The final case $E < F$ is symmetric to the previous one, here the size of a is reduced.

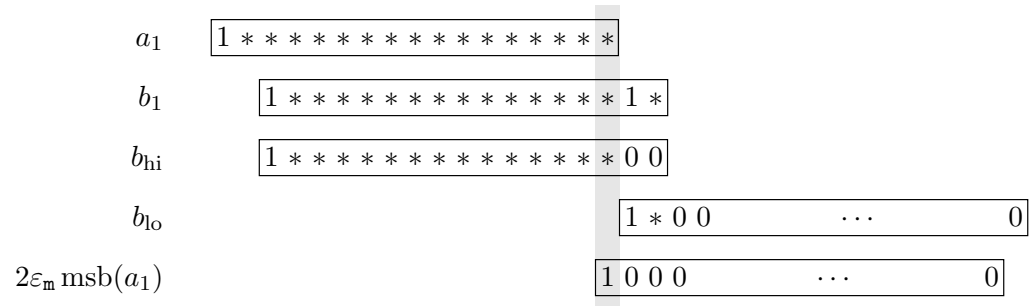
⟨$E < F$ 12.2⟩≡
```
double u = ldexp(1.0, E-1);
if ( a[1] == u ){
  b[1] -= u;
  restore_heap_from_top(b,n);
}else{
  u *= 2.0;
  if ( b[1] != u ){
    b[1] -= u;
    restore_heap_from_top(b,n);
    b[++n] = u - a[1];
    restore_heap_from_bottom(b,n);
  }else{
    b[1] = u - a[1];
    restore_heap_from_top(b,n);
  }
}
a[1]=a[m-];
restore_heap_from_top(a,m);
```

Ratschek and Rokne give an upper bound on the number of iterations original ESSA performs.

Theorem 9.
Run original ESSA *with initially* $l \leq \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}$ *summands. Then the algorithm terminates after at most* $t \cdot l^2$ *iterations of the main loop.* □

3.3 Modified ESSA

Gavrilova et al. [4] propose a modification to the error-free transformation of original ESSA in case the exponents of a_1 and b_1 differ. We now discuss this modification for the case $a_1 > b_1$. In original ESSA the value u can be almost twice as large as b_1. This has the negative effect that more than necessary is subtracted from a_1. The idea here is to compute a smaller u such that $x = a_1 \ominus u$ and $y = u \ominus b_1$ can be computed without rounding error.



Gavrilova et al. propose to split b into b_{hi} and b_{lo} by cutting of b_1 after the bit with value $2\varepsilon_{\mathtt{m}}\,\text{msb}(a_1)$. Setting $u = b_{\text{hi}}$ is possible, but in this case $x = a_1 - b_{\text{hi}} > 0$ and $y = b_{\text{hi}} - b_1 = -b_{\text{lo}} \leq 0$, which in general keeps the number of both the positive and negative summands constant. Therefore they alternatively suggest to set $u = b_{\text{hi}} + 2\varepsilon_{\mathtt{m}}\,\text{msb}(a_1)$. Gavrilova et al. remark "[...] that despite

having knowledge and programming tools sufficient enough for implementing bit manipulations in the program, we use standard processor floating-point operations, which work faster than an equivalent software implementation.", however no implementation for computing u is given. Our implementation uses `frexp`/`ldexp` to compute $\mathrm{msb}(a_1)$. (Rump et al. [17] give a way to compute the msb of a number using only $\oplus, \ominus, \odot$, however it might overflow and involves a branch).
Here is how we compute u. $\mathrm{msb}(a_1)+b_1$ is rounded to $\mathrm{msb}(a_1)\oplus b_1$ after the bit with value $2\varepsilon_{\mathtt{m}}\,\mathrm{msb}(a_1)$, the final subtraction of $\mathrm{msb}(a_1)$ is free from rounding error by Sterbenz Lemma. When run in round-towards-zero or round-down we get $u = b_{\mathrm{hi}}$, when run in round-up we get $u = b_{\mathrm{hi}} + 2\varepsilon_{\mathtt{m}}\,\mathrm{msb}(a_1)$. In round-to-nearest we may get any of these results.

⟨*modified* $E > F$ 13.1⟩≡
```
const double msba = ldexp(1.0,E-1);
const double u = (msba+b[1])-msba;
```

We know $x = a_1 - u \geq 0$, however y may have any sign. Depending on the rounding mode, there are more restrictions on the sign of x and y and the code may be simplified. We implement a complete version nevertheless and start with the case $a_1 > u$.

⟨*modified* $E > F$ 13.1⟩+≡
```
if(a[1] > u){
  a[1] -= u;
  restore_heap_from_top(a,m);
  if(u > b[1]){
    a[++m] = u - b[1];
    restore_heap_from_bottom(a,m);
    b[1] = b[n-];
  }else if(u < b[1]){
    b[1] -= u;
  }else{
    b[1] = b[n-];
  }
  restore_heap_from_top(b,n);
}
```

The remaining case is $a_1 = u$. It may occur, e.g., when $b_1 = \mathrm{msb}(a_1) - 2\varepsilon_{\mathtt{m}}\,\mathrm{msb}(a_1)$ and the rounding mode is round-up. $a_1 < u$ can not occur however.

⟨*modified* $E > F$ 13.1⟩+≡
```
else{ //(a[1] == u)
  if(u > b[1]){
    a[1] = u - b[1];
    b[1] = b[n-];
  }else if(u < b[1]){
    a[1] = a[m-];
    b[1] -= u;
  }else{
    a[1] = a[m-];
    b[1] = b[n-];
  }
  restore_heap_from_top(a,m);
  restore_heap_from_top(b,n);
}
```

In case $E < F$ and therefore $a_1 < b_1$ we proceed analogously.

⟨*modified* $E < F$ 13.4⟩≡
```
const double msbb = ldexp(1.0,F-1);
const double u = (msbb+a[1])-msbb;
if(b[1] > u){
  b[1] -= u;
  restore_heap_from_top(b,n);
```

```
    if(u > a[1]){
      b[++n] = u - a[1];
      restore_heap_from_bottom(b,n);
      a[1] = a[m-];
    }else if(u < a[1]){
      a[1] -= u;
    }else{
      a[1] = a[m-];
    }
    restore_heap_from_top(a,m);
  }else{ //(b[1] == u)
    if(u > a[1]){
      b[1] = u - a[1];
      a[1] = a[m-];
    }else if(u < a[1]){
      b[1] = b[n-];
      a[1] -= u;
    }else{
      b[1] = b[n-];
      a[1] = a[m-];
    }
    restore_heap_from_top(b,n);
    restore_heap_from_top(a,m);
  }
```

We complete the implementation reusing code chunks from original ESSA and revised ESSA.

⟨*modified essa loop body* 14.1⟩≡

```
⟨exponent extraction 10.2⟩
if (E == F){
  ⟨E = F 10.4⟩
}else if (E > F){
  ⟨modified E > F 13.1⟩
}else{ // (E < F)
  ⟨modified E < F 13.4⟩
}
```

The requirements are the same as for original and revised ESSA: a and b must provide enough space for up to l elements and the summands must be stored in positions $1, \ldots, m$ and $1, \ldots, n$.

⟨*modified essa* 14.2⟩≡

```
int modified_essa::
sign_of_sum(double *const a, double *const b, int m, int n)const{
  ⟨build heap 4.2⟩
  while (true){
    ⟨termination criteria 4.3⟩
    ⟨modified essa loop body 14.1⟩
  }
}
```

3.4 A simpler interface and wrapping ESSA

For simplicity and to achieve interface compatibility with other floating-point summation algorithms, we provide another to our ESSA variants. The following function computes the sign of $s = \sum_{i=0}^{l-1} a_i$. Here the summands must be stored at positions $0, \ldots, l-1$ and may have any sign. Still l may not exceed $\frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}$.

⟨*simpler interface for essa* 14.3⟩≡

```
inline int
sign_of_sum(double *const a,const int l)const{
```

```
⟨set up positive and negative summands 15.1⟩
⟨compute and return sign 15.2⟩
}
```

The function proceeds by redistributing the contents of a to two arrays a and b according to their sign. Positive summands are kept in a, negative summands are negated and moved to b.

⟨*set up positive and negative summands* 15.1⟩≡

```
double *b = new double[l];
int m=0,n=0;
for(int i=0;i<l;i++){
       if(a[i] > 0.0) a[m++] =  a[i];
  else if(a[i] < 0.0) b[n++] = -a[i];
}
```

Then the requirements for our `sign_of_sum` functions from above are met and the one residing in the same `struct` is called. The pointers for a and b have to be modified so that the summands are in the positions starting from 1.

⟨*compute and return sign* 15.2⟩≡

```
const int s = sign_of_sum(a-1,b-1,m,n);
delete[] b;
return s;
```

Finally we put revised ESSA, original ESSA and modified ESSA into stateless `structs`, which allows to use them as template parameter.

⟨*revised essa struct* 15.3⟩≡

```
struct revised_essa{
  int sign_of_sum(double *const a, double *const b, int m, int n)const;
  ⟨simpler interface for essa 14.3⟩
};
```

⟨*original essa struct* 15.4⟩≡

```
struct original_essa{
  int sign_of_sum(double *const a, double *const b, int m, int n)const;
  ⟨simpler interface for essa 14.3⟩
};
```

⟨*modified essa struct* 15.5⟩≡

```
struct modified_essa{
  int sign_of_sum(double *const a, double *const b, int m, int n)const;
  ⟨simpler interface for essa 14.3⟩
};
```

The ESSA variants are bundled into files. The header contains the `struct` definitions and with them the simpler interface so it can be inlined.

⟨*variants_of_essa.hpp* 15.6⟩≡

```
#ifndef VARIANTS_OF_ESSA_HPP
#define VARIANTS_OF_ESSA_HPP

⟨revised essa struct 15.3⟩
⟨original essa struct 15.4⟩
⟨modified essa struct 15.5⟩

#endif//VARIANTS_OF_ESSA_HPP
```

The actual implementations of the `sign_of_sum` functions are compiled separately. This assures that the heap maintenance functions are inlined.

⟨*variants_of_essa.cpp* 15.7⟩≡

```
#include <cmath>
#include "variants_of_essa.hpp"
#include "heap_maintenance_for_essa.hpp"
```

⟨*revised essa* 4.1⟩
⟨*original essa* 10.1⟩
⟨*modified essa* 14.2⟩

3.5 Running ESSA in a specified rounding mode

We can run any of our ESSA implementations in any rounding mode, but the algorithms will behave differently. Therefore we create a wrapper function, that uses CGAL functionality to set and reset the rounding mode of the CPU, then call a variant of ESSA.

