Stochastic Rounding: Implementation, Error Analysis, and Applications

Matteo Croci, Massimiliano Fasi, Nicholas J Higham, Théo Mary, Mantas Mikaitis

ROYAL SOCIETY OPEN SCIENCE

rsos.royalsocietypublishing.org

Research

Draft version of October 13, 2021

Subject Areas:
Mathematics, Computer Science

Keywords:
Floating-point Arithmetic, Rounding Error Analysis, Computer Arithmetic, IEEE 754, binary16, bfloat16, Low Precision, Ordinary Differential Equations, Partial Differential Equations

Author for correspondence:
Mantas Mikaitis
e-mail:
mantas.mikaitis@manchester.ac.uk

Stochastic Rounding: Implementation, Error Analysis, and Applications

Matteo Croci[1], Massimiliano Fasi[2], Nicholas J. Higham[3], Theo Mary[4], and Mantas Mikaitis[3]

[1]Mathematical Institute, University of Oxford, Oxford, OX2 6GG, United Kingdom.
[2]Department of Computer Science, Durham University, Durham, DH1 3LE, United Kingdom.
[3]Department of Mathematics, The University of Manchester, Manchester, M13 9PL, United Kingdom.
[4]Sorbonne Université, CNRS, LIP6, Paris, F-75005, France.

Stochastic rounding randomly maps a real number to one of the two nearest values in a finite precision number system. First proposed for use in computer arithmetic in the 1950s, it is attracting renewed interest. If used in floating-point arithmetic in the computation of the inner product of two vectors of length n, it yields an error bounded by $\sqrt{n}u$ with high probability, where u is the unit roundoff, which is not necessarily the case for round to nearest. A particular attraction of stochastic rounding is that, unlike round to nearest, it is immune to the phenomenon of stagnation, whereby a sequence of tiny updates to a relatively large quantity are lost. We survey stochastic rounding, covering its mathematical properties and probabilistic error analysis, its implementation, and its use in applications, including deep learning and the numerical solution of differential equations.

1. Introduction

Rounding is the act of mapping a given number to one having a certain number of digits in a given base. For illustration, consider the task of rounding in base 10 a 2-significant-digit number to 1 significant digit. If we round to the closest 1-digit number then 1.4 rounds to 1 and 1.7 rounds to 2, for example. We denote the rounding operator by fl, thus we write $\mathrm{fl}(1.4) = 1$ and $\mathrm{fl}(1.7) = 2$.

THE ROYAL SOCIETY PUBLISHING

This rounding rule, called round to nearest (RN), is deterministic: the value of $\mathrm{fl}(x)$ depends only on x, and repeating the rounding will not yield a different result.

Suppose we want to compute 1 + 0.1 in 1-digit base-10 arithmetic. With round to nearest we obtain $\mathrm{fl}(1+0.1)=1$. Another option is to round to either of the two nearest 1-digit numbers with a probability that depends on the distances to those numbers. If in our example we define $\mathrm{fl}(1+0.1)$ as 1 with probability 0.9 and as 2 with probability 0.1, then the expected result is $0.9\times 1+0.1\times 2=1.1$, which is the exact answer. This probabilistic rounding is called *stochastic rounding* (SR), and this simple example demonstrates why it can be useful in practice.

SR was first proposed over sixty years ago, but until recently had proved useful only in rather specialised contexts. In the last five years or so, however, this rounding mode has enjoyed a resurgence of interest, mainly because of the increasing availability of low precision floating-point arithmetic in hardware. When solving large problems in low precision, the accumulation of rounding errors can cause all accuracy to be lost with standard rounding modes. Using SR in place of round to nearest can attenuate the growth of worst-case error bounds and provide more accurate solutions, since statistical effects allow errors to cancel out, at least to some extent.

The aim of this work is to survey SR, describing

- its history,
- its basic properties compared with RN,
- how it can be implemented,
- its probabilistic rounding error analysis, and
- how and why it is being used in applications.

2. What is stochastic rounding?

Let $F\subseteq\mathbb{R}$ denote a number system. For $x\in\mathbb{R}$, define the two rounding candidates

$$\lfloor x\rfloor=\max\{\,y\in F: y\le x\,\},\quad \lceil x\rceil=\min\{\,y\in F: y\ge x\,\},$$

so that $\lfloor x\rfloor\le x\le\lceil x\rceil$, with equality throughout if $x\in F$. Note that when $x\notin F$, the two numbers $\lfloor x\rfloor$ and $\lceil x\rceil$ are adjacent in F. We denote by fl any rounding operator that maps numbers in $\mathbb{R}$ to either of the two nearest numbers in F.

For $x\in\mathbb{R}\setminus F$, SR is defined by

$$\mathrm{fl}(x)=\begin{cases}\lceil x\rceil \text{ with probability } q(x),\\ \lfloor x\rfloor \text{ with probability } 1-q(x),\end{cases}\tag{2.1}$$

where $q(x)\in[0,1]$. The simplest choice is $q(x)=0.5$, in which case we round up or down with equal probability, independently of x. As is customary in the literature [1], [2], we call this less commonly used form of SR *mode 2 SR*. Another choice is to set in (2.1)

$$q(x)=\frac{x-\lfloor x\rfloor}{\lceil x\rceil-\lfloor x\rfloor},\tag{2.2}$$

which means that we round x to the next (larger or smaller) number y with a probability that is 1 minus the relative distance of x to y. The choice (2.2) yields *mode 1 SR*, which is the most interesting stochastic rounding mode from a numerical point of view. Unless otherwise stated, here SR means mode 1 SR. See Figure 2.1 for an illustration.

For the rest of this paper, we take F to be a floating-point number system, unless otherwise stated, as this is the case of most interest.

In RN, which is the default rounding mode in most floating-point arithmetics, $\mathrm{fl}(x)$ is the number in F nearest to x, with some tie-breaking strategy for handling the case where x is equidistant from the next and previous floating-point numbers. While SR and RN share some important properties, they also differ in some important respects. We first describe some features that they have in common.

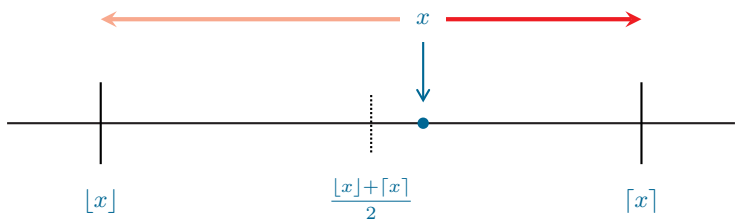

Figure 2.1. Stochastic rounding rounds the real number x to the next smaller number $\lfloor x \rfloor$ in F or to the next larger number $\lceil x \rceil$ in F. In this example, RN rounds x to $\lceil x \rceil$, whereas mode 1 SR can round to either $\lfloor x \rfloor$ or $\lceil x \rceil$ but is more likely to round to $\lceil x \rceil$.

- If $x \in \mathbb{R}$, then $\mathrm{fl}(\mathrm{fl}(x)) = \mathrm{fl}(x)$, that is, rounding a floating-point number leaves it unchanged.
- If x and y are floating-point numbers with $y/2 \le x \le 2y$ then $\mathrm{fl}(x - y) = x - y$ (assuming subnormal numbers are supported). This result, known as Sterbenz's lemma [3, Thm. 2.5], [4, Thm. 1.8.2], holds for any rounding mode.
- In base-2 arithmetic, any floating-point numbers x and y such that $x \le y$ satisfy the inequalities

$$x \le \frac{\mathrm{fl}(x + y)}{2} \le y, \tag{2.3}$$

  which do not hold for all bases.

SR and RN differ in some key properties, however. For example, if $x \notin F$ then in general $\mathrm{fl}(|x|) \ne |\mathrm{fl}(x)|$ and $\mathrm{fl}(-x) \ne -\mathrm{fl}(x)$ for SR. Moreover, SR is not monotonic: $x < y$ does not imply that $\mathrm{fl}(x) \le \mathrm{fl}(y)$, as can be seen by considering any pair of reals x and y lying between two adjacent floating-point numbers.

Several results are available that describe how simple identities for real numbers are (partially) preserved for floating-point numbers under RN. Generally similar, but weaker, results hold for SR. For a binary format with RN, for example, we have that $\mathrm{fl}(\sqrt{x^2}) = |x|$ if x is a floating-point number [3, Prob. 2.20], barring underflow and overflow. For SR, however, if x is a floating-point number in the interval $(1, 2)$ then $\mathrm{fl}(\sqrt{x^2}) \in \{|x| - \varepsilon_M, |x|, |x| + \varepsilon_M\}$, where ε_M is the machine epsilon defined in Section 4(a). A consequence of this fact is that the inequality $\mathrm{fl}(x/\sqrt{x^2 + y^2}) \le 1$, which is always satisfied by RN [3, Prob. 2.21], is not necessarily true when SR is used.

Full details of the above results, as well as other properties that differ between SR and RN, are given by Connolly, Higham, and Mary [1, Sec. 3]. Before replacing RN with SR it is vital to consider whether a certain computation relies on properties of RN that go beyond the standard model of floating-point arithmetic (6.1) below, and if so, whether these properties remain true for SR. The solution of the quadratic equation $ax^2 + bx + c = 0$ is a striking example of the subtle issues that may occur when switching from RN to SR: if evaluated using SR, the discriminant $b^2 - 4ac$ can be negative even when $b^2 > 4ac$. This is a consequence of the non-monotonicity of SR which in some cases may lead one to conclude, incorrectly, that an equation has no real solutions when in fact it has two almost identical real roots.

3. Early history of stochastic rounding

To the best of our knowledge, the earliest proposal of SR was in a one-paragraph abstract of a communication presented by Forsythe in 1949 at the fifty-second meeting of the American Mathematical Society [5]. The abstract claims that SR can be used to reduce the accumulation of round-off errors observed by Huskey [6] in solving a simple system of ordinary differential equations (ODEs). The numerical integration that Forsythe and Huskey consider entails a sum

of real values which is further reduced to a sum of integers, most likely intended as fixed-point representations of reals. The suggestion is to perform this rounding by *random round-off*, a suggestive name for mode 1 SR. The abstract concludes by stating that numerical tests on some unspecified IBM equipment confirm that SR can eliminate the "peculiarities" noticed by Huskey on the ENIAC[1].

The first hardware implementation of SR we are aware of was described by Barnes et al. [7] in 1951. The authors describe a digital computer with 8-digit decimal arithmetic and explain that using SR rather than RN in multipliers and dividers simplified the implementation. As their implementation rounds up or down with equal probability, this constitutes an early example of mode 2 SR.

A note by Forsythe, originally written in 1950 and reprinted in 1959 [8] (see [8, footnote 1]), provides more details about the proposal to round stochastically when solving ODEs. The document suggests to implement mode 1 SR for decimal arithmetic as follows:

> *"On a decimal machine, instead of adding a 5 in the most significant position of the digits to be dropped (ordinary rounding off), one adds a random decimal digit to each of the digital positions to be dropped. As with ordinary rounding off, the addition carry-over determines whether the rounding off is 'up' or 'down'."*

It is not clear whether this excerpt refers to a hardware implementation or to a modification that could be done in software on the computers of the time. This technique has been used in recent hardware implementations for rounding binary numbers [9], [10].

In a 1966 paper, Hull and Swenson [11] test various probabilistic rounding error models by comparing the results of stochastically rounded operations to the expected error predicted by the models. According to the description provided at the beginning of the section "Simulation of the Models" [11, p. 109], however, the implementation of SR that Hull and Swenson consider differs from the one we examine. In order to round stochastically the result of an arithmetic operation they first perform the operation in binary64 arithmetic, then add a pseudo-random number between $-1/2$ and $1/2$ of the unit in the last place of the upper 32 bits of the binary64 result. Subsequent calculations use the modified 64-bit value, which presumably includes the original quantity of the bottom 32 bits and the added random quantity. Despite the different spirit, we mention this contribution here as it is one of the earliest manuscripts we are aware of that considers non-deterministic rounding.

4. Floating-point arithmetics

Before describing the finer details of SR, we recall some necessary background on floating-point arithmetic. We discuss the formats in the IEEE 754 standard for floating-point arithmetic and two other formats of practical interest, bfloat16 and TensorFloat-32.

(a) IEEE 754 standard floating-point arithmetics

The IEEE standard 754 for floating-point arithmetic was first released in 1985 [12] and then revised in 2008 [13] and 2019 [14]. The standard dictates the encoding rules for binary and decimal floating-point data types, the precision and exponent range of some standard formats, and the accuracy requirements of basic arithmetic operations. It also prescribes how to handle exceptional cases and specifies a set of recommended mathematical functions that software and hardware floating-point libraries should provide in order to ensure a consistent numerical behaviour. Table 4.1 reports the parameters for the four binary floating-point data types defined in the latest revision of the standard. Most hardware implements the data types binary32 and binary64, previously known as *single* and *double precision*, respectively. Of the remaining formats, binary16 is defined only as a storage format, but it has been implemented in hardware by several

[1]Electronic Numerical Integrator and Computer—first programmable, general-purpose digital computer made in 1945.

