MANCHESTER
1824

The University
of Manchester

Algorithms for stochastically rounded elementary arithmetic operations in IEEE 754 floating-point arithmetic

Fasi, Massimiliano and Mikaitis, Mantas

2020

MIMS EPrint: **2020.9**

Manchester Institute for Mathematical Sciences
School of Mathematics

The University of Manchester

Reports available from: http://eprints.maths.manchester.ac.uk/
And by contacting: The MIMS Secretary
School of Mathematics
The University of Manchester
Manchester, M13 9PL, UK

ISSN 1749-9097

Algorithms for Stochastically Rounded Elementary Arithmetic Operations in IEEE 754 Floating-Point Arithmetic*

Massimiliano Fasi[†] Mantas Mikaitis[‡]

We present algorithms for performing the five elementary arithmetic operations ($+$, $-$, $\times$, $\div$, and $\sqrt{\ }$) in floating point arithmetic with stochastic rounding, and demonstrate the value of these algorithms by discussing various applications where stochastic rounding is beneficial. The algorithms require that the hardware be compliant with the IEEE 754 floating-point standard and that a floating-point pseudorandom number generator be available. The goal of these techniques is to emulate stochastic rounding when the underlying hardware does not support this rounding mode, as is the case for most existing CPUs and GPUs. Simulating stochastically rounded floating-point operations can be used to explore the behavior of this rounding, as well as to develop applications before hardware with stochastic rounding is available—once such hardware becomes available, the proposed algorithms can be replaced by calls to the relevant hardware routines. When stochastically rounding double precision operations, the algorithms we propose are between 7.3 and 19 times faster than the implementations that use the GNU MPFR library to simulate extended precision. We test our algorithms on various problems where stochastic rounding is expected to bring advantages, which includes summation algorithms and ordinary differential equation solvers.

Key words: floating-point arithmetic, error-free transformation, stochastic rounding, numerical analysis, numerical algorithm, IEEE 754

---

*Version of October 29, 2020. This work has been submitted to the IEEE for possible publication. Copyright may be transferred without notice, after which this version may no longer be accessible.

[†]School of Science and Technology, Örebro University, Örebro, 701 82, Sweden. This work was carried out when this author was at the Department of Mathematics, University of Manchester, Manchester, M13 9PL, UK (massimiliano.fasi@oru.se).

[‡]Department of Mathematics, University of Manchester, Manchester, M13 9PL, UK (mantas.mikaitis@manchester.ac.uk).

1 Introduction

The IEEE 754-1985 standard for floating-point arithmetic specifies four rounding modes: the default round-to-nearest, which we denote by RN, and three directed rounding modes, round-toward-$+\infty$, round-toward-$-\infty$, and round-toward-zero, which we denote by RU, RD, and RZ, respectively. The 2008 revision of the standard handles rounding by means of the attribute *rounding-direction*, which can take any of five possible values: *roundTiesToEven* and *roundTiesToAway* for round-to-nearest with two different tie-breaking rules, and *roundTowardPositive*, *roundTowardNegative*, and *RoundTowardZero* for directed rounding. The standard states, however, that it is not necessary for a binary format to implement roundTiesToAway, thus confirming that only four rounding modes are necessary for a floating-point hardware implementation to be IEEE compliant. The 2019 revision of the standard [1] does not introduce any major changes to this section, but recommends the use of *roundTiesTowardZero* for augmented operations [1, Sec. 9.5].

These five rounding modes are deterministic, in that the rounded value of a number is determined solely by the value of that number in exact arithmetic, and an arbitrary sequence of rounded elementary arithmetic operations will always produce the same result. Here we focus on *stochastic rounding*, a non-deterministic rounding mode that randomly chooses in which direction to round a number that cannot be represented exactly in the working precision. To the best of our knowledge, this rounding mode was first mentioned by Forsythe [2]. Informally speaking, the goal of stochastic rounding is to round a real number x to a nearby floating-point number y with a probability that depends on the proximity of x to y, that is, on the quantity $|x - y|$. We formalize this concept in Section 4.

Despite the similar name, stochastic rounding should not be confused with *stochastic arithmetic* [3], a custom rounding mode in which each number that is not exactly representable in the current precision is rounded to either of the closest floating-point numbers with equal probability. Stochastic arithmetic is used by the CADNA library [4] to estimate the propagation of rounding errors in floating-point programs. A similar device for the experimental analysis of rounding errors is *Monte Carlo arithmetic* [5], a technique that comprises both stochastic arithmetic, as used by the CADNA library, and stochastic rounding, which we consider here. Monte Carlo arithmetic is used by tools such as Verificarlo [6] and Verrou [7] in order to estimate the impact of round-off errors in floating-point computation, but we are not aware of any examples of use in numerical software. All these tools propose to run an a program multiple times using stochastic or Monte Carlo arithmetic, sample the result, and then use this set of answers to draw conclusions on the propagation of rounding errors and the numerical stability of the same code when run with deterministic rounding. None of them, however, considers the use of stochastic rounding for alleviating rounding errors when running a program (even if only once) for its intended purpose.

Stochastic rounding is inherently more expensive than the standard IEEE rounding modes, as it requires the generation of a floating-point pseudorandom number, and its advantages might not be entirely obvious, at first. Round-to-nearest maps an exact number to the closest floating-point number in the floating-point number system in use, and always produces the smallest possible roundoff error. In doing so, however, it discards most of the data encapsulated in the bits that are rounded off. Stochastic rounding aims to capture more of the information stored in the least significant of the bits that are lost when

rounding. This benefit should be understood in a statistical sense: stochastic rounding may produce an error larger than that of round-to-nearest on a single rounding operation, but over a large number of roundings it may help to obtain a more accurate result, as errors of opposite signs cancel out. This rounding strategy is particularly effective at alleviating stagnation [8], a phenomenon that often occurs when computing the sum of a large number of terms that are small in magnitude. A sum stagnates when the summands become so small—compared with the partial sum—that their values are "swamped" [9], causing a dramatic increase in forward error. We examine stagnation experimentally in Section 8.

2 Motivation

Stochastically rounded arithmetic operations are widely used in fixed- and floating-point arithmetic, both in software and hardware. The need for efficient software implementations of stochastic rounding arises in several contexts.

First of all, floating-point operations with stochastic rounding are useful in tools for the verification of rounding errors such as Verificarlo [6] and Verrou [7], which are used in industrial *Verification & Validation* processes. Févotte and Lathuilière [7] discuss plans for using Verrou in numerical simulations of electricity production units, in order to analyze the propagation or rounding errors in floating-point arithmetic, detect their origin in the source code, and verify that they are kept within some acceptable limits throughout the simulation.

Secondly, stochastic rounding has been shown to reduce the worst-case bounds on the backward error of various numerical linear algebra algorithms [10]. Connolly, Higham, and Mary [10] show that if stochastic rounding is used then 1) rounding errors are mean-independent random variables with zero mean, and 2) the worst-case bound on the backward error of inner products can be lowered from nu, which holds when round-to-nearest is used, to $\sqrt{n}u$, where n is the problem size and u the unit roundoff of the floating-point arithmetic in use. Advantages of stochastic rounding were also shown in other types of numerical algorithms such as, for example, those used for solving the ordinary differential equations arising in the Izhikevich neuron model [11]. The general floating-point format simulator developed by Higham and Pranesh [12] includes stochastic rounding because, as the they point out themselves, there is a need to better understand its behavior.

Furthermore, stochastic rounding is being increasingly used in machine learning [13–18]. When training neural networks, in particular, it can help compensate for the loss of accuracy caused by reducing the precision at which deep neural networks are trained in fixed-point [14] as well as floating-point [17] arithmetic. Graphcore Intelligence Processing Units (IPUs) include stochastic rounding in their mixed-precision matrix multiplication hardware [19].

Lastly, stochastic rounding plays an important role in neuromorphic computing. Intel uses it to improve the accuracy of biological neuron and synapse models in the neuromorphic chip Loihi [20]. The SpiNNaker2 neuromorphic chip [21] will be equipped with a hardware rounding accelerator designed to support, among others, fast stochastic rounding. In general, various patents from AMD, NVIDIA, and other companies propose hardware implementations of stochastic rounding [22–24]. Of particular interest is a patent from IBM [25], in which the entropy from the registers is proposed as a source of randomness.

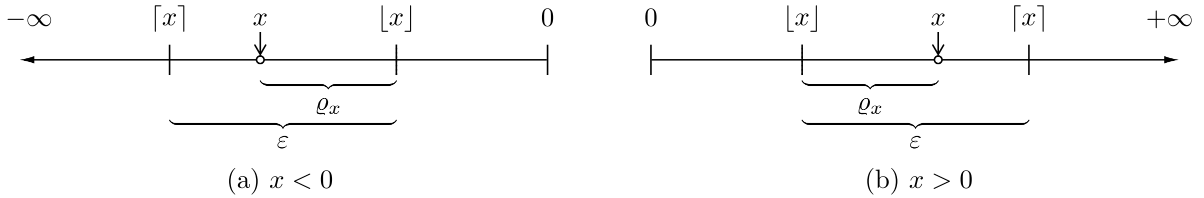


Figure 3.1: Illustration of the stochastic rounding of a negative (left) or positive (right) real number x which sits in-between two floating-point numbers $\lceil x \rceil$ and $\lfloor x \rfloor$. The number x is rounded to $\lceil x \rceil$ with probability r_x and to $\lfloor x \rfloor$ with probability $1 - r_x$. The residual ϱ_x in the diagrams and the probability r_x are connected by the identity in (4.3).

3 Contributions

Our contribution is twofold: on the one hand, we present algorithms for emulating stochastic rounding of addition/subtraction, multiplication, division, and square root; on the other, we discuss some examples in which using stochastic rounding can yield more accurate solutions, and even achieve convergence in cases where round-to-nearest would lead numerical methods to diverge.

In order to round the result of an arithmetic operation stochastically, it is necessary to know the error between the exact result of the computation and its truncation to working precision. Today's CPUs and GPUs typically do not return this value to the software layer, and the most common technique to emulate stochastic rounding via software relies on the use of two levels of precision. The operation is performed at higher precision and the direction of the rounding is chosen with probability proportional to the distance between this reference and its truncation to the target precision. The MATLAB `chop` function,[1] for instance, follows this approach [12].

In general, this strategy cannot guarantee an accurate implementation of stochastic rounding unless an extremely high precision is used to perform the computation. The sum of the two binary32 numbers 2^{127} and 2^{-126}, for instance, would require a floating point system with at least 253 bits of precision in order to be represented exactly, and up to 2045 bits may be necessary for binary64. The requirements would be even higher if subnormal numbers were allowed. This is hardly an issue in practice, and it is easy to check, theoretically as well as experimentally, that as long as enough extra digits of precision are used the results obtained with `chop` differ from those obtained using full precision only in a negligible portion of cases [10].

The main drawback of this technique is that it requires the availability of an efficient mechanism to perform high-precision computation, which may not be a viable option if one wants to simulate stochastic rounding when using the highest precision available in hardware.

