CPFloat: A C library for emulating low-precision arithmetic

Fasi, Massimiliano and Mikaitis, Mantas

2020

MIMS EPrint: **2020.22**

Manchester Institute for Mathematical Sciences
School of Mathematics

The University of Manchester

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

ISSN 1749-9097

CPFloat: A C library for emulating low-precision arithmetic*

Massimiliano Fasi † *Mantas Mikaitis* ‡

Abstract

Low-precision floating-point arithmetic can be simulated via software by executing each arithmetic operation in hardware and rounding the result to the desired number of significant bits. For IEEE-compliant formats, rounding requires only standard mathematical library functions, but handling subnormals, underflow, and overflow demands special attention, and numerical errors can cause mathematically correct formulae to behave incorrectly in finite arithmetic. Moreover, the ensuing algorithms are not necessarily efficient, as the library functions these techniques build upon are typically designed to handle a broad range of cases and may not be optimized for the specific needs of rounding algorithms. CPFloat is a C library that offers efficient routines for rounding arrays of binary32 and binary64 numbers to lower precision. The software exploits the bit level representation of the underlying formats and performs only low-level bit manipulation and integer arithmetic, without relying on costly library calls. In numerical experiments the new techniques bring a considerable speedup (typically one order of magnitude or more) over existing alternatives in C, C++, and MATLAB. To the best of our knowledge, CPFloat is currently the most efficient and complete library for experimenting with custom low-precision floating-point arithmetic available in any language.

Key words: low-precision arithmetic, floating-point arithmetic, mixed precision, IEEE 754 standard, binary16, bfloat16, round-to-nearest, directed rounding, round-to-odd, stochastic rounding

1. A plethora of floating-point formats and rounding modes. The 2019 revision of the IEEE 754 standard for floating-point arithmetic [32] specifies three basic binary formats for computation: binary32, binary64, and binary128. The majority of 64-bit CPUs equipped with a floating-point unit support natively both the 32-bit and the 64-bit formats, and 32-bit CPUs can emulate 64-bit arithmetic very efficiently by relying on highly optimized software libraries. The binary128 format, introduced in the 2008 revision [31] of the original IEEE 754 standard [30], has not gained much popularity among hardware manufacturers, and over ten years after having been standardized is

*Version of October 19, 2020. **Funding:** The work of the first author was supported by the Royal Society. The work of the second author was supported by an EPSRC Doctoral Prize Fellowship. This work was carried out when the first author was at the Department of Mathematics, The University of Manchester, Oxford Road, Manchester, M13 9PL, UK.

†School of Science and Technology, Örebro University, Örebro, Sweden (mssimiliano.fasi@oru.se).

‡Department of Mathematics, The University of Manchester, Manchester, UK (mantas.mikaitis@manchester.ac.uk).

supported only by the supercomputer-grade IBM POWER9 CPUs, which implement version 3.0 of the Power Instruction Set Architecture [24].

In fact, under the pressure of the low-precision requirements of artificial intelligence applications, hardware vendors have moved in the opposite direction, and in recent years have developed a wide range of fewer-than-32-bit formats. The first widely available 16-bit floating-point format is arguably binary16. Despite having been defined in the last two revisions of the IEEE 754 standard only as an interchange format, it has been supported as an arithmetic format by all NVIDIA microarchitectures since Pascal [28] and all AMD architectures since Vega [35]. Google has recently introduced the bfloat16 data type [22], a 16-bit format with approximately the same dynamic range as binary32. The latest Armv8 CPUs support a wide variety of floating-point formats, including binary32, binary64, bfloat16 [1, Sec. A1.4.5], binary16, and an alternative custom half-precision format [1, Sec. A1.4.2]. The latter is a 16-bit format that reclaims the 2,050 bit patterns (about 3%) that the binary16 format uses for infinities and NaNs (Not a Number) to double the dynamic range from [−65,504, +65,504] to [−131,008, +131,008]. The latest NVIDIA graphics card microarchitecture, NVIDIA Ampere, features a 19-bit floating-point format, called TensorFloat-32, which has the same exponent range as binary32, but the same precision as binary16 [29]. We discuss the general framework to which all these floating-point formats belong in Section 3.

Such a broad range of floating-point formats poses a major challenge to those developing mixed-precision algorithms for scientific computing, as studying the numerical behavior of an algorithm in different working precisions may require access to a number of high-end hardware platforms. To alleviate this issue, several software packages for simulating low-precision floating-point arithmetics have been proposed in recent years. We review the most widely adopted alternatives in Section 2.

Our contribution is two-fold. First, we discuss how all the operations underlying the rounding of a floating-point number x to lower precision can be performed directly on the binary representation of x in the floating-point format in which the numbers are stored. We present the new algorithms in Section 4, and in Section 5 we explain how to implement them efficiently using bitwise operations.

Second, we introduce `CPFloat`, a header-only C library that implements our algorithms and can be used to round numbers stored in binary32 or binary64 format to lower precision. The name of the package is a shorthand for *Custom-Precision Floats*. Section 6 describes how the library was implemented and tested, and Section 7 provides a minimal self-contained code snippet to exemplify how the library can be used in practice.

We remark that this is not the first library for simulating low-precision in C, as the GNU MPFR library [14] allows the programmer to work with arbitrarily few bits of precision. Unlike MPFR, however, `CPFloat` is intended only for simulating low-precision floating-point formats, and it supports only formats that require an exponent range and a number of digits of precision smaller than those offered by binary64. This narrower aim provides scope for the wide range of optimizations discussed in the following sections, which in turn yield more efficient implementations.

We provide a MEX interface to `CPFloat` for MATLAB and Octave, which we use to compare the new library with the MATLAB function `chop` [20] in terms of performance. Our experimental results in Section 8 show that the new codes bring a considerable speedup over the MATLAB implementation, as long as the matrices being rounded are large enough to offset the overhead due to calling C code from MATLAB.

Tab. 1: Synoptic table of available software packages for simulating low-precision floating-point arithmetic. The first three columns reports the name of the package, the main programming language in which the software is written, and what storage formats are supported. The following three columns describe the parameters of the target formats: whether the number of bits of precision in the significand is arbitrary (A) or limited to the number of bits in the significand of the storage format (R); whether the exponent range can be arbitrary (A), must be the same as the storage format (R), or a sub-range thereof (S); whether the target format supports subnormal numbers (S), does not support them (N), supports them only for builtin types (P), or supports them but allows the user to switch the functionality off (F). The following column lists the floating-point formats that are built into the system. The last five columns indicate whether the software supports round-to-nearest wih ties-to-even (RNE), ties-to-zero (RNZ), or ties-to-away (RNA), the three directed rounding modes of the IEEE 754 standard round-toward-zero (RZ), round-to-$+\infty$ and round-to-$-\infty$ (RUD), round-to-odd (RO), and the two variants of stochastic rounding discussed in Section 3 (SR). The abbreviations bf16, tf32, fp16, fp32, and fp64 denote the formats bfloat16, TensorFloat-32, binary16, binary32, and binary64, respectively.

Package name	Primary language	Storage format	Target format				RNE	RNZ	RNA	RZ	RUD	RO	RS
			p	e	s	builtin							
GNU MPFR	C	custom	A	A	F		✓			✓	✓		
rpe	Fortran	fp64	R	S	P	fp16				✓			
FloatX	C++	fp32/fp64	R	R	S		✓						
chop	MATLAB	fp32/fp64	R	R	F	fp16/bf16	✓			✓	✓		✓
QPyTorch	Python	fp32	R	R	N		✓						
FLOATP	MATLAB	fp64	R	A	N		✓			✓	✓		✓
CPFloat	C	fp32/fp64	R	R	F	fp16/bf16/tf32	✓	✓	✓	✓	✓	✓	✓

2. Related work. Low-precision floating-point arithmetic can be simulated using a number of existing packages. For each of these packages, Table 1 reports the main programming language in which the software is written, and details what storage formats, target formats, and rounding modes are supported.

The most comprehensive software package for simulating arbitrary-precision floating-point arithmetic is the GNU MPFR library [14], which extends the formats in the IEEE 754 standard to any number of bits of precision and an arbitrarily large exponent range. GNU MPFR is written in C, but interfaces for most programming languages are available, and this library is the de facto standard choice for simulating arbitrary precision via software. For this reason, GNU MPFR is typically used for higher-precision, rather than low-precision, floating-point arithmetics.

Dawson and Düben [8] recently developed a Fortran library, called rpe, for emulating reduced-precision floating-point arithmetic in large numerical simulations. In rpe, the reduced-precision floating-point values are stored as binary64 numbers, a solution that provides a very efficient technique for simulating floating-point formats with the same exponent range as binary64 and at most 53 bits of precision.