Table 4.1. Parameters of various binary floating-point formats: number of digits of precision including the implicit bit (p), endpoints of the exponent range ($e_{\min}$ and $e_{\max}$), machine epsilon (ε_M), smallest positive representable normal ($f_{\min}$) and subnormal ($s_{\min}$) numbers, and largest positive number ($f_{\max}$). The "binary*xy*" formats are from the IEEE 754 standard.

	bfloat16	binary16	binary32	binary64	binary128
p	8	11	24	53	113
$e_{\max}$	127	15	127	1023	16383
$e_{\min}$	-126	-14	-126	-1022	-16382
ε_M	2^{-7}	2^{-10}	2^{-23}	2^{-52}	2^{-112}
$f_{\min}$	2^{-126}	2^{-14}	2^{-126}	2^{-1022}	2^{-16382}
$s_{\min}$	2^{-133}	2^{-24}	2^{-149}	2^{-1074}	2^{-16494}
$f_{\max}$	$2^{127}(2-2^{-7})$	$2^{15}(2-2^{-10})$	$2^{127}(2-2^{-23})$	$2^{1023}(2-2^{-52})$	$2^{16384}(2-2^{-112})$

manufacturers. While binary128 is mainly supported in software, it is also available in hardware on the IBM Power9 [15] and z13 [16] processors.

We now briefly recall some key aspects of IEEE floating-point number systems and the definitions and main properties of *normalization* and *subnormal numbers*. We focus on binary formats, since most commercially available hardware implements only binary arithmetic. A binary floating-point number x has the form

$$(-1)^s \times m \times 2^{e-p+1},$$

where s is the sign bit, p is the *precision*, $m \in [0,\, 2^p - 1]$ is the integer significand, and $e \in [e_{\min},\, e_{\max}]$, with $e_{\min} = 1 - e_{\max}$, is the integer exponent. In order for $x \neq 0$ to have a unique representation, the number system is *normalised* so that the most significant bit of m—the *implicit bit* in IEEE 754 parlance—is always set to 1 if $|x| \geq 2^{e_{\min}}$. Therefore, all floating-point numbers with $m \geq 2^{p-1}$ are normalised. Numbers that have absolute value below that of the smallest normalised number $2^{e_{\min}}$ are said to be *subnormal*: they have exponent $e = e_{\min}$, integer significand $m < 2^{p-1}$, and therefore precision lower than that of normalised values (from $p-1$ bits to just 1 bit). Subnormal numbers provide the means to represent values in the *subnormal range* $(-2^{e_{\min}}, 2^{e_{\min}})$, and are necessary in order to ensure that a floating-point number system has desirable properties such as Sterbenz's lemma or gradual underflow. Because of the variable precision, however, they require special treatment in both software and hardware implementations of floating-point arithmetics. This is likely to cause performance and chip area overhead, therefore it is not uncommon for hardware manufacturers not to support subnormal numbers. Two important numbers related to the precision p are the *machine epsilon*

$$\varepsilon_M = 2^{1-p}, \tag{4.1}$$

which is the spacing of the floating-point numbers just to the right of 1, and the *unit roundoff*

$$u = 2^{-p} = \frac{1}{2}\varepsilon_M, \tag{4.2}$$

which is an upper bound on the relative error that occurs when rounding a real value to a precision-p floating-point representation using RN. For further details we refer the reader to [3, Ch. 1] and [17, Ch. 2].

The latest revision of the IEEE 754 standard defines six rounding modes, which are listed in Table 4.2. Four rounding modes are required for a floating-point arithmetic to be compliant: round to nearest with ties to even (RN), round toward positive (or toward $+\infty$, or up, RU), round toward negative (or toward $-\infty$, or down, RD), and round toward zero (RZ).

The IEEE 754-2019 standard recommends *extended or extendable precisions* [14, Sec. 3.7] to enhance the basic formats listed in Table 4.1. As an example, Intel provides an 80-bit extended precision format that has a 15-bit exponent and a 64-bit significand—the bit to the left of the

Table 4.2. Rounding modes defined in the 2019 revision of the IEEE 754 standard [14].

Round mode	Description
To nearest with ties to even (RN)	Round to a nearest floating-point value and if the two nearest floating-point values are equally close, round to the one with an even least significant digit. This is a default rounding mode.
To nearest with ties to away	Round to a nearest floating-point value and if the two nearest floating-point values are equally close, round to the number with larger magnitude. Only required for decimal floating-point data types.
To nearest with ties to zero	Round to a nearest floating-point value and if the two nearest floating-point values are equally close, round to the number with smaller magnitude. Only required for *augmented operations* [14, Sec. 9.5].
Toward positive (RU)	Round to a nearest floating-point value that is no less than the argument.
Toward negative (RD)	Round to a nearest floating-point value that is no larger than the argument.
Toward zero (RZ)	Round to a nearest floating-point value that is no larger in magnitude than the argument.

radix point stored explicitly in this case, as opposed to the IEEE754 formats, which rely on the implicit bit convention and use the value of the exponent field to determine the leading bit of the significand. Arithmetic operations can be performed in higher precision and the results need not be rounded to binary64 until the final result of a computation leaves the higher precision registers. Note that such use of 80-bit arithmetic is not immune to double rounding, whereby a value may be rounded incorrectly to the target format whenever is rounded to an intermediate format (extended precision, in this case) first [18], [19].

(b) Non-IEEE arithmetics

Among the non-IEEE floating-point formats implemented in recent hardware, we are particularly interested in those based on binary32: bfloat16 and TensorFloat-32, which lower the precision p from 23 to 8 and 11 bits, respectively. The main idea behind these formats is to reduce the memory and hardware arithmetic costs without narrowing the dynamic range; this contrasts with the aim behind the binary16 format, which allocates to the exponent field fewer bits than binary32 and therefore has a more limited dynamic range.

Bfloat16, which was originally proposed by Google and formalised by Intel [20], is available on the Armv8 architecture [21], on the NVIDIA Ampere chips [22] and on some Intel microarchitectures [23].

TensorFloat-32 is a format used internally in the tensor cores (matrix multiply-accumulate units) of the NVIDIA Ampere microarchitecture [22]. This 19-bit format is meant to be a low precision replacement for binary32, but is not used for data storage and is not available in any other arithmetic unit on these GPUs.

5. Implementation

Here we discuss how to implement SR. Table 5.1 summarises the features of SR in a number of implementations.

Table 5.1. Summary of SR implementations. Here, p is the target precision of rounding; k is the precision of the random number in SR; the "Type" column has "I" for integer or fixed-point arithmetic and "F" for floating-point arithmetic; the "Op" column indicates the class of operations supported or "Any" if rounding of any operation is supported; the H/S column has H if SR is in hardware and S if software; and in the "Applications" column ML and QC stand for machine learning and quantum computing, respectively.

Reference	Type	p	k	Op.	H/S	Applications
Barnes et al. (1951) [7]	I	8	1	$*, /$	H	General
Gupta et al. (2015) [9]	I	18	30	Dot prod.	H	ML
Davies et al. (2018) [24]	–	7	–	$*, +$	H	ML
Higham & Pranesh (2019) [2]	F	≤ 64	64	Any	S	General
Hopkins et al. (2020) [25]	I	32	2–32	$*$	S	ODE solve
Mikaitis (2020) [10], [26]	I, F	≤ 32	32	Any	H	General
Meurant (2020) [27]	I, F	≤ 64	64	Any	S	General
Fasi & Mikaitis (2020) [28]	F	≤ 64	64	Any	S	General
Croci & Giles (2020) [29]	F	≤ 32	32–64	$+, *, /$	S	General/PDEs
Fasi & Mikaitis (2021) [30]	F	p	p	$+, *, /, \sqrt{}$	S	General
Paxton et al. (2021) [31]	F	≤ 32	32–64	Any	S	General
Klöwer (2021)[2]	F	≤ 32	32–64	Any	S	General
Krishnakumar & Zeng (2021) [32]	I	n	$m = n$	$*$	-	QC

(a) SR expressed in terms of other rounding modes

We can express the SR operator in terms of other rounding operators by writing, for $x \notin F$,

$$\mathrm{fl}(x) = \begin{cases} \mathrm{RA}(x), & \text{with probability } q(x), \\ \mathrm{RZ}(x), & \text{with probability } 1 - q(x), \end{cases} \tag{5.1}$$

where RA denotes the operator that rounds away from zero and $q(x) \in [0, 1]$. For mode 1 SR, we can rewrite (2.2) as

$$q(x) = \frac{x - \mathrm{RZ}(x)}{\mathrm{RA}(x) - \mathrm{RZ}(x)}. \tag{5.2}$$

In order to implement (2.1) or (5.1) in practice, we need to define a discrete version of the SR operator. Given a positive integer k, which controls the number of bits used to approximate the continuous definition (2.1), let P be a random precision-k floating-point number drawn from the uniform distribution over the interval $[0, 1)$.[3] We have that, for $x \notin F$,

$$\mathrm{fl}(x) = \begin{cases} \mathrm{RA}(x), & \text{if } P < \dfrac{x - \mathrm{RZ}(x)}{\mathrm{RA}(x) - \mathrm{RZ}(x)}, \\ \mathrm{RZ}(x), & \text{if } P \geq \dfrac{x - \mathrm{RZ}(x)}{\mathrm{RA}(x) - \mathrm{RZ}(x)}, \end{cases} \tag{5.3}$$

where SR, RA, RZ round to some precision p. It is worth noting that the choice of the optimal k for implementing SR is one of the main open questions surrounding this mode 1 SR. A lower value of k makes a hardware implementation cheaper but is expected to reduce the accuracy benefit that SR may potentially bring: setting $k = 1$, for example, gives mode 2 SR.

(b) Proposed IEEE 754 style properties of SR

The definition in the previous section does not cover edge cases such as overflow, underflow, and rounding of infinities and NaNs (not-a-number). In the following we propose our definition of SR

[2] https://github.com/milankl/StochasticRounding.jl

[3] We remark that floating-point numbers are not uniformly distributed in $[0, 1]$, but here we need the precision-k random floating-point number to be sampled from that interval uniformly in the sense of real numbers. This is not a consideration required in the hardware algorithms in Section 5(d)ii since there SR is performed at the bit level, using uniformly distributed integers. Usage of other random number distributions in SR has been explored in [33].

for these edge cases by giving some properties of SR analogous to those of the rounding modes defined in the IEEE 754 standard [14].

- If the exact number is in the range of the target format, SR should be performed as though the number was originally held in $p + k$ bits and then rounded to p bits according to (2.1). Here k bits refer to the *precision of SR*, as well as the number of random bits required.
- Overflows: if the exact number lies between the maximum representable number $\pm f_{max}$ and the neighbouring value that is not representable (which has to be treated as $\pm\infty$), SR is performed in the usual way, as though the value is representable, to preserve the statistical information about the round off bits.
- When the exact number is smaller than the smallest value representable in the target format, SR should round stochastically to one of the two neighbouring floating-point values in the target format, either zero or the smallest representable value, maintaining the sign.
- When subnormals are disabled or not supported in the target format and the exact value is in the range of underflow, SR should round either to zero or to the smallest normalised value, again without changing the sign.
- $\pm\infty$ and ± 0 should not be rounded (changed) by the SR operation. NaNs with payloads that cannot be represented in the target format should not be stochastically rounded: a NaN with an implementation-defined payload may be returned, as per IEEE 754 [14, Sec. 6.2.3] (relevant in mixed-precision setting, for example converting binary64 to binary32).
- As in the standard rounding operations [14, Sec. 4.3], inexact, underflow, and overflow exceptions should be signalled by the SR operation.

(c) Simulation of SR in software

SR can be simulated in software in a straightforward fashion by relying on high precision floating-point arithmetic. The computation whose result is to be rounded stochastically is performed using higher precision and then rounding the high-precision result using (5.1), where $q(x)$ in (5.2) is based on the (higher-precision) approximation to x rather than on its exact value. This approach is easy to implement as long as higher-than-working-precision arithmetic is available, be it in hardware, for instance when emulating binary32 rounding using binary64 arithmetic, or in software, through arbitrary precision libraries such as the GNU Multiprecision Library (GNU MPFR) [34].

In practice, once the high precision solution has been computed, the rounding step can be performed in several ways. The MATLAB function `chop`[4] [2] and the FLOATP_Toolbox[5] for MATLAB [27] leverage the MATLAB random number generator to draw a random number r from the uniform distribution over the open interval $(0, 1)$ and choose the rounding direction depending on whether r is larger or smaller than $q(x)$.

Most software favors the use of integer random numbers, integer arithmetic, and bit manipulation.