In Section 6 we show how the five elementary arithmetic operations can be implemented stochastically with the same guarantees as `chop` without resorting to higher precision. This approach brings a performance gain, as we show in Section 7. In Section 8, we explore

[1] `https://github.com/higham/chop`

three applications showing that stochastic rounding may be more effective than the four rounding modes defined by the IEEE 754 standard. We summarize our contribution and point at directions for future work in Section 9.

4 Stochastic rounding

Let $\mathcal{F}$ be a normalized binary floating-point number system with p digits of precision and maximum exponent $e_{\max}$, and let $\varepsilon := 2^{1-p}$ be the *machine epsilon* of $\mathcal{F}$. The number $x \in \mathcal{F}$ can be written as $x = (-1)^s \cdot 2^e \cdot m$, where $s \in \{0, 1\}$ is the *sign bit* of x, the *exponent* e is an integer between $e_{\min} := 1 - e_{\max}$ and $e_{\max}$ inclusive, and the *significand* $m \in [0, 2)$ can be represented exactly with p binary digits, of which only $p - 1$ are stored explicitly. We remark that since $\mathcal{F}$ is normalized m can be smaller than 1 only when $e = e_{\min}$. For a number $x \in \mathcal{F}$ we denote s and e by $\operatorname{sign}(x)$ and $\operatorname{exponent}(x)$, respectively.

As mentioned in the previous section, here we consider stochastic rounding. In order to give a precise definition, let us denote the *truncation* of a number $m \in [0, 2)$ to its p most significant digits by $\lfloor m \rfloor := \operatorname{sign}(m) \cdot 2^{1-p} \lfloor 2^{p-1} |m| \rfloor$. The function $\mathrm{SR} : \mathbb{R} \to \mathcal{F}$ is a stochastic rounding if one has that (we use "w. p." as a shorthand for "with probability")

$$\mathrm{SR}(x) = \begin{cases} (-1)^s \cdot 2^e \cdot \lfloor m \rfloor, & \text{w. p. } 1 - r_x, \\ (-1)^s \cdot 2^e \cdot (\lfloor m \rfloor + \varepsilon), & \text{w. p. } r_x, \end{cases} \tag{4.1}$$

where

$$r_x = \frac{m - \lfloor m \rfloor}{\varepsilon}, \tag{4.2}$$

for any real number x with absolute value between the smallest and the largest representable numbers in $\mathcal{F}$. We note that $m - \lfloor m \rfloor \in [0, \varepsilon)$, which implies that $r_x \in [0, 1)$. The definition is illustrated pictorially in Fig. 3.1.

We note that this definition gives the desired result if x is subnormal. Let $x_{\max}$ be the largest floating-point number representable in $\mathcal{F}$. If $|x| \geq x_{\max}$, then the definition (4.1) cannot be used, as $(-1)^s \cdot 2^e \cdot (\lfloor m \rfloor + \varepsilon)$ is not representable in $\mathcal{F}$. For consistency with the informal definition of stochastic rounding, we could assume that if x is larger than $x_{\max}$ in absolute value, then $\mathrm{SR}(x) = \operatorname{sign}(x) \cdot x_{\max}$ rather than $\mathrm{SR}(x) = \operatorname{sign}(x) \cdot \infty$. In practice, we can have different overflow behavior based on intermediate rounding modes used to simulate stochastic rounding—we discuss this for each algorithm below. The quantity r_x in (4.2) is proportional to the rounding error when rounding toward zero, since

$$\begin{aligned} x - \mathrm{RZ}(x) &= \operatorname{sign}(x) \cdot 2^e \cdot (m - \lfloor m \rfloor) \\ &= \operatorname{sign}(x) \cdot 2^e \cdot \varepsilon \cdot r_x \\ &=: \varrho_x. \end{aligned} \tag{4.3}$$

As ϱ_x depends only on x, we call it the *residual* of x.

We now discuss how to implement (4.1). Let $X \sim \mathcal{U}_I$ denote a random variable X that follows the uniform distribution over the interval $I \subset \mathbb{R}$. Then for any $x \in \mathbb{R}$ we have that

$$\mathrm{SR}(x) = \begin{cases} (-1)^s \cdot 2^e \cdot \lfloor m \rfloor, & X \geq r_x, \\ (-1)^s \cdot 2^e \cdot (\lfloor m \rfloor + \varepsilon), & X < r_x, \end{cases} \tag{4.4}$$

where r_x is as in (4.2) and $X \sim \mathcal{U}_{[0,1)}$. Using the strict inequality for the second case ensures that if $x \in \mathcal{F}$ then $\mathrm{SR}(x) = x$, since $r_x = 0$ if x is exactly representable in $\mathcal{F}$.

Note that (4.4) can equivalently be rewritten as

$$\mathrm{SR}(x) = \begin{cases} (-1)^s \cdot 2^e \cdot \lfloor m \rfloor, & X' > r_x, \\ (-1)^s \cdot 2^e \cdot (\lfloor m \rfloor + \varepsilon), & X' \leq r_x, \end{cases} \tag{4.5}$$

for $X' \sim \mathcal{U}_{(0,1]}$. An alternative way of implementing (4.1) can be obtained by substituting $Y = 1 - X$ in (4.5), which yields

$$\mathrm{SR}(x) = \begin{cases} (-1)^s \cdot 2^e \cdot \lfloor m \rfloor, & Y < 1 - r_x, \\ (-1)^s \cdot 2^e \cdot (\lfloor m \rfloor + \varepsilon), & Y \geq 1 - r_x, \end{cases} \tag{4.6}$$

where $Y \sim \mathcal{U}_{[0,1)}$. We will rely on both (4.4) and (4.6) in later sections.

Definitions (4.4) and (4.6) are equivalent not only if X and Y are continuous random variables, but also in the discrete case. In particular, it is possible to show that both definitions round up for $2^p r_x$ cases out of 2^p. This is illustrated for the case $p = 2$ in Table 4.1, and can be proven by induction on the structure of the table for any p.

Here we provide only the proof for (4.4), that for (4.6) is analogous and therefore omitted. For $p = 1$, the result can be verified by exhaustion. Now we consider the inductive step $p = k$. For a k-digit floating-point number y, let us denote by $t(y)$ the integer obtained by interpreting the string containing all but the leading bit in the significand of y as an integer. If the leading bit of y is 1 then $y = 2^{-1} + 2^{-k} t(y)$, whereas if the leading bit of y is 0 then $y = 2^{-k} t(y)$.

If the leading bit of r_x is 0, then x is rounded up only when the leading bit of X is 0 and $t(r_x) > t(X)$, which by inductive hypothesis happens in $2^k r_x$ cases out of 2^{k-1}. Taking into account the 2^{k-1} cases in which the leading bit of X is 1 and x is rounded down, we obtain that x is rounded up in $2^k r_x$ cases out of 2^k.

If the leading bit of r_x is 1, on the other hand, x will be rounded up if 1) the leading bit of the significand of X is 0, which happens in 2^{k-1} cases; or 2) the leading bit of X is 1 but $t(r_x) > t(X)$. By inductive hypothesis, the former happens in $2^k t(r_x)$ cases out of 2^{k-1}, and accounting for the 2^{k-1} cases in which the leading bit of X is 0 and x is rounded up, we obtain that even in this case x is rounded up in $2^{k-1} + t(r_x) = 2^k r_x$ cases out of 2^k.

5 TwoSum and TwoProdFMA algorithms

The IEEE 754-2019 standard for floating-point arithmetic [1] includes, among the new recommended operations, three *augmented operations*: *augmentedAddition*, *augmented-Subtraction*, and *augmentedMultiplication*. These homogeneous operations take as input two values in any binary floating-point format and return two floating-point numbers in the same format: a correctly rounded result and an exact rounding error. The functionality of the new recommended operations in the standard are very similar to the classical FastTwoSum, TwoSum, and TwoProdFMA algorithms. The only difference is that the tie-breaking rule in round-to-nearest, which is ties-toward-zero rather ties-to-even [26, 27]. For simplicity, we will refer to these algorithms as *augmented addition/multiplication algorithms* (they are also called *error-free transformations*).

Table 4.1: Demonstration of stochastic roundings in a 2-bit case for every pair of X/Y and r_x. The table on the left considers (4.4), whereas (4.6) is shown on the right. The direction of the arrows corresponds to rounding directions, $\downarrow$ for round-toward $-\infty$ and $\uparrow$ for round-toward $+\infty$. Note that corresponding columns in the two tables have the same number of arrows pointing upward and arrows pointing downward: this shows that for any given r_x the probability of rounding up or down does not depend on which definition is used.

X \ r_x	0.00	0.01	0.10	0.11
0.00	$\downarrow$	$\uparrow$	$\uparrow$	$\uparrow$
0.01	$\downarrow$	$\downarrow$	$\uparrow$	$\uparrow$
0.10	$\downarrow$	$\downarrow$	$\downarrow$	$\uparrow$
0.11	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$

Y \ r_x	0.00	0.01	0.10	0.11
0.00	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
0.01	$\downarrow$	$\downarrow$	$\downarrow$	$\uparrow$
0.10	$\downarrow$	$\downarrow$	$\uparrow$	$\uparrow$
0.11	$\downarrow$	$\uparrow$	$\uparrow$	$\uparrow$

How to perform these tasks efficiently is a well-understood problem. Algorithms for augmented addition (and thus subtraction) and augmented multiplication are discussed in [28, Sec. 4.3] and [28, Sec. 4.4], respectively. The former can be performed efficiently by using the function TwoSum in Algorithm 5.1, due to Knuth [29, Th. B] and Møller [30], which for $\circ = \text{RN}$ computes the correctly rounded sum and the rounding error at the cost of six floating-point operations. If the two summands are ordered by decreasing magnitude, this task can be achieved more efficiently by using Dekker's FastTwoSum [31], which requires only 3 operations in round-to-nearest.

The names we use for these two routines were originally proposed by Shewchuck [32]. Boldo, Grillat, and Muller [33] explore the robustness of FastTwoSum and TwoSum with rounding modes other than round-to-nearest. They conclude that both algorithms return a very accurate approximation of the error of addition, and that FastTwoSum is immune to overflow in all the internal steps, while TwoSum is not only in rare cases, as long as the main addition does not overflow.

When dealing with augmented multiplication, extra care is required, as in this case it is necessary to ensure that underflow does not occur. In [28, Sec. 4.4] it is shown that if $a, b \in \mathcal{F}$ and

$$\text{exponent}(a) + \text{exponent}(b) \geq e_{\min} + p - 1, \tag{5.1}$$

then $\tau = a \cdot b - \circ(a \times b)$, with $\circ \in \{\text{RN}, \text{RD}, \text{RU}, \text{RZ}\}$, also belongs to $\mathcal{F}$. In other words, the error of a floating-point product is exactly representable in the same format as its arguments. If an FMA (*Fused Multiply-Add*) instruction is available, augmented multiplication can be realized very efficiently with the function TwoProdFMA in Algorithm 5.2, which requires only two floating-point operations and guarantees that if a and b satisfy (5.1), then $\sigma + \tau = a \cdot b$ regardless of the rounding mode used for the computation.

