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 and Mantas Mikaitis

Abstract—We present algorithms for performing the four elementary arithmetic operations ($+$, $-$, $\times$, and $\div$) in floating point arithmetic with stochastic rounding, and discuss a few examples where using stochastic rounding may be 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, either in software or in hardware. The goal of these techniques is to emulate operations with stochastic rounding when the underlying hardware does not support this rounding mode, as is the case for most existing CPUs and GPUs. When stochastically rounding double precision operations, the algorithms we propose are on average over 5 times faster than an implementation that uses extended precision. We test our algorithms on various problems where stochastic rounding is expected to bring advantages: harmonic sum, summation of small random numbers, and ordinary differential equation solvers.

Index Terms—Computer arithmetic, floating-point arithmetic, multiword arithmetic, stochastic rounding, numerical analysis, numerical algorithms

1 Introduction

The IEEE 754-1985 standard for floating-point arithmetic [1, Sec. 4] specifies four rounding modes: the default round-to-nearest, denoted 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 [2, Sec. 4] defines 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. No major changes to this section were introduced in the most recent revision of the standard [3].

• *The authors are with the Department of Mathematics, University of Manchester, Manchester, M13 9PL, UK (*mantas.mikaitis@manchester.ac.uk*,* massimiliano.fasi@manchester.ac.uk*).*

Version of March 31, 2020.

These four 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 elementary arithmetic operations will always produce the same results if repeated. 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 at the current working precision. 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 2.

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 due to errors of different signs cancelling out. This rounding strategy is particularly effective at alleviating stagnation [4], 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" [5], causing a dramatic increase in forward error. We examine stagnation experimentally in Section 6.

Stochastic rounding is being increasingly used in artificial intelligence as well as in disciplines that rely heavily on numerical simulations, where it often improves the accuracy of the computation. In machine learning, it can help compensate the loss of accuracy caused by reducing the precision at which deep neural networks are trained in fixed-point [6] as well as floating-point [7] arithmetic. Similar improvements were shown in the use of numerical

methods for the solution of ordinary differential equations arising in the Izhikevich neuron model [8]. The general floating-point format simulator developed by Higham and Pranesh [9] includes stochastic rounding because, as the authors point out, there is a need to better understand its behavior.

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 [10]. The SpiNNaker2 neuromorphic chip [11] will be equipped with a hardware rounding accelerator designed to support, among others, fast stochastic rounding [12]. In general, various patents from AMD, NVIDIA, and other companies propose stochastic rounding implementations [13], [14], [15]. Of particular interest are two patents from IBM [16], [17], where the entropy from the registers is proposed as a source of randomness. We note, however, that a pseudorandom sequence should be preferred to true randomness for reproducible results.

Our contribution is twofold: on the one hand, we present algorithms for emulating stochastic rounding of addition, multiplication, and division of signed numbers; 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 *residual*, that is, 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 [9].

In general, this strategy cannot guarantee a faithful 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 [18].

The main drawback of this technique is that it requires the availability of an efficient mechanism to perform high-precision computation. To date, no major hardware vendor supports precision higher than binary64, thus implementing stochastic rounding in binary64 would necessarily require emulating binary128, or an even more precise floating-point arithmetic, via software. In Section 4 we show how the four 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 5. In Section 6, we explore three applications showing that stochastic rounding may be beneficial over the four rounding modes defined by the IEEE 754 standard. We summarize our contribution and point at directions for future work in Section 7.

1. https://github.com/higham/chop

2 Stochastic rounding

Let $\mathcal{F}$ be a normalized family of binary floating-point numbers with p digits of precision and maximum exponent $e_{\max}$, and let $\varepsilon := 2^{1-p}$. 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 $e = e_{\min}$. We denote the exponent of x by $\mathrm{exponent}(x)$.

A *rounding* is any function that maps the real line to $\mathcal{F}$. 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 x to its p most significant digits by $\mathrm{tr}(x) := \mathrm{sign}(x) \cdot 2^{1-p} \lfloor 2^{p-1}|x| \rfloor = \mathrm{RZ}(x)$, where $\mathrm{sign}(x)$ is 1 if $x \geq 0$ and -1 otherwise. The function $\mathrm{SR} : \mathbb{R} \to \mathcal{F}$ is a stochastic rounding if for any real number x in the range of $\mathcal{F}$ one has that

$$\mathrm{SR}(x) = \begin{cases} (-1)^s \cdot 2^e \cdot \mathrm{tr}(m), & \text{with p. } 1 - r_x, \\ (-1)^s \cdot 2^e \cdot (\mathrm{tr}(m) + \varepsilon), & \text{with p. } r_x, \end{cases} \tag{2.1}$$

where

$$r_x = \frac{m - \mathrm{tr}(m)}{\varepsilon}. \tag{2.2}$$

Note that $m - \mathrm{tr}(m) \in [0, \varepsilon)$, which implies that $r_x \in [0, 1)$.

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 (2.1) does not work, as $(-1)^s \cdot 2^e \cdot (\mathrm{tr}(m) + \varepsilon)$ is not representable in $\mathcal{F}$. In this case, it is more reasonable to assume $\mathrm{SR}(x) = \mathrm{sign}(x) \cdot x_{\max}$ rather than $\mathrm{SR}(x) = \mathrm{sign}(x) \cdot \infty$. This convention has two main advantages: it is consistent with the informal definition of stochastic rounding given above, and it ensures that only computations, but not roundings, can produce infinities.

The quantity r_x in (2.2) is related to the rounding error when rounding toward zero, since

$$x - \mathrm{RZ}(x) = x - \mathrm{tr}(x) = \mathrm{sign}(x) \cdot 2^e \cdot (m - \mathrm{tr}(m)) =: \varrho_x.$$

When other rounding modes are used, however, this is not necessarily the case. As ϱ_x depends only on x, we call it the residual of x.