The implementation of SR in the QPyTorch[6] package [35], for example, rounds stochastically a binary32 number y to a floating-point format with precision $p < 23$ as follows. First, it generates the 32-bit integer m by zeroing out the leading $(p - 1) + 9$ bits of a 32-bit random integer, since the first 9 digits in the binary representation of y store its sign and exponent and the following $p - 1$ store the p most significant bits of the significand using the implicit bit convention. Next, the algorithm computes $n = \widetilde{y} + m$, where $\widetilde{y}$ is the binary representation of y seen as an integer, uses a bitmask to zero out the $24 - p$ trailing digits of n, as 24 is the number of precision bits in a binary32 number, and finally returns the value thus obtained as a 32-bit floating-point number. The implementations in the CPFloat[7] C library [28] use an analogous technique when rounding

[4] https://github.com/higham/chop
[5] https://gerard-meurant.pagesperso-orange.fr/floatp.zip
[6] https://github.com/Tiiiger/QPyTorch
[7] https://github.com/mfasi/cpfloat

binary32 as well as binary64 floating-point numbers to lower precision. The same approach is followed by Verificarlo[8] [36], an instrumentation tool which uses the GNU Compiler Collection (GCC) `quad` format as extended precision for binary64, and binary64 as extended precision for binary32.

The approach of simulating SR through extended precision is also used in the `mcaquad` backend of the Valgrind tool Verrou[9] [37], [38]. This tool offers also a second backend, which bears the same name as the tool and rounds stochastically without using higher precision. The technique implemented in the `verrou` backend is reminiscent of the algorithms used for double–double arithmetic, and it exploits reduction operations [14, Sec. 9.4], also known as error-free transformations, to approximate the distance between the result computed with $2p$ digits of precision and the two precision-p rounding candidates. The random direction is then chosen by comparing this value with that of a random integer.

Fasi and Mikaitis [30] propose a similar but more general approach for implementing in software stochastically rounded elementary arithmetic operations: addition/subtraction, multiplication, division, and extraction of the square root. For each operation, they propose two algorithms, one that uses only RN, and one that combines it with RZ, RU, and RD. Both variants are faster than an implementation that relies on the GNU MPFR library, and the RN-only versions are faster on x86 architectures, where switching the rounding mode incurs a high performance penalty [17, Sec. 12.3.2].

Klöwer's Julia software package StochasticRounding.jl[10] defines three new Julia floating-point types that automatically include SR. These correspond to bfloat16, binary16, and binary32, and use the Xoroshiro128Plus fast pseudo-random number generator (PRNG).[11] Composability and type flexibility in Julia enable SR computations in single and half precision in a large number of numerical software and mathematical libraries. Automatic application of SR operations is extremely advantageous from a user standpoint, as it allows significant code simplification.

In terms of fixed-point arithmetic with SR, Hopkins et al. [25] and Mikaitis [26] have recently implemented a set of rounding and multiplication operations[12] and used them on low power ARM integer processors. Multiplication routines for various fixed-point formats in the ISO 18037 embedded C standard [39] were developed by exploiting the fact that ARM processors return the full-precision result of integer multiplication using two registers: multiplying two 32-bit fixed-point values, for example, returns the exact 64-bit result with all the information of integer and fraction bits of products preserved. The bottom bits of the fraction can then be used to round the results to one of the standard fixed-point formats stochastically.

(d) Hardware with SR

Now we review hardware designs discussed in the literature, some of which are already available in commercial hardware, and describe how basic floating-point addition and multiplication algorithms can be modified to support SR.

(i) Overview of available devices and patents

The Graphcore Intelligence Processing Unit (IPU) is a highly parallel machine learning accelerator that supports SR for binary32 and binary16 arithmetic [40, Sec. 2.1], [41, Ch. 10], [42]. The patent filed by Graphcore [43] reveals some technical details not specified in the documentation that may reflect the hardware implementation of the IPU. The document explains how binary32 values are stochastically rounded to binary16 precision in hardware by using a PRNG, also implemented in hardware—this kind of conversion might be performed in the IPU, although this is not reported. The algorithm begins by generating a 24-bit random number, that is, one random bit for each bit

[8] https://github.com/verificarlo/verificarlo
[9] https://github.com/edf-hpc/verrou
[10] https://github.com/milankl/StochasticRounding.jl
[11] https://juliarandom.github.io/RandomNumbers.jl/stable/man/xorshifts/
[12] https://github.com/SpiNNakerManchester/spinn_common/blob/master/include/round.h

in the significand of a binary32 value. It then uses 13 or more of those bits to round a number to binary16. The number of random bits that are actually used depends on whether rounding will result in a normal or subnormal number.

Two patents filed by IBM disclose methods for implementing floating-point adders [44] and multipliers [45] that use SR. The authors demonstrate the techniques on an 8-bit data type, but mention binary32 and binary64 as examples of other formats to which the approach can be applied. The procedures require a fixed number of random bits be loaded into a register, but the patents do not explicitly mention how these bits should be generated. The adder [44] rounds stochastically by using the bits that drop off in the significand alignment step. It is not mentioned, however, whether the sum is normalised before being rounded, and if so whether the bits that are to be shifted out during the normalization are also taken into account when rounding.

A patent from AMD describes methods and circuits to use SR in conjunction with integer adders or accumulators [46]. The document shows the design of 1) an adder that computes the sum of two integers by using a random number passed in through a third input, and 2) a 32-bit accumulator which takes as inputs both the next 16-bit value to accumulate and a 16-bit random number, and outputs a 16-bit stochastically rounded sum. The pseudo-random numbers in the proposed SR unit are generated with a *linear-feedback shift register* (LFSR), but no specific algorithm is mentioned.

A patent from NVIDIA demonstrates a method to round stochastically floating-point values to lower precision, using a fixed, programmable, or computable rounding bit position [47]. The authors explain how to round binary64 values to binary32, and binary32 values to binary16 and bfloat16. A special feature of the design presented is that it performs SR by using a number of bottom bits of the source value's significand, without relying on a random number generator [47, Fig. 2B]. For example, the 23-bit fraction of a binary32 number can be rounded to the 10-bit fraction of a binary16 value by taking the bottom 8 bits, adding them to the top 8 bits of the bottom 13-bit part of the significand (a carry may propagate to the top 10 bits), and finally setting the 13 least significant bits to zero, in order to produce the output. The authors note that this method for performing SR has an advantage over using real random numbers, since it is deterministic and cheaper to implement. They do not mention, however, whether replacing the random number with part of the input causes SR to lose any of its desirable properties.

Gupta et at. [9] discuss the hardware prototype of a fixed-point matrix multiplier based on a 2D *systolic array architecture* and demonstrate experimental results from a *field-programmable gate array* (FPGA) implementation. Each node of the systolic array is a *multiply-and-accumulate digital signal processing* (MACC DSP) unit that multiplies two integers and accumulates the result into an internal register. Each element of the matrix product is produced by a single MACC DSP. The unit is generalised, but the authors report results for an implementation in which each MACC DSP accepts inputs of at most 18 bits and accumulate the partial results in an internal 48-bit register. When the matrix product is computed, each 48-bit element is passed through a SR unit (there is one for each column of the 2D array of MACC DSPs) to produce the 18-bit rounded and saturated results. The pseudo-random numbers needed to implement SR are generated using a LFSR, and are added to the least significant bits of the number in the internal register of the MACC before they are set to 0 and dropped off the 48-bit accumulator's value. The bit width of the LFSR is equivalent to the bit width of the part of the accumulator that is dropped, which is 30 in the example discussed above.

The Intel Loihi [24] and the SpiNNaker2 [10], [26], [48] digital *neuromorphic* processors include SR. The Intel Loihi processor has multiply-accumulate hardware that computes a 7-bit approximation to $x[t] = \alpha \cdot x[t-1] + \delta \cdot s[t]$ (with $s[t] \in \{0, 1\}$, α a decay factor, and δ an impulse amount added at each step). It is not specified where SR is applied in this computation, and what precision and type of random numbers are used. The SpiNNaker2 SR accelerator rounds and saturates 64-, 32-, or 16-bit to 32- or 16-bit fixed-point numbers with SR. As a special case it also includes rounding from IEEE 754 binary32 to bfloat16. The random bits needed for rounding are produced using the 32-bit hardware pseudo-random number generator available on

SpiNNaker2 [48]. The number of bits to be used to round the fixed-point number is programmable (it can be anything between 1 and 32 bottom bits of the input), while in the case of binary32 to bfloat16 rounding it is fixed to round the bottom 16 bits of the significand.

(ii) Modifying basic floating-point algorithms to include SR

Now we discuss how to modify classical algorithms for addition and multiplication of floating-point numbers [17, Ch. 7], [49, Sec. 4.2.1], [50, Ch. 8] in order to obtain algorithms that support SR and can readily be implemented in floating-point software or hardware. Algorithms for other operations to include SR such as fused multiply-add (FMA) or division can be similarly derived by modifying the original algorithms for the IEEE 754 arithmetic operations [17], [50].

Addition. The sum $r = \circ(x + y)$, where $\circ \in \{\mathrm{RN}, \mathrm{RZ}, \mathrm{RD}, \mathrm{RU}\}$ and x and y are binary floating-point numbers, can be computed by performing the following steps.

(i) Add or subtract the integer significands and set the exponent of the result.
(ii) Normalise the sum's significand and update the exponent.
(iii) Round the sum's significand, and renormalise and adjust the exponent.

In more detail, let $x = (-1)^{s_x} \times m_x \times 2^{e_x - p+1}$ and $y = (-1)^{s_y} \times m_y \times 2^{e_y - p+1}$ be two normalised precision-p floating-point numbers. We assume that $s_x = s_y = 0$, which implies that x and y are positive, in order to avoid considering sign interactions which may transform the addition into a subtraction. This restriction does not affect our main observations pertaining to the implementation of SR. The role that the sign of the operands plays in this algorithm is discussed, for instance, in [17, Sec. 7.3]. We make additional observations about subtraction when necessary.

The sum $r = \circ(x + y) = (-1)^{s_r} \times m_r \times 2^{e_r - p+1}$ is computed as follows.

(i) If $e_y > e_x$, swap x and y to ensure that $e_x \geq e_y$.
(ii) Alignment of the significands: compute $m_y \times 2^{-(e_x - e_y)}$ by shifting m_y to the right by $e_x - e_y$ places. Set $e_r = e_x$. It is not necessary to keep all the bits that are shifted out: maintaining only two bits plus a third *sticky bit* suffices—see below.
(iii) Sum of the significands: compute $m_t = m_x + m_y \times 2^{-(e_x - e_y)}$. At this step, m_t is an exact sum of the significands.
(iv) Normalization of the result: since $0 \leq m_t < 2^{p+1}$, we may need to normalise the result by shifting m_t to the right by one place (if $m_t \geq 2^p$) and increasing e_r by 1 (note that a shift left may be needed with subtraction—see below).
(v) Rounding: the significand of the rounded sum, m_r, is computed by rounding the normalised exact sum m_t to p significant bits according to $\circ$, and renormalizing if required. At this point r is the correctly rounded sum of x and y.

In order to perform the rounding at step (v) of the algorithm, it may seem necessary to preserve all the bits that, being after the bit in the pth position, are dropped off during steps (ii) and (iv). It can be shown, however, that for $\circ \in \{\mathrm{RN}, \mathrm{RZ}, \mathrm{RD}, \mathrm{RU}\}$ it suffices to keep only the first three discarded bits after the one in position p. These are the *guard bit* G, in position $p + 1$, the *round bit* R, in position $p + 2$, and the *sticky bit* T, in position $p + 3$, which is a logical OR of all the bits after the $(p + 2)$nd. Together, these three bits are called in short the GRT bits [50, Sec. 8.4.3]. In the algorithm above, they are formed in step (ii) and updated in step (iv) if normalization is required.

We now explain how the algorithm should be modified in order to include SR as an option in step (v). We use the same notation as in Section 5(a), and use k to denote the number of bits used for rounding, or equivalently the number of bits in the random number used to perform the rounding. In step (ii), instead of computing the GRT bits from the dropped off bits, we shift m_y by up to $p + k - 1$ places to the right and keep the k bottom bits beyond that in position p. Depending on the implementation, it might be necessary to manipulate appropriately those extra bits when subtraction is considered. An alignment of more than $p + k - 1$ bits is unnecessary, as in that case SR with k bits would not have any effect and the largest summand x would be returned unchanged. If a shift is required in step (iv), the whole significand has to be shifted by $p + k$ places, in order to keep the bottom k bits correct after the normalization. In order to perform

step (v), it is necessary to generate k bits from a stream of uniformly distributed random bits. This operation is expensive, but can be performed asynchronously at any point before reaching the last step, as it does not require any information about x or y. Finally, the k bits from the random stream are added to the bottom k bits of the normalised m_r; if this operation leads to a carry out, we increment the top p bits of m_t by 1 and truncate the bottom k bits to form the rounded significand m_r.

We need to consider normalization and whether the k bits required for rounding could be altered by shifting. Shifting is necessary in three cases. First, when the addition of the significands causes a carry out, m_t is shifted by one place to the right in order to normalise it, and this does not violate the bottom k bits. Secondly, when the difference of exponents $e_x - e_y$ is larger than 1, the smaller operand is aligned so that there are multiple zeros at the front and consequently only a left shift by one position may be required. For this reason, one extra bit is needed to make sure that a 1 that drops off the $p + k$ bits is shifted in correctly by the left shift. Thirdly, when the exponent difference is 1 cancellation may occur and multiple left shifts are required to normalise the result. Since the alignment was performed by shifting right by only one place, however, there is no risk of any bits being shifted beyond the $(p + k)$th position, and therefore no incorrect bits will be shifted in during the normalization. Therefore, only one extra bit is necessary, and the width of m_t should be $p + k + 1$ bits.