Algorithm 5.1: TwoSum augmented addition.

```
1 function TwoSum(a ∈ F, b ∈ F, ∘ : ℝ → F)
    Compute σ, τ ∈ F s.t. σ + τ = a + b.
2   σ ← ∘(a + b);
3   a′ ← ∘(σ − b);
4   b′ ← ∘(σ − a′);
5   δ_a ← ∘(a − a′);
6   δ_b ← ∘(b − b′);
7   τ ← ∘(δ_a + δ_b);
8   return (σ, τ);
```

Algorithm 5.2: TwoProdFMA augmented multiplication.

```
1 function TwoProdFMA(a ∈ F, b ∈ F, ∘ : ℝ → F)
    If a, b satisfy (5.1), compute σ, τ ∈ F s.t. σ + τ = a · b.
2   σ ← ∘(a × b);
3   τ ← ∘(a × b − σ);
4   return (σ, τ);
```

If an FMA is not available, another algorithm, due to Dekker [31], may be used to compute σ and τ. This algorithm requires 16 floating-point operations, and is therefore considerably more expensive than TwoProdFMA, which requires only 2. We do not reproduce the algorithm here, and in our pseudocode we denote by TwoProdDek the function that has the same interface as TwoProdFMA and implements [28, Alg. 4.10]. This algorithm also requires that condition (5.1) holds, but has been proven to work correctly only when round-to-nearest is used.

6 Operations with stochastic rounding

In order to round a real number x according to the definition in Section 4, we need to know r_x in (4.2). We refer the reader to Fig. 3.1 for a graphical demonstration. It may be possible to compute this quantity exactly, if the operation producing x is carried out in higher precision, but the value of r_x (or ϱ_x) is not available when one wishes to round the result of an arithmetic operation performed in hardware in the same precision as the arguments. When rounding the sum of two binary64 numbers of different magnitude, for example, in order to match the exponent of the two summands one must shift the fraction of the smaller operand in absolute value to the right, causing roundoff bits to appear. These leftmost bits form r_x, a quantity that is used in hardware for rounding purposes but is usually not returned to the user (this, however, might change in the future with the introduction of augmented operations in the IEEE 754-2019 standard [1]). In order to manipulate the rounded sum of the two values in a way that simulates stochastic rounding, we need to obtain r_x in order to control the probability of rounding up and down. The algorithms for basic operations below use error-free transformations or other techniques for

Algorithm 6.1: Stochastically rounded addition.

```
1  function ADD(a ∈ 𝓕, b ∈ 𝓕)
     Compute ϱ = SR(a + b) ∈ 𝓕.
2    Z ← rand();
3    (σ, τ) ← TWOSUM(a, b, RN);
4    η ← get_exponent(RZ(a + b));
5    π ← sign(τ) × Z × 2^η × ε;
6    if τ ≥ 0 then
7      ∘ = RD;
8    else
9      ∘ = RU;
10   ϱ ← ∘(⋄(τ + π) + σ);
11   return ϱ;
```

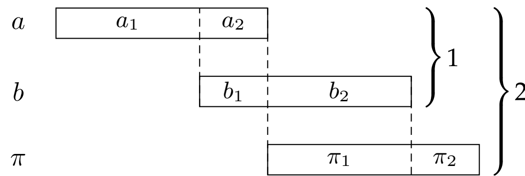


Figure 6.1: Alignment of the fractions of a, b, and π on line 10 of Algorithm 6.1.

approximating the error in order to obtain r_x or $1-r_x$ (alternatively ϱ_x or $2^{\text{exponent}(x)}\cdot\varepsilon-\varrho_x$) and conditionally change the result produced in hardware.

6.1 Addition

The solution we propose leverages the TWOSUM algorithm to round stochastically the sum of two floating-point numbers without explicitly computing the quantity r_x. This is achieved by exploiting the relation between the residual and the roundoff error in round-to-nearest, which can be computed exactly with the TWOSUM algorithm, provided that the sum does not overflow in round-to-nearest [33, Th. 6.2]. This approach is shown in Algorithm 6.1.

In the pseudocode, rand() returns a pseudorandom floating-point number in the interval $[0, 1)$. The algorithm first computes σ, the sum of a and b in round-to-nearest, the error term τ such that $\sigma + \tau = a + b$ in exact arithmetic, and the exponent η of the sum computed in round-toward-zero. Then it generates a p-digit floating-point number in the interval $[0, 1)$ which is scaled by the value of the least significant digit of $\mathrm{RZ}(a + b)$, so to have the same sign as the rounding error τ and absolute value in $[0, 2^\eta\varepsilon)$.

Finally, the operation on line 10 performs stochastic rounding. The alignment of a, b, and π in Algorithm 6.1 is illustrated in Fig. 6.1.

We now argue the correctness of the algorithm. We will assume, for now, that the quantity $\diamond(\tau + \pi)$ on line 10 is computed exactly; we will discuss the effect of errors striking this operation later. Note that if $\sigma = a + b$, then $\tau = 0$ and $0 \leq Z < 1$ guarantees that $\varrho = \sigma$ on line 10.

If σ and τ have the same sign, then $|\sigma| < |a+b|$, and it is easy to check that if the rounding mode $\circ$ used on line 10 is chosen according to the strategy on line 6, then $\varrho = \mathrm{RZ}(a+b)$ if and only if $|\diamond(\tau+\pi)| < 2^\eta\varepsilon$ or, under the assumption that $\diamond(\tau+\pi) = \tau+\pi$, if and only if $|\tau+\pi| < 2^\eta\varepsilon$. Since σ and τ have same sign, the latter condition can be rewritten as $|\pi| < 2^\eta\varepsilon - |\tau|$, or equivalently as $Z < 1 - r_{a+b}$. Similarly, $\varrho = \mathrm{RZ}(a+b) + 2^\eta\varepsilon$ if and only if $Z \geq 1 - r_{a+b}$, and we conclude that Algorithm 6.1 implements (4.6) when $\mathrm{sign}(\sigma) = \mathrm{sign}(\tau)$.

If σ and τ have opposite sign, on the other hand, then $|\sigma| > |a+b|$. In this case we have that $\varrho = \mathrm{RZ}(a+b)$ if and only if $|\diamond(\tau+\pi)| \geq 2^\eta\varepsilon$, which reasoning as above can be equivalently rewritten as $Z \geq r_{a+b}$, since $r_{a+b} = 1 - |\tau|/\varepsilon$. The case $\varrho = \mathrm{RZ}(a+b) + 2^\eta\varepsilon$ is analogous, which shows that Algorithm 6.1 implements (4.4) for $\mathrm{sign}(\sigma) = -\,\mathrm{sign}(\tau)$.

The diagram in Fig. 6.2 aids to clarify why the rounding operator $\circ$ used on line 10 depends on the sign of τ. The idea is that $|\diamond(\tau+\pi)|$ can become as large or larger than the least significant digit of σ, in which case the instruction on line 10 will revert the rounding performed by TwoSum. If, on the other hand, $|\diamond(\tau+\pi)|$ ends up being smaller than $2^\eta\varepsilon$, then the sum computed in round-to-nearest and is returned unchanged.

The error of Algorithm 6.1 depends on the magnitude of $\varphi := |\tau + \pi - \diamond(\tau+\pi)|$, which clearly depends on which rounding operator $\diamond$ is chosen on line 10. It is easy to see that for round-to-nearest and directed rounding we have that $\varphi < 2^{\eta-p}\varepsilon$ and $\varphi < 2^{\eta+1-p}\varepsilon = 2^\eta\varepsilon^2$, respectively.

The function get_exponent(x) returns the biased exponent of the floating-point number x as stored in the binary representation of x. It is important to stress that get_exponent(x) does not coincide with the exponent of x as computed by the C mathematical library functions `frexp` or `ilogb` if x represents 0 or a subnormal number whose leading bit is set to 0. In fact, when x is subnormal or zero, get_exponent(x) returns $0 - e_{\max} = e_{\min} - 1$, and not the exponent that x would have if it were a normal number, as `frexp` and `ilogb` would. This exponent is more efficient to obtain, as computing it does not require to count the leading zeros in the significand of x when the latter is subnormal, which in turn brings a performance gain. Our implementation of get_exponent(x) for binary64 arithmetic simply isolates the exponent bits from the binary representation of x with an appropriate mask, shifts them by 52 places to the right, and adds the bias of -1023. This ensures that no rounding is performed when $x = 0$, as it is the case, for example, when $a = -b$, since get_exponent$(0) = e_{min} - 1$ which ensures that $\pi = 0$ if line 5 is evaluated in round-to-nearest.

In the addition and all other algorithms below, the calculations of the type $\mathrm{sign}(\tau) \times Z \times 2^\eta \times \varepsilon$ are implemented by first isolating the sign bit of τ. The computation is then done by using `ldexp` to multiply the random number Z (with the sign of τ attached to it using bitwise disjunction) by $2^{\eta+\varepsilon}$.

Lastly, we comment on the behavior of overflow. The following applies to all the algorithms in this section that use a rounding mode other round-to-nearest. If overflow happens in the error-free transformation, TwoSum in this case, then we return a NaN since TwoSum sets τ as a NaN. If this is not acceptable, then an extra check (not shown in our algorithms) can be added to return $\sigma = \pm\infty$. However, if the operation on line 10 of Algorithm 6.1 overflows, then the maximum representable floating-point number of appropriate sign will be returned.

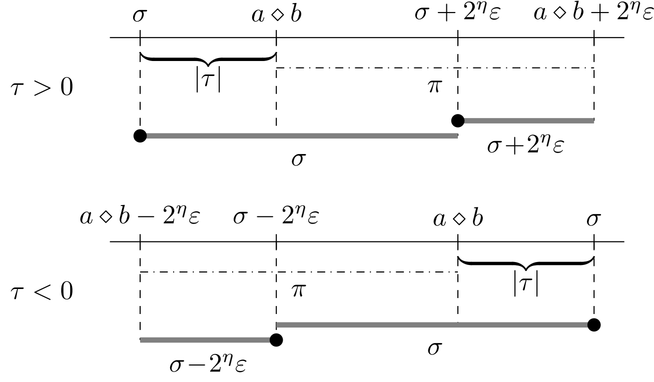


Figure 6.2: Diagram that motivates the use of different directed rounding modes depending on the sign of τ. The dot dashed line represents the range of the variable π, numbers that fall in the range of a thick grey line are rounded in the direction of the black dot at one end. The symbol $\diamond$ represents any of the elementary arithmetic operations, $a \diamond b$ and σ denote the result computed in exact arithmetic and in round-to-nearest, respectively.

Algorithm 6.2: Multiplication with stochastic rounding using the FMA instruction.

```
1 function MUL(a ∈ F, b ∈ F)
    If a, b satisfy (5.1), compute ϱ = SR(a · b) ∈ F.
2   Z ← rand();
3   (σ, τ) ← TWOPRODFMA(a, b, RZ);
4   η ← get_exponent(σ);
5   π ← sign(τ) × Z × 2^η × ε;
6   ϱ ← RZ(◇(τ + π) + σ);
7   return ϱ;
```

6.2 Multiplication

The function MUL in Algorithm 6.2 exploits TWOPRODFMA to compute $\mathrm{SR}(a \times b)$. Since the algorithm works with any rounding mode [28, Sec. 4.4.1], we prefer to use round-toward-zero for efficiency sake. In this way, $\varrho_x = \tau$ and the exponent can be calculated directly from σ, without requiring an extra floating-point operation as was the case in Algorithm 6.1.