We now discuss how to implement (2.1). 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

$$\mathrm{SR}(x) = \begin{cases} (-1)^s \cdot 2^e \cdot \mathrm{tr}(m), & X \geq r_x, \\ (-1)^s \cdot 2^e \cdot (\mathrm{tr}(m) + \varepsilon), & X < r_x, \end{cases} \tag{2.3}$$

where r_x is as in (2.2) and $X \sim \mathcal{U}_{[0,1)}$.

TABLE 2.1: Demonstration of stochastic roundings in a 2-bit case for every pair of X/Y and r_x. The table on the left considers (2.3), whereas (2.4) is shown on the right. Arrow directions correspond 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	$\downarrow$	$\uparrow$	$\uparrow$	$\uparrow$

Algorithm 3.1: TWOSUM augmented addition.

function TWOSUM($a \in \mathcal{F}$, $b \in \mathcal{F}$, $\circ : \mathbb{R} \to \mathcal{F}$)
Compute $\sigma, \tau \in \mathcal{F}$ s.t. $\sigma + \tau = a + b$.
  $\sigma \leftarrow \circ(a + b)$;
  $a' \leftarrow \circ(\sigma - b)$;
  $b' \leftarrow \circ(\sigma - a')$;
  $\delta_a \leftarrow \circ(a - a')$;
  $\delta_b \leftarrow \circ(b - b')$;
  $\tau \leftarrow \circ(\delta_a + \delta_b)$;
  return (σ, τ);

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 $\mathcal{F}$. An alternative way of implementing (2.1) can be obtained by substituting $Y = 1 - X$ in (2.3). The random variable Y thus defined takes values in the interval $(0, 1]$, but moving the equality from the first to the second case is enough to shift the domain of Y to $[0, 1)$ while ensuring that $\mathrm{SR}(x) = x$ when x belongs to $\mathcal{F}$. Therefore, definition (2.1) can be equivalently written as

$$\mathrm{SR}(x) = \begin{cases} (-1)^s \cdot 2^e \cdot \mathrm{tr}(m), & Y < 1 - r_x, \\ (-1)^s \cdot 2^e \cdot (\mathrm{tr}(m) + \varepsilon), & Y \geq 1 - r_x, \end{cases} \tag{2.4}$$

where $Y \sim \mathcal{U}_{[0,1)}$. We will rely on both (2.3) and (2.4) in later sections.

We note that definitions (2.3) and (2.4) are equivalent to (2.1) not only if X and Y are continuous random variables, but also in the discrete case. This is illustrated for the 2-bit rounding case in Table 2.1.

3 AUGMENTED OPERATIONS

The IEEE 754-2019 standard for floating-point arithmetic [3] includes, among the new recommended operations, three *augmented operations*: *augmentedAddition*, *augmentedSubtraction*, 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 a rounding error.

Algorithm 3.2: TWOPRODFMA augmented multiplication.

function TWOPRODFMA($a \in \mathcal{F}$, $b \in \mathcal{F}$, $\circ : \mathbb{R} \to \mathcal{F}$)
If a, b satisfy (3.1)*, compute $\sigma, \tau \in \mathcal{F}$ s.t. $\sigma + \tau = a \cdot b$.*
  $\sigma \leftarrow \circ(a \times b)$;
  $\tau \leftarrow \circ(a \times b - \sigma)$;
  return (σ, τ);

How to perform these tasks efficiently is a well-understood problem. Algorithms for augmented addition (and thus subtraction) and augmented multiplication are discussed in [19, Sec. 4.3] and [19, Sec. 4.4], respectively. The former can be performed efficiently by using the function TWOSUM in Algorithm 3.1, due to Knuth [20, Th. B] and Møller [21], which for $\circ = \mathrm{RN}$ 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 achieved more efficiently by using Dekker's FASTTWOSUM [22], which requires only 3 operations in round-to-nearest. The names we use for these two routines were originally proposed by Shewchuck [23].

When dealing with augmented multiplication, extra care is required, as in this case it is necessary to ensure that overflow does not occur. In [19, Sec. 4.4] it is shown that if $a, b \in \mathcal{F}$ and

$$\mathrm{exponent}(a) + \mathrm{exponent}(b) \geq e_{\min} + p - 1, \tag{3.1}$$

then $\tau = a \cdot b - \circ(a \times b)$, with $\circ \in \{\mathrm{RN}, \mathrm{RD}, \mathrm{RU}, \mathrm{RZ}\}$, 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 3.2, which requires only two floating-point operations and guarantees that if a and b satisfy (3.1), then $\sigma + \tau = a \cdot b$ regardless of the rounding mode used for the computation.

If an FMA is not available, another algorithm due to Dekker [22] 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 [19, Alg. 4.10]. This algorithm also requires that condition (3.1) hold, but is proven to work only when round-to-nearest is used.

4 OPERATIONS WITH STOCHASTIC ROUNDING

In order to perform stochastic rounding as defined in Section 2, we need to know r_x in (2.2). It may be possible to compute this quantity exactly, if the operation is carried out in higher precision and then rounded to lower precision, but the value of r_x is not available when one wishes to round the result of an arithmetic operation performed in hardware to the same precision as the arguments. One example is rounding the sum of two double-precision numbers with

Algorithm 4.1: Stochastically rounded addition.

```
1  function SRADD(a ∈ F, b ∈ F)
   Compute ϱ = SR(a + b) ∈ F.
2      Z ← rand();
3      (σ, τ) ← TWOSUM(a, b, RN);
4      η ← 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;
```

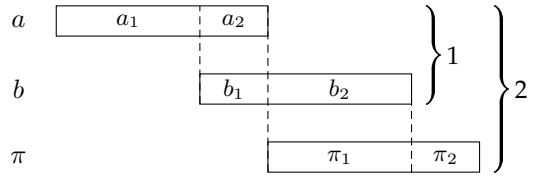


Fig. 4.1: Alignment of the fractions of a, b, and π on line 10 of Algorithm 4.1.

different exponents: the fraction of the operand smaller in magnitude will have to be shifted right to match the exponent of the other summand, causing roundoff bits to appear.