If the sum of two floating-point numbers is subnormal, then it is exact and no rounding is required [17, Thm. 4.2]. If one of the inputs is subnormal then the significand alignment step requires minor modification as per [17, Sec. 7.3.3] while the rest of the algorithm for the floating-point addition with SR remains the same.

Addition of floating-point numbers can result in the following exceptions: overflow, underflow, inexact, and NaN [50, p. 425]. With SR these may be handled as discussed Section 5(b).

Multiplication. Given two normalised positive floating-point numbers x and y as in the previous section, the product $r = \circ(x \times y)$ can be computed as follows.

(i) Product of the significands: compute the $2p$-bit integer $m_t = m_x \times m_y$. The fact that $2^{p-1} \leq m_x, m_y < 2^p$ implies that $2^{2p-2} \leq m_t < 2^{2p}$.

(ii) Sum of the exponents: compute $e_r = e_x + e_y$. At this point we have the exact product $m_t \times 2^{e_r - 2p + 2}$.

(iii) Normalization of the result: if $m_t \geq 2^{2p-1}$ we need to normalise the result by shifting the significand by one place to the right and increasing e_r by 1.

(iv) Rounding: the normalised exact product m_t is rounded to p significant bits according to $\circ$, giving m_r. At this point m_r is the rounded product of x and y.

As with addition, the G and T bits are required to perform the rounding in step (iv) (the R bit is not required since left shifts cannot occur here). After the normalization in step (iii), the unrounded result will be in the top p bits, whereas the bottom p bits will be used for rounding. In order to implement SR with k random bits (in this case $k \leq p$, as m_t has at most $2p$ bits and there is no need to consider larger values of k) we need to add a k-bit random number to the top k of the bottom p bits of the internal significand m_t: a carry will increment the pth bit of the top segment of m_t, causing the number to round up. The significand of the stochastically rounded result will consist of the top p bits of m_t.

For subnormal values, a few modifications have to be incorporated in the multiplication algorithm. First of all, if both inputs are subnormal, then the product will be in the underflow range and SR can be handled as in Section 5(b). If the inputs are normalised but yield a subnormal result, it is necessary to shift m_t right by more than one place in step (iii), depending on how much the product's exponent differs from that of subnormals in the target format. If only one of the inputs is subnormal, then the product can be either normal or subnormal. Two approaches are taken in this case: either the subnormal input is normalised before the multiplication, or the product's significand is normalised by a left shift. We do not go into details as this does not

change the SR algorithm (SR is performed in step (iv) after the product has been normalised or denormalised as required), and refer the reader to [17, Sec. 7.4.2].

Multiplication of floating-point numbers can result in the following exceptions: overflow, underflow, inexact, and NaN [50, p. 438]. With SR these may be handled as discussed in Section 5(b).

6. Rounding error analysis with SR

The standard model of floating-point arithmetic assumes that the elementary arithmetic operations are rounded to nearest (as is the case for IEEE standard arithmetic with the default rounding mode), so that they satisfy

$$\mathrm{fl}(x \mathrm{\ op\ } y) = (x \mathrm{\ op\ } y)(1+\delta), \quad |\delta| \le u, \quad \mathrm{op} \in \{+,-,*,/\}, \tag{6.1}$$

where the unit roundoff u is defined in (4.2). When multiple floating-point operations are performed in a sequence, rounding errors accumulate. For example, if $s = x_1y_1 + x_2y_2 + x_3y_3$ is computed in floating-point arithmetic, the computed $\widehat{s}$ satisfies

$$\begin{aligned}\widehat{s} &= \Big(\big(x_1y_1(1+\delta_1) + x_2y_2(1+\delta_2)\big)(1+\delta_3) + x_3y_3(1+\delta_4)\Big)(1+\delta_5)\\ &= x_1y_1(1+\delta_1)(1+\delta_3)(1+\delta_5) + x_2y_2(1+\delta_2)(1+\delta_3)(1+\delta_5) + x_3y_3(1+\delta_4)(1+\delta_5),\end{aligned}$$

for some $\delta_1, \ldots, \delta_5$ of magnitude at most u. It is clear from this example that rounding error analysis for vector and matrix operations involves dealing with multiple terms of the form $\prod_{i=1}^n (1+\delta_i)$. The following lemma [3, Lem. 3.1] bounds the distance of the product of n terms of the form $(1+\delta_i)^{\pm 1}$ from 1 by

$$\gamma_n = \frac{nu}{1-nu},$$

a ubiquitous constant in rounding error analysis.

Lemma 6.1. *If $|\delta_i| \le u$ and $\rho_i = \pm 1$ for $i = 1, \ldots, n$, and $nu < 1$, then*

$$\prod_{i=1}^n (1+\delta_i)^{\rho_i} = 1 + \theta_n, \quad |\theta_n| \le \gamma_n. \tag{6.2}$$

Under SR we define the elementary floating-point operations $+, -, *, /$ to be the stochastically rounded exact ones. Therefore for SR, (6.1) holds with u replaced by $2u$:

$$\mathrm{fl}(x \mathrm{\ op\ } y) = (x \mathrm{\ op\ } y)(1+\delta), \quad |\delta| \le 2u, \quad \mathrm{op} \in \{+,-,*,/\}. \tag{6.3}$$

Standard rounding error analysis based on the model (6.1) clearly remains valid for (6.3), with u replaced by $2u$, but the statistical properties of SR can be exploited to obtain smaller, probabilistic rounding error bounds.

(a) Probabilistic error analysis

Modeling rounding errors as random variables to obtain probabilistic error bounds is an old idea going back to von Neumann and Goldstine [51], Henrici [52], [53], [54], and Hull and Swenson [11], among others. This line of thought has led to the rule of thumb that in a worst-case rounding error bound one can take the square root of some error constants to obtain a more realistic bound, because of statistical effects in rounding error propagation; see, for example, [55, p. 318]. Higham and Mary [56] provided the first rigorous proof of the validity of this criterion: they showed that for random independent zero-mean rounding errors δ_i, the constant γ_n in (6.2),

can be replaced by

$$\widetilde{\gamma}_n(\lambda) = \exp\left(\frac{\lambda\sqrt{n}u + nu^2}{1-u}\right) - 1 = \lambda\sqrt{n}u + O(u^2) \tag{6.4}$$

with high probability. Subsequently, Higham and Mary [57] and Ipsen and Zhou [58] obtained a probabilistic error bound for inner products that only requires mean independence of rounding errors, an assumption weaker than independence. Finally, Connolly, Higham, and Mary [1] derived the following probabilistic version of Lemma 6.1 under these assumptions [1, Thm. 4.6]. Here, $\mathbb{E}$ denotes the expectation of a random variable.

Theorem 6.2. *Let $\delta_1, \delta_2, \dots, \delta_n$ be random variables of mean zero such that $\mathbb{E}(\delta_{i+1} \mid \delta_1, \dots, \delta_i) = \mathbb{E}(\delta_{i+1}) = 0$, for $i = 1, \dots, n-1$. If $|\delta_i| \le u$ and $\rho_i = \pm 1$, for $i = 1, \dots, n$, then for any constant $\lambda > 0$,*

$$\prod_{i=1}^{n}(1+\delta_i)^{\rho_i} = 1 + \theta_n, \quad |\theta_n| \le \widetilde{\gamma}_n(\lambda) \tag{6.5}$$

holds with probability at least $1 - 2\exp(-\lambda^2/2)$.

The assumptions in Theorem 6.2 on the zero mean and mean independence of rounding errors are not guaranteed to hold for RN, and indeed it is easy to construct examples where either of the assumptions fails and the backward error almost attains the worst-case bound (6.2) (and thus exceeds the probabilistic bound (6.5) by a factor $\sqrt{n}$).

Importantly, however, SR satisfies the conditions of Theorem 6.2, as shown by the following result [1, Lem. 5.2].

Theorem 6.3. *Let the computation of interest generate rounding errors $\delta_1, \delta_2, \dots$, in that order. If SR is used then the δ_i are random variables of mean zero such that $\mathbb{E}\big(\delta_i \mid \delta_1, \dots, \delta_{i-1}\big) = \mathbb{E}(\delta_i)$ $(= 0)$.*

It follows that the probabilistic bound (6.5) holds unconditionally for SR (with u replaced by $2u$ in view of (6.3)). Hence for SR, the rule of thumb that one can replace nu in a worst-case error bound by $\sqrt{n}u$ is a *rule*.

The probabilistic bound is most favorable when the worst-case bound is approximately attained, which happens when many tiny increments are applied to a relatively large quantity. If $\phi \in F$ is updated by increments $h_1, h_2, \dots$, which have magnitude smaller than half of the spacing around ϕ, then using RN gives $\phi = \mathrm{fl}(\phi + h_1) = \mathrm{fl}(\mathrm{fl}(\phi + h_1) + h_2) = \dots$, and the information in the updates is lost. This phenomenon, known as *stagnation*, commonly occurs in practical applications. It arises, for example, in neural networks, when parameter updates become very small, or in numerical methods for ODEs and partial differential equations (PDEs), when a very small time step is chosen. SR avoids stagnation, as some of the updates will produce rounding that changes the partial sum. This can be seen from the following result [1, Thm. 6.2].

Theorem 6.4 (inner products)**.** *Let $y = a^T b$, where $a, b \in \mathbb{R}^n$, be evaluated in floating-point arithmetic. Under SR, the computed $\widehat{y}$ satisfies $\mathbb{E}(\widehat{y}) = y$ regardless of the order in which the sums of products are evaluated.*

Taking $b_i \equiv 1$ in the theorem we see that the expected value of a sum is the true value under SR. As a simple example, suppose we run the code

```
x = 1
for i = 1: 10
   x = x + ε_M/4
end
```

in floating-point arithmetic. Since the spacing of the floating-point numbers between 1 and 2 is u, with RN every addition rounds down and the computed $\widehat{x} = 1$. With SR, however, each addition has a probability $1/4$ of rounding up, giving an increment of ε_M. Hence the expected result is $1 + 10 \cdot (1/4) \cdot \varepsilon_M$, which is the exact result (albeit not a floating-point number).

7. Applications

In applications, SR can replace existing rounding modes (usually RN) either globally or in certain parts of an algorithm, and either true random numbers or pseudo-random numbers can be used. The latter are often preferred as they ensure reproducibility of the result.

(a) Numerical linear algebra

For most numerical linear algebra algorithms rounding error analysis is built on Lemma 6.1 or some variation of it, thus these algorithms can benefit from the smaller probabilistic bound $\widetilde{\gamma}_n(\lambda)$ guaranteed for SR by the probabilistic error analysis of Theorem 6.2. For inner product, in particular, we have the following result [1, Thm. 4.8].

Theorem 7.1 (inner products). *Let $y = a^T b$, where $a, b \in \mathbb{R}^n$, be evaluated in floating-point arithmetic with SR. Then the computed $\widehat{y}$ satisfies*

$$\widehat{y} = (a + \Delta a)^T b = a^T (b + \Delta b), \quad |\Delta a| \le \widetilde{\gamma}_n(\lambda)|a|, \ |\Delta b| \le \widetilde{\gamma}_n(\lambda)|b| \tag{7.1}$$

with probability at least $1 - 2n\exp(-\lambda^2/2)$ regardless of the order of evaluation.

As a special case we can take $b_i \equiv 1$ and deduce that

$$\left| \sum_{i=1}^n a_i - \mathrm{fl}\left(\sum_{i=1}^n a_i \right) \right| \le \widetilde{\gamma}_n \sum_{i=1}^n |a_i|.$$

Figure 7.1 plots the relative errors for the sum $\sum_{i=1}^n \mathrm{fl}(1/i)$ computed in binary16 with RN and SR for a range of n. Note that the summands are already converted to binary16 (with RN), so the only errors are in the summation. This example models a very slowly growing sum of decaying summands. We see that SR has much smaller errors than RN for larger n and that the errors for SR are mostly well within the probabilistic bound with $\lambda = 1$.

Matrix products are considered in the following result [1, Thm. 4.9].

Theorem 7.2 (matrix–matrix products). *Let $C = AB$ with $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times p}$ be evaluated in floating-point arithmetic with SR. The jth column of the computed $\widehat{C}$ satisfies*

$$\widehat{c}_j = (A + \Delta A_j) b_j, \quad |\Delta A_j| \le \widetilde{\gamma}_n(\lambda)|A|, \quad j = 1, \ldots, n, \tag{7.2}$$

with probability at least $1 - 2mn\exp(-\lambda^2/2)$, and hence

$$|C - \widehat{C}| \le \widetilde{\gamma}_n(\lambda)|A||B| \tag{7.3}$$

with probability at least $1 - 2mnp\exp(-\lambda^2/2)$.

This result is illustrated in Figure 7.2, which plots the backward error for computing a matrix-vector product $y = Ax$ where $A \in \mathbb{R}^{100\times n}$ and $x \in \mathbb{R}^n$ have entries sampled from the uniform distribution over $[0, 1]$. We see that RN attains its worst-case rate of error growth and hits a relative error of 1, whereas SR has slower error growth and maintains some accuracy for all n.