⟨*essa with rounding mode* 16.1⟩≡
```
inline int
sign_of_sum(double *const a, double *const b,const int m,const int n)const{
  CGAL::FPU_CW_t rounding_mode = CGAL::FPU_get_and_set_cw(ROUNDING_MODE);
  int s = ESSA().sign_of_sum(a,b,m,n);
  CGAL::FPU_set_cw(rounding_mode);
  return s;
}
```

`ROUNDING_MODE` and `ESSA` are template parameters to the `struct` which encapsulates this function. It provides the same interface as the other ESSA implementations.

⟨*essa with rounding mode struct* 16.2⟩≡
```
template <class ESSA,CGAL::FPU_CW_t ROUNDING_MODE=CGAL_FE_TONEAREST>
struct essa_with_rounding_mode{
  ⟨essa with rounding mode 16.1⟩
  ⟨simpler interface for essa 14.3⟩
};
```

We put the `struct` into a header file. `CGAL/Interval_nt.h` provides the functionality for setting the rounding mode.

⟨*essa_with_rounding_mode.hpp* 16.3⟩≡
```
#ifndef ESSA_WITH_ROUNDING_MODE_HPP
#define ESSA_WITH_ROUNDING_MODE_HPP

#include <CGAL/basic.h>
#include <CGAL/Interval_nt.h>

⟨essa with rounding mode struct 16.2⟩

#endif//ESSA_WITH_ROUNDING_MODE_HPP
```

3.6 Heap maintenance

Next we describe heap maintenance. We use the same functions for heap manipulation for all our ESSA variants. The function `build_heap` creates the heap bottom-up in linear time.

⟨*heap maintenance* 16.4⟩≡
```
inline void
build_heap(double *const a,const int n){
 int l = n >> 1;
 while ( l >= 1 ){
   int i = l;
   int j = l << 1;
   const double top = a[l-];
   ⟨walk down and establish heap property 17.1⟩
 }
}
```

⟨*walk down and establish heap property* 17.1⟩≡
```
while (j <= n){
  if( j < n && a[j] < a[j+1] ) ++j;
  if( top >= a[j] ) break;

  a[i] = a[j];
  i = j;
  j «= 1;
}
a[i] = top;
```

The function `restore_heap_from_top` restores the heap property for the top element a_1 by pushing it downwards in the heap.

⟨*heap maintenance* 16.4⟩+≡
```
inline void
restore_heap_from_top(double *const a,const int n){
  int i = 1;
  int j = 2;
  const double top = a[1];
  ⟨walk down and establish heap property 17.1⟩
}
```

The function `restore_heap_from_bottom` restores the heap property for a leaf a_i in the heap, by pushing it upwards in the heap.

⟨*heap maintenance* 16.4⟩+≡
```
inline void
restore_heap_from_bottom(double *const a,int i){
  int j = i » 1;
  const double last = a[i];
  while (j > 0 && a[j] < last){
    a[i] = a[j];
    i = j;
    j »= 1;
  }
  a[i]= last;
}
```

Heap maintenance is put into a file.

⟨*heap_ maintenance_for_ essa.hpp* 17.4⟩≡
```
#ifndef HEAP_MAINTENANCE_FOR_ESSA_HPP
#define HEAP_MAINTENANCE_FOR_ESSA_HPP

⟨heap maintenance 16.4⟩

#endif//HEAP_MAINTENANCE_FOR_ESSA_HPP
```

4 Examples and experiments

Original ESSA as described by Ratschek and Rokne has the property that the sequence with the smaller leading element has an element less in the next iteration. For revised ESSA we can achieve this by running it in round-towards-zero, for modified ESSA by running it in round-up. Then either $y = 0$ or y has the same sign as x. If $a_1 > b_1$ an element is removed from b but none is inserted. Similarly the size of a is reduced if $b_1 > a_1$. This property might force termination, as the length of the sequences has a large influence in the termination criteria. On the other hand the sequence whose length is reduced might actually be the dominant one. Thus it is unclear whether this property is beneficial.

4.1 Examples

To illustrate how the variants of ESSA work and to show that they perform a different number of iterations, we give two examples. In Table 1 original ESSA needs less iterations than our revised ESSA in the default rounding mode. To construct such an example we exploit the fact that in the original ESSA both results of an operation always have the same sign, while the roundoff error in our revised version may have any sign. This is not the case when running revised ESSA in round-towards-zero and consequently it needs as many steps as original ESSA in this example. In Table 2 both variants of revised ESSA need one iteration, while original ESSA needs four.

4.2 Test data generation

To test the performance of our ESSA variants, we need some input data. Instead of using data from a certain application or predicate, we decided to create artificial data. Using artificial data covering a broad range (hopefully) allows us to draw conclusions for a majority of applications. We will shortly mention two generators for random sums, taken from the literature, and then describe our own solution. Our goal is to generate sums with varying exponent ranges, including zero sums, as they correspond to degenerate constellations in geometric algorithms which occur occasionally.
To test the original ESSA, Ratschek and Rokne [14] generate sums with 10000 summands, using floating-point numbers with a mantissa of 24 bits and an exponent in the range of $[-127, 127]$. To generate easy examples, simply 10000 numbers are taken from that range. To generate harder sums, a number c is selected, then $-c$ is inserted eight times into the sum, together with $8c$, keeping the sum zero. This is repeated 1111 times, resulting in 9999 summands. Finally $\pm 2^{-60}$ is inserted and the summands are shuffled, so the sum of the now 10000 summands is $\pm 2^{-60}$. The latter scheme can be generalized by varying the final summand and therefore the value of the sum. We think however, that the property of 9 or any fixed number of summands canceling out exactly, rarely occurs in practice. Furthermore this could penalize one algorithm and help another one in an unapparent way, so we refrained from using this scheme.
Ogita et al. [12] propose a generator for random dot products with a prespecified condition number. The generator can also be used to generate sums, by transforming the dot product into a sum. Such a transformation however also imposes a structure on the sum and a generator for sums is easy to derive and just as easy to implement. The following function returns in a a sum with n summands and a condition of approximately 2^c. Naturally a must provide space for n summands.

⟨*random sum generation* 18.1⟩≡
```
void
generate_random_sum(double *const a,const int n,const int c){
  CGAL::MP_Float sum(0);
  const int m = n/2;
  ⟨generate first part of sum 18.2⟩
  ⟨generate second part of sum 21.1⟩
  random_integer_generator rig;
  std::random_shuffle(a,a+n,rig);
}
```

The first half of the summands are generated as $\pm\alpha \cdot 2^e$ with α uniformly drawn from $[0, 1]$ and e uniformly drawn from $0, \ldots, c$. It is assured, that 0 and c actually occur as exponent.

⟨*generate first part of sum* 18.2⟩≡
```
for(int i=1;i<m-1;i++){
   const int e = static_cast<int>(drand48()*c);
   a[i] = ldexp((2*drand48()-1),e);
   sum += CGAL::MP_Float(a[i]);
}
a[0]   = ldexp((2*drand48()-1),0);
a[m-1] = ldexp((2*drand48()-1),c);
```

revised ESSA	original ESSA	modified ESSA in round-up
$a_1 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^0$ $a_2 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^0$ $b_1 = 1.0000000000000000000000000000000000000000000000000001_2 \cdot 2^{-2}$ $b_2 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-2}$ $b_3 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-54}$ $b_4 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-54}$	$a_1 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^0$ $a_2 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^0$ $b_1 = 1.0000000000000000000000000000000000000000000000000001_2 \cdot 2^{-2}$ $b_2 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-2}$ $b_3 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-54}$ $b_4 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-54}$	$a_1 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^0$ $a_2 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^0$ $b_1 = 1.0000000000000000000000000000000000000000000000000001_2 \cdot 2^{-2}$ $b_2 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-2}$ $b_3 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-54}$ $b_4 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-54}$
In the first iteration, x gets rounded up, so y is negative. $x = 1.1000000000000000000000000000000000000000000000000000_2 \cdot 2^{-1}$ $y = -1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-54}$	Original ESSA computes two new positive summands. $x = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-1}$ $y = 1.1111111111111111111111111111111111111111111111111110_2 \cdot 2^{-3}$	Modified ESSA in round-up also computes two positive summands, yet x is a better approximation to $a_1 - b_1$ than x in original ESSA. $x = 1.0111111111111111111111111111111111111111111111111110_2 \cdot 2^{-1}$ $y = 1.1000000000000000000000000000000000000000000000000000_2 \cdot 2^{-53}$
The size of both sequences stays the same and the largest elements have almost the same magnitude as before. Revised ESSA does not terminate yet. $a_1 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^0$ $a_2 = 1.1000000000000000000000000000000000000000000000000000_2 \cdot 2^{-1}$ $b_1 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-2}$ $b_2 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-54}$ $b_3 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-54}$ $b_4 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-54}$	Although the largest elements in both sequences are now almost the same magnitude as in the first iteration, there are fewer negative summands, $a_1 > 3b_1$ and original ESSA terminates. $a_1 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^0$ $a_2 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-1}$ $a_3 = 1.1111111111111111111111111111111111111111111111111110_2 \cdot 2^{-3}$ $b_1 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-2}$ $b_2 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-54}$ $b_3 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-54}$	Modified ESSA terminates just like original ESSA. $a_1 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^0$ $a_2 = 1.0111111111111111111111111111111111111111111111111110_2 \cdot 2^{-1}$ $a_3 = 1.1000000000000000000000000000000000000000000000000000_2 \cdot 2^{-53}$ $b_1 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-2}$ $b_2 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-54}$ $b_3 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-54}$
Although Sterbenz Lemma is not applicable, the next computation is exact and revised ESSA terminates. $x = 1.1000000000000000000000000000000000000000000000000000_2 \cdot 2^{-1}$ $y = 0$		
$a_1 = 1.1000000000000000000000000000000000000000000000000000_2 \cdot 2^{-1}$ $a_2 = 1.1000000000000000000000000000000000000000000000000000_2 \cdot 2^{-1}$ $b_1 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-54}$ $b_2 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-54}$ $b_3 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-54}$		

Table 1: We create a sum with positive sign. The two positive summands a_1 and a_2 are chosen to be equal. The two larger negative summands b_1 and b_2 are chosen such that $4b_i \geq a_i$. Furthermore b_1 is chosen such that the transformation of $a_1 - b_1$ into $x + y$ will result in a negative y for revised ESSA. b_3 and b_4 are just there to make the set of negative summands appear large. Note that in round-to-zero revised ESSA would also terminate after the first iteration.