4.1 Addition

The first 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. This approach is shown in Algorithm 4.1.

In the pseudocode, $\mathrm{rand}()$ 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 of the sum computed in round-toward-zero. Then it generates a p-digit floating-point number in the interval $[0, 1)$ and scales it, by multiplying by the value of the least significant digit of $\mathrm{RZ}(a + b)$, so that it has the same sign as the rounding error τ and absolute value in $[0, 2^\eta\varepsilon)$.

Finally, stochastic rounding is performed by computing $(\tau + \pi) + \sigma$, in the order indicated by the parentheses, using round-toward-zero. The alignment of a, b, and π in Algorithm 4.1 is illustrated in Fig. 4.1.

We now argue the correctness of the algorithm. Note that if $\sigma = a+b$, then $\tau = 0$ and $0 \le Z < 1$ guarantees that $\tau = \sigma$ on line 10.

If σ and τ have the same sign, then $|\sigma| < |a + b|$, and $\varrho = \mathrm{tr}(a+b)$ if and only if $|\pi| < 2^\eta\varepsilon - |\tau|$, or equivalently if and only if $Z < 1-r_{a+b}$. Similarly, $\varrho = \mathrm{tr}(a+b)+2^\eta\varepsilon$ if and only if $Z \ge 1 - r_{a+b}$, and we conclude that Algorithm 4.1 implements (2.4) when $\mathrm{sign}(\sigma) = \mathrm{sign}(\tau)$.

Algorithm 4.2: Fast stochastically rounded addition.

```
1  function SRADD2(a ∈ F, b ∈ F)
   Compute ϱ = SR(a + b) ∈ F.
2      Z ← rand();
3      (σ, τ) ← TWOSUM(a, b, RZ);
4      η ← exponent(σ);
5      π ← sign(τ) × Z × 2^η × ε;
6      ϱ ← RZ(RZ(τ + π) + σ);
7      return ϱ;
```

Algorithm 4.3: Multiplication with stochastic rounding using the FMA instruction.

```
1  function SRMULFMA(a ∈ F, b ∈ F)
   If a, b satisfy (3.1), compute ϱ = SR(a · b) ∈ F.
2      Z ← rand();
3      (σ, τ) ← TWOPRODFMA(a, b, RZ);
4      η ← exponent(σ);
5      π ← sign(τ) × Z × 2^η × ε;
6      ϱ ← RZ(⋄(τ + π) + σ);
7      return ϱ;
```

If σ and τ have opposite sign, on the other hand, then $|\sigma| > |a + b|$. Thus $\varrho = \mathrm{tr}(a + b)$ if and only if $|\pi| \ge 2^\eta\varepsilon - |\tau|$, which can be equivalently rewritten as $Z \ge r_{a+b}$, since $r_{a+b} = 1 - \tau/\varepsilon$ in this case. The case $\varrho = \mathrm{tr}(a + b) + 2^\eta\varepsilon$ is analogous, which shows that Algorithm 4.1 implements (2.3) for $\mathrm{sign}(\sigma) = -\mathrm{sign}(\tau)$.

The diagram in Fig. 4.2 justifies the use of different rounding modes. The idea is that $|\mathrm{RZ}(\tau + \pi)|$ 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, $|\mathrm{RZ}(\tau + \pi)|$ ends up being smaller than $2^\eta\varepsilon$, then the sum computed in rounding to nearest remains unchanged and is returned.

The error on the quantity r_{a+b} used by Algorithm 4.1 depends on which rounding operator $\diamond$ is chosen on line 10. If we denote by $\widehat{r}_{a+b}$ the approximate quantity used by the algorithm, it is easy to see that for round-to-nearest and directed rounding we have that $|\widehat{r}_{a+b} - r_{a+b}| < 2^\eta\varepsilon^2$ and $|\widehat{r}_{a+b} - r_{a+b}| < 2^{\eta-p}\varepsilon$, respectively.

Algorithm 4.2 is a faster variant of Algorithm 4.1 in which round-toward-zero is used everywhere. We note that TWOSUM does not return an exact unevaluated sum of the two arguments with this rounding mode. Specifically, for floating-point systems with $p \ge 4$ bits of precision, [19, Th. 4.7] shows that if overflow does not occur then TWOSUM satisfies $\tau = (a + b) - \sigma + \alpha$ with $|\alpha| < 2^{-p+1} \cdot \mathrm{ulp}(a + b)$, where $\mathrm{ulp}(a + b)$ is an upper error bound in the addition operation with round-toward-zero [19, Sec. 2.3.2]. In our terms, $|\alpha| < 2^\eta\varepsilon^2$.