As this rounding error analysis and the examples illustrate, SR is especially useful for large scale and/or low precision computations.

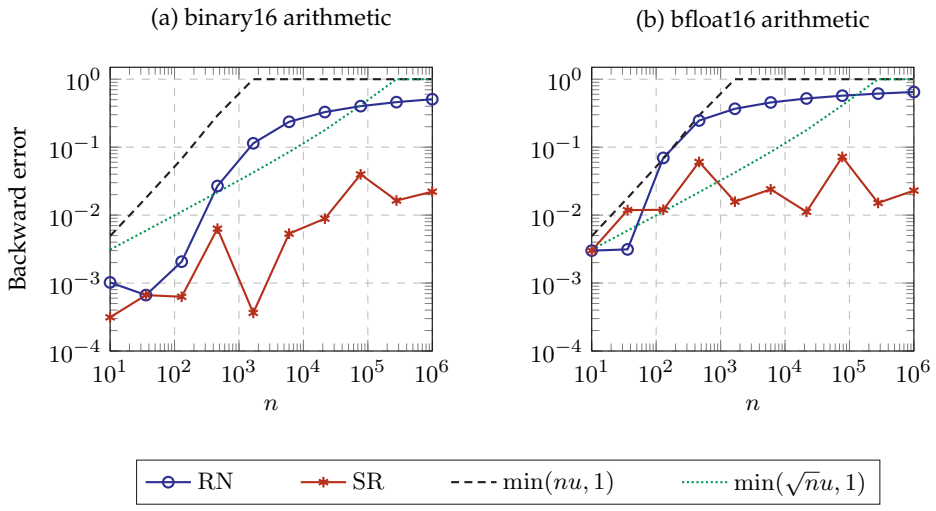

Figure 7.1. Relative errors for computing $\sum_{i=1}^{n} 1/i$ with RN and SR. The dashed and dotted lines are the worst-case error bound for RN and the probabilistic error bound for SR (with $\lambda = 1$), respectively.

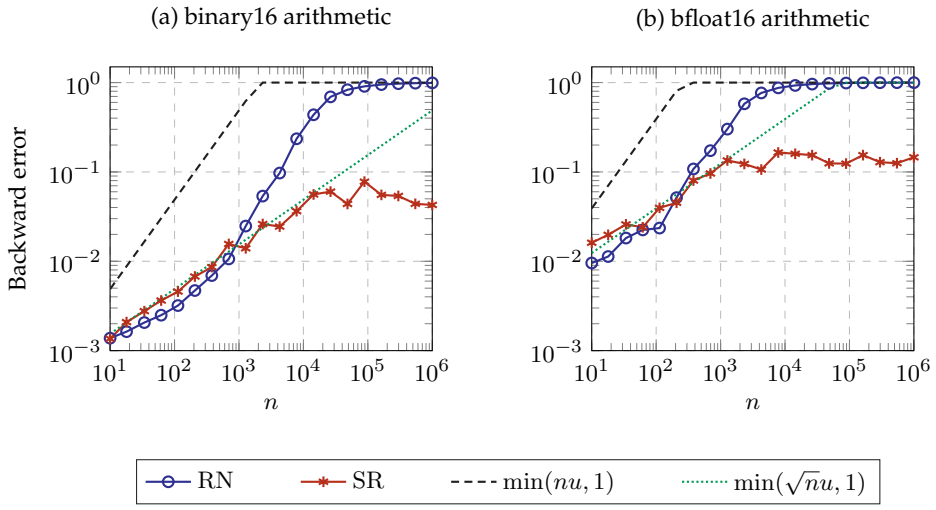

Figure 7.2. Backward error for computing $y = Ax$ with RN and SR, where $A \in \mathbb{R}^{100\times n}$ is a random matrix with uniform $[0, 1]$ entries. The dashed and dotted lines are the worst-case error bound for RN and the probabilistic error bound for SR (with $\lambda = 1$), respectively.

(b) Machine learning

The use of SR in neural networks is not a new idea. Höhfeld and Fahlman [59], [60] proposed it in 1992, calling it probabilistic rounding. Today, SR is being used in machine learning in conjunction with half precision arithmetic, not least because of its ability to avoid the problem of stagnation that affects RN. Gupta et al. [9] show that SR can be used for training deep neural networks in half precision fixed-point arithmetic, with little or no degradation in the classification accuracy. Su et al. [61] successfully train neural networks in 8-bit fixed-point arithmetic using SR and offer

some suggestions as to why SR is beneficial in this context. Similarly, Wang et al. [62] train neural networks in 8-bit floating-point arithmetic with SR, obtaining factor 2–4 speedups over 32-bit training. Zamirai et al. [63] find that either of SR and compensated summation [3, Sec. 4.3] enables training in bfloat16 to match 32-bit training. Liu et al. [64] propose a modification of (2.1) and show faster convergence in training using 16-bit fixed-point numbers.

We note that the survey by Wang et al. [65] of custom hardware for deep learning includes a review of work that uses SR.

Muller and Indivieri [66] use SR (called *randomised rounding* or *online stochastic*) to map continuous neural network weights to a discrete low precision fixed-point representation. It was shown that with SR the networks could perform well with lower precision weights than required with the standard rounding.

Essam et al. [67] combine SR with dynamic precision fixed-point arithmetic formats (variable integer scaling factors) in training neural networks. Na et al. [68] implemented SR with dynamic fixed-point in hardware for neural network applications. Used to train neural networks with 24-bit fixed-point numbers, SR provided performance similar to that of binary32, but with lower hardware area and energy costs. The authors also show that without SR even 64-bit is not enough to train the kinds of neural networks they used.

Mellempudi et al. [69] show that training neural networks using SR with 8-bit floating-point numbers yields performance comparable to that of binary32. Ortiz et al. [70] compare 12-bit fixed- and floating-point formats with and without SR with binary32 arithmetic in the training of neural networks. They find that SR can be very useful in improving the accuracy: without SR training accuracy was just 10%, whereas with SR 12-bit fixed-point improved substantially and the 12-bit floating-point closely matched binary32 [70, Table 2].

Joardar et al. [71] implemented a 32- to 16-bit SR unit in an in-memory computing device for neural network training, based on *resistive random access memory* (ReRAM), which sues an LFSR 16-bit pseudo-random number generator.

There are many more examples of SR being of use in low precision machine learning applications. For more details, see the references cited in the papers discussed above.

(c) Numerical verification software

Numerical verification software uses SR to explore the propagation of rounding errors in applications: a particular computation is repeated multiple times and the distribution of errors from these runs is used to draw conclusions about the sensitivity of a code to rounding errors. Mode 2 SR, known as stochastic arithmetic [72], is used for example in the CADNA library [73].

An approach that includes as options both mode 1 and mode 2 SR is Monte Carlo arithmetic [74], [75], a method used by tools such as Verificarlo [36] and Verrou [37], [38]. Monte Carlo arithmetic is more general than SR, not least because as well as randomly rounding the result of a floating-point operation it can also randomly perturb its inputs and output.

(d) Ordinary differential equations

The analysis of rounding errors in ODEs typically follows the classical convergence theory of timestepping methods [52], [76], in which the global error introduced by the ODE integration procedure is expressed in terms of the local errors introduced at each time step, and the global error is bounded in terms of the stepsize, h, which we will assume is fixed. These local errors are comprised of both (local) truncation errors and (local) rounding errors. It is clear that unless the contribution to the global error from rounding errors decays to zero at the same rate as the contribution from truncation error the overall convergence of the method can be impacted. Unfortunately, the analysis by Henrici [52], [76] shows that the global error of an order-p ODE solver under RN is $O(uh^{-1} + h^p)$. The $O(uh^{-1})$ term is often overlooked in the literature, and indeed in binary64 arithmetic the unit roundoff u is usually small enough to make it negligible

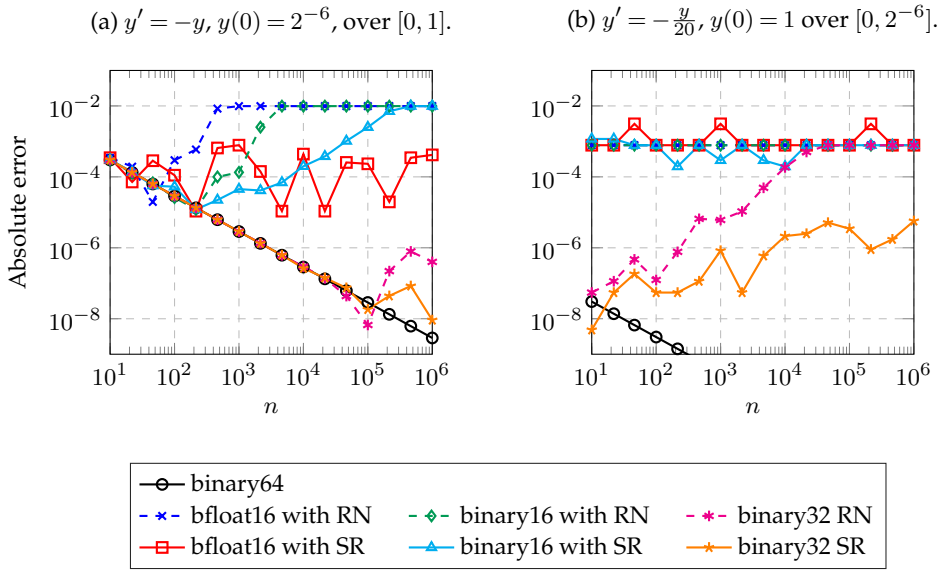

Figure 7.3. Absolute errors in the forward Euler method for an ODE with exponentially decaying solutions with different floating-point arithmetics and rounding modes. The stepsize is the interval length divided by n. Diagrams adapted from [30].

for the stepsizes h of interest, but the $O(uh^{-1})$ term cannot be neglected when computations are performed in reduced precision arithmetic.

While not explicitly mentioning SR, the early work by Henrici in the 1960s [52], [76] and by Arató in the 1980s [77] considers rounding errors arising in ODE solvers as independent (rather than mean-independent) random variables of zero mean. Henrici indicates that whenever rounding errors have this random structure the term $O(u\Delta t^{-1})$ can be replaced with a more mildly growing term with respect to Δt. The analysis by Arató in [77] rewrites the problem of estimating the global rounding error as the solution of a stochastic differential equation. It is curious that stochastic differential equations have not yet appeared in the actual analysis of SR errors for ODEs and PDEs.

It has been shown experimentally that SR can alleviate the accumulation of rounding errors in ODE solvers. Hopkins et al. [25] and Mikaitis [26] use, on an ODE that models neurons in two configurations, four different solvers including RK2 Midpoint and RK3 Heun. They compare the results obtained in fixed-point arithmetic with those obtained using the same solvers on the ODE configurations run in binary32 and binary64 arithmetics. The fixed-point solvers had three rounding variants in the multiplication operation: bit truncation, RN, and mode 1 SR. In the experiments, 32-bit fixed-point arithmetic with SR in multipliers showed accuracy similar to that of binary64 arithmetic in all cases, while fixed-point arithmetic with RN and bit truncation, as well as binary32 arithmetic, accumulated errors significantly in the progression of the ODE system, ending up with very different neuron behaviour.

Some experiments have also been performed using floating-point arithmetics (binary16, bfloat16, binary32) with SR in adds and multiplies by Fasi and Mikaitis [30]. ODEs exhibiting exponential decay were solved with Euler, midpoint, and Heun solvers. For very small timesteps, where rounding errors dominate the overall error of the solution, using SR produced a final solution error lower than that of RN. Figure 7.3 shows this for the solution errors with the forward Euler method solved in various arithmetics.