revised ESSA	original ESSA	modified ESSA in round-up
$a_1 = 1.0000000000000000000000000000000000000000000000000011_2 \cdot 2^1$ $b_1 = 1.0000000000000000000000000000000000000000000000000011_2 \cdot 2^0$ $b_2 = 1.0000000000000000000000000000000000000000000000000010_2 \cdot 2^{-1}$ $b_3 = 1.0000000000000000000000000000000000000000000000000010_2 \cdot 2^{-1}$	$a_1 = 1.0000000000000000000000000000000000000000000000000011_2 \cdot 2^1$ $b_1 = 1.0000000000000000000000000000000000000000000000000011_2 \cdot 2^0$ $b_2 = 1.0000000000000000000000000000000000000000000000000010_2 \cdot 2^{-1}$ $b_3 = 1.0000000000000000000000000000000000000000000000000010_2 \cdot 2^{-1}$	$a_1 = 1.0000000000000000000000000000000000000000000000000011_2 \cdot 2^1$ $b_1 = 1.0000000000000000000000000000000000000000000000000011_2 \cdot 2^0$ $b_2 = 1.0000000000000000000000000000000000000000000000000010_2 \cdot 2^{-1}$ $b_3 = 1.0000000000000000000000000000000000000000000000000010_2 \cdot 2^{-1}$
Conforming to Sterbenz Lemma, the first computation $a_1 \ominus b_1$ in revised ESSA is exact. $x = 1.0000000000000000000000000000000000000000000000000011_2 \cdot 2^0$ $y = 0$	The exponents of a_1 and b_1 differ, therefore we get two new summands. Additionally this is a case where $\|x\| < \|y\|$. $x = 1.1000000000000000000000000000000000000000000000000000_2 \cdot 2^{-50}$ $y = 1.1111111111111111111111111111111111111111111111111010_2 \cdot 2^{-1}$	In round-up modified ESSA computes two new summands of the same sign, if any. $x = 1.0000000000000000000000000000000000000000000000000010_2 \cdot 2^0$ $y = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-52}$
Now already $a_1 > 2b_1$ and revised ESSA terminates. $a_1 = 1.0000000000000000000000000000000000000000000000000011_2 \cdot 2^0$ $b_1 = 1.0000000000000000000000000000000000000000000000000010_2 \cdot 2^{-1}$ $b_2 = 1.0000000000000000000000000000000000000000000000000010_2 \cdot 2^{-1}$	In contrast to revised ESSA the positive part of the sum is split into two summands. The termination criteria does not apply. $a_1 = 1.1111111111111111111111111111111111111111111111111010_2 \cdot 2^{-1}$ $a_2 = 1.1000000000000000000000000000000000000000000000000000_2 \cdot 2^{-50}$ $b_1 = 1.0000000000000000000000000000000000000000000000000010_2 \cdot 2^{-1}$ $b_2 = 1.0000000000000000000000000000000000000000000000000010_2 \cdot 2^{-1}$	We differ from revised ESSA only in one bit that is stored in a_2 instead of a_1, but this suffices to prevent termination. $a_1 = 1.0000000000000000000000000000000000000000000000000010_2 \cdot 2^0$ $a_2 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-52}$ $b_1 = 1.0000000000000000000000000000000000000000000000000010_2 \cdot 2^{-1}$ $b_2 = 1.0000000000000000000000000000000000000000000000000010_2 \cdot 2^{-1}$
	Now the exponents of a_1 and b_1 are equal and we get only one new summand. Still the termination criterion is not fulfilled. $x = 1.1111111111111111111111111111111111111111111111110000_2 \cdot 2^{-2}$ $y = 0$	Although the exponents differ, we get rid of one summand. $x = 1.0000000000000000000000000000000000000000000000000010_2 \cdot 2^{-1}$ $y = 0$
	$a_1 = 1.1111111111111111111111111111111111111111111111110000_2 \cdot 2^{-2}$ $a_2 = 1.1000000000000000000000000000000000000000000000000000_2 \cdot 2^{-50}$ $b_1 = 1.0000000000000000000000000000000000000000000000000010_2 \cdot 2^{-1}$	$a_1 = 1.0000000000000000000000000000000000000000000000000010_2 \cdot 2^{-1}$ $a_2 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-52}$ $b_1 = 1.0000000000000000000000000000000000000000000000000010_2 \cdot 2^{-1}$
	Original ESSA computes two new negative summands and thus reduces the size of the actually dominant sequence! $x = -1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-52}$ $y = -1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-50}$	Fortunately $a_1 = b_1$, so we loose two summands and modified ESSA terminates in the next step. $x = 0$ $y = 0$
	$a_1 = 1.1000000000000000000000000000000000000000000000000000_2 \cdot 2^{-50}$ $b_1 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-50}$ $b_2 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-52}$	$a_1 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-52}$
	$x = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-51}$ $y = 0$	
	$a_1 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-51}$ $b_1 = 1.0000000000000000000000000000000000000000000000000000_2 \cdot 2^{-52}$	

Table 2: The input data is chosen such that the first iteration in revised ESSA reduces the number of summands, while this is not the case for original ESSA. Furthermore the sum is only marginally positive, to prevent termination of original ESSA in the first step.

```
sum += CGAL::MP_Float(a[0]);
sum += CGAL::MP_Float(a[m-1]);
```

To generate the second half, for each summand first an accurate floating-point approximation s of the already generated sum is computed. Then $s \ominus \alpha \cdot 2^e$ is inserted as a summand, with α chosen as before and e decreasing linearly from c to 0.

⟨*generate second part of sum* 21.1⟩≡

```
for(int i=m;i<n;i++){
  const double factor = 1.0 - static_cast<double>(i-m)/static_cast<double>(n-m-1);
  const int e = static_cast<int>(factor * c);
  a[i] =  ldexp((2*drand48()-1),e)-CGAL::to_double(sum);
  sum += CGAL::MP_Float(a[i]);
}
```

Here is how we create zero sums. We select a random summand and compute a best floating-point approximation s' to the sum of the remaining summands. We replace the selected summand by $-s'$ and iterate until the overall sum is zero.

⟨*random sum generation* 18.1⟩+≡

```
void
modify_sum_to_zero(double *const a,const int n){
  CGAL::MP_Float sum(a[0]);
  for(int i=1;i<n;i++) sum += CGAL::MP_Float(a[i]);

  random_integer_generator rig;
  while(CGAL::to_double(sum) != 0.0){
    const int i = rig(n);
    sum -= CGAL::MP_Float(a[i]);
    a[i] = -CGAL::to_double(sum);
    sum += CGAL::MP_Float(a[i]);
  }
}
```

To assert reproducibility we use our own random integer generator for shuffling the summands.

⟨*random integer generator* 21.3⟩≡

```
struct random_integer_generator{
  inline int
  operator()(const int n){
    double d = drand48();
    while(d == 1) d = drand48();
    return static_cast<int>(floor(n*d));
  }
};
```

⟨*sum_generators.hpp* 21.4⟩≡

```
#ifndef SUM_GENERATORS_H
#define SUM_GENERATORS_H

double
generate_random_sum(double *const a, const int n,
                                      const int c);

double
modify_sum_to_zero(double *const a, const int n);
#endif
```

⟨*sum_generators.cpp* 21.5⟩≡

```
#include <cstdlib>
#include <cmath>
```

```
#include <algorithm>

#include <CGAL/basic.h>
#include <CGAL/MP_Float.h>
```

⟨*random integer generator* 21.3⟩
⟨*random sum generation* 18.1⟩

4.3 Experimental results

We performed experiments on a notebook with an Intel Core 2 Duo T5500 processor with 1.66 Ghz, using `g++ 4.3.2` and CGAL 3.3.1. Using the generator from above, we generate 10000 sums with l summands and approximate condition number c for $l \in \{16, 23, 64, 128, 256, 515\}$ and $c \in \{4, 8, 16, 23, 64, 128\}$. We measure running time, as well as the number of iterations of the ESSA main loop for each sum. Figure 1 and Figure 2 show the average running time in milliseconds for computing 10 times the sign of each sum. Figure 3 shows the minimum, average and maximum number of steps needed to compute the sign. We repeat the experiment, after modifying each sum to have a value of zero. Note that this changes the condition number to ∞ and ESSA is required to eliminate all summands. The results are shown in Figure 4, Figure 5 and Figure 6. As competitors we use original ESSA, modified ESSA in round-up, revised ESSA (in round-to-nearest) and revised ESSA in round-towards-zero.
Revised ESSA distributes new summands to both sequences, while all other three competitors move both new summands to the sequence which previously had the larger leading summand. However revised ESSA and revised ESSA in round-towards-zero are nearly indistinguishable, both with respect to running time and number of iterations. Controlling where new summands are moved to does not seem to have a large impact.
With respect to running time, modified ESSA is an improvement over original ESSA but revised ESSA is a clear winner. This seems partly to be caused by using standard floating-point operations only, since modified ESSA and revised ESSA are similar with respect to the number of iterations used. The number of iterations of original ESSA also shows why it performs badly. It has a larger variance and in general is larger than for any other ESSA variant. For many summands and large condition numbers its minimum is larger than the maximum number of iterations for any other ESSA variant.
The performance of revised ESSA seems to be invariant with respect to the condition number, both in the number of iterations and the running time. It rarely uses more than l iterations, leaving a large gap to the bound from Theorem 8. While the same holds for the number of iterations of modified ESSA, its running time increases with the condition number. The reason is probably that modified ESSA is not as good in reducing the number of summands, this makes heap maintenance more expensive.

5 Conclusion

Our revised ESSA is clearly an improvement over previously existing ESSA variants. Revised ESSA reduces the number of summands if and only if the subtraction of both leading elements can be performed without rounding error. It also maximizes the cancellation in each iteration. Both properties lead to a more compact representation of the sum in each step, which helps to meet the termination criteria early. It is however not competitive to other algorithms that allow to compute the sign of a sum of floating-point numbers exactly [11]. These algorithms avoid branches in their inner loops, which are essential for the heap maintenance in ESSA. Unlike these algorithms however revised ESSA is completely immune to overflow and underflow and runs correctly in any rounding mode. It might therefore have applications where these properties are required.