4.2 Multiplication

The function SRMULFMA in Algorithm 4.3 exploits TWOPRODFMA to compute $\mathrm{SR}(a \times b)$. Since the algorithm works with any rounding mode [19, Sec. 4.4.1], we use round-toward-zero in order to obtain a more efficient algorithm. In this way, we compute the *residual* as τ and

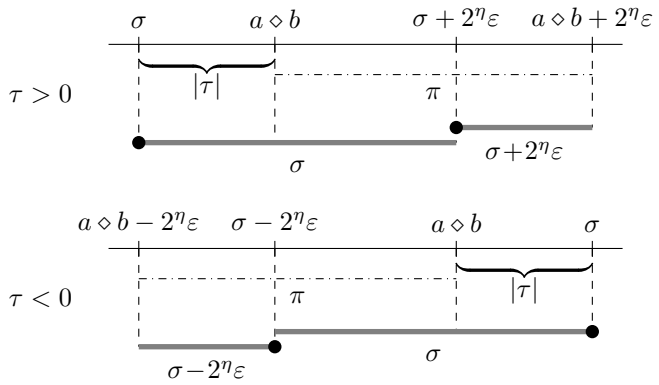


Fig. 4.2: Diagram that motivates the use of different directed rounding modes depending on the sign of τ. The dot dashed line represents the range of the variable π, numbers that fall in the range of a thick grey line are rounded in the direction of the black dot at one end. The symbol $\diamond$ represents 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 4.4: Multiplication with stochastic rounding using Dekker's algorithm.

1 **function** SRMULDEKKER($a \in \mathcal{F}$, $b \in \mathcal{F}$)
  If a, b satisfy (3.1), *compute* $\varrho = \mathrm{SR}(a \cdot b) \in \mathcal{F}$.
2 $Z \leftarrow \mathrm{rand}()$;
3 $(\sigma, \tau) \leftarrow$ TWOPRODDEK(a, b);
4 $\eta \leftarrow \mathrm{exponent}(\mathrm{RZ}(a \times b))$;
5 $\pi \leftarrow \mathrm{sign}(t) \times Z \times 2^\eta \times \varepsilon$;
6 **if** $\tau \geq 0$ **then**
7 $\circ = \mathrm{RD}$;
8 **else**
9 $\circ = \mathrm{RU}$;
10 $\varrho \leftarrow \circ(\diamond(\tau + \pi) + \sigma)$;
11 **return** ϱ;

the exponent can be calculated directly from σ, without requiring an extra floating-point operation as was the case in Algorithm 4.1.

The correctness of Algorithm 4.3 can be shown with an argument analogous to that used for Algorithm 4.1. 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 4.4. As it 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 $\mathrm{tr}(a \times b)$ is necessary. This corresponds to the operation on line 4 of SRADD in Algorithm 4.1.

4.3 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 $1/11 = 0.\overline{01} = 0.010101\ldots$.

Algorithm 4.5: Division with stochastic rounding.

1 **function** SRDIV($a \in \mathcal{F}$, $b \in \mathcal{F}$)
  Compute $\varrho = \mathrm{SR}(a \div b) \in \mathcal{F}$.
2 $Z \leftarrow \mathrm{rand}()$;
3 $\sigma \leftarrow \mathrm{RZ}(a \div b)$;
4 $\tau' \leftarrow \mathrm{RZ}(-\sigma \times b + a)$;
5 $\tau \leftarrow \mathrm{RZ}(\tau' \div b)$;
6 $\eta \leftarrow \mathrm{exponent}(\sigma)$;
7 $\pi \leftarrow \mathrm{sign}(\tau) \times Z \times 2^\eta \times \varepsilon$;
8 $\varrho \leftarrow \mathrm{RZ}(\diamond(\tau + \pi) + \sigma)$;
9 **return** ϱ;

In order to obtain an algorithm for division, we exploit a result by Bohlender et al. [24]. 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 [25, Th. 4]) $\tau' := a - \sigma \cdot b$ 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. If an FMA is not available, then Dekker's multiplication algorithm can be used to compute τ'. 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 4.5.

The error $|\widehat{r}_{a\div b} - r_{a\div b}|$ is larger than that of the other algorithms discussed so far, since only the approximate residual $\tau \geq 0$ is available. We note, however, that this error is of the same magnitude as that introduced by rounding $\tau + \pi$, which suggests that $|\widehat{r}_{a\div b} - r_{a\div b}| < 2^{\eta+1}\varepsilon$.

5 PERFORMANCE

In this section we evaluate experimentally the performance of the techniques in Section 4. We implemented Algorithms 4.1, 4.2, 4.3, and 4.5 in C, and compiled our experiments with GCC 8.3 using the optimization flags `-O3` and `-mach=native`. The experiments were run on a GNU/Linux machine equipped with an Intel Core i5-6500 CPU and 16 GiB of RAM. We enabled the use of FMA with `-mfma`, and disabled the use of optimizations that assume default floating-point rounding behavior and the storage of floating-point variables in registers using the flags `-frounding-math` and `-ffloat-store`, respectively. Note that the latter flag is necessary as the Intel processor we are using has 80-bit extended format registers.

We compared our methods with a C port of the stochastic rounding functionalities of the MATLAB `chop` function [9], which simulates floating-point arithmetics with a choice of rounding modes and a wide range of precisions. As our focus in this section is on binary64 arithmetic, we used the GNU MPFR library [26] (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 are available on GitHub.[2]

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

TABLE 5.1: Throughput (in Mop/s) of our 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 throughtput of the MPFR variant that uses 113 bits to perform the same operation.