The correctness of Algorithm 6.2 can be shown with an argument analogous to that used for Algorithm 6.1. Note that the proof is easier in this case, as the use of round-to-zero implies that either $\tau = 0$ or $\mathrm{sign}(\sigma) = \mathrm{sign}(\tau)$.

A method that exploits TWOPRODDEK in place of TWOPRODFMA is given in Algorithm 6.3. As Dekker's multiplication algorithm has not been shown to be exact for rounding modes other than round-to-nearest, an extra floating-point operation to get the correct exponent of $\lfloor a \times b \rceil$ is necessary. This corresponds to the operation on line 4 of ADD in Algorithm 6.1.

As discussed in Section 5, the error-free transformation for multiplication does not work

Algorithm 6.3: Multiplication with stochastic rounding using Dekker's algorithm.

```
1  function MulDekker(a ∈ F, b ∈ F)
     If a, b satisfy (5.1), compute ϱ = SR(a · b) ∈ F.
2    Z ← rand();
3    (σ, τ) ← TwoProdDek(a, b);
4    η ← get_exponent(RZ(a × b));
5    π ← sign(t) × Z × 2^η × ε;
6    if τ ≥ 0 then
7      ∘ = RD;
8    else
9      ∘ = RU;
10   ϱ ← ∘(⋄(τ + π) + σ);
11   return ϱ;
```

if the error is smaller than the smallest number representable in the working precision. Our algorithms for multiplication do not try to solve this issue, and we return an unrounded result in the case of underflow in τ. In terms of overflow, the behavior of Mul hinges on what TwoProdFMA returns when the multiplication overflows. Depending on the implementation of the FMA, τ on line 3 of Algorithm 6.2 may be either $\pm\infty$ or a NaN. If $\tau = +\infty$, then Mul will return $+\infty$ correctly, whereas if τ is $-\infty$ or a NaN, then an extra check will be required to ensure that $+\infty$ is correctly returned.

6.3 Division

We note that it would not be possible to derive an algorithm for stochastically rounded division in the spirit of the other algorithms in this section, as the binary expansion of the error arising in the division of two floating-point numbers may have, in general, infinitely many nonzero digits. An example of this is the binary number $1/11 = 0.\overline{01} = 0.010101\ldots$.

In order to obtain an algorithm for division, we exploit a result by Bohlender et al. [34]. Let a and b be floating-point numbers and let $\sigma := \circ(a \div b)$ where $\circ$ is any of the IEEE rounding functions. If σ is neither an infinity nor a NaN, then under some mild assumptions (see [35, Th. 4]) $\tau' := a - \sigma \times b$ is exactly representable. In our algorithm, we first compute σ, then obtain τ' using a single FMA operation, and estimate the rounding error in the division by computing τ'/b. When an FMA is not available, τ' can be computed with Dekker's multiplication algorithm. The stochastic rounding step is performed as in previous algorithms. The method we propose to stochastically round this operation without relying on higher precision is illustrated in Algorithm 6.4.

The error of $\varphi := |\tau + \pi - \diamond(\tau + \pi)|$ is larger than that of the other algorithms discussed so far, since only an approximation to the actual residual τ is available. We note, however, that this error is of the same magnitude as that introduced by rounding $\tau + \pi$, which suggests that $\varphi < 2^{\eta}\varepsilon$ for round-to-nearest and $\varphi < 2^{\eta+1}\varepsilon$ for directed rounding.

Algorithm 6.4: Division with stochastic rounding.

```
1 function Div(a ∈ F, b ∈ F)
    Compute ϱ = SR(a ÷ b) ∈ F.
2     Z ← rand();
3     σ ← RZ(a ÷ b);
4     τ' ← RZ(−σ × b + a);
5     τ ← RZ(τ' ÷ b);
6     η ← get_exponent(σ);
7     π ← sign(τ) × Z × 2^η × ε;
8     ϱ ← RZ(◇(τ + π) + σ);
9     return ϱ;
```

Algorithm 6.5: Square root with stochastic rounding.

```
1 function Sqrt(a ∈ F)
    Compute ϱ = SR(√a) ∈ F.
2     Z ← rand();
3     σ ← RZ(√a);
4     τ' ← RZ(−σ^2 + a);
5     τ ← RZ(τ' ÷ (2 × σ));
6     η ← get_exponent(σ);
7     π ← sign(τ) × Z × 2^η × ε;
8     ϱ ← RZ(◇(τ + π) + σ);
9     return ϱ;
```

6.4 Square root

The algorithm for the square root, given in Algorithm 6.5, is very similar to that for division. The only difference is the computation of the approximate error τ, which is performed as discussed by Brisebarre et al. [36].

6.5 Algorithms without the change of the rounding mode

Most floating-point hardware uses round-to-nearest by default, thus we now discuss how the algorithms discussed so far can be modified to rely on this rounding mode only. It is widely known that changing the rounding mode of a processor can result in a severe performance degradation, therefore the algorithms in this section should be much faster, yet more complex. Algorithms 6.7, 6.8, 6.9, and 6.10 show how to adapt Algorithms 6.1, 6.2, 6.4, and 6.5, respectively. The function SRound in Algorithm 6.6 is an auxiliary routine on which the stochastic rounding algorithms rely on. The quantity called "ulp" in the algorithm is a gap between the two floating-point values surrounding $\sigma + \tau$, except when σ is zero or subnormal, in which case it is a quantity half-way between zero and the smallest subnormal. The function $\mathrm{pred}(x) := (1 - 2^{-p}) \times x$ returns the floating-point number next to x in the direction of 0, if $|x| > 2^{e_{min}}$, and the number x itself otherwise, as per line 1 of [27, Alg. 4]. Note that $\mathrm{pred}(x) = x$ when $|x| \leq 2^{e_{min}}$, which includes subnormals and

Algorithm 6.6: A helper function for stochastic rounding.

```
1  function SRROUND(σ ∈ F, τ ∈ F, Z ∈ F)
     Compute round ∈ F.
2    if sign(τ) ≠ sign(σ) then
3      η ← get_exponent(pred(σ));
4    else
5      η ← get_exponent(σ);
6    ulp ← sign(τ) × 2^η × ε;
7    π ← ulp × Z;
8    if |RN(τ + π)| ≥ |ulp| then
9      round = ulp;
10   else
11     round = 0;
12   return round;
```

Algorithm 6.7: Stochastically rounded addition without the change of the rounding mode.

```
1  function ADD2(a ∈ F, b ∈ F)
     Compute ϱ = SR(a + b) ∈ F.
2    Z ← rand();
3    (σ, τ) ← TWOSUM(a, b, RN);
4    round ← SRROUND(σ, τ, Z);
5    ϱ ← RN(σ + round);
6    return ϱ;
```

the smallest normal value, as shown in [27]. However, for the purposes of Algorithm 6.6 we only need this functions when get_exponent(σ) from get_exponent(pred(σ)), which is never the case in the subnormal range.

The function pred(σ) in our C implementation of the algorithms is calculated, in the case of binary64 arithmetic, by multiplying σ by the constant `1-ldexp(1, -53)`.

We now discuss the behavior of the modified algorithms in case of overflow. If there is no overflow in the error-free transformation, then the exact results of magnitude at least $2^{e_{\max}}(2-2^{-p})$ overflow to the closest infinity, whereas those below this threshold but of magnitude larger than the maximum representable value will be rounded stochastically to the corresponding infinity. This is due to use of round-to-nearest in the final step of the computation of ϱ. If, on the other hand, the computation performed during the error-free transformation overflows, then $\sigma = \pm\infty$ is returned.

7 Performance

In this section we evaluate experimentally the performance of a C implementation of the techniques in Section 6.

Algorithm 6.8: Multiplication with stochastic rounding using the FMA instruction without the change of the rounding mode.

```
1 function Mul2(a ∈ F, b ∈ F)
    If a, b satisfy (5.1), compute ϱ = SR(a · b) ∈ F.
2     Z ← rand();
3     (σ, τ) ← TwoProdFMA(a, b, RN);
4     round ← SRround(σ, τ, Z);
5     ϱ ← RN(σ + round);
6     return ϱ;
```

Algorithm 6.9: Division with stochastic rounding without the change of the rounding mode.

```
1 function Div2(a ∈ F, b ∈ F)
    Compute ϱ = SR(a ÷ b) ∈ F.
2     Z ← rand();
3     σ ← RN(a ÷ b);
4     τ′ ← RN(−σ × b + a);
5     τ ← RN(τ′ ÷ b);
6     round ← SRround(σ, τ, Z);
7     ϱ ← RN(σ + round);
8     return ϱ;
```

We compared our methods with a C port of the stochastic rounding functionalities of the MATLAB `chop` function [12]. As our focus in this section is on binary64 arithmetic, we used the GNU MPFR library [37] (version 4.0.1) to compute in higher-than-binary64 precision. We denote by `sr_<mpfr_op>` the function that uses the MPFR operator `<mpfr_op>` to compute the high-precision result that is subsequently stochastically rounded to binary64. The codes we used for this benchmark (as well as experiments of the next section) are available on GitHub.[2]

In Table 7.1 we consider the throughput (in Mop/s, millions of operations per second) of the functions we implemented on a test set of 100 pairs of uniformly distributed binary64 random numbers in the interval $[f_{\min}, 1+f_{\min})$, where $f_{\min} := 2^{-1022}$ is the smallest positive normal number in binary64. We remove subnormals from the interval from which we draw the random samples, in order to avoid the possible performance degradation should subnormals be handled in software rather than in hardware. For each pair of floating-point inputs, we estimate the throughput by running each algorithm 10,000,000 times, and in the table report the minimum, maximum, and mean value over the 100 test cases, as well as the value of the standard deviation and the speedup with respect to the 113-bit variant of the GNU MPFR-based algorithm.

Our experiments were performed on a machine equipped with an Intel Xeon Gold 6130 CPU running CentOS GNU/Linux release 7 (Core). The codes were compiled with GCC 8.2.0 using the options `-mfpmath=sse` and `-march=native`, which includes the flags

[2] `https://github.com/mmikaitis/stochastic-rounding-evaluation`

Algorithm 6.10: Square root with stochastic rounding without the change of the rounding mode.

```
1 function SQRT2(a ∈ F, b ∈ F)
    Compute ϱ = SR(√a) ∈ F.
2   Z ← rand();
3   σ ← RN(√a);
4   τ′ ← RN(−σ² + a);
5   τ ← RN(τ′ ÷ (2 × σ));
6   round ← SROUND(σ, τ, Z);
7   ϱ ← RN(σ + round);
8   return ϱ;
```

`-mfma` and `-msse2`, since the Skylake CPU we used supports the FMA instruction and the Streaming SIMD Extensions 2 (SSE2) supplementary instruction set. The flags `-msse2` and `-mfpmath=sse` together ensure that 80-bit extended precision is not used at any point in the computation [38]. For the optimization level, we were forced to use `-O0` for the implementation of Algorithms 6.1, 6.2, 6.4, and 6.5 (more strict optimization causes some issues with the changes of the rounding mode so that the algorithms do not pass basic tests), but used `-O3` for the functions based on MPFR and for implementation of Algorithms 6.7, 6.8, 6.9, and 6.10.