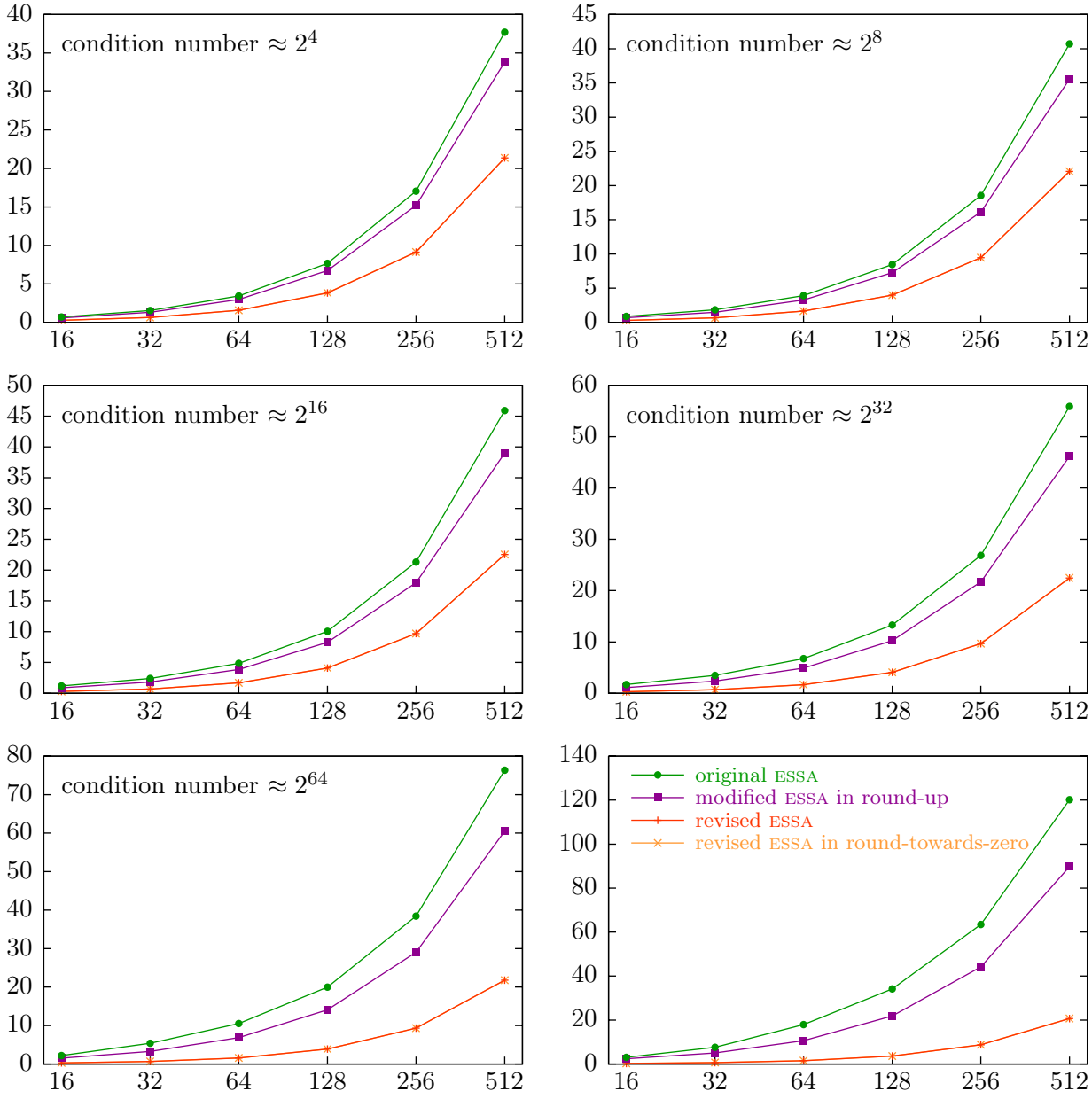


Figure 1: Running times for nonzero sums. The x-axis is labeled with the number of summands.

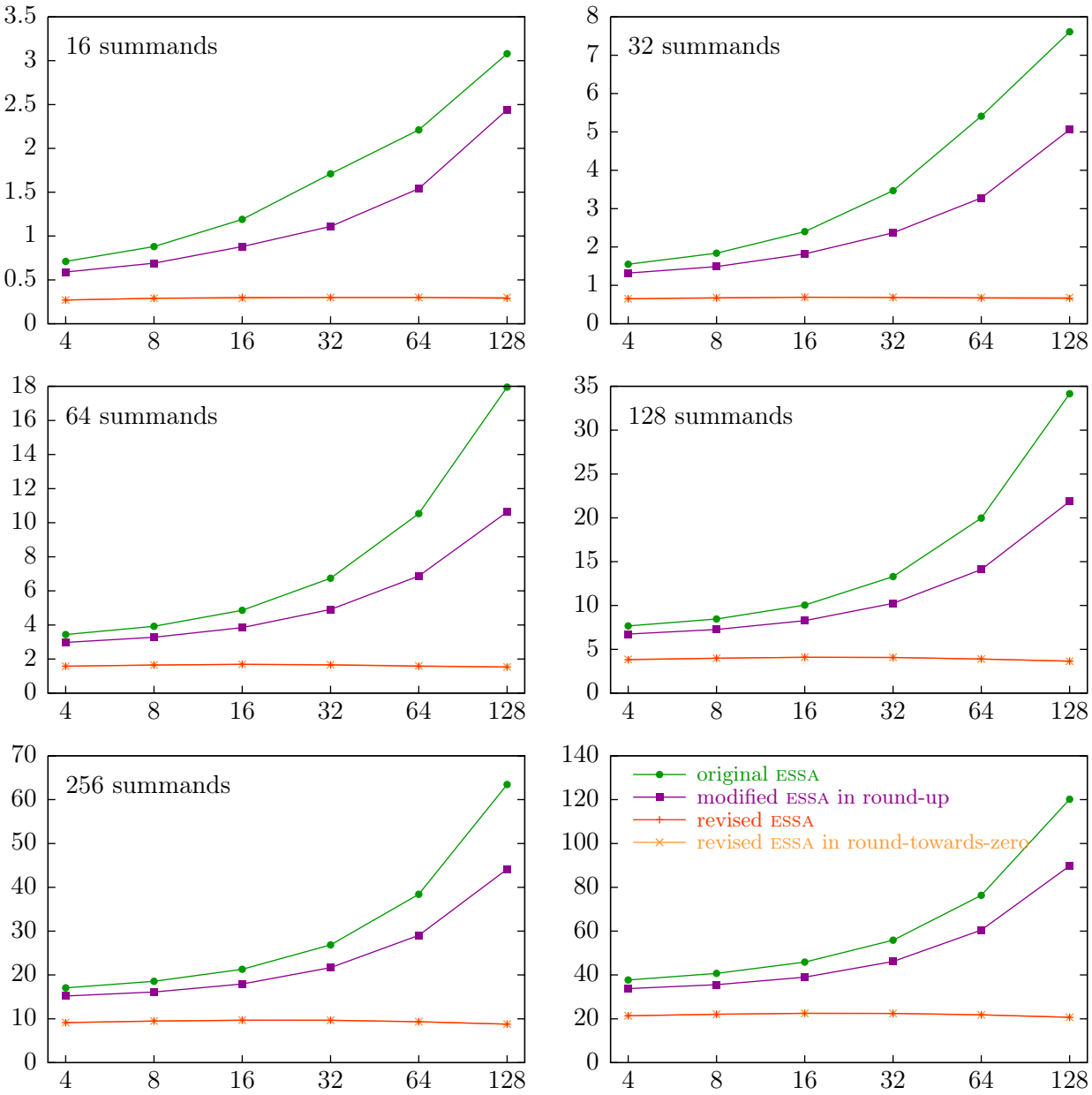


Figure 2: Running times for nonzero sums. The x-axis is labeled with c, where 2^c is the approximate condition number.

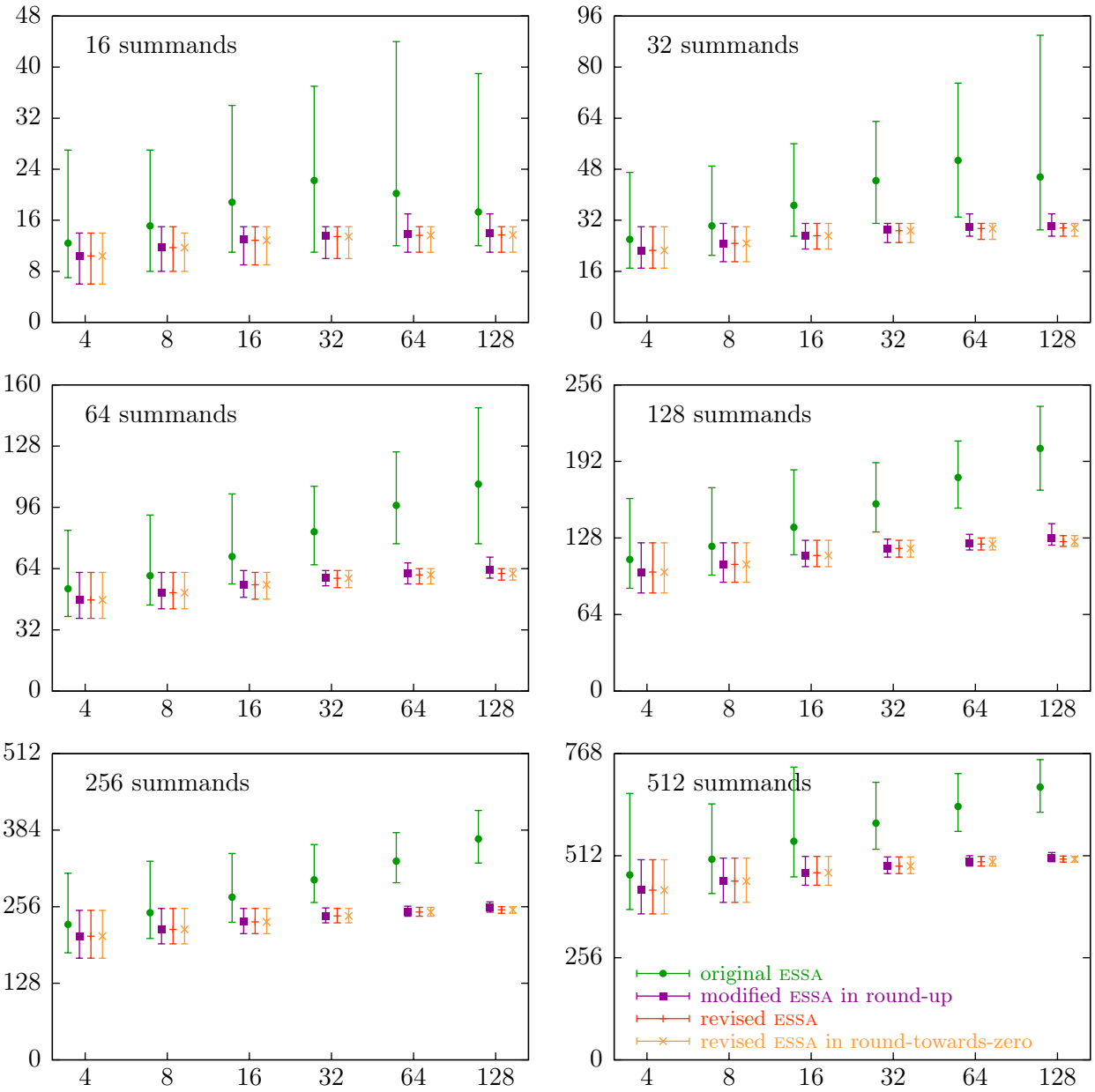


Figure 3: Number of iterations of the main loop for nonzero sums. The x-axis is labeled with c, where 2^c is the approximate condition number.