	`sr_mpfr_sum`			SRADD	SRADD2	`sr_mpfr_mul`			SRMUL	`sr_mpfr_div`			SRDIV
p	61	88	113	–	–	61	88	113	–	61	88	113	–
min	5.31	4.85	5.02	20.77	22.82	4.45	4.53	4.28	23.69	4.38	4.41	4.13	21.53
max	6.71	6.49	6.56	28.96	34.46	6.34	5.91	5.98	32.10	6.14	5.74	5.70	29.55
mean	6.29	6.05	6.08	26.81	32.23	5.73	5.42	5.37	30.25	5.51	5.26	5.35	28.11
↪ *speedup*	1.03×	0.99×	1.00×	4.41×	5.30×	1.07×	1.01×	1.00×	5.63×	1.03×	0.98×	1.00×	5.25×
deviation	0.26	0.27	0.24	1.34	1.96	0.17	0.22	0.26	1.52	0.21	0.17	0.14	1.36

In Table 5.1 we consider the throughput (in Mop/s, millions of operations per second) of the functions we implemented on a test set of 1,000 pairs of uniformly distributed binary64 random numbers in the interval $[0, 1)$. For each pair of floating-point inputs, we estimate the throughput by running each algorithm 100,000 times, and in the table report the minimum, maximum, and mean value over the 1,000 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.

The new algorithms that work only in binary64 arithmetic are up to 5 times more efficient than those relying on the GNU MPFR library, regardless of the number of extra digits of precision used.

6 NUMERICAL EXPERIMENTS

Now we gauge the accuracy of the new algorithms in Section 4. 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). Our codes can be found on GitHub.[3] The experiments were run in MATLAB 9.7 (2019b) using the Stochastic Rounding Toolbox we developed, also available on GitHub.[4] Reduced-precision floating-point formats were simulated on binary64 hardware using the MATLAB `chop` function [9].

6.1 Harmonic series

In exact arithmetic, the harmonic series

$$\sum_{i=1}^{\infty} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \dots \tag{6.1}$$

is divergent. If the partial sums of (6.1) are evaluated in finite precision, however, this is not the case: using binary64 arithmetic and round-to-nearest, Malone [27] 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 [9], 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

$$H_k(s_0) := s_0 + \sum_{i=1}^{k} \frac{1}{i} = s_0 + 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{k} \tag{6.2}$$

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 4.1 and one that relies on the MATLAB `chop` function [9]. We use a single stream of random numbers produced by the `mrg32k3a` generator seeded with 300, and at each step we generate only one random number and use it for both algorithms. For round-to-nearest we consider, besides recursive summation at working precision, compensated summation [28], which at each step computes the rounding error with TWOSUM and adds it to the next summand, and cascaded summation [29], 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 [19, Sec. 5.3.2], [30] 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; this is easily achieved with bfloat16 and binary16 in 10^5 steps. 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. 6.1(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 4.1 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.

3. https://github.com/mmikaitis/stochastic-rounding-evaluation
4. https://github.com/mfasi/srtoolbox

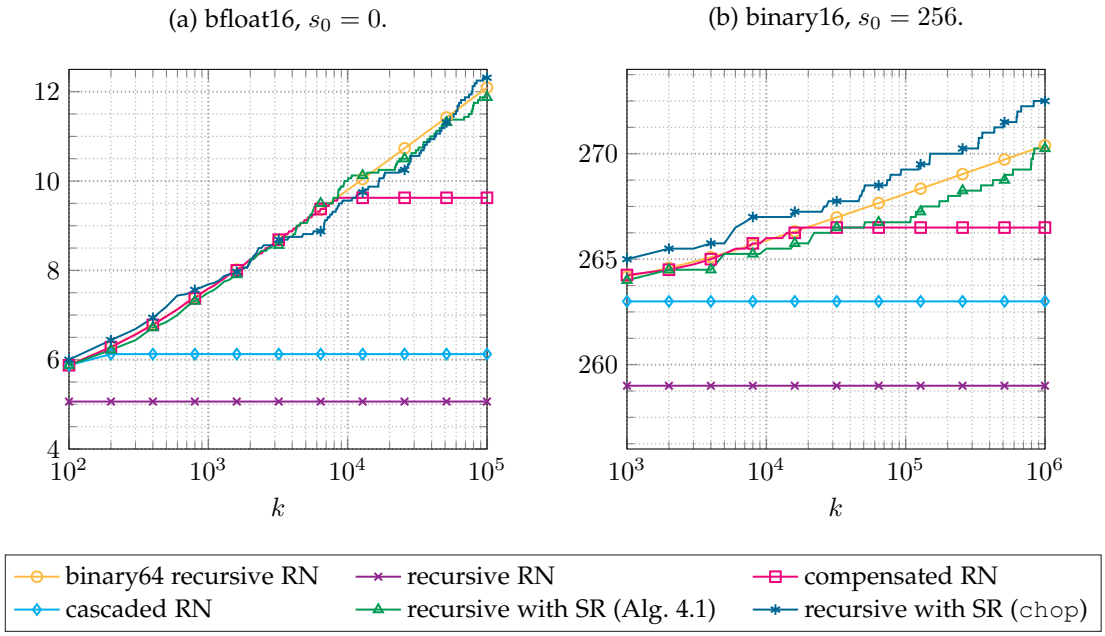


Fig. 6.1: Numerical value of the sum $H_k(s_0)$ in (6.2) accumulated in bfloat16 (left) and binary16 (right) arithmetics with various summation algorithms. The sum computed in binary64 precision is taken as reference. The algorithms use round-to-nearest (RN) or stochastic rounding (SR) as indicated in the legend.

6.2 Sum of random values

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

$$S_k(s_0) = s_0 + \sum_{i=1}^{k} x_i, \tag{6.3}$$

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 second stream of random numbers seeded with a value of 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. 6.2 shows these results. 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 as per the IEEE 754 standard, and 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 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. 6.3. 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.