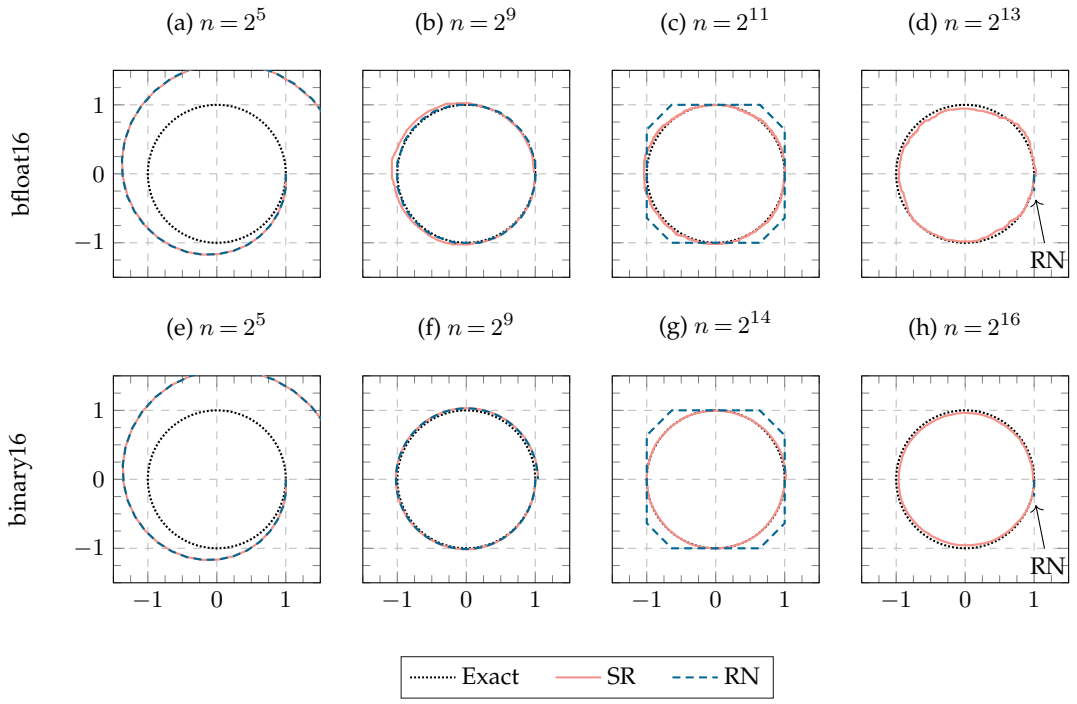

Figure 7.4. Computed solutions from the forward Euler method with SR and RN for the system (7.4). The exact solution is the unit circle. The x- and y-axis represent u and v, respectively. Note that in (d) and (h) only a small part of the solution computed with round-to-nearest is visible (marked with an arrow) since the ODE solver failed because of stagnation. Diagrams adapted from [30].

The ODE system

$$\begin{aligned} u'(t) &= v(t), \qquad u(0) = 1, \\ v'(t) &= -u(t), \quad v(0) = 0, \end{aligned} \tag{7.4}$$

whose solution is the unit circle in the (u, v) plane, was solved using binary16 and bfloat16 arithmetics for increasingly smaller integration steps [30, Sec. 8.3.2]. The results in Figure 7.4 for the forward Euler method with $h = 2\pi/n$ demonstrate that with RN the computed solutions are not meaningful for very small integration steps, while with SR the computed solution reproduces the unit circle quite well.

(e) Partial differential equations

Little is known about the interplay between SR and the typical algorithmic components of PDEs solvers, namely sparse iterative solvers, preconditioning, optimization, and timestepping methods.

Croci and Giles [29] analyze the effects of RN and SR in the solution of the heat equation with Runge–Kutta methods and finite differences, and explain how the numerical scheme should be implemented to reduce rounding errors. The analysis for RN yields the same $O(u\Delta t^{-1})$ rate in all dimensions as in Henrici's work on ODEs [52], [76], and in related work on the heat equation by Jézéquel [78]. On the other hand, using SR yields considerably smaller error bounds. In fact, Croci and Giles prove that the leading-order component of the rounding errors introduced by SR are zero-mean random variables that are independent in space and mean-independent in time. Thanks to this lack of correlation, much milder blow-up rates are obtained

for the global rounding errors, essentially $O(u\Delta t^{-1/4})$ in 1D, $O(u|\log(\Delta t)|^{1/2})$ in 2D, and $O(u)$ in 3D. Interestingly, rounding errors become asymptotically smaller as the dimension increases: the larger the dimension, the more error cancellation happens due to the spatial independence of the SR errors.

The lack of error correlation and the zero-mean property are not solely responsible for the success of SR for this problem. Croci and Giles also show that the RN solution is prone to stagnation, and in fact the phenomenon may occur from the very first step if Δt is small enough to cause the RN solution to never move away from the initial condition. On the other hand, SR is resilient to stagnation, which did not affect the SR solution in numerical experiments.

In unpublished work, these results have been extended to linear parabolic PDEs and the finite element method and, together with numerical experimentation in binary16 and bfloat16 precision, show that while RN can fail to compute meaningful solutions, SR computations always exhibit near-machine-precision accuracy for sufficiently small timesteps and mesh sizes. We expect similar results as in the parabolic case to hold for hyperbolic PDEs, with the exception perhaps of the stagnation behaviour.

Here we consider a diffusion equation with Dirichlet boundary conditions,

$$\begin{cases} u_t(t,x) = \nabla\cdot(D(x)\nabla u(t,x)) + f(x), & x\in\mathcal{D}=[0,1]^d, \quad t\in[0,1], \\ u(0,x) = u_0(x), \quad u(t,s) = 1, & s\in\partial\mathcal{D}, \qquad\quad t\in[0,1], \end{cases} \tag{7.5}$$

where

$$D(x) = \begin{cases} \frac{1}{2}(\sin(\pi x)^2+1), & \text{in 1D}, \\ \frac{1}{3}\begin{bmatrix} \sin(\pi x_1)^2+1 & \sin(\pi x_1)\cos(\pi x_2) \\ \sin(\pi x_1)\cos(\pi x_2) & \cos(\pi x_2)^2+1 \end{bmatrix}, & \text{in 2D}, \end{cases}$$

and $f(x)$ is chosen so that the exact solutions to (7.5) at steady-state are

$$u_{1\text{D}}(\infty,x) = 16(x(1-x))^2+1, \quad u_{2\text{D}}(\infty,x) = (16x_1x_2(1-x_1)(1-x_2))^2+1.$$

By using the bfloat16 format with RN and SR, in Figure 7.5 we show the effect of stagnation on the numerical steady-state solution of this problem in 1D as we vary the initial condition. We solve (7.5) using the finite element method with piecewise linear basis functions using forward and backward Euler. We note that the RN solution always stagnates close to the initial condition, while SR successfully captures the correct steady-state solution.

In Figure 7.6 we plot the relative (i.e., normalised by the roundoff unit) global rounding errors for both RN and SR against the theoretical bounds derived in [29]. While the RN error indeed grows linearly with Δt^{-1} until stagnation, the SR error increases very mildly in 1D and almost unnoticeably in 2D. For a similar 3D problem (not shown), the errors are just bounded by a multiple of machine precision. It therefore seems that SR is able to control the growth of rounding errors without requiring more accurate summation techniques such as that in [79].

Paxton et al. [31] investigate experimentally the effects of RN and SR in chaotic ODE and PDE systems related to climate modelling: the Lorenz system, heat diffusion, a nonlinear shallow water model for turbulent flow over a ridge, and a coarse resolution global atmospheric model with simplified parameterisations. They simulate these models in various precisions using from 62- to 10-bit floating point formats and they compare their results via the Wasserstein distance, a metric used to measure the discrepancy between probability distributions. They find that SR can effectively mitigate rounding error growth in both simple heat diffusion and turbulent models. Furthermore, they report the occurence of stagnation when RN is used to solve the heat equation, confirming the results in [29].

Overall, the findings by Paxton et al. show that reduced precision with SR is a valid alternative to standard double precision computations. The authors also suggest that SR might become relevant in next-generation climate models.

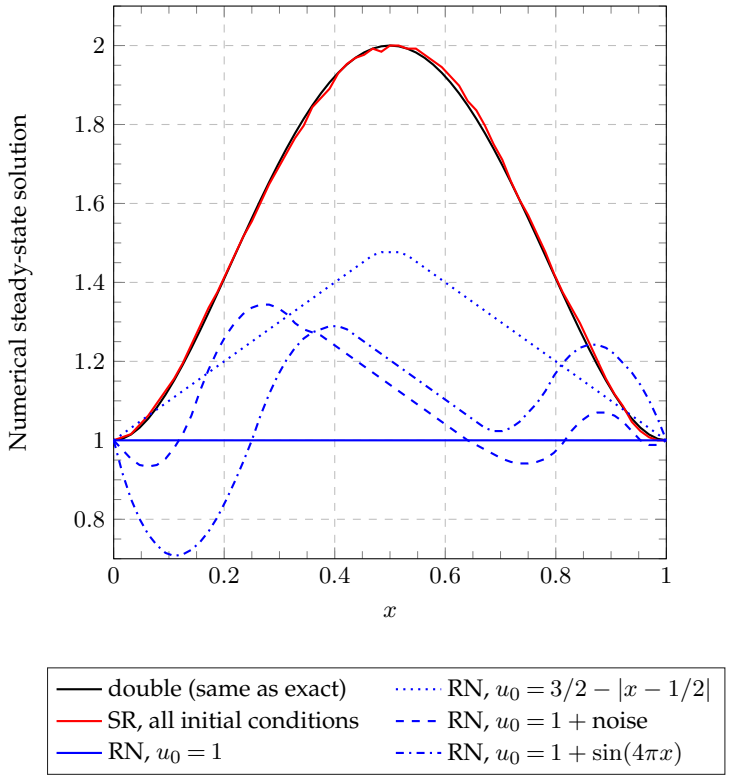

Figure 7.5. Comparison between the numerical steady-state solutions obtained with RN and SR with forward Euler and the bfloat16 format for different initial conditions. All SR solutions essentially converge to the same steady-state. On the other hand when RN is used different initial conditions lead to different steady-state solutions. The noise term in the initial condition has been obtained by sampling independent standard Gaussian random variables at each mesh node.

(f) Quantum mechanics

In quantum mechanics, an integer variant of SR has been used by several authors in order to estimate the dominant eigenvalues of Hamiltonian matrices using Monte Carlo versions of the power iteration. The goal is to compute the ground-state eigenvector φ_0 of a Hamiltonian matrix H as a linear combination of a set of *basis states* $|0\rangle, \ldots, |n\rangle$. The coefficients of the linear combination, or *basis-state amplitudes*, are the inner products $c_i = \langle i|\varphi_0\rangle$. At each step of the power method, the coefficient c_i is approximated by an integer $n_i^{(k)}$, and for all i the approximations at step $k+1$ are computed from those at step k. Once the iteration has converged numerically, the basis-state amplitude for the state $|i\rangle$ will be estimated as the average value of $n_i^{(k)}$ over k.

Nightingale and Blöte [80] were the first to suggest the use of SR for the solution of this problem. They used a random walk model, and in their work the integers $n_i^{(k)}$ count the number of random walkers that are in state $|i\rangle$ at iteration k. The integer SR function used in this work is

$$\mathrm{fl}(x) = \begin{cases} [x] + 1, & \text{with probability } x - [x], \\ [x], & \text{with probability } 1 - (x - [x]), \end{cases} \tag{7.6}$$

where $[x]$ denotes the integer part of $x \in \mathbb{R}$, defined by

$$[x] = \begin{cases} \lfloor x \rfloor & x \geq 0, \\ \lceil x \rceil & x < 0. \end{cases}$$

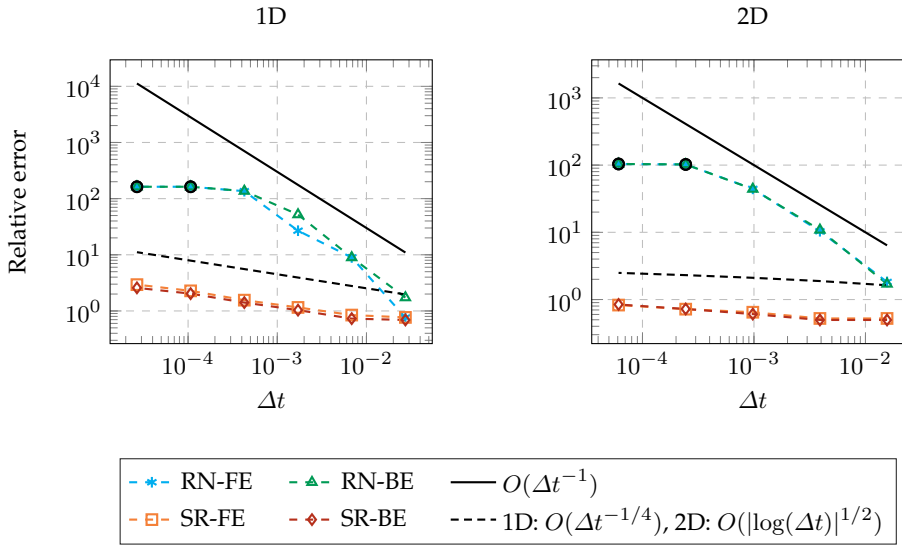

Figure 7.6. Plot of the relative global rounding error in the L^2 norm for the solution of (7.5) in 1D (left) and 2D (right) with forward and backward Euler (FE and BE respectively) and the bfloat16 format. We circled the RN data points for which the solution stagnates at the initial condition. The error behavior matches the theoretical predictions from [29].

Allton, Yung, and Hamer [81] improved on this idea by suggesting a new scheme, called *stochastic truncation*, which was further developed by Hamer et al. [82] and Hamer and Court [83]. All these variants of the stochastic truncation method use essentially the same rounding function (7.6).

A variation of (7.6) which essentially applies integer SR only to a specific interval was proposed by Price, Hamer, and O'Shaughnessy [84], who suggested the use the rounding function

$$\mathrm{fl}(x) = \begin{cases} x, & \text{if } x \geq 1, \\ 1, & \text{if } 1 > x > P, \\ 0, & \text{if } P \geq x > 0, \end{cases} \tag{7.7}$$

where P is a random number from the interval $[0, 1]$. This rounding operator keeps the "exact" value of x for large x but allows some of the values below 1 to be stochastically replaced by 0.