FloatX [11] is a header-only C++ library for simulating low-precision arithmetic that supports both binary32 and binary64 as storage formats. This software is more flexible than rpe, in that it allows the user to choose also the number of bits used to represent the exponent of the floating-point numbers in the reduced-precision format, and not only

the number of significant digits in their fraction.

The only rounding mode currently implemented by both rpe and FloatX is round-to-nearest with ties-to-even, which may be too restrictive when one wants to simulate hardware units that provide only truncation or stochastic rounding.

Higham and Pranesh [20] have recently proposed `chop`, a MATLAB function for rounding arrays of binary32 or binary64 numbers to lower precision. This solution is more efficient and flexible than the `fp16` and `vfp16` MATLAB data types proposed by Moler [26], as it allows the user to specify not only the boundaries of the dynamic exponent range and the number of binary digits of precision for the fraction, but also the rounding mode to be used and whether subnormals are supported. In particular, `chop` supports six rounding modes, the four default rounding modes prescribed by the IEEE 754-2019 for single operations, and two variants of stochastic rounding. This function can be used only from within the MATLAB programming environment, and the underlying algorithms rely on mathematical operations involving the exponent and fraction of the represented floating-point numbers, which makes them less suitable for efficient implementations in a low-level language such as C. For example, `chop` uses built-in MATLAB functions such as `abs`, `sign`, `ceil`, `floor`, `log2`, `pow2`, which are not necessarily optimized for the narrow range of inputs required in order disassemble and reassemble floating-point numbers.

The QPyTorch library is a low-precision arithmetic simulator written natively in PyTorch, whose primary aim is to facilitate the training of neural networks in low precision without the overhead of building low-precision hardware [39]. The approach taken by the developers of this library is similar to that of the MATLAB function `chop`, as the numbers are stored in binary32 before as well as after rounding. QPyTorch supports custom floating-point formats that can fit into the binary32 format, fixed-point formats of arbitrary precision but not wider than 24 bits, and block floating-point formats [38]. Infinities, NaNs, and subnormal numbers are not supported for efficiency reasons and because, as the authors point out, these typically do not appear when training neural networks and may not be supported by the underlying low-precision hardware. In terms of rounding, it supports stochastic rounding and round to nearest with three tie-breaking rules: ties-to-away, ties-to-zero, and ties-to-even.

FLOATP_Toolbox [25] is a MATLAB toolbox for simulating reduced-precision fixed-point and floating-point arithmetics which uses binary64 as storage format. This library supports the same six rounding modes as `chop` [20], and implements a number of mathematical functions, such as `log`, `exp`, `sin`, and others. The functionalities of the FLOATP_Toolbox can be used in two ways: either by calling the functions that work directly on the binary64 data structure, or by relying on the methods of the `floatp` class, which override a number of built-in MATLAB functions. A similar library is available for posit arithmetic [19, 9] from the same author.

The algorithms presented here complement existing software by proposing efficient techniques for implementing rounding using low-level bitwise instructions. Our library is intended as a software package that enables the use of these rounding functionalities in lower-level languages such as C and C++, but can benefit all those high-level languages that allow the user to call C routines either directly or indirectly.

3. Storage formats for floating-point numbers, rounding. A family of binary floating-point numbers $\mathcal{F}\langle p, e_{\min}, e_{\max}, s_n\rangle$ is a finite subset of the real line. In our notation, the three integer parameters p, $e_{\min}$, and $e_{\max}$ represent the number of binary digits

of precision, the minimum exponent, and the maximum exponent, respectively, and the Boolean flag s_n indicates whether subnormal numbers are supported. A real number $x := (s, m, e)$ in $\mathcal{F}\langle p, e_{\min}, e_{\max}, s_n\rangle$ can be written as

$$x = (-1)^s \cdot m \cdot 2^{e-p+1}, \tag{3.1}$$

where s is the sign bit, set to 0 if x is positive and to 1 otherwise, the integral siginificand m is a natural number not greater than $2^p - 1$, and the exponent e is an integer between $e_{\min}$ and $e_{\max}$ inclusive.

In order to ensure a unique representation for all numbers in $\mathcal{F}\langle p, e_{\min}, e_{\max}, s_n\rangle \setminus \{0\}$, it is customary to normalize the system by assuming that if $x \geq 2^{e_{\min}}$ then $2^{p-1} \leq m \leq 2^p - 1$, that is, the number is represented using the smallest possible exponent and the largest possible significand. In such systems, the number $(s, m, e) \in \mathcal{F}\langle p, e_{\min}, e_{\max}, s_n\rangle \setminus \{0\}$ is normal if $m \geq 2^{p-1}$, and subnormal otherwise. The exponent of subnormal numbers is always $e_{\min}$, and in a normalized system any number $x = (s, m, e) \neq 0$ has a unique p-digit binary expansion $(-1)^s \cdot \widetilde{m} \cdot 2^e$, where

$$\widetilde{m} = m \cdot 2^{1-p} = d_0 + \frac{d_1}{2} + \cdots + \frac{d_{p-1}}{2^{p-1}} = d_0.d_1 \ldots d_{p-1}, \tag{3.2}$$

for some $d_0, d_1, \ldots, d_{p-1} \in \{0, 1\}$, is called the normal significand of x. One can verify that if x is normal then $d_0 = 1$ and $1 \leq \widetilde{m} < 2$, whereas if x is subnormal then $d_0 = 0$ and $0 < \widetilde{m} < 1$. Conventionally, 0 is considered neither normal nor subnormal. In a normalized system, the smallest subnormal is $x_{\text{minsub}} := 2^{e_{\min}-p+1}$, whereas the smallest and largest positive normal numbers are $x_{\min} := 2^{e_{\min}}$ and $x_{\max} := 2^{e_{\max}}(2 - 2^{1-p})$, respectively. Their negative counterparts can be obtained by observing that a floating point number system is symmetric with respect to 0. In our notation, subnormal numbers are kept if $s_n = \textbf{true}$, and then rounded to either 0 or the smallest floating-point number of appropriate sign if $s_n = \textbf{false}$.

Now we discuss how the floating-point numbers in $\mathcal{F}\langle p, e_{\min}, e_{\max}, s_n\rangle$ can be represented efficiently as binary strings. The sign is stored in the leftmost bit of the representation, and the following b_e bits are used to store the exponent. Under the IEEE 754 assumption that $e_{\min} = 1 - e_{\max}$, the most efficient way of representing the exponent is obtained by setting $e_{\max} = 2^{b_e-1} - 1$ and using a representation biased by $e_{\max}$, so that $00\cdots01_2$ is the smallest allowed exponent and $11\cdots10_2$ is the largest. The trailing $p-1$ bits are used to store the significand of x. It is not necessary to store the value of d_0 explicitly, as the IEEE standard uses an encoding often referred to as "implicit bit": d_0 is assumed to be 1 unless the binary encoding of the exponent is the all-zero string, in which case $d_0 = 0$ and the floating-point number represents 0 if $m = 0$ and a subnormal number otherwise. If the exponent field contains the reserved all-one string, then the number represents $+\infty$ or $-\infty$ if the significand is set to 0, and a NaN otherwise. Infinities are needed to express values whose magnitude exceeds that of the largest positive and negative numbers that can be represented in $\mathcal{F}\langle p, e_{\min}, e_{\max}, s_n\rangle$, whereas NaNs represent the result of invalid operations, such as taking the square root of a negative number, dividing 0 by 0, or multiplying an infinity by 0. These are needed in order to ensure that the semantics of all floating-point operations is well specified and the resulting floating-point number system is closed.

A rounding is an operator that maps a real number x to one of the floating-point numbers closest to x in a given floating-point family. The IEEE 754 standard [32, Sec. 4]

defines four rounding modes for binary formats, the default round-to-nearest with ties-to-even, and three directed rounding modes, round-toward-$+\infty$, round-toward-$-\infty$, and round-toward-zero. We consider also two less usual rounding strategies, round-to-odd and stochastic rounding, and two other tie-breaking rules for round-to-nearest.

Round-to-odd is the rounding mode that requires setting the least significant bit of the rounded number to 1, unless the infinitely-precise number is already representable exactly in the target format. This rounding has applications in several domains. In computer arithmetic, for instance, it can be used to emulate a correctly rounded fused multiply-and-add (FMA) operator when a hardware FMA unit is not available [2], and the fact that it can be implemented at low cost in hardware has recently prompted interest from the machine learning community [5].