6.3 ODE solvers

6.3.1 *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 [8]. Here we use the algorithms developed in Section 4 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 [9] 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 [31, Sec. 4.3]. We use their code[5] and compare round-to-nearest and stochastic rounding modes 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

5. https://github.com/SrikaraPranesh/LowPrecision_Simulation

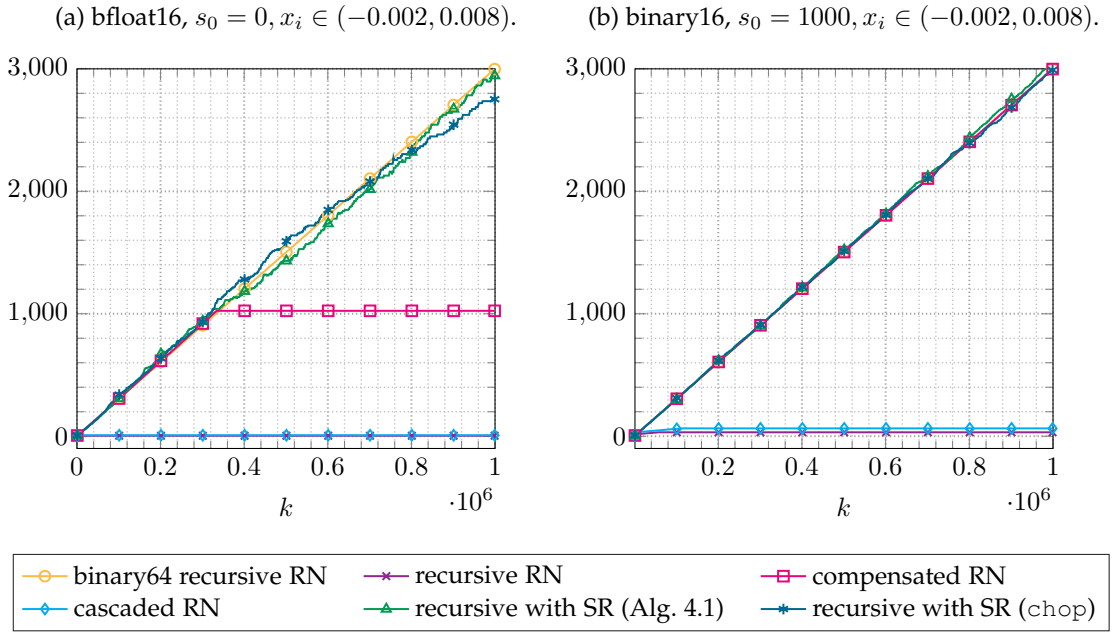


Fig. 6.2: Numerical value of the sum $S_k(s_0)$ in (6.3) accumulated in bfloat16 (left) and binary16 (right) arithmetics using various algorithms. The algorithms use round-to-nearest (RN) or stochastic rounding (SR) as indicated in the legend.

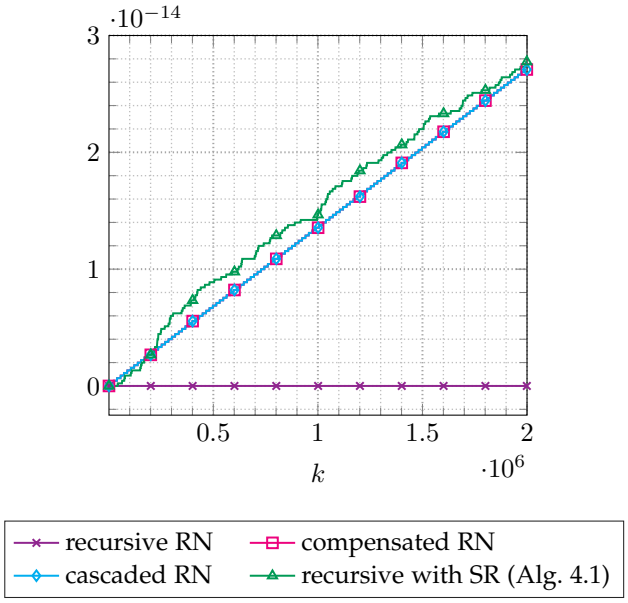


Fig. 6.3: Numerical value of $S_k(s_0) - 1$, for the sum $S_k(s_0)$ as defined in (6.3), accumulated in binary64 arithmetic using various algorithms with $s_0 = 1$, $x_i \in (0, 2^{-65})$. The algorithms use round-to-nearest (RN) or stochastic rounding (SR) as indicated in the legend.

$y_{n+1} = y_n + hf(t_n, y_n)$ with $h = 1/n$ for $n \in [10, 10^6]$. Fig. 6.4(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.

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

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

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

- Heun's method

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

The results for these two methods are reported in Fig. 6.4(b) and Fig. 6.4(c), respectively.

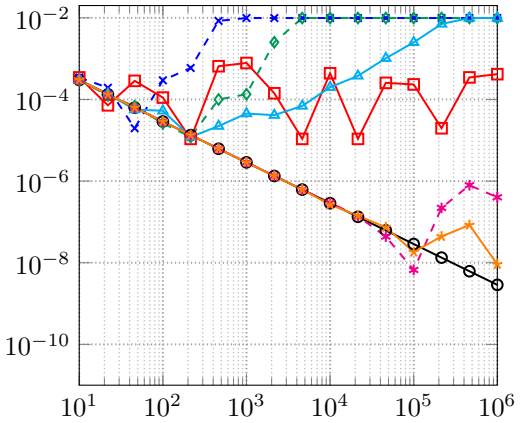
To cause stagnation in binary32 we reformulated the problem to have a smaller integration period and a larger initial condition. For instance we could consider the same ODE $y' = -y$, but take as initial condition $y(0) = 1$ over $[0, 2^{-6}]$ with $h = 2^{-6}/n$ for $n \in [10, 10^6]$. We only need 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. 6.4(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.