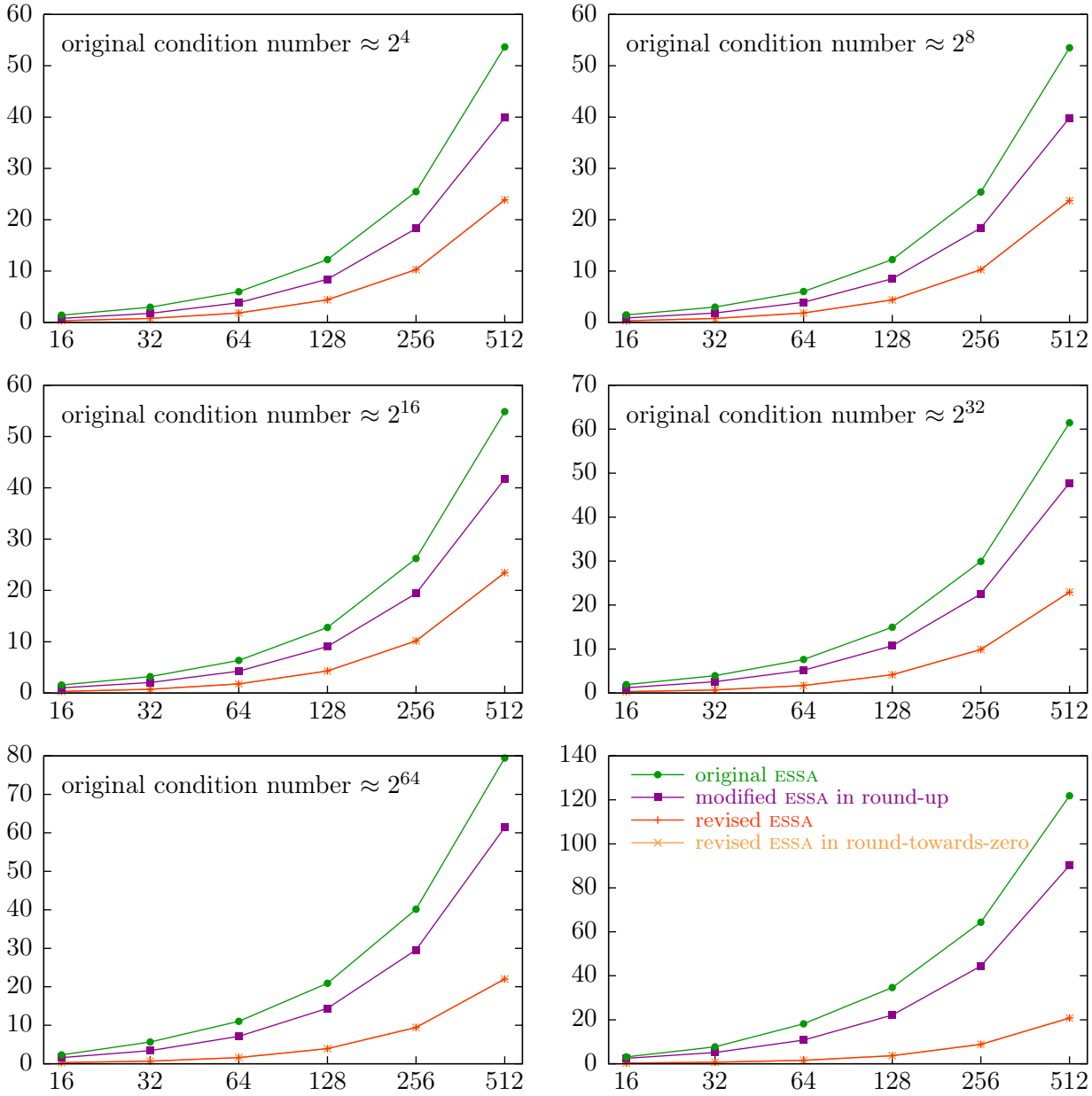


Figure 4: Running times for zero sums. The x-axis is labeled with the number of summands.

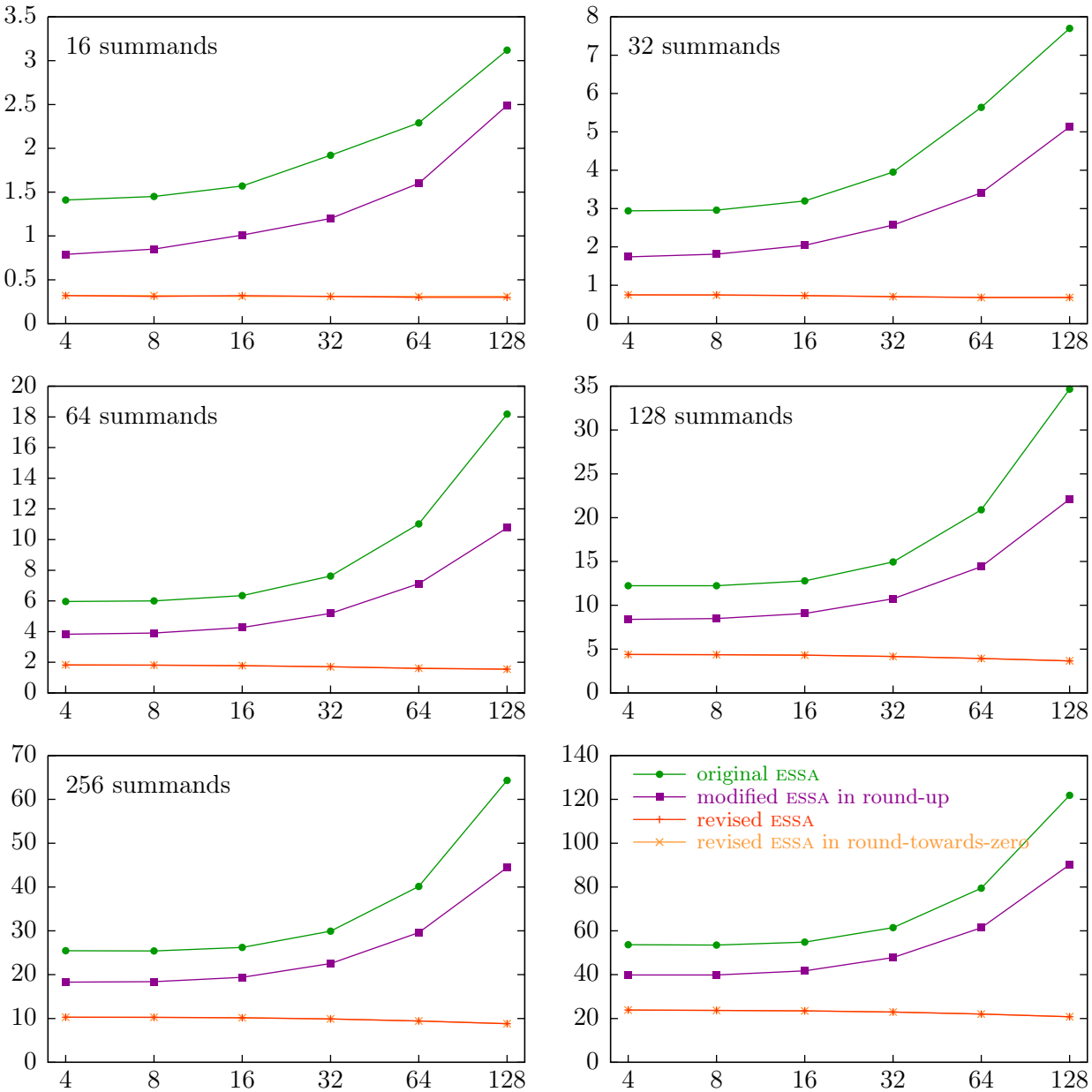


Figure 5: Running times for zero sums. The x-axis is labeled with c, where 2^c is the approximate condition number of the sum before it was modified to be zero.

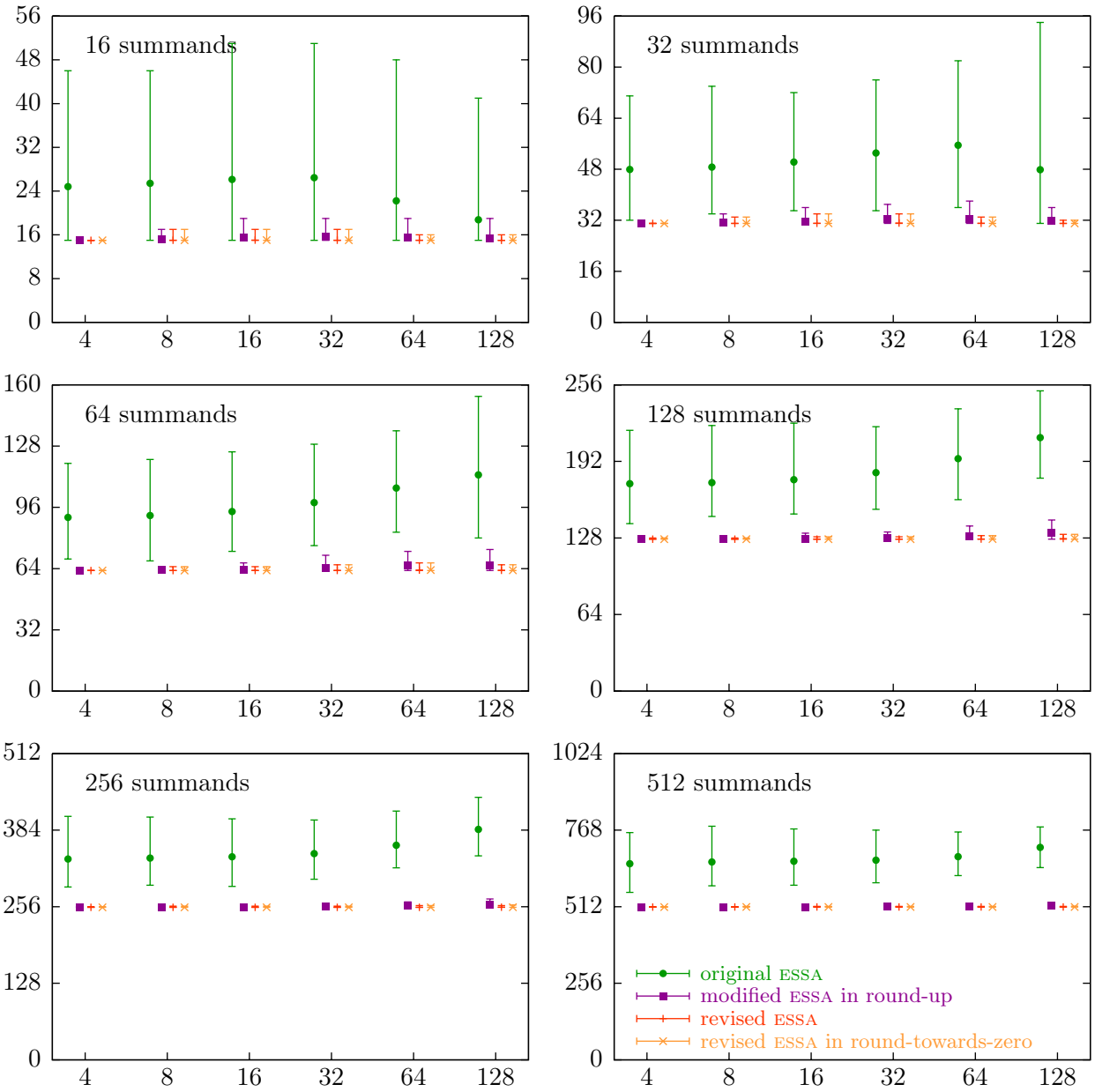


Figure 6: Number of iterations of the main loop for zero sums. The x-axis is labeled with c, where 2^c is the approximate condition number of the sum before it was modified to be zero.