(g) Quantum computing

Krishnakumar and Zeng [32] show how to implement mode 1 and 2 SR for quantum computing applications and demonstrate that mode 1 provides accuracy or circuit complexity improvements. Mode 1 SR in this work is called *quantum rounding*. It is shown that quantum rounding can be implemented by utilizing the fact that quantum computing has a probabilistic component—measuring a state of a quantum register can return different results with certain probabilities. The authors show that a quantum rounding circuit can be made to round with proportional probabilities according to mode 1 SR of (2.1). Once such a circuit is used multiple times to measure the value of a quantum register (as is commonly done in quantum computing in order to improve confidence in the results), the average value will be more accurate due to the properties of mode 1 SR. Using a fixed-point quantum multiplication operation as a benchmark application, the authors show that 2 to 3 times less resources are required to implement it, compared with when round to nearest is used, for the same accuracy targets.

(h) Other applications

Various other applications use SR in one way or another. We give overviews of a few such applications.

SR was applied in digital signal processing [85] (called *random rounding*) using fixed-point arithmetic. The authors demonstrated a simple filter in 16 bits that is more accurate with SR than with the standard rounding. Two hardware implementations of SR are also demonstrated, and one of them interestingly does not require random number generation but uses a value that is perturbed on each rounding operation.

Bargh et al. [86] and Tran et al. [87] address the problem of preserving privacy when publicly releasing data sets. Their goal is to find the best ways to minimise the disclosure of personal information and share only data that does not infringe peoples' privacy. One of the aspects considered is how to transform sensitive information in specific cells of tabulated data. In [86, Sec. 4.2.2], rounding is discussed as an alternative to suppression, which is the simple removal of values that are at risk of disclosing private information, a process which may potentially delete useful data. In this research SR, under the name of *random rounding* [88, Sec. 5.4.3], is used to round numerical data to one of the two nearest integer multiples of a given base. In base 10, for example, the number 26 would be rounded to 20 or to 30 with probabilities 40% and 60% respectively. SR is useful here as it does not always increase large values and decrease small ones as round to nearest would [86]. Being unbiased, moreover, SR can hide the information about the original data [88], and may even provide protection against *differencing*, where sensitive information can be extracted from the differences in multiple tables [88].

Rounding to integer in a stochastic fashion is also considered by Gösgens et at. [89] in the study of models for the spread of infections, by Matter and Potgieter [90], to solve a problem of resource allocation, and by Hörls and Balac [91], for exploring travel demand in cities using transport simulations.

Wu [92] explores SR and a modification of it called *dither rounding* in the context of stochastic computing. Dither rounding is more complex than SR, as it requires that one keeps track of the number of rounding operations that have been performed. However, Wu shows that dither rounding can produce similar accuracy results, but with lower variance, in matrix multiplication and digit classification machine learning applications.

There is some connection between SR and the technique of *dither* that is a common component in audio analog-to-digital conversion and back [93–95]. In analog-to-digital conversion, dither relates to the randomization of the analog signal before it is converted to some low precision quantised digital representation [94]. However, the same term is used by the authors to refer to the randomization at the other end, digital back to analog conversion. For example, the following excerpt from [94] discusses a method of dither of the digital signal that is similar to an implementation of SR mode 1 where a set of random bits are added to the part of fraction to be truncated.

> *If a digital manipulation (such as a gain reduction) is performed, there may be a tendency to take the intermediate higher precision numbers generated by the multiplication and simply truncate or round them to the bit width of the system. This will in many cases leave the signal improperly dithered. [...] The fractional truncated bits have some influence on the dither, in keeping with their relative position. If cost or processing time were no object, then any digital manipulation should be carried out with full accuracy, and the dither carry bit (0 or 1) can be determined by an appropriate digital random number added to the bits to be truncated. In practice such schemes would probably work well by considering only the first 3 or 4 bits to be truncated.*

See also [94, Fig. 9] for a diagram that sketches an implementation of SR mode 1 in an integer multiplier.

8. Conclusions

Hardware with stochastic rounding in computer arithmetic units is not yet widely available, but has started to appear: as we explained in Section 5(d), Graphcore and Intel have produced processors with SR built-in. Patents from AMD, NVIDIA, IBM, and other computing companies describing implementations of SR in fixed- or floating-point arithmetic units show that this rounding mode could become more widely available in the future.

When hardware is not available, simulation in software of arithmetics with SR can be used to explore its behaviour. Multiple simulators have been developed, as discussed in Section 5(c). These are available in various forms in MATLAB, C, Julia, and Python.

Rounding error analysis with SR, discussed in Section 6, shows that compared with the standard rounding modes SR promises smaller probabilistic errors bounds in applications that involve inner products and avoids the problem of stagnation (Section 6(a)), where small values are lost to rounding when they are added to an increasingly large accumulator. This explains the success of SR in the applications described in Section 7. We covered work utilizing SR in various forms, in numerical linear algebra, machine learning, ODE and PDE solvers, quantum computing, and other areas. The wide array of applications in which SR, once tried, has lead to improved accuracy demonstrates that it is a useful technique to consider when arithmetics with standard rounding modes fail or are insufficiently accurate. SR is a competitive alternative to extended precision registers, arbitrary precision libraries, multi-word representation and arithmetic, compensated algorithms, and other methods for improving accuracy.

Ethics. This statement is not relevant to this work

Data Accessibility. This article has no additional data.

Authors' Contributions. All authors collected and reviewed the literature, wrote drafts of the survey, and approved the final version for publication.

Competing Interests. The authors declare no competing interests.

Funding. MC, NJH, and MM were supported by Engineering and Physical Sciences Research Council grant EP/P020720/1. MF was supported by Wenner-Gren Foundations grant UPD2019-0067. NJH was also supported by the Royal Society. TM was supported by the InterFLOP project (ANR-20-CE46-0009) of the French National Agency for Research.

References

1. Connolly MP, Higham NJ, Mary T. 2021 Stochastic Rounding and Its Probabilistic Backward Error Analysis. *SIAM J. Sci. Comput.* **43**, A566–A585. (Cited on pp. 2, 3, 14, 15.)
2. Higham NJ, Pranesh S. 2019 Simulating Low Precision Floating-Point Arithmetic. *SIAM J. Sci. Comput.* **41**, C585–C602. (Cited on pp. 2, 7, 8.)
3. Higham NJ. 2002 *Accuracy and Stability of Numerical Algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics second edition. (Cited on pp. 3, 5, 13, 17.)
4. Sterbenz PH. 1974 *Floating-Point Computation*. Englewood Cliffs, NJ, USA: Prentice-Hall. (Cited on p. 3.)
5. Forsythe GE. 1950 Round-off errors in numerical integration on automatic machinery. *Bull. Amer. Math. Soc.* **56**, 55–64. (Cited on p. 3.)
6. Huskey HD. 1949 On the Precision of a Certain Procedure of Numerical Integration. *J. Res. Nat. Bur. Standards* **42**, 57–62. With an appendix by Douglas R. Hartree. (Cited on p. 3.)
7. Barnes RCM, Cooke-Yarborough EH, Thomas DGA. 1951 An Electronic Digital Computor Using Cold Cathode Counting Tubes for Storage. *Electronic Eng.* **23**, 286–291. (Cited on pp. 4, 7.)
8. Forsythe GE. 1959 Reprint of a Note on Rounding-Off Errors. *SIAM Rev.* **1**, 66âĂŞ67. (Cited on p. 4.)
9. Gupta S, Agrawal A, Gopalakrishnan K, Narayanan P. 2015 Deep Learning with Limited Numerical Precision. In Bach F, Blei D, editors, *Proceedings of the 32nd International Conference*

on Machine Learning vol. 37*Proceedings of Machine Learning Research* pp. 1737–1746 Lille, France. PMLR. (Cited on pp. 4, 7, 10, 16.)
10. Mikaitis M. 2021 Stochastic Rounding: Algorithms and Hardware Accelerator. In *2021 International Joint Conference on Neural Networks (IJCNN)* Shenzhen, China. Institute of Electrical and Electronics Engineers. (Cited on pp. 4, 7, 10.)
11. Hull TE, Swenson JR. 1966 Tests of Probabilistic Models for Propagation of Roundoff Errors. *Comm. ACM* **9**, 108âĂŞ113. (Cited on pp. 4, 13.)
12. IEEE Computer Society. 1985 *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers. Reprinted in SIGPLAN Notices, 22(2):9–25, 1987. (Cited on p. 4.)
13. IEEE Computer Society. 2008 *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. (Cited on p. 4.)
14. IEEE Computer Society. 2019 *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. (Cited on pp. 4, 5, 6, 8, 9.)
15. Trader T. 2016 IBM Advances Against x86 with Power9. Accessed 21 May 2021. (Cited on p. 5.)
16. Lichtenau C, Carlough S, Mueller SM. 2016 Quad Precision Floating Point on the IBM z13. In *Proocedings of the 23rd IEEE Symposium on Computer Arithmetic* pp. 87–94. (Cited on p. 5.)
17. Muller JM, Brunie N, de Dinechin F, Jeannerod CP, Joldes M, Lefèvre V, Melquiond G, Revol N, Torres S. 2018 *Handbook of Floating-Point Arithmetic*. Birkhäuser 2nd edition. (Cited on pp. 5, 9, 11, 12, 13.)
18. Roux P. 2014 Innocuous Double Rounding of Basic Arithmetic Operations. *J. Formaliz. Reason.* **7**, 131–142. (Cited on p. 6.)
19. Rump SM. 2017 IEEE754 Precision-k base-β Arithmetic Inherited by Precision-m Base-β Arithmetic for $k < m$. *ACM Trans. Math. Software* **43**, 1âĂŞ15. (Cited on p. 6.)
20. Intel Corporation. 2018 BFLOAT16—Hardware Numerics Definition. White paper. Document number 338302-001US. (Cited on p. 6.)
21. Limited A. 2020 Arm Architecture Reference Manual. Technical Report ARM DDI 0487G.b (ID072021). Accessed 9 August 2021. (Cited on p. 6.)
22. NVIDIA Corporation. 2020 NVIDIA A100 Tensor Core GPU architecture. NVIDIA whitepaper v1.0. (Cited on p. 6.)
23. Intel Corporation. 2021 Intel Architecture Instruction Set Extensions and Future Features Programming Reference. Technical Report 319433-044. Accessed 9 August 2021. (Cited on p. 6.)
24. Davies M, Srinivasa N, Lin TH, Chinya G, Cao Y, Choday SH, Dimou G, Joshi P, Imam N, Jain S, Liao Y, Lin CK, Lines A, Liu R, Mathaikutty D, McCoy S, Paul A, Tse J, Venkataramanan G, Weng YH, Wild A, Yang Y, Wang H. 2018 Loihi: A Neuromorphic Manycore Processor with On-Chip Learning. *IEEE Micro* **38**, 82–99. (Cited on pp. 7, 10.)
25. Hopkins M, Mikaitis M, Lester DR, Furber S. 2020 Stochastic rounding and reduced-precision fixed-point arithmetic for solving neural ordinary differential equations. *Philos. Trans. R. Soc. A* **378**. (Cited on pp. 7, 9, 18.)
26. Mikaitis M. 2020 *Arithmetic Accelerators for a Digital Neuromorphic Processor*. PhD thesis University of Manchester. (Cited on pp. 7, 9, 10, 18.)
27. Meurant G. 2020 FLOATP_Toolbox. Matlab software, variable precision floating point arithmetic. (Cited on pp. 7, 8.)
28. Fasi M, Mikaitis M. 2020 CPFloat: A C Library for Emulating Low-Precision Arithmetic. MIMS EPrint 2020.22 Manchester Institute for Mathematical Sciences, The University of Manchester UK. (Cited on pp. 7, 8.)
29. Croci M, Giles MB. 2020 Effects of round-to-nearest and stochastic rounding in the numerical solution of the heat equation in low precision. arXiv:2010.16225 [math.NA]. (Cited on pp. 7, 19, 20, 22.)
30. Fasi M, Mikaitis M. 2021 Algorithms for Stochastically Rounded Elementary Arithmetic Operations in IEEE 754 Floating-Point Arithmetic. *IEEE Trans. Emerg. Topics Comput.* **9**, 1451âĂŞ1466. (Cited on pp. 7, 9, 18, 19.)