6.3.2 *Unit circle ODE*

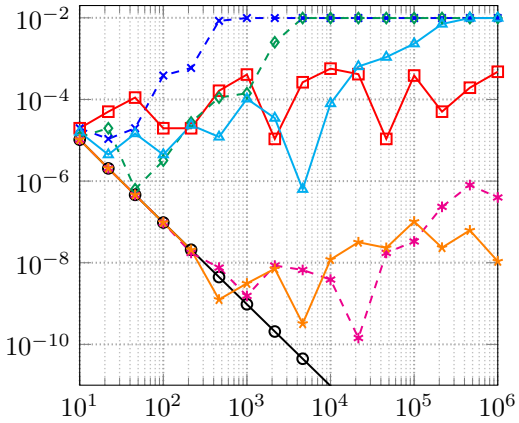
The solution to the ODE

$$u'(t) = v(t),$$
$$v'(t) = -u(t),$$

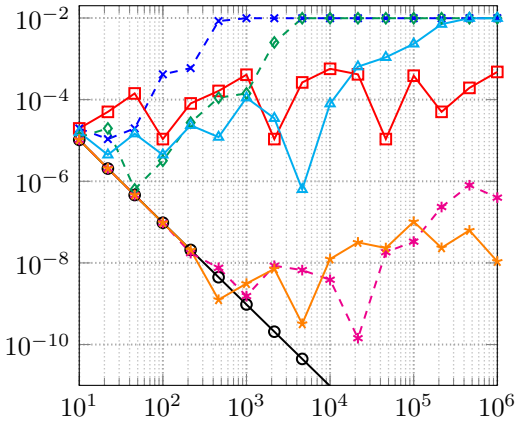
(a) Euler for $y' = -y$, $y(0) = 2^{-6}$, over $[0, 1]$.



(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 $[0, 2^{-6}]$.

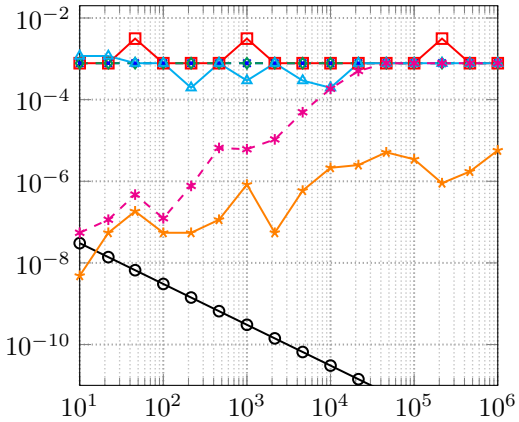


Fig. 6.4: Absolute errors in Euler, Midpoint, and Heun methods for the exponential decay ODE solutions with different floating point arithmetics and rounding modes. The x-axis represents n while y-axis represents the error.

with initial values $u(0) = 1$ and $v(0) = 0$ represents the unit circle in the uv-plane [32, p. 51]. Higham [32, p. 51] shows also that the forward Euler scheme

$$\begin{aligned} u_{k+1} &= u_k + hv_k, \\ v_{k+1} &= v_k - hu_k, \end{aligned} \tag{6.4}$$

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 unit circle at a higher total 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 (6.4). Fig. 6.5 shows some circles drawn when solving (6.4) for various step sizes $h = 2\pi/n$.

As expected, for large step sizes the solution spirals out of the unit circle, then gets gradualy 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. 6.5c and 6.5g 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