References

[1] IEEE Standards Comitee 754. IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985. Institute of Electrical and Electronics Engineers, New York, 1985. Reprinted in SIGPLAN Notices, 22(2):9-25, 1987.

[2] David Cordes and Marcus Brown. The literate-programming paradigm. *Computer*, 24(6):52–61, 1991.

[3] Theodorus J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18(2):224–242, June 1971.

[4] Marina Gavrilova, Dmitri Gavrilova, and Jon G. Rokne. New algorithms for the exact computation of the sign of algebraic expressions. In *Canadian Conference on Electrical and Computer Engineering*, pages 314–317, May 1996.

[5] Marina Gavrilova, Helmut Ratschek, and Jon G. Rokne. Exact computation of Voronoi diagram and Delaunay triangulation. *Journal of Reliable Computing*, 6(1):39–60, 2000.

[6] Marina Gavrilova and Jon G. Rokne. Reliable line segment intersection testing. *Computer-Aided Design*, 32(12):737–745, October 2000.

[7] Marina Gavrilova and Jon G. Rokne. Computing line intersections. *International Journal of Image and Graphics*, 1(2):1–14, May 2001.

[8] Donald E. Knuth. Literate programming. *Comput. J.*, 27(2):97–111, 1984.

[9] Georg Mackenbrock, Helmut Ratschek, and Jon G. Rokne. Experimental reliable code for 2D convex hull construction, 1998. http://pages.cpsc.ucalgary.ca/~rokne/convex/.

[10] Daniel Mall. web pages on literate programming. http://www.literateprogramming.com/.

[11] Marc Mörig and Stefan Schirra. On the design and performance of reliable geometric predicates using error-free transformations and exact sign of sum algorithms. In *19th Canadian Conference on Computational Geometry (CCCG'07)*, pages 45–48, August 2007.

[12] Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing*, 26(6):1955–1988, 2005.

[13] Michael L. Overton. *Numerical Computing with IEEE Floating-Point Arithmetic*. SIAM, 2001.

[14] Helmut Ratschek and Jon G. Rokne. Exact computation of the sign of a finite sum. *Appl. Math. Comput.*, 99(2-3):99–127, 1999.

[15] Helmut Ratschek and Jon G. Rokne. Exact and optimal convex hulls in 2D. *Int. J. Comput. Geometry Appl.*, 10(2):109–129, 2000.

[16] Helmut Ratschek and Jon G. Rokne. *Geometric computations with interval and new robust methods: applications in computer graphics, GIS and computational geometry*. Horwood Publishing, Ltd., Chichester, USA, 2003.

[17] Siegfried M. Rump, Takeshi Ogita, and Shin'ichi Oishi. Accurate floating-point summation. Technical Report 05.1, Faculty of Information and Communication Science, Hamburg University of Technology, 2005.

[18] Pat H. Sterbenz. *Floating-Point Computation*. Prentice-Hall, 1974.

A Additional code

A.1 Checking ESSA

The next chunk can be inserted in any of the ESSA variants. It checks, that both heaps contain only positive elements and that the topmost elements a_1 and b_1 are the largest in a and b respectively.

⟨*assert a and b* 30.1⟩≡

```
for (int i=2; i<=m; i++)
{ if ( (a[1] < a[i]) || (a[i] <= 0) )
  { std::cout « "i: " « i « " m: " « m « " n: " « n « std::endl;
    for (int j=1;j<=m;j++) { std::cout « a[j] « std::endl; }
  }
  assert( (a[1] >= a[i]) );
  assert( (a[i] > 0) );
}
for (int i=2;i<=n;i++)
{ if ( (b[1] < b[i]) || (b[i] <= 0) )
  { std::cout « "i: " « i « " n: " « n « " m: " « m « std::endl;
    for (int j=1;j<=n;j++) { std::cout « b[j] « std::endl; }
  }
  assert( (b[1] >= b[i]) );
  assert( (b[i] > 0) );
}
```

The next chunk contains a function that checks that all essa variants return the same sign for a given sum. The sum given to the function is not modified.

⟨*assert consistency of essa variants* 30.2⟩≡

```
void
assert_essa(const double *const c, const int l){
  double* cc = new double[l];

  copy_sum(cc,c,l);
  int s1 = revised_essa().sign_of_sum(cc,l);
  copy_sum(cc,c,l);
  int s2 = revised_essa_in_round_toward_zero().sign_of_sum(cc,l);
  copy_sum(cc,c,l);
  int s3 = original_essa().sign_of_sum(cc,l);
  copy_sum(cc,c,l);
  int s4 = modified_essa_in_round_up().sign_of_sum(cc,l);
  copy_sum(cc,c,l);
  int s5 = modified_essa_in_round_toward_zero().sign_of_sum(cc,l);

  if( s1 != s2 || s2 != s3 || s3 != s4 || s4 != s5 ){
    std::cout « "ESSA variants inconsistent, exiting." « std::endl
              « s1 « " " « s2 « " " « s3 « " " « s4 « " " « s5
              « std::endl « std::endl;
    std::cout.precision(20);
    for(int i =0;i<l;i++) std::cout « c[i] « std::endl;
    exit(1);
  }

  delete[] cc;
}
```

A.2 Counting and verbose versions of all ESSA variants

Here we give modified versions of all three ESSA variants. They are augmented with code for printing the content of the heaps and the results of the error-free transformations. Additionally they do not return the sign, but the number of iterations taken by the algorithm.
To count the number of iterations a variant of ESSA takes, we need a counter and a modified termination criteria.

⟨*counter for steps of essa* 31.1⟩≡
```
int steps = 0;
```

⟨*termination criteria returning steps* 31.2⟩≡
```
if ( m == 0 || n == 0  || a[1] > n*b[1] || b[1] > m*a[1] ) return steps;
steps++;
```

Here we have all three ESSA variants again, composed of the same chunks as above. Again we encapsulate the functions into a `struct`.

⟨*counting original essa* 31.3⟩≡
```
int
sign_of_sum(double *const a, double *const b, int m, int n)const{
  ⟨counter for steps of essa 31.1⟩
  ⟨build heap 4.2⟩
  while (true){
    ⟨print a and b 33.2⟩
    ⟨termination criteria returning steps 31.2⟩
    ⟨print a-b=a-u+u-b 34.2⟩
    ⟨original essa loop body 10.3⟩
  }
}
```

⟨*counting original essa struct* 31.4⟩≡
```
struct counting_original_essa{
  ⟨counting original essa 31.3⟩
  ⟨simpler interface for essa 14.3⟩
};
```

⟨*counting modified essa* 31.5⟩≡
```
int
sign_of_sum(double *const a, double *const b, int m, int n)const{
  ⟨counter for steps of essa 31.1⟩
  ⟨build heap 4.2⟩
  while (true){
    ⟨print a and b 33.2⟩
    ⟨termination criteria returning steps 31.2⟩
    ⟨print a-b=a-u+u-b modified 34.3⟩
    ⟨modified essa loop body 14.1⟩
  }
}
```

⟨*counting modified essa struct* 31.6⟩≡
```
struct counting_modified_essa{
  ⟨counting modified essa 31.5⟩
  ⟨simpler interface for essa 14.3⟩
};
```

⟨*counting revised essa* 32.1⟩≡
```
int
sign_of_sum(double *const a, double *const b, int m, int n)const{
  ⟨counter for steps of essa 31.1⟩
  ⟨build heap 4.2⟩
  while (true){
    ⟨print a and b 33.2⟩
    ⟨termination criteria returning steps 31.2⟩
    ⟨print a-b=x+y 33.3⟩
    ⟨revised essa loop body 7.1⟩
  }
}
```

⟨*counting revised essa struct* 32.2⟩≡
```
struct counting_revised_essa{
  ⟨counting revised essa 32.1⟩
  ⟨simpler interface for essa 14.3⟩
};
```

The ESSA variants are bundled into one file.

⟨*counting_variants_of_essa.hpp* 32.3⟩≡
```
#ifndef COUNTING_VARIANTS_OF_ESSA_HPP
#define COUNTING_VARIANTS_OF_ESSA_HPP

#include <cmath>
#include "heap_maintenance_for_essa.hpp"

⟨includes for output 32.4⟩
⟨double to binary string conversion 32.5⟩

⟨counting original essa struct 31.4⟩
⟨counting modified essa struct 31.6⟩
⟨counting revised essa struct 32.2⟩

#endif//COUNTING_VARIANTS_OF_ESSA_HPP
```

A.3 Printing informations from within ESSA

The code chunks are useful to print information from the various ESSA variants. They are inserted into the code, but do nothing unless `ESSA_VERBOSE` is defined.

⟨*includes for output* 32.4⟩≡
```
#ifdef ESSA_VERBOSE
  #include <algorithm>
  #include <iostream>
  #include <sstream>
  #include <iomanip>
  #include <bitset>
#endif //ESSA_VERBOSE
```

The following code converts a `double` into a string showing its binary representation such that it is usable in a tex file.

⟨*double to binary string conversion* 32.5⟩≡
```
#ifdef ESSA_VERBOSE

union ieee_double{
  double d;
  unsigned long long l;
```

```
    ieee_double(const double a):d(a){};
  };
  #endif//ESSA_VERBOSE
```

⟨*double to binary string conversion* 32.5⟩+≡

```
  #ifdef ESSA_VERBOSE
  std::string tex_string(const ieee_double dble){

    std::ostringstream tex;
    tex « "\\verb2";

    unsigned long low  = (dble.l &         0xFFFFFFFFULL);
    unsigned long high = (dble.l &    0xFFFFF00000000ULL) » 32;
    int bexp           = (dble.l & 0x7FF0000000000000ULL) » 52;
    int bsig           = (dble.l & 0x8000000000000000ULL) » 63;

    if(bexp == 2047){
      if(high == 0 && low == 0){
        if ( bsig == 1 ) { tex « "-";  } else { tex « "+"; }
        tex « "inf" « std::setw(52) « 2 « "$\\hphantom{_2}$&         ";
      }else{
        tex « " NaN" « std::setw(52) « 2 « "$\\hphantom{_2}$&         ";
      }
      return tex.str();
    }else if(bexp == 0 && high == 0 && low == 0){
      tex « " 0" « std::setw(54) « 2 « "$\\hphantom{_2}$&         ";
      return tex.str();
    }

    std::bitset<32> L(low);
    std::bitset<20> H(high);
    int expo = (bexp == 0) ? -1022 : bexp - 1023;

    if ( bsig == 1 ) { tex « "-";  } else { tex « " "; }
    if ( bexp == 0 ) { tex « "0."; } else { tex « "1."; }
    tex « H « L « "2$_2$ &  $\\cdot 2^{" « std::setw(5) « expo « "}$";

    return tex.str();
  }
  #endif//ESSA_VERBOSE
```

A chunk for printing the contents of both heaps. Modifies, i.e., sorts the heaps.