Unlike all rounding modes discussed so far, stochastic rounding is non-deterministic, in the sense that the direction in which to round is chosen randomly and repeating the rounding may yield different results. The simplest variant of stochastic rounding rounds an infinitely-precise number that is not representable in the target format to either of the two closest representable floating-point numbers with equal probability. A more interesting flavor of stochastic rounding rounds a non-representable number x to either of the closest floating-point numbers with probability proportional to the distance between x and the two rounding candidates. This rounding mode dates back to the fifties [12, 13], and has recently gained prominence owing to the surge of interest in low-precision floating-point formats. It has been shown to be particularly effective at alleviating swamping in long sums [6] and ordinary differential equation solvers [21, 10], as well as at counteracting the loss of accuracy observed when the precision used to train neural networks is reduced [18, 37]. Stochastic rounding is not widely available in general-purpose hardware, but it has started to appear in some specialized processors, such as the Intel Loihi neuromorphic chips [7] and the Graphcore IPU, an accelerator for machine learning [17].

4. Efficient rounding to a lower-precision format. Now we discuss how to exploit the binary representation in the storage format to develop efficient algorithms for rounding $x \in \mathcal{F}^{(h)}$ to $\widetilde{x} \in \mathcal{F}^{(\ell)}$, where

$$\mathcal{F}^{(h)} := \mathcal{F}\langle p^{(h)}, e_{\min}^{(h)}, e_{\max}^{(h)}, s_n^{(h)}\rangle, \qquad \text{and} \qquad \mathcal{F}^{(\ell)} := \mathcal{F}\langle p^{(\ell)}, e_{\min}^{(\ell)}, e_{\max}^{(\ell)}, s_n^{(\ell)}\rangle, \tag{4.1}$$

and the same superscript notation is used for x_{minsub}, $x_{\min}$, and $x_{\max}$. We assume that $e_{\max}^{(\ell)} \le e_{\max}^{(h)}$, so that numbers in $\mathcal{F}^{(\ell)}$ are representable exactly in $\mathcal{F}^{(h)}$, and that $p^{(\ell)} \le p^{(h)}/2 - 1$, which guarantees that double rounding will be innocuous for the four elementary arithmetic operations ($+$, $-$, $\times$, and $\div$) and for the square root [36]. We do not assume that the storage format supports subnormals, but we note that if that is not the case then one must additionally require that $e_{\min}^{(h)} \le e_{\min}^{(\ell)} - p^{(\ell)}$, in order to ensure that the smallest subnormal number in $\mathcal{F}^{(\ell)}$ is representable as a normal number in $\mathcal{F}\langle p^{(h)}, e_{\min}^{(h)}, e_{\max}^{(h)}, \textbf{false}\rangle$.

We now list the high-level functions we need to operate on floating-point numbers. In the description, x denotes a floating-point number in $\mathcal{F}^{(h)}$, n denotes a positive integer, and i is an integer index between 0 and $p^{(h)} - 1$ inclusive.

- $\textsc{abs}(x)$ returns the absolute value of x.

Algorithm 4.1: Round a number from $\mathcal{F}^{(h)}$ to $\mathcal{F}^{(\ell)}$ in (4.1).

```
function CPFloat(x ∈ F^(h), F^(ℓ), roundFun : F^(h) × ℕ⁺ × F^(h) → F^(h))
    if s_n^(ℓ) then
        ζ ← x_minsub^(ℓ)
    else
        ζ ← x_min^(ℓ)
    if abs(x) < x_min^(ℓ) and e_min^(ℓ) > e_min^(h) then
        p ← p^(ℓ) − (e_min^(ℓ) − (exponent(x) − 1))
    else
        p ← p^(ℓ)
    x̃ ← roundFun(x, p, ζ)
```

- $\textsc{digit}(x, i)$ returns the ith digit of the fraction of x from the left. The indexing starts from 0, so that $\textsc{digit}(x, i)$ is d_i in (3.2).
- $\textsc{exponent}(x)$ returns the exponent of x, that is, the signed integer e in (3.1).
- $\textsc{fraction}(x)$ returns the integral significand of x, that is, the positive integer m in (3.1).
- $\textsc{rand}(n)$ returns a string of n randomly generated bits.
- $\textsc{sign}(x)$ returns -1 if the floating-point number x is negative and $+1$ otherwise.
- $\textsc{tail}(x, i)$ returns the trailing $p^{(h)} - i$ bits of the fraction of x as an unsigned integer.
- $\textsc{trunc}(x, i)$ returns the number x with the last $p^{(h)} - i$ bits of the fraction set to zero.
- $\textsc{ulp}(x, i)$ returns the number $2^{\textsc{exponent}(x)-i+1}$, that is, the gap between x and its successor in a floating-point number system with i bits of precision. As noted by Muller [27], this function corresponds to the unit in the last place as defined by Overton [34, p. 14] and Goldberg [15].

How to implement these efficiently will be discussed in detail in Section 5.

Our rounding strategy is summarized in Algorithm 4.1. The function CPFloat computes the representation of the floating-point number $x \in \mathcal{F}^{(h)}$ in a lower-precision format $\mathcal{F}^{(\ell)}$. In the pseudocode, $\mathbb{N}^+$ denotes the set of positive integers. The input parameter roundFun is a pointer to one of the functions in Algorithm 4.2, 4.4, or 4.5. A call to $\textsc{roundFun}(x, p, \zeta)$ returns the floating-point number $\widetilde{x} \in \mathcal{F}^{(h)}$ which represents the rounding of x to $\mathcal{F}^{(\ell)}$. The function starts by setting the underflow threshold ζ, which corresponds to the smallest subnormal number if $s_n^{(\ell)} = \textbf{true}$ and to the smallest normal number if $s_n^{(\ell)} = \textbf{false}$. This value will be used by roundFun to flush to zero numbers that are too small to be represented. Then the algorithm computes the number p of significant digits in the significand of the binary representation of $\widetilde{x}$. If x falls within the normal range of $\mathcal{F}^{(\ell)}$, then its fraction has $p^{(\ell)}$ significant digits and the algorithm sets

Algorithm 4.2: Function for round-to-nearest with ties-to-even.

```
function roundToNearest(x ∈ F^(h), p ∈ N^+, ζ ∈ F^(h))
    if abs(x) < ζ/2 or (abs(x) = ζ/2 and s_n^(ℓ)) then
        x̃ ← 0
    else if abs(x) ≤ ζ then
        x̃ ← ζ
    else if abs(x) ≥ 2^(e_max^(ℓ)) (2 − 2^(−p^(ℓ))) then
        x̃ ← +∞
    else
        x̃ ← trunc(abs(x), p)
        if tail(x, p) > 2^(p^(h)−p−1) or (tail(x, p) =
          2^(p^(h)−p−1) and digit(x, p^(h) − p) = 1) then
            x̃ ← x̃ + ulp(x̃, p)
    return sign(x) · x̃
```

$p = p^{(\ell)}$. If, on the other hand, $|x|$ is between $x^{(\ell)}_{\text{minsub}}$ and $x^{(\ell)}_{\text{min}}$, then the exponent of x is smaller than $e^{(\ell)}_{\text{min}}$ and the number p of significant binary digits has to be reduced unless $e^{(\ell)}_{\text{min}} = e^{(h)}_{\text{min}}$, in which case x is subnormal in both storage and destination format and has the same number of leading zeros in both. If not, then p is given by the difference between $p^{(\ell)}$ and the number of leading zeros after the binary point in the representation of $\widetilde{x}$.

In the coming sections we discuss how the function roundFun can be implemented for the six rounding strategies we consider.

4.1. Round-to-nearest. Our algorithm for rounding a floating-point number in $\mathcal{F}^{(h)}$ to the closest floating-point number in the lower-precision format $\mathcal{F}^{(\ell)}$ is given in Algorithm 4.2. Initially, the function checks if the number to be rounded is too small to be represented in $\mathcal{F}^{(\ell)}$. The direction in which the two tie values $x \in \mathcal{F}^{(h)}$ such that $|x| = \zeta/2$ are rounded changes depending on whether subnormal numbers are supported or not. The significand of the smallest normal number $x^{(\ell)}_{\text{min}}$ representable in $\mathcal{F}^{(\ell)}$ is even, thus if subnormals are not supported x is equidistant from two numbers with even significands. If there is a tie and both closest floating-point numbers have same parity, the IEEE 754 standard requires rounding to the candidate largest in magnitude [32, Sec. 4.3.1], thus we round x to $\text{sign}(x) \cdot \zeta$. The significand of the smallest subnormal $x^{(\ell)}_{\text{minsub}}$, on the other hand, is odd, and when subnormals are available ($s^{(\ell)}_n$ = **true**), 0 is the floating-point number with even fraction that is closer to the the mid-point value $\zeta/2 = x^{(\ell)}_{\text{min}}/2$, thus x underflows to 0 in this case. Next, a number $x \in \mathcal{F}^{(h)}$ that is too large to underflow but has absolute value below the threshold ζ is rounded to $\text{sign}(x) \cdot \zeta$.