31. Paxton EA, Chantry M, Klöwer M, Saffin L, Palmer T. 2021 Climate Modelling in Low-Precision: Effects of both Deterministic & Stochastic Rounding. arXiv:2104.15076 [physics.ao-ph]. (Cited on pp. 7, 20.)
32. Krishnakumar R, Zeng W. 2021 Quantum Rounding. arXiv:2108.05949 [quant-ph]. (Cited on pp. 7, 22.)
33. Xia L, Anthonissen M, Hochstenbach M, Koren B. 2020 Improved stochastic rounding. arXiv:2006.00489 [math.NA]. (Cited on p. 7.)
34. Fousse L, Hanrot G, Lefèvre V, Pélissier P, Zimmermann P. 2007 MPFR: A Multiple-precision Binary Floating-Point Library with Correct Rounding. *ACM Trans. Math. Software* **33**, 13:1–13:15. (Cited on p. 8.)
35. Zhang T, Lin Z, Yang G, De Sa C. 2019 QPyTorch: A Low-Precision Arithmetic Simulation Framework. arXiv:1910.04540 [cs.LG]. (Cited on p. 8.)
36. Denis C, De Oliveira Castro P, Petit E. 2016 Verificarlo: Checking Floating Point Accuracy through Monte Carlo Arithmetic. *Proceedings of the 23rd IEEE Symposium on Computer Arithmetic* pp. 55–62. (Cited on pp. 9, 17.)
37. Févotte F, Lathuilière B. 2016 VERROU: Assessing Floating-Point Accuracy Without Recompiling. . (Cited on pp. 9, 17.)
38. Févotte F, Lathuilière B. 2019 Debugging and Optimization of HPC Programs with the Verrou Tool. *Proceedings of the 3rd IEEE/ACM International Workshop on Software Correctness for HPC Applications*. (Cited on pp. 9, 17.)
39. ISO/IEC. 2008 *Programming languages — C — Extensions to support embedded processors, ISO/IEC TR 18037:2008*. ISO/IEC. Technical committee ISO/IEC JTC 1/SC 22. (Cited on p. 9.)
40. Graphcore Limited. 2021a IPU Programmer's Guide. Version 2.0.0. (Cited on p. 9.)
41. Graphcore Limited. 2021b Targeting the IPU from TensorFlow 1. Version 2.0.0-dc2. (Cited on p. 9.)
42. Graphcore Limited. 2021c AI-Float™- Mixed Precision Arithmetic for AI: A Hardware Perspective. Version latest: Aug 25, 2021. (Cited on p. 9.)
43. Felix S, Gore M, Alexander AG. 2021 Converting floating point numbers to reduce the precision. Patent Status: Active. (Cited on p. 9.)
44. Bradbury JD, Carlough SR, Prasky BR, Schwarz EM. 2019a Stochastic rounding floating-point add instruction using entropy from a register. Patent Status: Active. (Cited on p. 10.)
45. Bradbury JD, Carlough SR, Prasky BR, Schwarz EM. 2019b Stochastic rounding floating-point multiply instruction using entropy from a register. Patent Status: Active. (Cited on p. 10.)
46. Loh GH. 2019 Stochastic rounding logic. Patent Status: Active. (Cited on p. 10.)
47. Alben JM, Micikevicius P, Wu H, Siu MY. 2019 Stochastic rounding of numerical values. Patent Status: Active. (Cited on p. 10.)
48. Höppner S, Mayr C. 2018 SpiNNaker2—Towards Extremely Efficient Digital Neuromorphics and Multi-scale Brain Emulation. In *Neuro-inspired Computional Elements Workshop* Portland, OR, US. (Cited on pp. 10, 11.)
49. Knuth DE. 1997 *The Art of Computer Programming* vol. 2: Seminumerical Algorithms. Reading, MA, USA: Addison-Wesley 3rd edition. (Cited on p. 11.)
50. Ercegovac MD, Lang T. 2004 *Digital Arithmetic*. San Francisco, CA, USA: Morgan Kauffmann. (Cited on pp. 11, 12, 13.)
51. von Neumann J, Goldstine HH. 1947 Numerical Inverting of Matrices of High Order. *Bull. Amer. Math. Soc.* **53**, 1021–1099. (Cited on p. 13.)
52. Henrici P. 1962 *Discrete Variable Methods in Ordinary Differential Equations*. New York: Wiley. (Cited on pp. 13, 17, 18, 19.)
53. Henrici P. 1964 *Elements of Numerical Analysis*. New York: Wiley. (Cited on p. 13.)
54. Henrici P. 1966 Test of Probabilistic Models for the Propagation of Roundoff Errors. *Comm. ACM* **9**, 409–410. (Cited on p. 13.)
55. Wilkinson JH. 1961 Error Analysis of Direct Methods of Matrix Inversion. *J. ACM* **8**, 281–330. (Cited on p. 13.)
56. Higham NJ, Mary T. 2019 A New Approach to Probabilistic Rounding Error Analysis. *SIAM J. Sci. Comput.* **41**, A2815–A2835. (Cited on p. 13.)

57. Higham NJ, Mary T. 2020 Sharper Probabilistic Backward Error Analysis for Basic Linear Algebra Kernels with Random Data. *SIAM J. Sci. Comput.* **42**, A3427–A3446. (Cited on p. 14.)
58. Ipsen ICF, Zhou H. 2020 Probabilistic Error Analysis for Inner Products. *SIAM J. Matrix Anal. Appl.* **41**, 1726–1741. (Cited on p. 14.)
59. Höhfeld M, Fahlman SE. 1992 Probabilistic rounding in neural network learning with limited precision. *Neurocomputing* **4**, 291–299. (Cited on p. 16.)
60. Höhfeld M, Fahlman SE. 1992 Learning with Limited Numerical Precision Using the Cascade-Correlation Algorithm. *IEEE Trans. Neural Netw.* **3**, 602–611. (Cited on p. 16.)
61. Su C, Zhou S, Feng L, Zhang W. 2020 Towards high performance low bitwidth training for deep neural networks. *J. Semicond.* **41**. (Cited on p. 16.)
62. Wang N, Choi J, Brand D, Chen CY, Gopalakrishnan K. 2018 Training Deep Neural Networks with 8-bit Floating Point Numbers. In Bengio S, Wallach H, Larochelle H, Grauman K, Cesa-Bianchi N, Garnett R, editors, *Advances in Neural Information Processing Systems 31* , pp. 7675–7684. Montreal, Canada: Curran Associates. (Cited on p. 17.)
63. Zamirai P, Zhang J, Aberger CR, De Sa C. 2020 Revisiting BFloat16 Training. arXiv:2010.06192 [cs.LG]. Revised March 2021. (Cited on p. 17.)
64. Xia L, Anthonissen M, Hochstenbach M, Koren B. 2021 A Simple and Efficient Stochastic Rounding Method for Training Neural Networks in Low Precision. arXiv:2103.13445 [cs.LG]. (Cited on p. 17.)
65. Wang E, Davis JJ, Zhao R, Ng HC, Niu X, Luk W, Cheung PYK, Constantinides GA. 2019 Deep Neural Network Approximation for Custom Hardware. *ACM Comput. Surv.* **52**, 1âĂŞ39. (Cited on p. 17.)
66. Müller LK, Indiveri G. 2015 Rounding Methods for Neural Networks with Low Resolution Synaptic Weights. arXiv:1504.05767 [cs.NE]. (Cited on p. 17.)
67. Essam M, Tang TB, Ho ETW, Chen H. 2017 Dynamic point stochastic rounding algorithm for limited precision arithmetic in Deep Belief Network training. In *2017 8th International IEEE/EMBS Conference on Neural Engineering (NER)* pp. 629–632 Shanghai, China. (Cited on p. 17.)
68. Na T, Ko JH, Kung J, Mukhopadhyay S. 2017 On-chip training of recurrent neural networks with limited numerical precision. In *International Joint Conference on Neural Networks* pp. 3716–3723 Anchorage, AK, USA. (Cited on p. 17.)
69. Mellempudi N, Srinivasan S, Das D, Kaul B. 2019 Mixed Precision Training With 8-bit Floating Point. arXiv:1905.12334 [cs.LG]. (Cited on p. 17.)
70. Ortiz M, Cristal A, AyguadÃľ E, Casas M. 2018 Low-Precision Floating-Point Schemes for Neural Network Training. arXiv:1905.12334 [cs.LG]. (Cited on p. 17.)
71. Joardar BK, Doppa JR, Pande PP, Li H, Chakrabarty K. 2020 AccuReD: High Accuracy Training of CNNs on ReRAM/GPU Heterogeneous 3D Architecture. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.* Early Access Article. (Cited on p. 17.)
72. Vignes J. 1993 A stochastic arithmetic for reliable scientific computation. *Math. Comput. Simulation* **35**, 233âĂŞ261. (Cited on p. 17.)
73. Jézéquel F, Chesneaux JM. 2008 CADNA: A library for estimating round-off error propagation. *Comput. Phys. Comm.* **178**, 933âĂŞ955. (Cited on p. 17.)
74. Parker DS, Pierce B, Eggert PR. 2000 Monte Carlo Arithmetic: How to Gamble with Floating Point and Win. *Computing in Science and Engineering* **2**, 58–68. (Cited on p. 17.)
75. Stott PD. 1997 Monte Carlo Arithmetic: Exploiting Randomness in Floating-Point Arithmetic. Technical Report CSD-970002 Computer Science Department, University of California, Los Angeles. (Cited on p. 17.)
76. Henrici P. 1963 *Error Propagation for Difference Methods*. New York: Wiley. (Cited on pp. 17, 18, 19.)
77. Arató M. 1983 Round-Off Error Propagation in the Integration of Ordinary Differential Equations by One Step Methods. *Acta Sci. Math.* **45**, 23–31. (Cited on p. 18.)
78. Jézéquel F. 1995 Round-Off Error Propagation in the Solution of the Heat Equation by Finite Differences. *J. Univers. Comput. Sci.* **1**, 469–483. (Cited on p. 19.)

79. Blanchard P, Higham NJ, Mary T. 2020 A Class of Fast and Accurate Summation Algorithms. *SIAM J. Sci. Comput.* **42**, A1541–A1557. (Cited on p. 20.)
80. Nightingale MP, Blöte HWJ. 1986 Gap of the Linear Spin-1 Heisenberg Antiferromagnet: A Monte Carlo Calculation. *Phys. Rev. B* **33**, 659–661. (Cited on p. 21.)
81. Allton CR, Yung CM, Hamer CJ. 1989 Stochastic truncation method for Hamiltonian lattice field theory. *Phys. Rev. D* **39**, 3772âĂŞ3777. (Cited on p. 22.)
82. Hamer CJ, Aydin M, Oitmaa J, He HX. 1990 The 3-state Potts model in (2+1) dimensions. *J. Phys. A: Math. Gen.* **23**, 4025âĂŞ4038. (Cited on p. 22.)
83. Hamer CJ, Court J. 1990 Stochastic truncation approach to the Z_2 gauge model in 3+1 dimensions. *Phys. Rev. D* **42**, 2835âĂŞ2840. (Cited on p. 22.)
84. Price PF, Hamer CJ, O'Shaughnessy D. 1993 Stochastic truncation for the $(2+1)$D Ising model. *J. Phys. A: Math. Gen.* **26**, 2855–2871. (Cited on p. 22.)
85. Callahan AC. 1976 Random Rounding: Some Principles and Applications. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing* vol. 1 pp. 501–504. (Cited on p. 23.)
86. Bargh MS, Latenko A, van den Braak S, Vink M, Meijer R. 2020 On Statistical Disclosure Control Technologies for Protecting Personal Data in Tabular Data Sets. Technical report Research and Documentation Centre (WODC), Dutch Ministry of Justice and Security. (Cited on p. 23.)
87. Tran C, Fioretto F, Van Hentenryck P, Yao Z. 2021 Decision Making with Differential Privacy under a Fairness Lens. In Zhou ZH, editor, *Proceedings of the 30th International Joint Conference on Artificial Intelligence* pp. 560–566. Main Track. (Cited on p. 23.)
88. Hundepool A, Domingo-Ferrer J, Franconi L, Giessing S, Nordholt ES, Spicer K, De Wolf PP. 2012 *Statistical disclosure control*. New York: Wiley. (Cited on p. 23.)
89. Gösgens M, Hendriks T, Boon M, Keuning S, Steenbakkers W, Heesterbeek H, van der Hofstad R, Litvak N. 2020 Containment strategies after the first wave of COVID-19 using mobility data. arXiv:2010.14209 [physics.soc-ph]. (Cited on p. 23.)
90. Matter D, Potgieter L. 2021 Allocating epidemic response teams and vaccine deliveries by drone in generic network structures, according to expected prevented exposures. *PLOS ONE* **16**, 1–29. (Cited on p. 23.)
91. Hörl S, Balac M. 2021 Synthetic population and travel demand for Paris and Île-de-France based on open and publicly available data. *Transp. Res. Part C Emerg. Technol.* **130**. (Cited on p. 23.)
92. Wu CW. 2021 Dither computing: a hybrid deterministic-stochastic computing framework. arXiv:2102.10732 [cs.AR]. To appear in *Proceedings of the 28th IEEE Symposium on Computer Arithmetic*. (Cited on p. 23.)
93. Vanderkooy J, Lipshitz SP. 1983 Resolution Below the Least Significant Bit in Digital Systems and Dither. *J. Audio Eng. Soc* **32**, 106–113. (Cited on p. 23.)
94. Vanderkooy J, Lipshitz SP. 1987 Dither in Digital Audio. *J. Audio Eng. Soc* **35**, 966–975. (Cited on p. 23.)
95. Lipshitz SP, Wannamaker RA, Vanderkooy Ja. 1992 Quantization and Dither: A Theoretical Survey. *J. Audio Eng. Soc* **40**, 355–375. (Cited on p. 23.)