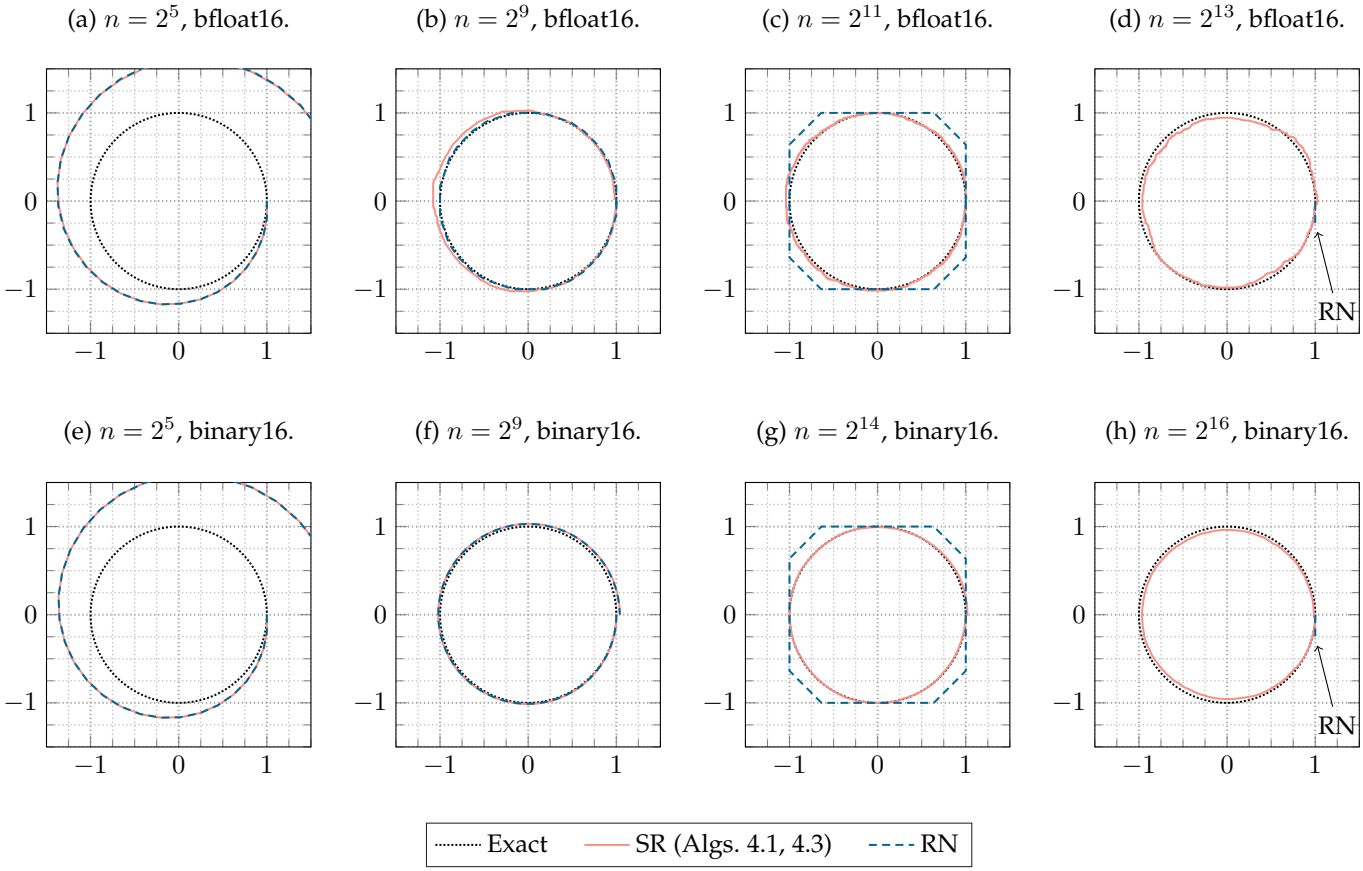


Fig. 6.5: Unit circle drawn by Euler's method in (6.4) with various arithmetics and rounding modes compared to the exact solution. The default MATLAB random number generator seeded with 500 was used. The x- and y-axis represent u and v, respectively. Note that in (d) and (h) only a very small part of the solution with RN is visible (marked with an arrow) since the ODE solver failed due to stagnation.

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,

$$\begin{aligned} u_7 &= 1 - 21h^2 + 36h^4 - 7h^6, \\ v_7 &= -7h + 35h^3 - 21h^5 + h^7. \end{aligned} \tag{6.5}$$

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. 6.5 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, as is evident from the diagrams, 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. 6.5d and 6.5h, 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 (6.5): 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 (6.5) 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 [33, 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.

7 Conclusions

There is growing interest in stochastic rounding [12], [18]. 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, our new 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 well above 5x over an MPFR-based multiple-precision approach.

We have 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 starting to appear in hardware.

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

8 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 work of M. Mikaitis was supported by an EPSRC Doctoral Prize Fellowship.

References

[1] *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, 1985, reprinted in SIGPLAN Notices, 22(2):9–25, 1987. Available: https://doi.org/10.1109/IEEESTD.1985.82928

[2] *IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2008 (revision of IEEE Std 754-1985)*. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, 2008. Available: https://doi.org/10.1109/IEEESTD.2008.4610935

[3] *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, 2019. Available: https://doi.org/10.1109/IEEESTD.2008.4610935

[4] P. Blanchard, N. J. Higham, and T. Mary, "A class of fast and accurate summation algorithms," Manchester Institute for Mathematical Sciences, The University of Manchester, UK, MIMS EPrint 2019.6, Apr. 2019, revised February 2020. To appear in SIAM J. Sci. Comput. Available: http://eprints.maths.manchester.ac.uk/2748/

[5] 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

[6] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proceedings of the 32nd International Conference on Machine 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

[7] 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 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Curran Associates, Inc., 2018, pp. 7675–7684. Available: http://papers.nips.cc/paper/7994-training-deep-neural-networks-with-8-bit-floating-point-numbers.pdf

[8] 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

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

[10] 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, 2018. Available: https://doi.org/10.1109/MM.2018.112130359

[11] S. Höppner and C. Mayr, "SpiNNaker2-Towards Extremely Efficient Digital Neuromorphics and Multi-scale Brain Emulation," in *2018 NICE Workshop*, 2018. Available: https://niceworkshop.org/wp-content/uploads/2018/05/2-27-SHoppner-SpiNNaker2.pdf

[12] M. Mikaitis, "Stochastic rounding: Algorithms and hardware accelerator," arXiv:2001.01501 [cs.AR], 2020. Available: https://arxiv.org/abs/2001.01501

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

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

[15] 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.

[16] 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.

[17] ——, "Stochastic rounding floating-point add instruction using entropy from a register," U.S. Patent 10 489 152B2, Nov 26, 2019.

[18] M. P. Connolly, N. J. Higham, and T. Mary, "Stochastic rounding and its probabilistic backward error analysis," 2020, in preparation.

[19] 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

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

[21] 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

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

[23] 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

[24] G. Bohlender, W. Walter, P. Kornerup, and D. W. Matula, "Semantics for exact floating point operations," *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, 1991. Available: https://doi.org/10.1109/ARITH.1991.145529

[25] S. Boldo and M. Daumas, "Representable correcting terms for possibly underflowing floating point operations," *Proceedings of the 16th IEEE Symposium on Computer Arithmetic*, 2003. Available: https://doi.org/10.1109/ARITH.2003.1207663

[26] 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, 2007. Available: https://doi.org/10.1145/1236463.1236468

[27] 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

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

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

[30] D. M. Priest, "Algorithms for arbitrary precision floating point arithmetic," in *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, 1991. Available: https://doi.org/10.1109/arith.1991.145549

[31] 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

[32] ——, "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. Available: https://assets.press.princeton.edu/chapters/s1-5_10592.pdf

[33] 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, 1991. Available: https://doi.org/10.1086/115850