If $|x|$ is larger than the threshold, the algorithm checks whether its exponent is within the exponent range of $\mathcal{F}^{(\ell)}$. According to the IEEE 754 standard, a number of magnitude at least $2^{e^{(\ell)}_{\text{max}}}(2-2^{-p^{(\ell)}})$ should overflow to infinity without changes in sign [32, Sec. 4.3.1].

If x is within the range of numbers presentable in $\mathcal{F}^{(\ell)}$, then the algorithm truncates

Algorithm 4.3: Function for round-to-odd.

```
1  function roundToOdd(x ∈ F^(h), p ∈ N^+, ζ ∈ F^(h))
2      if abs(x) < ζ and x ≠ 0 then
3          x̃ ← ζ
4      else
5          x̃ ← trunc(x̃, p)
6          if tail(x, p) ≠ 0 and p > 1 then
7              digit(x̃, p^(h) − p) ← 1
8          if x̃ > x_max^(ℓ) and x̃ ≠ +∞ then
9              x̃ ← x_max^(ℓ)
10     return sign(x) · x̃
```

$x \in \mathcal{F}^{(h)}$ to p significant digits to compute $\widetilde{x} \in \mathcal{F}^{(h)}$, which is the largest nonnegative number such that $\widetilde{x} \le |x|$. In general, $\widetilde{x}$ is the representation of one of the two floating-point numbers in $\mathcal{F}^{(\ell)}$ closest to $|x|$, the other candidate being $\widetilde{x} + \textsc{ulp}(\widetilde{x}, p)$. In order to choose a rounding direction, it is necessary to examine the value of the discarded bits. The unsigned integer $d := \textsc{tail}(x, p)$ represents the trailing $p^{(h)} - p$ digits of the fraction of x. Thus $0 \le d \le 2^{p^{(h)}-p} - 1$, and it is easy to see that if $d < \gamma := 2^{p^{(h)}-p-1}$ then $\widetilde{x}$ is the absolute value of x rounded to p significant digits, whereas if $d > \gamma$ then it is necessary to add $\textsc{ulp}(\widetilde{x}, p)$ to $\widetilde{x}$ in order to obtain the correct value. If $d = \gamma$, then we have a tie and we need to round to the nearest even number, that is, a number whose fraction has a zero in position $p^{(h)} - p$. Therefore we increment $\widetilde{x}$ by $\textsc{ulp}(\widetilde{x}, p)$ if the $(p^{(h)} - p)$th bit of $\widetilde{x}$ is a 1, and we leave $\widetilde{x}$ unchanged otherwise.

Finally, we change the sign of $\widetilde{x}$ so to match that of x, and return the number thus obtained.

The latest revision of the IEEE 754 standard mentions two other tie-breaking rules for round-to-nearest: ties-to-zero, to be used for the recommended augmented operations, and ties-to-away, which is required for decimal formats. These can be easily implemented by changing the conditions of the if statements on lines 2 and 10 in Algorithm 4.2 to

```
2  if abs(x) ≤ ζ/2 then
   [...]
10 if tail(x, p) > 2^(p^(h)−p−1) then
```

for ties-to-zero, and to

```
2  if abs(x) < ζ/2 then
   [...]
10 if tail(x, p) ≥ 2^(p^(h)−p−1) then
```

for ties-to-away.

Note that this implementation preserves the sign of zero, maps infinities to infinities, and does not change the encoding of quiet and signaling NaN values. The same observation is true for the rounding functions in Algorithm 4.3, 4.4, and 4.5.

4.2. Round-to-odd. The function roundToOdd in Algorithm 4.3 implements round-to-odd according to the definition in [2, Sec. 3.1], as this rounding mode is not part of the IEEE standard. Informally speaking, if x is exactly representable in $\mathcal{F}^{(\ell)}$, then the function returns it unchanged, otherwise it returns the number closest to x with an odd fraction, that is, a fraction with a trailing 1. In the spirit of the algorithms discussed so far, one could obtain $\widetilde{x}$ by truncating $|x|$ to the first p significant digits, checking the parity of the fraction of $\widetilde{x}$, and adding $\text{ulp}(\widetilde{x}, p)$ to the result if $\widetilde{x}$ is even. However, in this case we know that the result of the truncation requires a correction only if the least significant digit of $\widetilde{x}$ is a 0, in which case adding $\text{ulp}(\widetilde{x}, p)$ amounts to setting that bit to 1. Therefore, we check the bits obliterated by the truncation, and if $\text{tail}(x, p) \neq 0$ then x is not exactly representable in $\mathcal{F}^{(\ell)}$ and the fraction of the rounded $\widetilde{x}$ must be odd. We can ensure this by setting the $(p^{(h)} - p)$th bit of $\widetilde{x}$ to 1, as long as this digit is stored explicitly, which is the case if and only if p is not 1. The core idea of this algorithm is the same as that of the second of the two methods for round-to-odd discussed in [2, Sec. 3.4].

The algorithm must round $\widetilde{x}$ to the closest odd number in $\mathcal{F}^{(\ell)}$ if it falls within the underflow or the overflow range. Note that since 0 is even, underflow is not possible when this rounding mode is used, and numbers smaller than the smallest representable number in absolute value must be rounded to the smallest floating-point number with an odd fraction of corresponding sign. When subnormal numbers are supported, the smallest representable number $x^{(\ell)}_{\text{minsub}}$ is odd, thus numbers smaller than $x^{(\ell)}_{\text{minsub}}$ are simply rounded up to $x^{(\ell)}_{\text{minsub}}$. If subnormal numbers are not supported, on the other hand, the smallest representable number $x^{(\ell)}_{\text{min}}$ is even, and we are faced with a choice. We could round $\widetilde{x} < x^{(\ell)}_{\text{min}}$ to the smallest number larger than $x^{(\ell)}_{\text{min}}$ that has odd fraction, that is, $x^{(\ell)}_{\text{min}} + \text{ulp}(x^{(\ell)}_{\text{min}}, p)$, but this choice feels unnatural, since the operator thus defined would not be rounding to either of the floating-point numbers closest to $\widetilde{x}$. In fact, in our pseudocode we prefer to round $\widetilde{x}$ to the rounding candidate largest in magnitude, that is, $\text{sign}(x) \cdot x^{(\ell)}_{\text{min}}$.

The definition given by Boldo and Melquiond cannot be applied directly to values in the overflow range, as in principle the fraction of $\pm\infty$ is neither odd nor even. Since $-x^{(\ell)}_{\text{max}}$ and $x^{(\ell)}_{\text{max}}$ are necessarily odd, we prefer to round values outside the range of floating-point number representable in $\mathcal{F}^{(\ell)}$ to the closest finite number. Infinities themselves represent an exception to this rule, and the algorithm leaves them unchanged.

Finally, we restore the sign of the input, and return the result.

4.3. Directed rounding. The functions in Algorithm 4.4 show how to implement the three directed rounding modes prescribed by the IEEE 754 standard. The idea underlying the three functions is similar to that discussed for the function roundToNearest in Algorithm 4.2, the main differences being the use of the sign, which is relevant when the rounding direction is not symmetric with respect to 0, and the conditions under which a unit in the last place has to be added.