The benchmark results show that the new algorithms that work only in binary64 arithmetic but switch rounding mode are 7.3 to 9.1 times faster than those relying on the GNU MPFR library, regardless of the number of extra digits of precision used. The alternative algorithms discussed in Section 6.5, which do not require the change of rounding mode, are 16.3 to 19 faster than the reference implementation based on GNU MPFR.

8 Numerical experiments

Now we gauge the accuracy of the new algorithms in Section 6. We do so by illustrating their numerical behavior on three benchmark problems on which stochastic rounding outperforms round-to-nearest when low-precision arithmetic is used. These are the computation of partial sums of the harmonic series in finite precision, the summation of badly scaled random values, and the solution of simple ordinary differential equations (ODEs). The experiments were run in MATLAB 9.7 (2019b) using the Stochastic Rounding Toolbox we developed, also available on GitHub.[3] Reduced-precision floating-point formats were simulated on binary64 hardware using the MATLAB `chop` function [12].

8.1 Harmonic series

In exact arithmetic, the harmonic series

$$\sum_{i=1}^{\infty} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \dots \tag{8.1}$$

[3] https://github.com/mfasi/srtoolbox

Table 7.1: Throughput (in Mop/s) of the C implementations of the algorithms discussed in the paper. The parameter p represents the number of significant digits in the fraction of the MPFR numbers being used; algorithms that do not use MPFR have a missing value in the corresponding row. The baseline for the speedup is the mean thoughtput of the MPFR variant that uses 113 bits to perform the same operation.

	`sr_mpfr_add`			Add	Add2	`sr_mpfr_mul`			Mul	Mul2
p	61	88	113	–	–	61	88	113	–	–
min	3.5	3.7	3.5	18.5	62.5	3.7	3.7	3.7	32.2	66.6
max	3.8	4.2	4.0	31.2	71.4	4.2	4.1	4.0	34.4	76.9
mean	3.7	3.8	3.8	28.2	68.8	3.9	3.9	3.8	33.9	72.4
↪ *speedup*	0.9×	1.0×	1.0×	7.3×	17.9×	1.0×	1.0×	1.0×	8.7×	18.6×
deviation	0.1	0.1	0.1	2.3	2.5	0.1	0.1	0.1	0.6	2.3

	`sr_mpfr_div`			Div	Div2	`sr_mpfr_sqrt`			Sqrt	Sqrt2
p	61	88	113	–	–	61	88	113	–	–
min	3.3	3.4	3.4	31.2	62.5	3.6	3.0	2.8	28.5	52.6
max	3.6	3.6	3.6	33.3	71.4	4.2	3.6	3.6	30.3	58.8
mean	3.5	3.5	3.5	32.2	67.2	4.1	3.5	3.5	29.5	57.4
↪ *speedup*	1.0×	1.0×	1.0×	9.1×	19.0×	1.1×	1.0×	1.0×	8.3×	16.3×
deviation	0.1	0.1	0.1	0.4	1.9	0.1	0.1	0.1	0.4	1.7

is divergent. If the partial sums of (8.1) are evaluated in finite precision, however, this is not the case: using binary64 arithmetic and round-to-nearest, Malone [39] showed that the series converges numerically to the value $S_{2^{48}} \approx 34.122$ after $N = 2^{48}$ terms. In the experiment, the author evaluated the sum by simply adding the terms from left to right, and convergence was achieved on an AMD Athlon 64 processor after 24 days. The same experiment was run in fp8 (an 8-bit floating-point format), bfloat16, binary16, and binary32 arithmetics by Higham and Pranesh [12], who showed that in binary32 arithmetic with round-to-nearest the series converges to $S_{2^{21}} \approx 15.404$ on iteration $N = 2^{21} = 2{,}097{,}152$.

Here we use the computation of

$$H_k(s_0) := s_0 + \sum_{i=1}^{k} \frac{1}{i} = s_0 + 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{k} \tag{8.2}$$

as a simple test problem to compare the behavior of stochastic summation with classic summation algorithms in round-to-nearest. We include two variants of stochastically rounded recursive summation, one that simulates stochastic rounding using Algorithm 6.1 and one that relies on the MATLAB `chop` function [12]. We use a single stream of random numbers produced by the `mrg32k3a` generator seeded with the arbitrarily chosen integer 300, and at each step we generate only one random number and use it for both algorithms. For

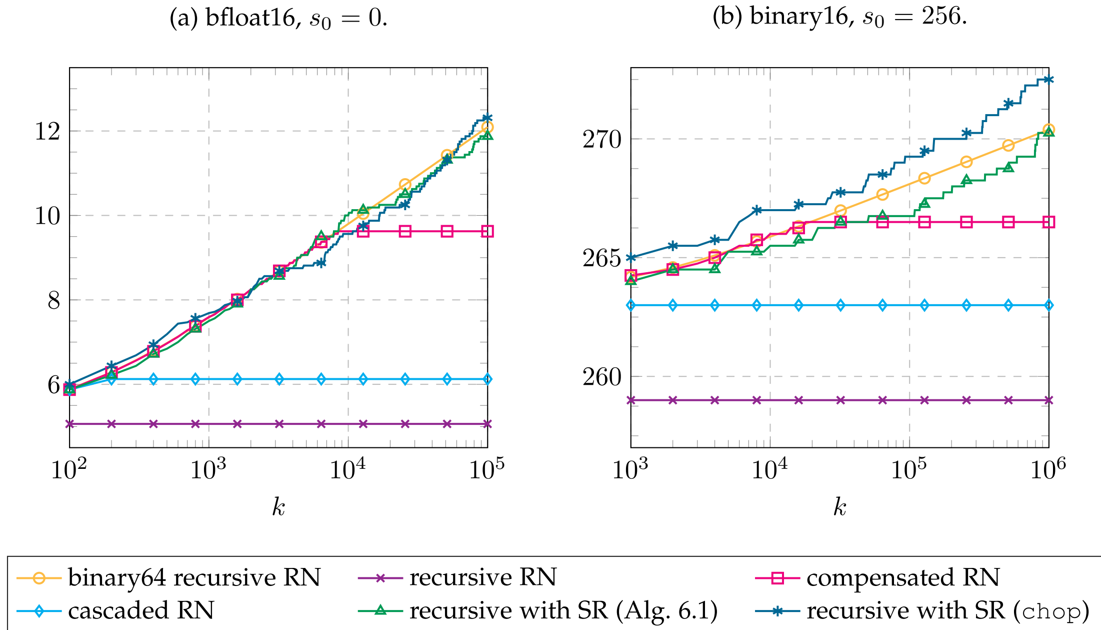


Figure 8.1: Numerical value of the sum $H_k(s_0)$ in (8.2) accumulated in bfloat16 (left) and binary16 (right) arithmetics with various summation algorithms. The sum computed in binary64 precision is taken as reference. The algorithms use round-to-nearest (RN) or stochastic rounding (SR) as indicated in the legend.

round-to-nearest we consider, besides recursive summation at working precision, compensated summation [40], which at each step computes the rounding error with TwoSum and adds it to the next summand, and cascaded summation [41], which accumulates all the rounding errors in a temporary variable which is eventually added to the total sum. We do not include doubly compensated summation [28, Sec. 5.3.2], [42] because its results are indistinguishable from those of compensated summation on this example. As reference we take the sum computed by recursive summation in binary64 arithmetic.

Our goal is to show that recursive and compensated summation stagnate with the standard IEEE 754 rounding mode but not when stochastic rounding is used; stagnation is easily achieved in bfloat16 and binary16 arithmetics for k well below 10^5. For binary16, we had to set $s_0 = 256$ to cause stagnation. In other words, for binary16 we computed $256+\sum_{i=1}^{\infty} 1/i$, obtaining the results in Fig. 8.1(b). As expected, recursive summation is the first method to fail, while compensated summation follows the reference quite accurately before starting to stagnate. Cascaded summation stagnates after recursive summation but before compensated summation. When paired with stochastic rounding, on the other hand, recursive summation suffers from an error larger than that of compensated summation, but does not stagnate. Observe that Algorithm 6.1 and `chop` perform differently, despite the fact that the same random number was used at each step: this is expected, as the two algorithms follow a totally different approach for the computation of the stochastically rounded sum.

8.2 Sum of random values

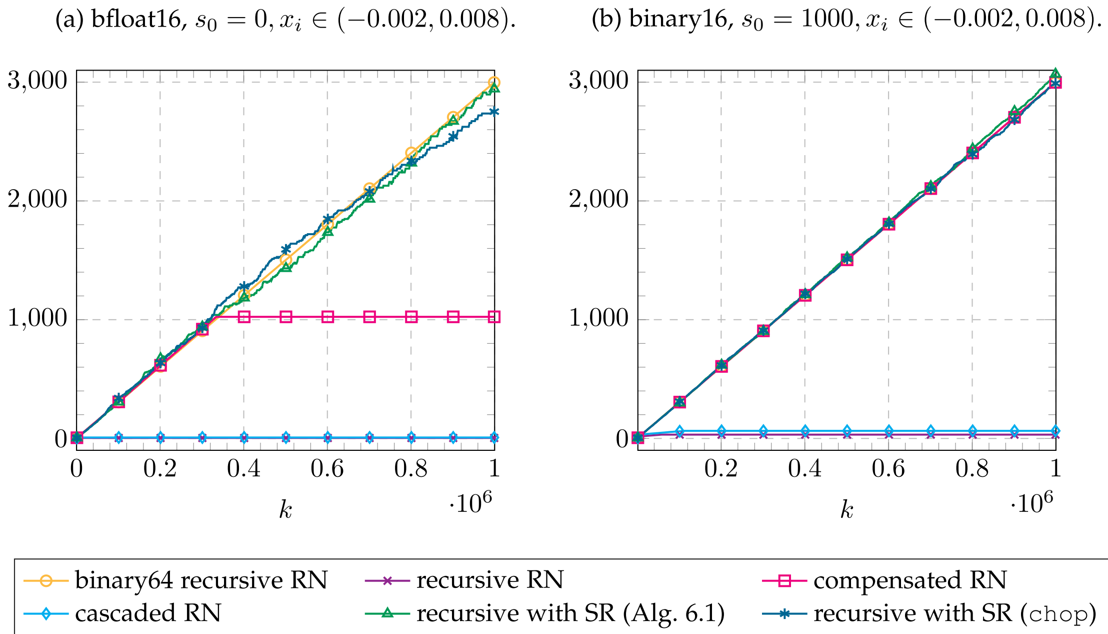


Figure 8.2: Numerical value of the sum $S_k(s_0)$ in (8.3) accumulated in bfloat16 (left) and binary16 (right) arithmetics using various algorithms. The algorithms use round-to-nearest (RN) or stochastic rounding (SR) as indicated in the legend.