⟨*print a and b* 33.2⟩≡

```
  #ifdef ESSA_VERBOSE
  std::sort(a+1,a+m+1,std::greater<double>());
  for (int j=1;j<=m;j++) {
    std::cout « "$a_" « j « "=\\,$ & "
              « tex_string(a[j]) « "\\\\" « std::endl; }
  std::sort(b+1,b+n+1,std::greater<double>());
  for (int j=1;j<=n;j++) {
    std::cout « "$b_" « j « "=\\,$ & "
              « tex_string(b[j]) « "\\\\" « std::endl; }
  std::cout « std::endl;
  #endif//ESSA_VERBOSE
```

A chunk for printing the results of the error-free transformation from revised ESSA.

⟨*print a-b=x+y* 33.3⟩≡

```
#ifdef ESSA_VERBOSE
{ double s = a[1] - b[1];
  double t = 0.0;
       if(s > 0.0) t = (a[1] - s) - b[1];
  else if(s < 0.0) t = a[1] - (b[1] + s);
  ⟨print s and t 34.1⟩
}
#endif//ESSA_VERBOSE
```

The actual output code.

⟨*print s and t* 34.1⟩≡

```
std::cout « "$  x=\\,$ & " « tex_string(s) « "\\\\" « std::endl
          « "$  y=\\,$ & " « tex_string(t) « "\\\\" « std::endl
          « std::endl « std::endl;
```

A chunk for printing the results of the error-free transformation from original ESSA.

⟨*print a-b=a-u+u-b* 34.2⟩≡

```
#ifdef ESSA_VERBOSE
{
  ⟨exponent extraction 10.2⟩
  double s,t;
  if (E == F){
    s = a[1] - b[1];
    t = 0.0;
  }else if (E > F){
    double u = ldexp(1.0,F-1);
    if( b[1] != u ) u *= 2;
    s = a[1] - u;
    t = u - b[1];
  }else{ // (E < F)
    double u = ldexp(1.0, E-1);
    if ( a[1] != u ) u *= 2;
    s  = u - b[1];
    t  = a[1] - u;
  }
  ⟨print s and t 34.1⟩
}
#endif//ESSA_VERBOSE
```

A chunk for printing the results of the error-free transformation from modified ESSA.

⟨*print a-b=a-u+u-b modified* 34.3⟩≡

```
#ifdef ESSA_VERBOSE
{
  ⟨exponent extraction 10.2⟩
  frexp(a[1],&E);
  frexp(b[1],&F);
  double s,t;
  if(E == F){
    s = a[1] - b[1];
    t = 0.0;
  }else if (E > F){
    const double msba = ldexp(1.0,E-1);
    const double u = (msba+b[1])-msba;
    s = a[1] - u;
    t = u - b[1];
  }else{ // (E < F)
    const double msbb = ldexp(1.0,F-1);
    const double u = (msbb+a[1])-msbb;
    s = u - b[1];
```

```
    t = a[1] - u;
  }
  ⟨print s and t 34.1⟩
}
#endif//ESSA_VERBOSE
```

We use the code above to print the examples from Section 4.1.

⟨*examples.cpp* 35⟩≡

```
#include <cmath>

#define ESSA_VERBOSE
#include "essa_with_rounding_mode.hpp"
#include "counting_variants_of_essa.hpp"

typedef essa_with_rounding_mode<counting_modified_essa,CGAL_FE_UPWARD>
        counting_modified_essa_in_round_up;

using namespace std;

⟨copy sum 36.1⟩

void run_essa(const double *const c,const int l){
  double* cc = new double[l];

  copy_sum(cc,c,l);
  cout « "revised_essa" « endl;
  counting_revised_essa().sign_of_sum(cc,l);

  copy_sum(cc,c,l);
  cout « "original_essa" « endl;
  counting_original_essa().sign_of_sum(cc,l);

  copy_sum(cc,c,l);
  cout « "modified_essa_in_round_up" « endl;
  counting_modified_essa_in_round_up().sign_of_sum(cc,l);

  delete[] cc;
}


int main(int /*argc*/, char * /*argv*/[]){

  double a[6] = { ldexp(5217238486728575.,5),
                  ldexp(5217238486728575.,5),
                  ldexp(8591413602653375.,1),
                 -ldexp(6980364834504767.,5),
                 -ldexp(8564466927325181.,3),
                 -ldexp(4523292875232383.,0)};
  run_essa(a,6);

  double b[6] = { ldexp(1.,0),
                  ldexp(1.,0),
                 -ldexp(1.,-2)-ldexp(1.,-54),
                 -ldexp(1.,-2),
                 -ldexp(1.,-54),
                 -ldexp(1.,-54)};
  run_essa(b,6);
```

```
  double c[4] = { ldexp(1.,1)+ldexp(1.,-50)+ldexp(1.,-51),
                 -ldexp(1., 0)-ldexp(1.,-51)-ldexp(1.,-52),
                 -ldexp(1.,-1)-ldexp(1.,-52),
                 -ldexp(1.,-1)-ldexp(1.,-52)};
  run_essa(c,4);

  return 0;
}
```

A.4 Performing Experiments

⟨*copy sum* 36.1⟩≡
```
inline void
copy_sum(double *const dest,const double *const source,const int n){
  for(int i=0;i<n;i++) dest[i] = source[i];
}
```

⟨*experiments.cpp* 36.2⟩≡
```
#include <fstream>
#include <iostream>
#include <sstream>
#include <iomanip>
#include <vector>
#include <iterator>

#include <CGAL/Timer.h>
#include <CGAL/Threetuple.h>

#include "sum_generators.hpp"
#include "variants_of_essa.hpp"
#include "counting_variants_of_essa.hpp"
#include "essa_with_rounding_mode.hpp"

using namespace std;

⟨typedefs for essa in special rounding mode 39.3⟩
⟨copy sum 36.1⟩
⟨assert consistency of essa variants 30.2⟩
⟨summands and condition number parameters 36.3⟩
```

⟨*summands and condition number parameters* 36.3⟩≡
```
int num_sums;
int repetitions;
int max_summands;
vector<int> summands;
vector<int> exponents;
```

⟨*summands and condition number parameters* 36.3⟩+≡
```
string trimline(string s){
 s.erase(s.find_first_of('#'));
 if(s.length() == 0) return s;
 if(s.at(0) == ' ') s.erase(0,s.find_first_not_of(' '));
 if(s.length() == 0) return s;
 if(s.at(s.length()-1) == ' ') s.erase(s.find_last_not_of(' ')+1);
 return s;
}
```

⟨*get line from config file* 37.1⟩≡
```
getline(cfg,line);
if(cfg.fail()){
   cout « "Error while reading '" « filename « "'." « endl;
   exit(1);
}
linestrm.clear();
linestrm.str(trimline(line));
```

⟨*error while parsing line* 37.2⟩≡
```
if(linestrm.fail()){
  cout « "Error while reading '" « filename
       « "' in line '" « line « "'." « endl;
  exit(1);
}
```

⟨*summands and condition number parameters* 36.3⟩+≡
```
void read_config_file(string filename){
  ifstream cfg(filename.c_str());
  if(!cfg){
    cout « "Could not open '" « filename « "' for reading." « endl;
    exit(1);
  }

  string       line;
  stringstream linestrm;

  ⟨get line from config file 37.1⟩
  linestrm » num_sums;
  ⟨error while parsing line 37.2⟩

  ⟨get line from config file 37.1⟩
  linestrm » repetitions;
  ⟨error while parsing line 37.2⟩

  ⟨get line from config file 37.1⟩
  while(!linestrm.eof()){
    int s;
    linestrm » s;
    ⟨error while parsing line 37.2⟩
    summands.push_back(s);
  }
  max_summands = *max_element(summands.begin(),summands.end());

  ⟨get line from config file 37.1⟩
  while(!linestrm.eof()){
    int s;
    linestrm » s;
    ⟨error while parsing line 37.2⟩
    exponents.push_back(s);
  }

  cfg.close();
}
```

⟨*experiments.cpp* 36.2⟩+≡
```
void
flush_results(vector<vector<vector<double> > >& results,
```

```
                string zname,
                vector<int>& znumbers,
                vector<int>& xnumbers,
                bool transpose = false){

  for(size_t i=0;i<znumbers.size();i++){

    stringstream filename;
    filename « "results_" « zname « "_"
             « setfill('0') « setw(5) « znumbers[i] « "_"
             « setw(5) « num_sums « "_"
             « setw(5) « repetitions;

    cout « filename.str() «  endl;
    ofstream ostrm(filename.str().c_str());
    ostrm.setf(ofstream::fixed);
    ostrm.precision(2);

    ostrm « "# " « zname « "=" « znumbers[i]
          « " sums=" « num_sums
          « " repetitions=" « repetitions « endl;

    for(size_t j=0;j<xnumbers.size();j++){
      ostrm « setw(3) « j « " " « setw(5) « xnumbers[j] « " ";

      vector<double>& ynumbers = transpose ? results[j][i] : results[i][j];

      for(size_t k=0;k<ynumbers.size();k++)
        ostrm « setw(6) « ynumbers[k] « " ";
      ostrm « endl;
    }
  }
}
```

⟨*experiments.cpp* 36.2⟩+≡

```
template <class SSA> double
measure_time(vector<double*>& sums,const int num_summands){

  double *sum = new double[num_summands];
  CGAL::Timer tmr;
  tmr.start();

  for(size_t i=0;i<sums.size();i++){
    for(int j=0;j<repetitions;j++){
      copy_sum(sum,sums[i],num_summands);
      SSA().sign_of_sum(sum,num_summands);
    }
  }

  tmr.stop();
  delete[] sum;
  return tmr.time();
}
```