We start by discussing the function roundTowardPlusInfinity. First, we check whether x is within the range of numbers that are representable in $\mathcal{F}^{(\ell)}$. Numbers that are too small to be represented are rounded up to 0 if negative and to ζ if positive. Finite positive numbers larger than the largest representable number $x^{(\ell)}_{\text{max}}$ overflow to $+\infty$, whereas negative numbers smaller than the smallest representable number $-x^{(\ell)}_{\text{max}}$

Algorithm 4.4: Functions for directed rounding modes.

function roundTowardPlusInfinity($x \in \mathcal{F}^{(h)}$, $p \in \mathbb{N}^+$, $\zeta \in \mathcal{F}^{(h)}$)
  if $x > 0$ **and** $x < \zeta$ **then**
    $\widetilde{x} \leftarrow \zeta$
  else if $x \leq 0$ **and** $x > -\zeta$ **then**
    $\widetilde{x} \leftarrow \text{sign}(x) \cdot 0$
  else
    $\widetilde{x} \leftarrow \text{TRUNC}(x, p)$
    if $\text{TAIL}(x, p) \neq 0$ **and** $x > 0$ **then**
      $\widetilde{x} \leftarrow \widetilde{x} + \text{ULP}(\widetilde{x}, p)$
    if $\widetilde{x} > x_{\max}^{(\ell)}$ **then**
      $\widetilde{x} \leftarrow +\infty$
    else if $\widetilde{x} < -x_{\max}^{(\ell)}$ **and** $\widetilde{x} \neq -\infty$ **then**
      $\widetilde{x} \leftarrow -x_{\max}^{(\ell)}$
  return $\widetilde{x}$
function roundTowardMinusInfinity($x \in \mathcal{F}^{(h)}$, $p \in \mathbb{N}^+$, $\zeta \in \mathcal{F}^{(h)}$)
  if $x < 0$ **and** $x > -\zeta$ **then**
    $\widetilde{x} \leftarrow -\zeta$
  else if $x \geq 0$ **and** $x < \zeta$ **then**
    $\widetilde{x} \leftarrow \text{sign}(x) \cdot 0$
  else
    $\widetilde{x} \leftarrow \text{TRUNC}(x, p)$
    if $\text{TAIL}(x, p) \neq 0$ **and** $x < 0$ **then**
      $\widetilde{x} \leftarrow \widetilde{x} - \text{ULP}(\widetilde{x}, p)$
    if $\widetilde{x} > x_{\max}^{(\ell)}$ **and** $\widetilde{x} \neq +\infty$ **then**
      $\widetilde{x} \leftarrow x_{\max}^{(\ell)}$
    else if $\widetilde{x} < -x_{\max}^{(\ell)}$ **then**
      $\widetilde{x} \leftarrow -\infty$
  return $\widetilde{x}$
function roundTowardZero($x \in \mathcal{F}^{(h)}$, $p \in \mathbb{N}^+$, $\zeta \in \mathcal{F}^{(h)}$)
  if $x < \zeta$ **then**
    $\widetilde{x} \leftarrow 0$
  else if $x \geq x_{\max}^{(\ell)}$ **and** $\widetilde{x} \neq +\infty$ **then**
    $\widetilde{x} \leftarrow x_{\max}^{(\ell)}$
  else
    $\widetilde{x} \leftarrow \text{TRUNC}(\text{ABS}(x), p)$
  return $\text{SIGN}(x) \cdot \widetilde{x}$

are rounded up to $-x_{\max}^{(\ell)}$.

Next, the function computes $\widetilde{x}$, that is, the number x with fraction truncated to p significant digits, and checks whether $\widetilde{x}$ is smaller than the smallest number representable in $\mathcal{F}^{(\ell)}$. The rounding can be easily performed by noting that the truncation $\widetilde{x}$ computed at the beginning is the correct result if x is negative or exactly representable in $\mathcal{F}^{(\ell)}$. Otherwise, $\widetilde{x}$ is incremented by $\textsc{ulp}(\widetilde{x}, p)$.

The function roundTowardMinusInfinity is identical, modulo some sign adjustments to take the opposite rounding direction into account. The algorithm starts by checking that x is within the range of numbers representable in $\mathcal{F}^{(\ell)}$. Numbers between $-\zeta$ and 0 are rounded down to $-\zeta$, whereas those between 0 and the smallest representable number underflow and are flushed to 0. Numbers that are smaller than the smallest number representable in $\mathcal{F}^{(\ell)}$ underflow to $-\infty$, whereas finite numbers greater than the largest number representable in $\mathcal{F}^{(\ell)}$ are rounded to $x_{\max}^{(\ell)}$. In order to round the number that falls within the range of $\mathcal{F}^{(\ell)}$, we compute $\widetilde{x}$ by truncating the fraction of x to p significant digits, and then subtract $\textsc{ulp}(\widetilde{x}, p)$ from $\widetilde{x}$ if x is negative and not exactly representable with a p-digit fraction.

The function roundTowardZero is simpler than the other two, as truncation is sufficient to correctly round the fraction of x to p significant digits. Underflow and overflow are also easier to handle: finite numbers that are smaller than the smallest representable number in absolute value are flushed to 0, whereas numbers outside the interval of representable finite numbers are rounded to the closest representable finite number. In order to simplify the function we work with the absolute value of the truncation of x, and restore the sign just before returning the result.

4.4. Stochastic rounding. The functions in Algorithm 4.5 describe how to implement the two variants of stochastic rounding we are concerned with, which are those discussed by Higham and Pranesh [20].

The function roundStochastic implements the strategy that rounds $x \in \mathcal{F}^{(h)}$ to one of the two closest floating-point numbers with probability proportional to the distance. First, the algorithm considers numbers in the underflow range, whose rounding candidates are 0 and the threshold value ζ, which equals x_{minsub} if subnormals are supported and $x_{\min}$ if they are not. The distance between $|x|$ and 0 depends not only on the fraction but also on the magnitude of x, thus the algorithm starts by computing the two values e_x and m_x, which represent the exponent and the integral significand of x, respectively. Being the exponent of a floating-point number in $\mathcal{F}^{(h)}$, e_x can be much smaller than $e_{\min}^{(\ell)}$, in which case it may be necessary to rescale m_x in order to align its exponent to $e_{\min}^{(\ell)}$. This is achieved by multiplying m_x by $2^{e_x+1-e_{\min}^{(\ell)}}$; in the pseudocode we take the floor of the result in order to keep it integer, although this is not strictly necessary. We prefer to work with integer arithmetic here so to be able to use directly the integers generated by the random number generator without any further post-processing. This is desirable not only from a performance point of view, but also because drawing floating-point numbers from the uniform distribution over an interval is not a trivial task, even if a good pseudo-random number generator for integers is available [16]. Finally, a random integer with $p^{(h)}$ digits of precision is generated, and the rounding direction is chosen according to the definition of stochastic rounding.

The procedure for numbers in the representable range is easier. In this case it suffices

Algorithm 4.5: Functions for stochastic rounding modes.

function roundStochastic($x \in \mathcal{F}^{(h)}$, $p \in \mathbb{N}^+$, $\zeta \in \mathcal{F}^{(h)}$)
  $\widetilde{x} \leftarrow x$
  if abs(x) $< \zeta$ **then**
    $e_{\min} \leftarrow$ exponent(ζ)
    $e_x \leftarrow$ exponent(x)
    $m_x \leftarrow$ fraction(x)
    $t \leftarrow \lfloor m_x \cdot 2^{e_x+1-e_{\min}} \rfloor$
    if $t >$ rand($p^{(h)}$) **then**
      $\widetilde{x} \leftarrow \zeta$
    else
      $\widetilde{x} \leftarrow 0$
  else if abs(x) $< 2^{e_{\max}^{(\ell)}}(2 - 2^{-p^{(\ell)}})$ **then**
    $\widetilde{x} \leftarrow$ trunc(abs(x), p)
    if tail(x, p) $>$ rand($p^{(h)} - p$) **then**
      $\widetilde{x} \leftarrow \widetilde{x} +$ ulp($\widetilde{x}, p$)
  if $\widetilde{x} \geq 2^{e_{\max}^{(\ell)}}(2 - 2^{-p^{(\ell)}})$ **then**
    $\widetilde{x} \leftarrow +\infty$
  return sign(x) $\cdot \widetilde{x}$
function roundStochasticEqual($x \in \mathcal{F}^{(h)}$, $p \in \mathbb{N}^+$, $\zeta \in \mathcal{F}^{(h)}$)
  $\widetilde{x} \leftarrow x$
  if abs(x) $< \zeta$ **and** $x \neq 0$ **then**
    $\widetilde{x} \leftarrow$ randSelect($0, \zeta$)
  else if abs(x) $> x_{\max}^{(\ell)}$ **and** abs(x) $\neq +\infty$ **then**
    $\widetilde{x} \leftarrow$ randSelect($x_{\max}^{(\ell)}, +\infty$)
  else if tail(x, p) $\neq 0$ **then**
    $\widetilde{x} \leftarrow$ trunc(abs(x), p)
    $\widetilde{x} \leftarrow$ randSelect($\widetilde{x}, \widetilde{x} +$ ulp($\widetilde{x}, p$))
  return sign(x) $\cdot \widetilde{x}$
function randSelect($x \in \mathcal{F}^{(h)}$, $y \in \mathcal{F}^{(h)}$)
  if rand(1) $= 1$ **then**
    return x
  else
    return y

to compute $\widetilde{x}$, the value of x truncated to p significant binary digits, and then generate a random integer r between 0 and $2^{p^{(h)}-p}$. Since $\textsc{tail}(x,p)$ represents the distance between x and $\widetilde{x}$, we increment $\widetilde{x}$ by $\textsc{ulp}(\widetilde{x},p)$ if $r < \textsc{tail}(x,p)$, and leave it unchanged otherwise. For overflow, we use the threshold value that the IEEE 754 standard recommends for round-to-nearest, and round numbers whose absolute value after rounding is larger than the threshold $2^{e_{\max}^{(\ell)}}(2-2^{-p^{(\ell)}})$ to infinity, leaving the sign unchanged.