In this second test we compare the different summation algorithms on the task of computing the sum

$$S_k(s_0) = s_0 + \sum_{i=1}^{k} x_i, \tag{8.3}$$

where the x_i are uniformly distributed over an open interval that contains both negative and positive numbers, but is biased towards positive values to ensure that the value of $S_k(s_0)$ is increasing for large k. These random numbers were generated from a stream of random numbers, this time seeded with the arbitrarily chosen integer 500. We initialized the sum to a positive number s_0 large enough compared with the range of the random numbers to cause stagnation.

Fig. 8.2 shows the results of this experiment. As in the previous experiment, in binary16 both recursive and cascaded summation stagnate very early, but compensated summation does not in this test. We note, however, that all three algorithms would face this problem if smaller random numbers were used.

In order to test the algorithms at precision natively supported by the hardware without using simulated arithmetics, we ran some experiments in binary64. In MATLAB, the rounding behavior of the underlying hardware can be controlled in an IEEE-compliant way: the commands `feature('setround', 0)` and `feature('setround', 0.5)` switch to round-towards-zero and round-to-nearest respectively. Our test problem is similar to those above, as we aim to sum random values small enough for stagnation to occur (random

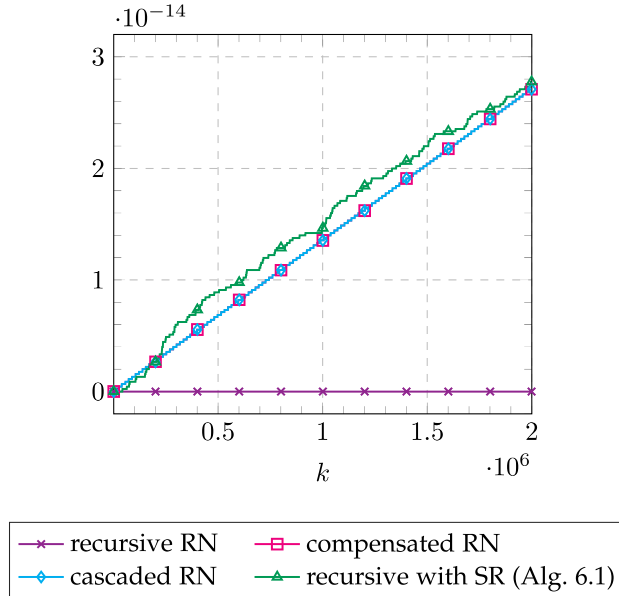


Figure 8.3: Numerical value of $S_k(s_0) - 1$, for the sum $S_k(s_0)$ as defined in (8.3), accumulated in binary64 arithmetic using various algorithms with $s_0 = 1$, $x_i \in (0, 2^{-65})$. The algorithms use round-to-nearest (RN) or stochastic rounding (SR) as indicated in the legend.

numbers required to observe stagnation in binary64 are so small that this phenomenon is unlikely to be observed in real applications).

The results of this experiments are reported in Fig. 8.3. While recursive summation stagnated as expected, we were unable to find any combination of parameters that caused compensated summation to stagnate in binary64. Therefore, compensated summation seems to be the best choice in binary64 arithmetic, whereas lower precision appears to benefit from recursive summation with stochastic rounding, as in this case both compensated and cascaded summation stagnate in our experiments.

8.3 ODE solvers

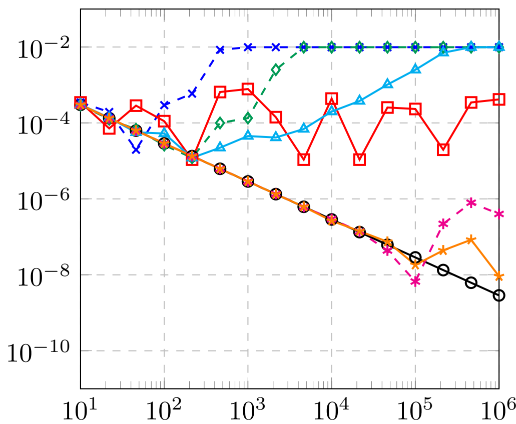
8.3.1 Exponential decay ODE

Explicit solvers for ODEs of the type $y' = f(x, y)$ have the form $y_{t+1} = y_t + h\phi(x_t, y_t, h, f)$ for a fixed step size h. For small h, they are therefore susceptible to stagnation. In fixed-point arithmetic, stochastic rounding was shown to be very beneficial on four different ODE solvers [11]. Here we use the algorithms developed in Section 6 to show that stochastic rounding brings similar benefits in floating-point arithmetic, as it increases the accuracy of the solution for small values of h. For these experiments we used the default MATLAB random number generator seeded with the value 1.

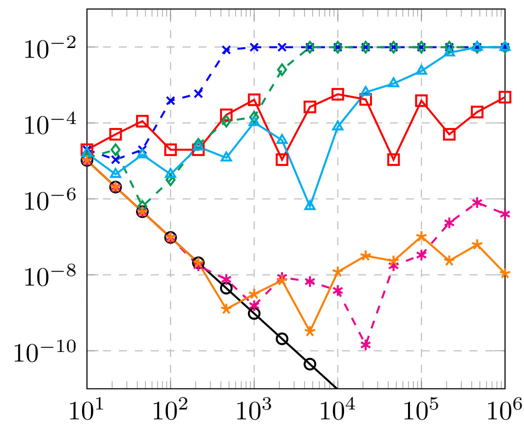
Higham and Pranesh [12] tested Euler's method on the equation $y' = -y$ using different reduced precision floating-point formats, and showed the importance of subnormal numbers. A similar experiment for time steps as small as 10^{-8} is shown in [43, Sec. 4.3]. We use

their code[4] and compare round-to-nearest and stochastic rounding on the same test problem. The ODE with initial condition $y(0) = 2^{-6}$ (chosen so that it is representable exactly in all arithmetics) is solved over $[0, 1]$ using the explicit scheme $y_{n+1} = y_n + hf(t_n, y_n)$ with $h = 1/n$ for $n \in [10, 10^6]$. Fig. 8.4(a) shows the absolute errors of the ODE solution at $x = 1$ for increasing values of the discretization parameter n. For small integration steps, the error is around four orders of magnitude smaller when stochastic rounding is enabled for the 16-bit arithmetics.

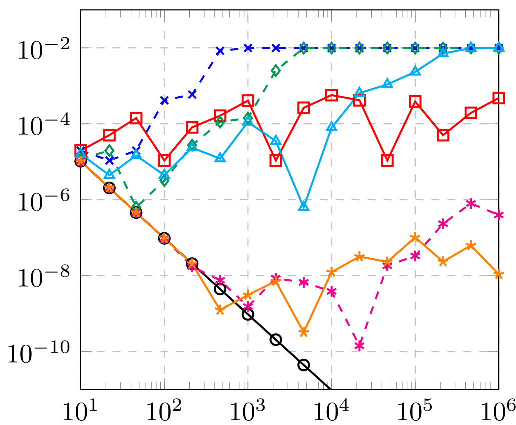
(a) Euler for $y' = -y$, $y(0) = 2^{-6}$, over $[0, 1]$.



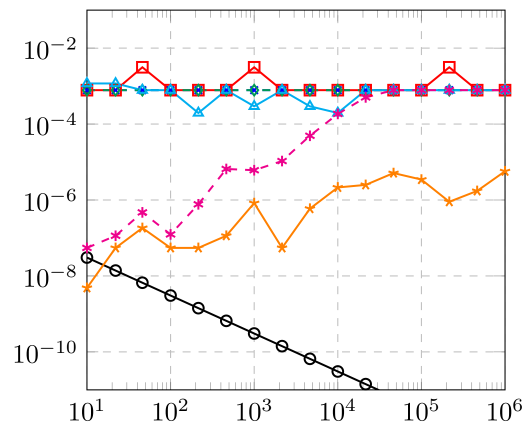
(b) Midpoint for $y' = -y$, $y(0) = 2^{-6}$, over $[0, 1]$.



(c) Heun for $y' = -y$, $y(0) = 2^{-6}$, over $[0, 1]$.



(d) Euler for $y' = -\frac{y}{20}$, $y(0) = 1$ over $[0, 2^{-6}]$.



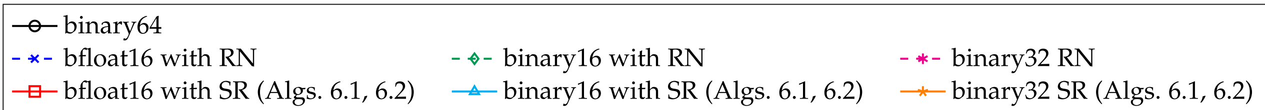


Figure 8.4: Absolute errors in Euler, Midpoint, and Heun methods for the exponential decay ODE solutions with different floating point arithmetics and rounding modes. The x-axis represents n while y-axis represents the error.

[4]`https://github.com/SrikaraPranesh/LowPrecision_Simulation`

We tested two other algorithms for the numerical integration of ODEs:

- the midpoint second-order Runge–Kutta method (RK2)

$$y_{n+1} = y_n + hf\Big(t_n + \frac{1}{2}h, y_n + \frac{1}{2}hf(t_n, y_n)\Big),$$

- Heun's method

$$\begin{cases} y'_n &= y_n + hf(t_n, y_n), \\ y_{n+1} &= y_n + \frac{1}{2}h\Big(f(t_n, y_n) + f(t_n + h, y'_n)\Big). \end{cases}$$

The results for these two methods are reported in Fig. 8.4(b) and Fig. 8.4(c), respectively.

To cause stagnation in binary32 we reformulated the problem to have a smaller integration period and a larger initial condition. For instance we could consider the same ODE $y' = -y$, but take as initial condition $y(0) = 1$ over $[0, 2^{-6}]$ with $h = 2^{-6}/n$ for $n \in [10, 10^6]$. We only need the constant in the initial condition to be large relative to the time step size, the number 1 was an arbitrary choice. Now at every step of the integration, we would subtract from 1 a very small positive value whose magnitude decreases with the time step size, and we expect that even binary32 will show more significant errors. Another way to increase the errors is to introduce a decay time constant other than 1 into the differential equation. The ODE $y' = -y/20$, for instance, will cause the updates at each step of a solver to be even smaller. Fig. 8.4(d) shows this scenario using Euler's method. In this case only binary64 and binary32 with stochastic rounding manage to avoid stagnation for small time steps.

8.3.2 Unit circle ODE

The solution to the system of ODEs

$$\begin{cases} u'(t) = v(t), \\ v'(t) = -u(t), \end{cases}$$

with initial values $u(0) = 1$ and $v(0) = 0$ represents the unit circle in the uv-plane [44, p. 51]. Higham [44, p. 51] shows also that the forward Euler scheme

$$\begin{cases} u_{k+1} &= u_k + hv_k, \\ v_{k+1} &= v_k - hu_k, \end{cases} \tag{8.4}$$

with $h = 2\pi/32$ produces a curve that spirals out of the unit circle. Euler's method can be improved by using a smaller time step, which gives a more accurate approximation to the unit circle at a higher computational cost. From the previous section we know, however, that smaller time steps are more likely to cause issues with rounding errors.