⟨*experiments.cpp* 36.2⟩+≡

```
template <class SSA> CGAL::Threetuple<double>
measure_steps(vector<double*>& sums,const int num_summands){
```

```
    double *sum = new double[num_summands];
    CGAL::Threetuple<double> steps(0,53.0*num_summands*num_summands,0);

    for(size_t i=0;i<sums.size();i++){
        copy_sum(sum,sums[i],num_summands);
        const double s = SSA().sign_of_sum(sum,num_summands);
        steps.e0 += s;
        steps.e1 = min(steps.e1,s);
        steps.e2 = max(steps.e2,s);
    }

    steps.e0 /= sums.size();
    delete[] sum;
    return steps;
  }
```

⟨*generate nonzero sums* 39.1⟩≡

```
  for(size_t k=0;k<sums.size();k++){
    generate_random_sum(sums[k],summands[i],exponents[j]);
    assert_essa(sums[k],summands[i]);
  }
```

⟨*modify sums to zero* 39.2⟩≡

```
  for(size_t k=0;k<sums.size();k++){
    modify_sum_to_zero(sums[k],summands[i]);
    assert_essa(sums[k],summands[i]);
  }
```

⟨*typedefs for essa in special rounding mode* 39.3⟩≡

```
  typedef essa_with_rounding_mode<revised_essa,CGAL_FE_TOWARDZERO>
          revised_essa_in_round_toward_zero;
  typedef essa_with_rounding_mode<modified_essa,CGAL_FE_UPWARD>
          modified_essa_in_round_up;
  typedef essa_with_rounding_mode<modified_essa,CGAL_FE_TOWARDZERO>
          modified_essa_in_round_toward_zero;

  typedef essa_with_rounding_mode<counting_modified_essa,CGAL_FE_UPWARD>
          counting_modified_essa_in_round_up;
  typedef essa_with_rounding_mode<counting_revised_essa,CGAL_FE_TOWARDZERO>
          counting_revised_essa_in_round_toward_zero;
```

⟨*perform experiments for sums* 39.4⟩≡

```
  *resi++ = measure_time<modified_essa_in_round_up        >(sums,summands[i]);
  *resi++ = measure_time<original_essa                    >(sums,summands[i]);
  *resi++ = measure_time<revised_essa                     >(sums,summands[i]);
  *resi++ = measure_time<revised_essa_in_round_toward_zero>(sums,summands[i]);

  steps = measure_steps<counting_modified_essa_in_round_up        >(sums,summands[i]);
  *resi++ = steps.e0; *resi++ = steps.e1; *resi++ = steps.e2;
  steps = measure_steps<counting_original_essa                    >(sums,summands[i]);
  *resi++ = steps.e0; *resi++ = steps.e1; *resi++ = steps.e2;
  steps = measure_steps<counting_revised_essa                     >(sums,summands[i]);
  *resi++ = steps.e0; *resi++ = steps.e1; *resi++ = steps.e2;
  steps = measure_steps<counting_revised_essa_in_round_toward_zero>(sums,summands[i]);
  *resi++ = steps.e0; *resi++ = steps.e1; *resi++ = steps.e2;
```

⟨*experiments.cpp* 36.2⟩+≡

```
int main(int argc, char* argv[]){

  if(argc < 2){
    cout « "usage: " « argv[0] « " config_file" « endl;
    exit(1);
  }

  read_config_file(argv[1]);
  cout « "num_sums: " « num_sums
       « " repetitions: " « repetitions « endl;


  unsigned short seed[3] = {0x7d1b, 0xa934, 0xbf10};
  time_t s = time(0);
  seed[0] = s » 16;
  seed[2] = (s & 0x0000FFFF);
  seed48(seed);

  vector<double*> sums(num_sums);
  for(size_t i=0;i<sums.size();i++) sums[i] = new double[max_summands];

  vector<vector<double> >          tmp2(exponents.size(),vector<double>());
  vector<vector<vector<double> > > results(summands.size(),tmp2);
  tmp2.resize(0);

  for(int i=static_cast<int>(summands.size())-1;i>=0;i-){
  //for(int i=0;i<summands.size();i++){
  cout « setw(5) « summands[i] « " :" « flush;
    for(int j=static_cast<int>(exponents.size())-1;j>=0;j-){
    //for(int j=0;j<exponents.size();j++){
      cout « setw(5) « exponents[j] « flush;

      CGAL::Threetuple<double> steps;
      back_insert_iterator<vector<double> > resi(results[i][j]);

      ⟨generate nonzero sums 39.1⟩
      ⟨perform experiments for sums 39.4⟩
      ⟨modify sums to zero 39.2⟩
      ⟨perform experiments for sums 39.4⟩

    }
    cout « endl;
  }

  flush_results(results,"summands",summands,exponents,false);
  flush_results(results,"exponent",exponents,summands,true);

  for(size_t i=0;i<sums.size();i++) delete[] sums[i];

  return 0;
}
```

A.5 Plotting experimental results

⟨*plotting.cpp* 40.2⟩≡

```
#include <iostream>
```

```
#include <fstream>
#include <sstream>
#include <iomanip>
#include <vector>

using namespace std;

⟨summands and condition number parameters 36.3⟩
```

⟨plotting.cpp 40.2⟩+≡

```
void
plot(string zname, int znumber, vector<int>& xitems,
     string type, int offset, bool plotkey){

    stringstream infile;
    infile « "results_" « zname « "_"
           « setfill('0') « setw(5) « znumber « "_"
           « setw(5) « num_sums « "_"
           « setw(5) « repetitions;

    stringstream outfile;
    outfile « "images/" « type « "_" « zname « "_"
            « setfill('0') « setw(5) « znumber « ".tex";

    ofstream gpfile("tmp.gp");

    ⟨gnuplot setup 41.2⟩

    if(type == "zero_times" || type == "nonzero_times"){
      ⟨gnuplot plotting times 42.1⟩
    }else{
      ⟨gnuplot plotting steps 42.2⟩
    }

    gpfile.close();
    if(system("gnuplot tmp.gp")) exit(1);
    cout « outfile.str() « endl;
}
```

⟨gnuplot setup 41.2⟩≡

```
gpfile « "set terminal epslatex color size 8.3cm, 5.5cm font \"\" 8" « endl
       « "set lmargin 4" « endl
       « "set output \"tmp.tex\"" « endl
       « "set key off" « endl
       « endl
       « endl
       « "set style line 1 lt 1 lw 2 lc rgb '#33ff33' pt 9" « endl
       « "set style line 2 lt 1 lw 2 lc rgb '#009900' pt 7" « endl
       « "set style line 3 lt 1 lw 2 lc rgb '#ff9933' pt 2" « endl
       « "set style line 4 lt 1 lw 2 lc rgb '#ff3300' pt 1" « endl
       « "set style line 5 lt 1 lw 2 lc rgb '#8f008f' pt 5" « endl
       « endl;

gpfile « "set xtics (";
for(size_t j=0;j<xitems.size()-1;j++) gpfile « "\"" « xitems[j] « "\" " « j « ",";
gpfile « "\"" « xitems[xitems.size()-1] « "\"" « xitems.size()-1 « ") nomirror" « endl;

if(type == "zero_steps" || type == "nonzero_steps"){
```

```
  if(zname == "summands") gpfile « "set ytics autofreq " « znumber/2 « endl;
  else gpfile « "set ytics autofreq " « xitems[xitems.size()/2] « endl;
}
```

⟨*gnuplot plotting times* 42.1⟩≡

```
if(plotkey){
  gpfile « "set key left top Left reverse invert spacing 1.2" « endl;
}else{
  gpfile « "set label \"";
  if(zname == "summands") gpfile « znumber « " " « zname;
  if(zname == "exponent") gpfile « "exponent range 0..." « znumber;
  gpfile « "\" at first 0, graph 0.9" « endl;
}
gpfile « "set xrange [" « -0.2 « ":" « xitems.size()-0.8 « "]" « endl
       « "set yrange [0:] writeback" « endl
       « endl
       « "set output \"" « outfile.str() « "\"" « endl
       « "plot \'" « infile.str() « "\' using 1:" « offset+3
       « " title \"\\\\footnotesize \\\\rtzESSA\" with linespoints ls 3,\\" « endl
       « "     \'" « infile.str() « "\' using 1:" « offset+2
       « " title \"\\\\footnotesize \\\\revESSA\" with linespoints ls 4,\\" « endl
       « "     \'" « infile.str() « "\' using 1:" « offset+0
       « " title \"\\\\footnotesize \\\\modESSA\" with linespoints ls 5,\\" « endl
       « "     \'" « infile.str() « "\' using 1:" « offset+1
       « " title \"\\\\footnotesize \\\\impESSA\" with linespoints ls 2 " « endl
       « endl;
```

⟨*gnuplot plotting steps* 42.2⟩≡

```
if(plotkey) gpfile « "set key left bottom Left reverse invert spacing 1.2" « endl;
gpfile « "set label \"";
if(zname == "summands") gpfile « znumber « " " « zname;
if(zname == "exponent") gpfile « "exponent range 0..." « znumber;
gpfile « "\" at first 0, graph 0.9" « endl;
gpfile « "set xrange [" « -0.4 « ":" « xitems.size()-0.6 « "]" « endl
       « "set yrange [0:]" « endl
       « endl
       « "set output \"" « outfile.str() « "\"" « endl
       « "plot \'" « infile.str() « "\' using ($1+0.21):"
       « offset+9 « ":" « offset+10 « ":" « offset+11
       « " title \"\\\\footnotesize \\\\rtzESSA\" with errorbars ls 3,\\" « endl
       « "     \'" « infile.str() « "\' using ($1+0.07):"
       « offset+6 « ":" « offset+7 « ":" « offset+8
       « " title \"\\\\footnotesize \\\\revESSA\" with errorbars ls 4,\\" « endl
       « "     \'" « infile.str() « "\' using ($1-0.07):"
       « offset+0 « ":" « offset+1 « ":" « offset+2
       « " title \"\\\\footnotesize \\\\modESSA\" with errorbars ls 5,\\" « endl
       « "     \'" « infile.str() « "\' using ($1-0.21):"
       « offset+3 « ":" « offset+4 « ":" « offset+5
       « " title \"\\\\footnotesize \\\\impESSA\" with errorbars ls 2 " « endl
       « endl;
```

⟨*plotting.cpp* 40.2⟩+≡

```
int main(int argc,char*argv[]){

  if(argc < 2){
    cout « "usage: " « argv[0] « " configfile" « endl;
    return 1;
  }
```

```
  read_config_file(argv[1]);

  string types[4] = {"nonzero_times","nonzero_steps","zero_times","zero_steps"};
  int  offsets[4] = {              3,              7,          19,          23};

  for(int j = 0;j<4;j++){
    for(size_t i=0;i<summands.size();i++)
      plot("summands",summands[i],exponents,types[j],offsets[j],
           summands[i] == summands[summands.size()-1]);

    for(size_t i=0;i<exponents.size();i++)
      plot("exponent",exponents[i],summands,types[j],offsets[j],
           exponents[i] == exponents[exponents.size()-1]);
  }
  return 0;
}
```