The function roundStochasticEqual deals with the simpler strategy that rounds x up or down with equal probability. Depending on the interval in which x falls, the function selects the two closest representable numbers in $\mathcal{F}^{(\ell)}$ and calls the function randSelect to select one of them with equal probability. In the pseudocode, we use a single bit generated randomly to discriminate between the two rounding directions.

5. Efficient implementation for IEEE-like representation formats. Now we explain how the subroutines used in the previous section can be implemented efficiently assuming that the numbers are represented using the floating-point format described in Section 3. We begin by defining the semantics of the operators for bit manipulation that we will need in this section. These are available in most programming languages, although the notation varies greatly from language to language. For clarity, we use a prefix notation for all the operators.

Let a and b be strings of n bits. As it is customary in computer engineering, the bits are indexed from right to left, so that a_{n-1} and a_0 denote the leftmost and the rightmost bit of a, respectively.

For $p \in \mathbb{N}$, we define the following operators.

- Conjunction: $\mathsf{c} = \textsc{and}(\mathsf{a},\mathsf{b})$ is an n-bit string such that $c_i = 1$ if a_i and b_i are both set to 1 and $\mathsf{c}_i = 0$ otherwise.
- Disjunction: $\mathsf{c} = \textsc{or}(\mathsf{a},\mathsf{b})$ is an n-bit string such that $\mathsf{c}_i = 1$ if at least one of a_i and b_i is set to 1 and $\mathsf{c}_i = 0$ otherwise.
- Negation: $\mathsf{c} = \textsc{not}(\mathsf{a})$ is an n-bit string such that $\mathsf{c}_i = 1$ if $\mathsf{a}_i = 0$ and $\mathsf{c}_i = 0$ otherwise.
- Logical shift left: $\mathsf{c} = \textsc{lsl}(\mathsf{a},p)$ is an n-bit string such that $\mathsf{c}_i = 0$ if $i < p$ and $\mathsf{c}_i = \mathsf{a}_{i-p}$ otherwise.
- Logical shift right: $\mathsf{c} = \textsc{lsr}(\mathsf{a},p)$ is an n-bit string such that $\mathsf{c}_i = 0$ if $i > (n-1)-p$ and $\mathsf{c}_i = \mathsf{a}_{i+p}$ otherwise.

Most of the operations used in the previous section require extracting a certain subset of the bits in the binary representation of the floating-point number $x \in \mathcal{F}\langle p, e_{\min}, e_{\max}, s_n\rangle$ while zeroing out the remaining ones. This can be achieved by using the bitwise conjunction between the binary string that represents x and a bitmask, that is, a string as long as the binary representation of x that has ones in the positions corresponding to the bits to be extracted and zeros everywhere else. More generally, the functions in Section 4 can be performed using the operators above as follows. In the descriptions, x denotes the floating-point representation of x, n denotes a positive integer, and i denotes an integer index between 0 and $p-1$.

- $\textsc{abs}(x)$ can be implemented as $\textsc{and}(\mathsf{x}, \mathsf{m}_{\text{abs}})$, where m_{abs} is constituted by a single leading 0 followed by ones.

- $\textsc{digit}(x, i)$ can be implemented as $\textsc{and}(\mathsf{x}, \mathsf{m}_{\text{digit}})$, where $\mathsf{m}_{\text{digit}}$ has a 1 in the position corresponding to the digit to be extracted and 0 everywhere else. We note that checking whether this digit is 0 or 1 does not require any additional operations in programming languages such as C where 0 is interpreted as **false** and any other integer is interpreted as **true**.

- $\textsc{exponent}(x)$ can be implemented as a sequence of logic and arithmetic operations. The raw bits of the exponents can be extracted with $\mathsf{c} = \textsc{and}(\mathsf{x}, \mathsf{m}_{\text{exp}})$, where m_{exp} has 1 in the positions corresponding to the exponent bits of the binary representation of x. This can be converted into the unsigned integer $\textsc{lsr}(\mathsf{c}, p-1)$, and the signed exponent can be obtained by subtracting the bias of the storage floating-point format. If x is subnormal in $\mathcal{F}^{(h)}$, then the value computed in this way is $-e_{\max}^{(h)} = e_{\min}^{(h)} - 1$, and the correct value to return in this case is $e_{\min}^{(h)}$.

- $\textsc{fraction}(x)$ can be implemented leveraging the function $\textsc{exponent}$. The digits to the right of the radix point can be obtained as $\mathsf{c} = \textsc{and}(\mathsf{x}, \mathsf{m}_{\text{frac}})$ where m_{frac} is the bitmask that has the $p^{(h)} - 1$ trailing bits set to 1 and the remaining bits set to 0. If $x_{\min} \leq |x| \leq x_{\max}$, then $\textsc{exponent}(x) > e_{\min}$ and the implicit bit must be set to 1 using, for instance, $\textsc{or}(\mathsf{c}, \textsc{lsr}(1, p^{(h)}))$.

- $\textsc{rand}(n)$ can be implemented by concatenating numbers produced by a pseudo-random number generator. Two m-bit strings a and b can be joined together by $\textsc{or}(\textsc{lsl}(\mathsf{a}, m), \mathsf{b})$, and the unnecessary bits can be set to zero using a suitable bitmask.

- $\textsc{sign}(x)$ is relatively expensive to implement by means of bit manipulation. However, note that we only need to compute the product $\text{sign}(x) \cdot \widetilde{x}$ where $\widetilde{x}$ is a positive floating-point number. This operation can be implemented as $\textsc{or}(\textsc{and}(\mathsf{x}, \mathsf{m}_{\text{sign}}), \widetilde{\mathsf{x}})$, where m_{sign} is the bitmask with a leading 1 followed by zeros and the string $\widetilde{\mathsf{x}}$ denotes the floating-point representation of $\widetilde{x}$.

- $\textsc{tail}(x, i)$ can be implemented as $\textsc{and}(\mathsf{x}, \mathsf{m}_{\text{tail}})$, where the trailing $p - i$ bits of m_{tail} are set to 1 and the remaining bits are set to 0.

- $\textsc{trunc}(x, i)$ can be implemented as $\textsc{and}(\mathsf{x}, \mathsf{m}')$, where $\mathsf{m}' = \textsc{not}(\mathsf{m}_{\text{tail}})$.

- $\textsc{ulp}(x, p)$ is a rather expensive function to implement, because it requires extracting the exponent from the binary representation of x and then performing arithmetic operations on it. Increasing or decreasing x by $\textsc{ulp}(x, p)$, on the contrary, can be achieved efficiently using only one bit shift and one integer arithmetic operation. In particular, it suffices to add to the binary representation of x, seen as an unsigned integer, a number that has 0 everywhere but in position $p^{(h)} - p$. We note that this technique could fail if $x = \pm\infty$, since adding $\textsc{ulp}(x, p)$ in this fashion would turn infinities into NaNs. It is easy to check that this is not a problem in our setting, as we only add or subtract $\textsc{ulp}(x, p)$ when x is finite.

It is possible to implement some of the rounding routines more efficiently yet by extending to other rounding modes the technique for round-to-nearest with ties-to-even developed in [23, p. 2-17], which manipulates the binary representation of the floating-point number by using only integer arithmetic. To fix ideas, here we show the concrete values of the bitmasks, expressed in hexadecimal notation, that one should use in order to round a binary32 number y to binary16, but these methods are easily generalized to other combinations of storage and target formats. We show this by explaining how to round the floating-point number $x \in \mathcal{F}\langle p, e_{\min}, e_{\max}, s_n\rangle$ to p digits of precision. We denote the 32-bit string containing the floating-point representation of y by y, and use the uppercase Latin letters X and Y to denote the unsigned integers that can be obtained by interpreting x and y as unsigned integers in radix 2. All the usual underflow and overflow checks are not included here—the aim is to demonstrate the core ideas for performing each type of rounding efficiently. We recall that the sign of a floating-point number can be determined by checking the leftmost bit, and that x (resp. y) is positive if x_{n-1} (resp. y_{31}) is set and negative otherwise. The rounding modes that can benefit from this approach can be implemented as follows.