The goal of this experiment is therefore to see what curve the methods draw when using round-to-nearest and stochastic rounding at small step sizes. We note that here stochastic rounding is used for both addition and multiplication operations in (8.4). Fig. 8.5 shows some circles drawn when solving (8.4) for various step sizes $h = 2\pi/n$.

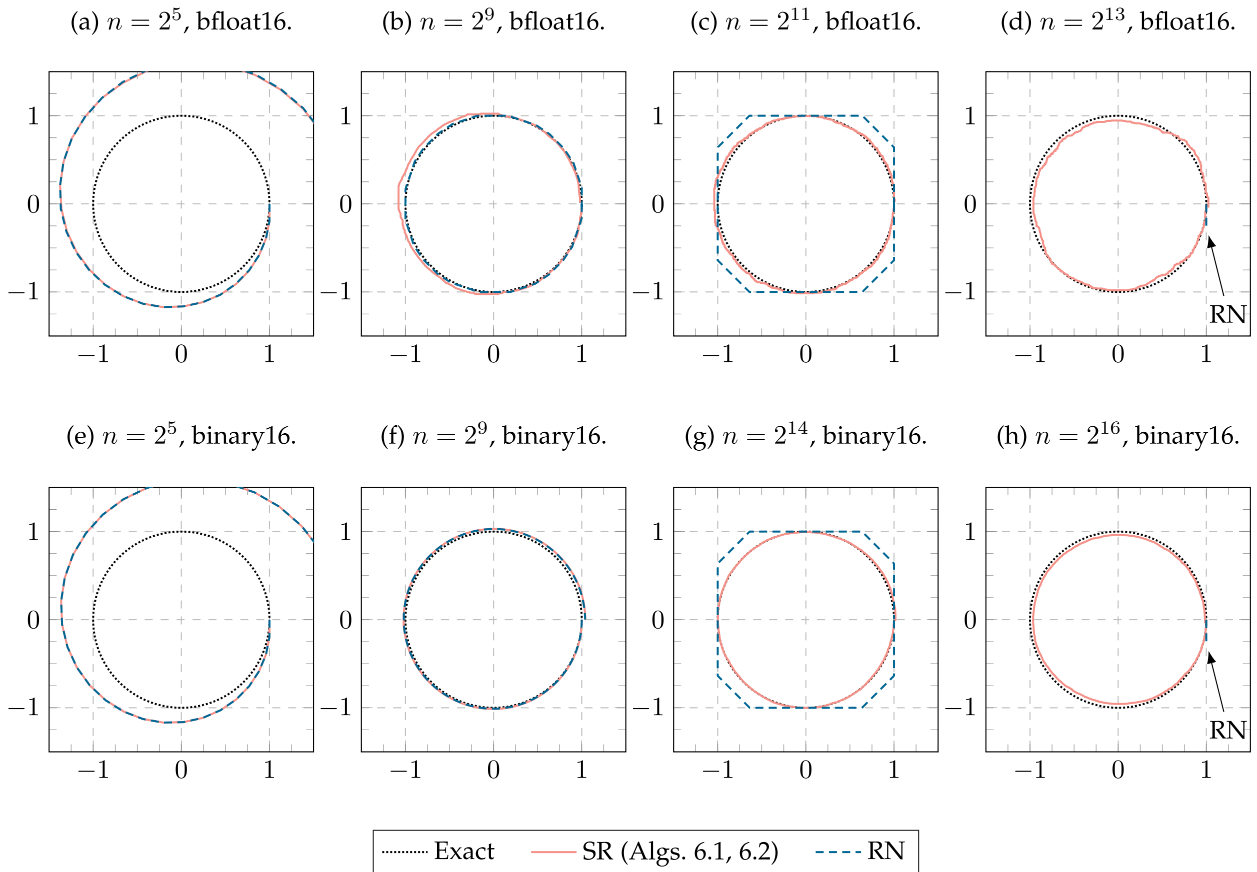


Figure 8.5: Unit circle drawn by Euler's method in (8.4) with various arithmetics and rounding modes compared to the exact solution. The default MATLAB random number generator seeded with 500 was used. The x- and y-axis represent u and v, respectively. Note that in (d) and (h) only a very small part of the solution with RN is visible (marked with an arrow) since the ODE solver failed due to stagnation.

As expected, for large step sizes the solution spirals out of the unit circle, then gets gradually closer to the right solution as the step size decreases, until rounding errors start to dominate the computation causing major issues: for a small enough step size the solution computed using round-to-nearest looks like an octagon. Stochastic rounding seems to avoid this problem and keeps the solution near the unit circle. We do not report the results for binary32, as we found its behavior to be the same as that of binary16/bfloat16 at $n = 2^5$ regardless of the step size.

The octagonal shape of the circle approximation with round-to-nearest is interesting and worth looking at in more detail. In Fig. 8.5c and 8.5g we can see that during the first few iterations, v changes while u remains constant. In theory, we would expect u to start decreasing because of the negative values of v, but the number being subtracted from $u_0 = 1$ is too small for the 16-bit floating-point number systems considered in this experiment, as we now explain.

In order to simplify the analysis, we now assume exact arithmetic. If no rounding errors strike the computation, after 7 integration steps we get, for a given h,

$$\begin{aligned} u_7 &= 1 - 21h^2 + 36h^4 - 7h^6, \\ v_7 &= -7h + 35h^3 - 21h^5 + h^7. \end{aligned} \tag{8.5}$$

It is clear that the value of v_7 will depend on h even for very small time steps, but the value of u_7 might not, as this coordinate has a constant term and the update is a second-order term in h that can potentially be much smaller. The other terms in this expression for u_7 are even smaller, so we focus only on the the first two. If we expand them so to make the sequence of operations performed by Euler's method explicit, we obtain that

$$u_7 \approx 1 - 21h^2 = 1 - h^2 - 2h^2 - 3h^2 - 4h^2 - 5h^2 - 6h^2,$$

where each multiplication and subtraction can potentially cause a rounding error. If h^2 is significantly smaller than 1, in particular, the subtraction $1 - kh^2$ might result in stagnation due to the rounding returning 1 and yielding $u_{k+1} = 1 = u_0$. That is why in Fig. 8.5 the value of u initially remains constant with round-to-nearest but changes immediately with stochastic rounding: the latter manages to erode 1 by rounding up some of the kh^2 terms. As k increases, the terms kh^2 will eventually become large enough for subtractions to start taking effect with round-to-nearest, at which point the curve will move to a different edge of the octagon.

The situation is similar at the bottom of the circle, where $v_{N/4} = -1$ and $u_{N/4} = 0$. At first, the value of hu_i is so small that $v_{i+1} = -1 - hu_i$ evaluates to -1 in finite precision. As the magnitude of $u_i < 0$ increases, so does $-hu_i$, which eventually becomes large enough for round-to-nearest to round up the result of $-1 - hu_i$. When rounding stochastically, this is not as problematic, since any nonzero value of hu_i has a nonzero probability of causing the subtraction to round up. The expanded expression for the first two terms of $v_{N/4+7}$ is similar to u_7, with increasingly larger multiples of h^2 being added to -1 at each step of Euler's method

$$v_{N/4+7} \approx -1 + 21h^2 = -1 + h^2 + 2h^2 + 3h^2 + 4h^2 + 5h^2 + 6h^2.$$

In Fig. 8.5d and 8.5h, on the other hand, the step size h is so small that even v stops progressing in round-to-nearest, and only a small portion of the octagon is drawn. This can be explained by looking at (8.5): the largest term supposed to decrease $v_0 = 0$ is the first order term $-h$, therefore for large enough h in finite-precision arithmetic one will have $v_k = -kh$. As can be seen from the figure, this works for the first few iterations, during which v grows in magnitude while h remains constant, eventually causing stagnation to occur. Note that u is also fixed at 1 at that point, which means that the other terms in the expansion of v in (8.5) vanish and the whole system of ODEs cannot progress any further. This does not happen when rounding stochastically, as this rounding mode avoids stagnation of both variables.

This simple experiment resembles the integration of planetary orbits. For example, Quinn, Tremaine, and Duncan [45, Sec. 3.2] use multistep methods to integrate orbits over a time span of millions of years with a time step of 0.75 days. The authors comment that roundoff errors arising in the additions within the integration algorithm can become

a dominant source of the total error, and propose to keep track of these errors and add them back to the partial sum as soon as their sum exceeds the value of the least significant bit. This technique is similar to the approach taken by cascaded summation. The use of stochastic rounding in the floating-point addition might alleviate this issue by reducing the total summation error without requiring any additional task-specific code or extra storage space at runtime.

We believe that the exploration of stochastic rounding in this particular application should be a main direction of future work. Our algorithms for emulating stochastically rounded elementary arithmetic operations, along with the code for binary64 precision arithmetic that we provide, will allow those interested in looking into this problem to easily access arithmetic with stochastic rounding without requiring the use MPFR or alternative multiple-precision libraries.

9 Conclusions

There is growing interest in stochastic rounding in various domains [6, 7, 10, 14–18], and this rounding mode has started appearing in hardware devices produced by Graphcore [19] and Intel [20]. In this work we proposed and compared several algorithms for simulating stochastically rounded elementary arithmetic operations via software. The main feature of our techniques is that they only assume an IEEE-compliant floating-point arithmetic, but do not require higher-precision computations. This is a major advantage in terms of both applicability and performance. On the one hand, these methods can be readily implemented on a wide range of platforms, including those, such as GPUs, for which multiple-precision libraries are not available. On the other hand, the new techniques lead to more efficient implementations: our experiments in double precision show a speedup of order 10 or more over an MPFR-based multiple-precision approach.

We also discussed some applications where stochastic rounding is capable of curing the instabilities to which round-to-nearest is prone. We showed that, in applications where stagnation is likely to occur, using stochastic rounding can lead to much more accurate results than standard round-to-nearest or even compensated algorithms. This is especially relevant for binary16 and bfloat16, two 16-bit formats that are becoming increasingly common in hardware.

We feel that other applications would benefit from the use of stochastic rounding at lower precision, and believe that this rounding mode will play an important role as hardware that does not support 32/64-bit arithmetics becomes more prevalent. This will be the subject of future work.

Acknowledgment

We thank Nicholas J. Higham for fruitful discussions on stochastic rounding, for suggesting the unit circle ODE applications, and for his feedback on early drafts of this work. We also thank Michael Connolly for his comments on the manuscript. The work of M. Fasi was supported by the Royal Society, the Wenner–Gren Foundation, and the Istituto Nazionale di Alta Matematica, INdAM–GNCS Project 2020. The work of M. Mikaitis was supported

by an Engineering and Physical Sciences Research Council Doctoral Prize Fellowship and grant EP/P020720/1.