- Round-to-nearest with ties-to-even: isolate the bit in position $n-p-1$ of x, and then compute $\textsc{trunc}(X + \textsc{lsr}(\mathsf{m}_{\text{tail}}, 1) + \mathsf{x}_{n-p-1}, p)$. For rounding a binary32 number to binary16, the formula becomes $\textsc{trunc}(Y + \text{0x7FFF} + \mathsf{y}_{15}, 16)$.
- Round-to-nearest with ties-to-away: return $\textsc{trunc}(X + \textsc{lsl}(\text{0x1}, n - p - 1), p)$, which in our example becomes $\textsc{trunc}(Y + \text{0x8000}, 16)$.
- Round-to-nearest with ties-to-zero: return $\textsc{trunc}(X + \textsc{lsr}(\mathsf{m}_{\text{tail}}, 1), p)$, which in our example $\textsc{trunc}(Y + \text{0x7FFF}, 16)$.
- Round-toward-$+\infty$: return $\textsc{trunc}(X, p)$ if x_{n-1} is set and $\textsc{trunc}(X + \mathsf{m}_{\text{tail}}, p)$ otherwise. For our example, return $\textsc{trunc}(Y, 16)$, if y_{31} is set and $\textsc{trunc}(Y + \text{0xFFFF}, 16)$ if not.
- Round-toward-$-\infty$: return $\textsc{trunc}(X, p)$ if x_{n-1} is not set and $\textsc{trunc}(X + \mathsf{m}_{\text{tail}}, p)$ otherwise. For our example, return $\textsc{trunc}(Y, 16)$ if y_{31} is not set and $\textsc{trunc}(Y + \text{0xFFFF}, 16)$ otherwise.
- Round-toward-zero: return $\textsc{trunc}(X, p)$. For our example, return $\textsc{trunc}(Y, 16)$.
- Stochastic rounding with probability of rounding proportional to distance: return $\textsc{trunc}(X + \textsc{rand}(p), p)$ in the general case and $\textsc{trunc}(Y + \textsc{rand}(16), 16)$ for our example.

6. Implementation and validation of the code. Our C implementation of the algorithms discussed in Section 4 and Section 5 is available on GitHub.[1] The code is provided as a header-only library, to allow the users to take advantage of the inlining feature of the C language for maximum efficiency. This also enhances the portability of the code, as packaging of the binaries and installation of the library are not required. The main drawback of this approach is an increase in the compilation time, which we alleviate

[1] https://github.com/mfasi/cpfloat/

by dividing the library into two separate units which can be selected by including the header files `cpfloat_binary32.h` and `cpfloat_binary64.h`, respectively.

In order to achieve a better performance when working on large amount of data, our functions were designed to work directly on C arrays. All the algorithms discussed in Section 4 are embarrassingly parallel, and each element of an array can be rounded independently from all the others. Therefore, our code was written so to take advantage of the OpenMP library, if available on the system in use.

In general, OpenMP brings significant gains in terms of performance, but greatly increases the execution time for arrays with a limited number of elements. This well-known phenomenon is due to the fact that synchronization and loop scheduling in OpenMP brings an additional overhead [3] which, while negligible for large arrays, can be significant when only a small amount of work is allocated to each OpenMP thread. The impact of this overhead is hard to quantify in general, as it depends on the hardware platform as well as the number of OpenMP threads and the compiler used [4]. In our library we provide both a parallel and a sequential version of the rounding functions, but we were unable to provide a single threshold that would allow the code to switch automatically from one variant to the other for optimal performance. Thus we devised a simple autotuning strategy that tries to determine the optimal threshold for the system in use by running the rounding function on several arrays of different lengths and using a binary search to determine the optimal array size.

For generating the pseudo-random numbers required for stochastic rounding, we rely on algorithms from the family of permuted congruential generators developed by O'Neill [33], who provides a pure C implementation available on GitHub.[2] In our code we use the functions `pcg32_random_r` and `pcg64_random_r` to generate 32-bit and 64-bit random numbers, respectively, and we initialize the random number generator with `pcg32_srandom_r` and `pcg64_srandom_r`, respectively. As initial state, we use a linear combination of the current time as returned by `time(NULL)` and the OpenMP thread number as returned by `omp_get_thread_num()`. For completeness, the option of using the default C pseudo-random number generator is also provided.

In order to validate our code experimentally, we wrote a suite of extensive unit tests. We considered two storage formats, binary32 and binary64, which are available in C through the native data types `float` and `double`, respectively, and three target formats, binary16, bfloat16, and TensorFloat-32. For each combination of storage and target formats, we performed three types of tests. First we checked that all the numbers that can be represented exactly in the target format, including subnormal numbers and special values such as infinities and NaNs, are not altered by any of the rounding routines. As the target formats we consider do not have an unduly large cardinality, we could test that this property is true for all representable numbers. When checking the correctness of the rounding routines when rounding is necessary, exhaustive testing is not an option, as there are too many distinct numbers in the storage format, thus we opted for testing only a set of representative values.

The correctness of the function for deterministic rounding can be assessed by checking that the output of the rounding routine matches the value predicted by the definition. For each pair of numbers $x_1, x_2 \in \mathcal{F}^{(\ell)}$ such that x_1 and x_2 are consecutive in $\mathcal{F}^{(\ell)}$ and $x_1 < x_2$, we considered five values in $\mathcal{F}^{(h)}$: nextafter$(x_1, +\infty)$, nextafter$(x_m, -\infty)$,

[2] https://github.com/imneme/pcg-c

x_m, nextafter$(x_m, +\infty)$, and nextafter$(x_2, -\infty)$, where the nextafter(x, y) denotes the next number in $\mathcal{F}^{(h)}$ after x in the direction of y, and x_m denotes the mid point between x_1 and x_2. We used the same technique for numbers in the underflow range, whereas for testing the correctness of overflow we used the values nextafter$(\pm x_{\max}^{(\ell)}, \pm\infty)$, nextafter$(\pm x_{\text{bnd}}^{(\ell)}, \mp\infty)$, $\pm x_{\text{bnd}}^{(\ell)}$, nextafter$(\pm x_{\text{bnd}}^{(\ell)}, \pm\infty)$, where $x_{\text{bnd}}^{(\ell)} = 2^{e_{\max}^{(\ell)}}(2 - 2^{-p^{(\ell)}})$ is the IEEE 754 threshold for overflow in round-to-nearest.

This technique would not work for stochastic rounding, as each value that is not representable in $\mathcal{F}^{(\ell)}$ can be rounded to two different values. We produced a test set by taking, for each pair of numbers $x_1, x_2 \in \mathcal{F}^{(\ell)}$ such that x_1 and x_2 are consecutive in $\mathcal{F}^{(\ell)}$ and $x_1 < x_2$, the numbers $(3x_1 + x_2)/4$, $(x_1 + x_2)/2$, and $(x_1 + 3x_2)/4$. We rounded each number 1,000 times and confirmed that the rounding routines always return either x_1 or x_2, and that the empirical probability distribution matches the expected one. We validated the correctness of the implementation for values in the underflow range by using the same technique, whereas for inputs in the overflow range, we repeated the test on the three values: $(3x_{\max}^{(\ell)} + x_{\text{bnd}}^{(\ell)})/4$, $(x_{\max}^{(\ell)} + x_{\text{bnd}}^{(\ell)})/2$, and $(x_{\max}^{(\ell)} + 3x_{\text{bnd}}^{(\ell)})/4$. The Makefile target

```
$ make ctest
```

runs the test suite for the C implementations.

We designed a MEX interface for MATLAB and Octave which is in charge of parsing and checking the input, allocating the output, and calling our library to perform the rounding. In order to show that our function is fully compatible with `chop`, we designed a set of tests by modifying the MATLAB function `test_chop`, the default test suite for `chop`. We confirmed that our interface is fully compatible with that of the MATLAB function `chop`. The tests for the MEX interface can be run with

```
$ make mtest
```

in MATLAB, and with

```
$ make otest
```

in Octave.

7. Usage example. When in the `CPFloat` source directory, the optional auto-tuning procedure for the C library can be triggered with the command

```
$ make autotune
```

which computes the optimal threshold for switching between the sequential and the parallel version of the rounding routines defined in `cpfloat_binary32.h` and `cpfloat_binary64.h`.