References

[1] *IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2019 (revision of IEEE Std 754-2008).* Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, Jul. 2019. Available: https://doi.org/10.1109/IEEESTD.2019.8766229

[2] G. E. Forsythe, "Round-off errors in numerical integration on automatic machinery," *Bull. Amer. Math. Soc.*, vol. 56, pp. 55–64, 1950. Available: https://doi.org/10.1090/S0002-9904-1950-09343-4

[3] J. Vignes, "A stochastic arithmetic for reliable scientific computation," *Math. Comput. Simulation*, vol. 35, no. 3, p. 233–261, Sep. 1993. Available: https://doi.org/10.1016/0378-4754(93)90003-d

[4] F. Jézéquel and J.-M. Chesneaux, "CADNA: A library for estimating round-off error propagation," *Comput. Phys. Comm.*, vol. 178, no. 12, p. 933–955, Jun. 2008. Available: https://doi.org/10.1016/j.cpc.2008.02.003

[5] S. Parker, "Monte Carlo arithmetic: Exploiting randomness in floating-point arithmetic," Computer Science Department, University of California, Los Angelse, Los Angeles, Technical Report CSD-970002, Mar. 1997.

[6] C. Denis, P. De Oliveira Castro, and E. Petit, "Verificarlo: Checking floating point accuracy through Monte Carlo arithmetic," in *Proc. 23rd IEEE Symp. Comput. Arithmetic*, Santa Clara, CA, USA, Jul. 2016, pp. 55–62. Available: https://doi.org/10.1109/ARITH.2016.31

[7] F. Févotte and B. Lathuilière, "VERROU: Assessing Floating-Point Accuracy Without Recompiling," Tech. Rep. hal-01383417, version 1, Oct. 2016. Available: https://hal.archives-ouvertes.fr/hal-01383417

[8] P. Blanchard, N. J. Higham, and T. Mary, "A class of fast and accurate summation algorithms," *SIAM J. Sci. Comput.*, vol. 42, no. 3, pp. A1541–A1557, Jan. 2020. Available: https://doi.org/10.1137/19M1257780

[9] N. J. Higham, "The accuracy of floating point summation," *SIAM J. Sci. Comput.*, vol. 14, no. 4, pp. 783–799, Jul. 1993. Available: https://doi.org/10.1137/0914050

[10] M. P. Connolly, N. J. Higham, and T. Mary, "Stochastic rounding and its probabilistic backward error analysis," Manchester Institute for Mathematical Sciences, The University of Manchester, UK, MIMS EPrint 2020.12, Aug. 2020. Available: http://eprints.maths.manchester.ac.uk/2778/

[11] M. Hopkins, M. Mikaitis, D. R. Lester, and S. Furber, "Stochastic rounding and reduced-precision fixed-point arithmetic for solving neural ordinary differential

equations," *Philos. Trans. R. Soc. A*, vol. 378, no. 2166, Jan. 2020, article ID 20190052. Available: https://doi.org/10.1098/rsta.2019.0052

[12] N. J. Higham and S. Pranesh, "Simulating low precision floating-point arithmetic," *SIAM J. Sci. Comput.*, vol. 41, no. 5, pp. C585–C602, Oct. 2019. Available: https://doi.org/10.1137/19M1251308

[13] M. Höhfeld and S. E. Fahlman, "Probabilistic rounding in neural network learning with limited precision," *Neurocomputing*, vol. 4, no. 6, pp. 291 – 299, Dec. 1992. Available: http://www.sciencedirect.com/science/article/pii/092523129290014G

[14] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proc. 32nd Int. Conf. Mach. Learning*, ser. Proceedings of Machine Learning Research, F. Bach and D. Blei, Eds., vol. 37. Lille, France: PMLR, Jul. 2015, pp. 1737–1746. Available: http://proceedings.mlr.press/v37/gupta15.html

[15] T. Na, J. H. Ko, J. Kung, and S. Mukhopadhyay, "On-chip training of recurrent neural networks with limited numerical precision," in *Int. Joint Conf. Neural Networks*, Anchorage, AK, USA, Jul. 2017, pp. 3716–3723. Available: https://doi.org/10.1109/IJCNN.2017.7966324

[16] S. Wu, G. Li, F. Chen, and L. Shi, "Training and inference with integers in deep neural networks," in *Int. Conf. Learning Representations*, Vancouver, BC, Canada, Apr. 2018.

[17] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, "Training deep neural networks with 8-bit floating point numbers," in *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Curran Associates, Inc., Dec. 2018, no. 31, pp. 7675–7684. Available: http://papers.nips.cc/paper/7994-training-deep-neural-networks-with-8-bit-floating-point-numbers.pdf

[18] C. Su, S. Zhou, L. Feng, and W. Zhang, "Towards high performance low bitwidth training for deep neural networks," *J. Semicond.*, vol. 41, no. 2, Feb. 2020. Available: https://doi.org/10.1088/1674-4926/41/2/022404

[19] Graphcore, "IPU programmer's guide," Online, 2020. Available: https://www.graphcore.ai/docs/ipu-programmers-guide

[20] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C.-K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y.-H. Weng, A. Wild, Y. Yang, and H. Wang, "Loihi: A neuromorphic manycore processor with on-chip learning," *IEEE Micro*, vol. 38, no. 1, pp. 82–99, Jan. 2018. Available: https://doi.org/10.1109/MM.2018.112130359

[21] S. Höppner and C. Mayr, "SpiNNaker2—towards extremely efficient digital neuromorphics and multi-scale brain emulation," in *Neuro-inspired Computional*

Elements Workshop, Portland, OR, US, Feb. 2018. Available: https://niceworkshop.org/wp-content/uploads/2018/05/2-27-SHoppner-SpiNNaker2.pdf

[22] S. Lifsches, "In-memory stochastic rounder," U.S. Patent 20 200 012 708A1, Jan. 9, 2020, application pending.

[23] G. H. Loh, "Stochastic rounding logic," U.S. Patent 20 190 294 412A1, Sep. 26, 2019, application pending.

[24] J. M. Alben, P. Micikevicius, H. Wu, and M. Y. Siu, "Stochastic rounding of numerical values," U.S. Patent 20 190 377 549A1, Dec. 12, 2019, application pending.

[25] J. D. Bradbury, S. R. Carlough, B. R. Prasky, and E. M. Schwarz, "Stochastic rounding floating-point multiply instruction using entropy from a register," U.S. Patent 10 445 066B2, Oct. 15, 2019.

[26] J. Riedy and J. Demmel, "Augmented arithmetic operations proposed for IEEE-754 2018," in *Proc. 25th IEEE Symp. Comput. Arithmetic*. Amherst, MA, USA: Institute of Electrical and Electronics Engineers, Jun. 2018, pp. 45–52. Available: https://doi.org/10.1109/ARITH.2018.8464813

[27] S. Boldo, C. Q. Lauter, and J.-M. Muller, "Emulating round-to-nearest-ties-to-zero "augmented" floating-point operations using round-to-nearest-ties-to-even arithmetic," *IEEE Trans. Comput.*, p. 1–1, Jun. 2020. Available: https://doi.org/10.1109/TC.2020.3002702

[28] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres, *Handbook of Floating-Point Arithmetic*, 2nd ed. Birkhäuser, 2018. Available: https://doi.org/10.1007/978-3-319-76526-6

[29] D. E. Knuth, *The Art of Computer Programming*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997, vol. 2: Seminumerical Algorithms.

[30] O. Møller, "Quasi double-precision in floating point addition," *BIT*, vol. 5, no. 1, pp. 37–50, Mar. 1965. Available: https://doi.org/10.1007/bf01975722

[31] T. J. Dekker, "A floating-point technique for extending the available precision," *Numer. Math.*, vol. 18, no. 3, pp. 224–242, Jun. 1971. Available: https://doi.org/10.1007/BF01397083

[32] J. R. Shewchuk, "Adaptive precision floating-point arithmetic and fast robust geometric predicates," *Discrete Comput. Geom.*, vol. 18, no. 3, pp. 305–363, Oct. 1997. Available: https://doi.org/10.1007/pl00009321

[33] S. Boldo, S. Graillat, and J.-M. Muller, "On the robustness of the 2Sum and Fast2Sum algorithms," *ACM Trans. Math. Software*, vol. 44, no. 1, p. 1–14, Jul. 2017. Available: https://doi.org/10.1145/3054947

[34] G. Bohlender, W. Walter, P. Kornerup, and D. W. Matula, "Semantics for exact floating point operations," in *Proc. 10th IEEE Symp. Comput. Arithmetic.* Grenoble, France: Institute of Electrical and Electronics Engineers, Jun. 1991. Available: https://doi.org/10.1109/ARITH.1991.145529

[35] S. Boldo and M. Daumas, "Representable correcting terms for possibly underflowing floating point operations," in *Proc. 16th IEEE Symp. Comput. Arithmetic.* Santiago de Compostela, Spain: Institute of Electrical and Electronics Engineers, Jun. 2003. Available: https://doi.org/10.1109/ARITH.2003.1207663

[36] N. Brisebarre, M. Joldes, P. Kornerup, E. Martin-Dorel, and J. Muller, "Augmented precision square roots and 2-D norms, and discussion on correctly rounding $\sqrt{x^2 + y^2}$," in *Proc. 20th IEEE Symp. Comput. Arithmetic*, Tubingen, Germany, Jul. 2011, pp. 23–30. Available: https://doi.org/10.1109/ARITH.2011.13

[37] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A multiple-precision binary floating-point library with correct rounding," *ACM Trans. Math. Software*, vol. 33, no. 2, pp. 13:1–13:15, Jun. 2007. Available: https://doi.org/10.1145/1236463.1236468

[38] "Semantics of floating point math in GCC," Online, Nov. 2019. Available: https://gcc.gnu.org/wiki/FloatingPointMath

[39] D. Malone, "To what does the harmonic series converge?" *Irish Math. Soc. Bull.*, no. 71, pp. 59–66, 2013. Available: https://www.maths.tcd.ie/pub/ims/bull71/recipnote.pdf

[40] W. M. Kahan, "Pracniques: Further remarks on reducing truncation errors," *Comm. ACM*, vol. 8, no. 1, p. 40, Jan. 1965. Available: https://doi.org/10.1145/363707.363723

[41] T. Ogita, S. M. Rump, and S. Oishi, "Accurate sum and dot product," *SIAM J. Sci. Comput.*, vol. 26, no. 6, pp. 1955–1988, Jul. 2005. Available: https://doi.org/10.1137/030601818

[42] D. M. Priest, "Algorithms for arbitrary precision floating point arithmetic," in *Proc. 10th IEEE Symp. Comput. Arithmetic*, Grenoble, France, Jun. 1991. Available: https://doi.org/10.1109/arith.1991.145549

[43] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2002. Available: https://doi.org/10.1137/1.9780898718027

[44] ——, "Goals of applied mathematical research," in *The Princeton Companion to Applied Mathematics*, N. J. Higham, M. R. Dennis, P. Glendinning, P. A. Martin, F. Santosa, and J. Tanner, Eds. Princeton, NJ, USA: Princeton University Press, 2015, pp. 48–55.

[45] T. R. Quinn, S. Tremaine, and M. Duncan, "A three million year integration of the Earth's orbit," *Astron. J.*, vol. 101, pp. 2287–2305, Jun. 1991. Available: https://doi.org/10.1086/115850