The following example, which can be compiled with

```
$ make example
```

illustrates how to round a small C array from binary64 to binary16 and bfloat16.

```
#include <math.h>
#include <stdio.h>

#include "cpfloat_binary64.h"

int main()
{
```

```
  // Allocate the data structure for target formats and rounding
   parameters.
  static optstruct *fpopts;
  fpopts = malloc(sizeof(optstruct));

  // Set up the parameters for binary16 target format.
  fpopts->precision = 11;      // Bits in the significand + 1.
  fpopts->emax = 15;           // The maximum exponent value.
  fpopts->subnormal = 1;       // Support for subnormals is on.
  fpopts->round = 2;           // Round toward +infinity.
  fpopts->flip = 0;            // Bit flips are off.
  fpopts->p = 0;               // Bit flip probability (not used here).
  fpopts->explim = 1;          // Exponent in the target format is
   limited.

  // Validate the parameters in fpopts.
  int retval = cpfloat_validate_optstruct(fpopts);
  printf("The validation function returned %d.\n", retval);

  // Initialize a 2x2 matrix with four arbitrary elements
  double X[4] = { (double)1/3, M_PI, M_E, M_SQRT2 };
  double Y[4];
  printf("Values in binary64:\n   %.15e %.15e\n   %.15e %.15e \n",
    X[0], X[1], X[2], X[3]);

  // Round the values of X to the binary16 format and store in Y
  cpfloat(&Y[0], &X[0], 4, fpopts);
  printf("Rounded to binary16:\n   %.15e %.15e\n   %.15e %.15e \n",
    Y[0], Y[1], Y[2], Y[3]);

  // Set the precision of the significand to 8 bits,
  // and the maximum exponent to 127, which gives the bfloat16 format
  fpopts->precision = 8;
  fpopts->emax = 127;

  // Round the values of X to the bfloat16 and store in Y
  cpfloat(&Y[0], &X[0], 4, fpopts);
  printf("Rounded to bfloat16:\n   %.15e %.15e\n   %.15e %.15e \n",
    Y[0], Y[1], Y[2], Y[3]);

  return 0;
}
```

When executed, this program produces the following output, which shows the original truncated values and the exact rounded values in binary16 and bfloat16.

```
The validation function returned 0.
Values in binary64:
   3.333333333333333e-01 3.141592653589793e+00
   2.718281828459045e+00 1.414213562373095e+00
Rounded to binary16:
   3.334960937500000e-01 3.142578125000000e+00
   2.718750000000000e+00 1.415039062500000e+00
Rounded to bfloat16:
   3.339843750000000e-01 3.156250000000000e+00
   2.718750000000000e+00 1.421875000000000e+00
```

8. Performance evaluation. The experiments were run on a machine equipped with a 12-core Intel Xeon E5-2690 v3 CPU running at 2.6 GHz (Haswell microarchitecture) and 125 GiB of RAM. The C code was compiled with version 6.4.0 of the GNU Compiler Collection (GCC) with the optimization flag `-O3` and the architecture options `-mfma` and `-march=native`. The MATLAB experiments were run using the GNU/Linux version of MATLAB 9.8 (R2020a) Update 2. For the parallel version of our code, we used version 4.5 of the OpenMP library. In this section, we compare the following codes.

- `cpfloat_seq` is the sequential C implementation of the algorithms in Section 4.
- `cpfloat` is a C function that employs the auto-tuning technique discussed in Section 6 to switch between the sequential and the parallel implementation of the algorithm described in Section 4.
- `chop_mpfr` performs the rounding to a lower-precision target format in two steps by relying on the GNU MPFR library. Our implementation sets the precision and exponent range of MPFR to those of the target format, then converts the binary64 input to to the `mpfr_t` data type using the function `mpfr_set_d`, and finally uses the function `mpfr_get_d` to obtain the representation in the storage format. For C arrays we set the parameters of MPFR only once, and allocate only one variable of type `mpfr_t`, in order to reduce the memory footprint of the application and the overall execution time.
- `floatx` converts binary64 numbers to a lower-precision target format by invoking the constructor of the `floatx` class from the FloatX library.[3] This code requires the parameters of the target format to be specified at compile time, as the `floatx` class uses C++ templates and the method is compiled only for the low-precision formats declared at compile-time.
- `floatxr` invokes the constructor of the `floatxr` class from the FloatX library. This function is more flexible than `floatx` in that the number of bits of precision and the maximum exponent allowed for the target format can be specified at runtime.
- `cpfloat_ml` is our MEX interface to `cpfloat` compiled in MATLAB. For large matrices, this function relies on the parallel version of our C codes.
- `chop_ml` is the MATLAB function `chop`, available on GitHub.[4]
- `floatp_ml` is the MATLAB function `f_d_dec2floatp` from the FLOATP_Toolbox, available from the website of the author.[5]

In the plots, we use the shorthand notation $\langle f_s \,|\, f_t \rangle$ to denote the conversion of numbers in the storage format f_s to the target format f_t. As the numerical validation of the code has already been discussed in Section 6, here we focus on performance, which we measure in terms of execution time. We time the C or C++ code by comparing the value returned by the function `clock_gettime` with `CLOCK_MONOTONIC` before and after the execution, and take the median of 1,000 repetitions in order to reduce the influence

[3] https://github.com/oprecomp/FloatX
[4] https://github.com/higham/chop/
[5] https://gerard-meurant.pagesperso-orange.fr/floatp.zip

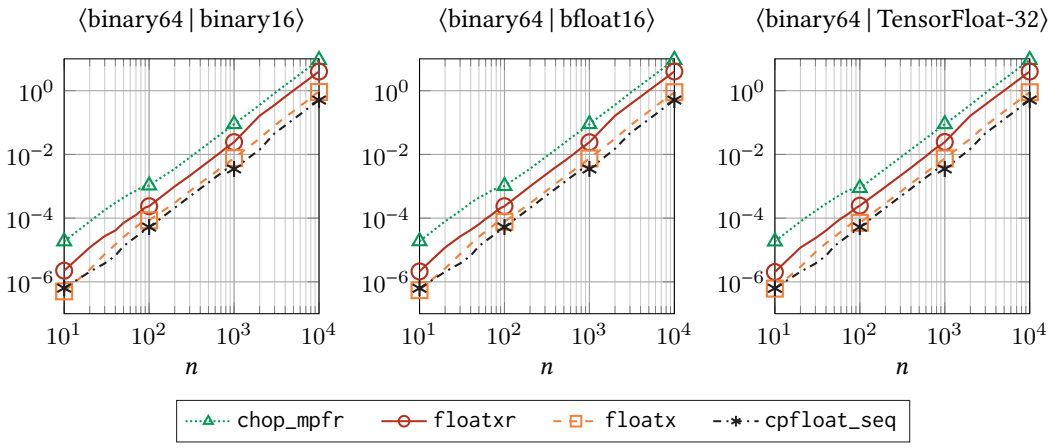

Fig. 1: Execution time of `chop_mpfr`, `floatxr`, `floatx`, and `cpfloat_seq` to convert matrices of size n from binary64 to binary16 (left), bfloat16 (middle), and TensorFlow-32 (right) using round-to-nearest with even-on-ties.

of possible outliers. For the MATLAB code, we rely on the function `timeit`, which runs a portion of code several times and returns the median of the measurements.

8.1. Performance of the C interface. Figure 1 compares the time required by `chop_mpfr`, `floatxr`, `floatx`, and `cpfloat_seq` to convert square matrices of increasing order n to lower precision. In all the experiments the storage format is binary64, and we consider three target formats: binary16 (left column), bfloat16 (middle column), and TensorFloat-32 (right column). In this experiment we use only round-to-nearest with ties-to-even, as the FloatX library currently does not support any other rounding modes. We observe, however, that `chop_mpfr` also supports directed rounding as prescribed by the IEEE 754 floating-point standard.

Broadly speaking, the execution time of the four algorithms grows quadratically with the order of the matrix to be converted, and thus linearly with the number of entries. The execution time of `floatxr` and `floatx` increases at a regular pace, with the former being about five times slower than the latter. For $n \geq 20$, `cpfloat_seq` is always the fastest of the four implementations we consider: it is typically about one order of magnitude faster than `floatxr`, with a gap that seems to widen as the size of the matrix to be converted increases, and about a factor two faster than `floatx`. For the three target formats in Figure 1, `chop_mpfr` is the slowest of the four algorithms, particularly for matrices of very small order.

We remark that `chop_mpfr`, `floatxr`, and `cpfloat_seq` are more flexible than `floatx`, as the latter requires the parameters of the target format to be known at compile time, in order for the compiler to instantiate the templates appropriately.

8.2. Performance of the MATLAB interface. Figure 2 reports the speedup of `cpfloat_ml` over `chop_ml`. In each plot we consider the conversion of square matrices of size n between 10 and 10,000 to binary16 (left column), bfloat16 (middle column), and TensorFloat-32 (right column) using the six rounding modes implemented in `chop_ml`. As storage format, we consider both binary32 (top row) and binary64 (bottom row).

The input data is obtained by generating an $n \times n$ matrix X of pseudo-random numbers

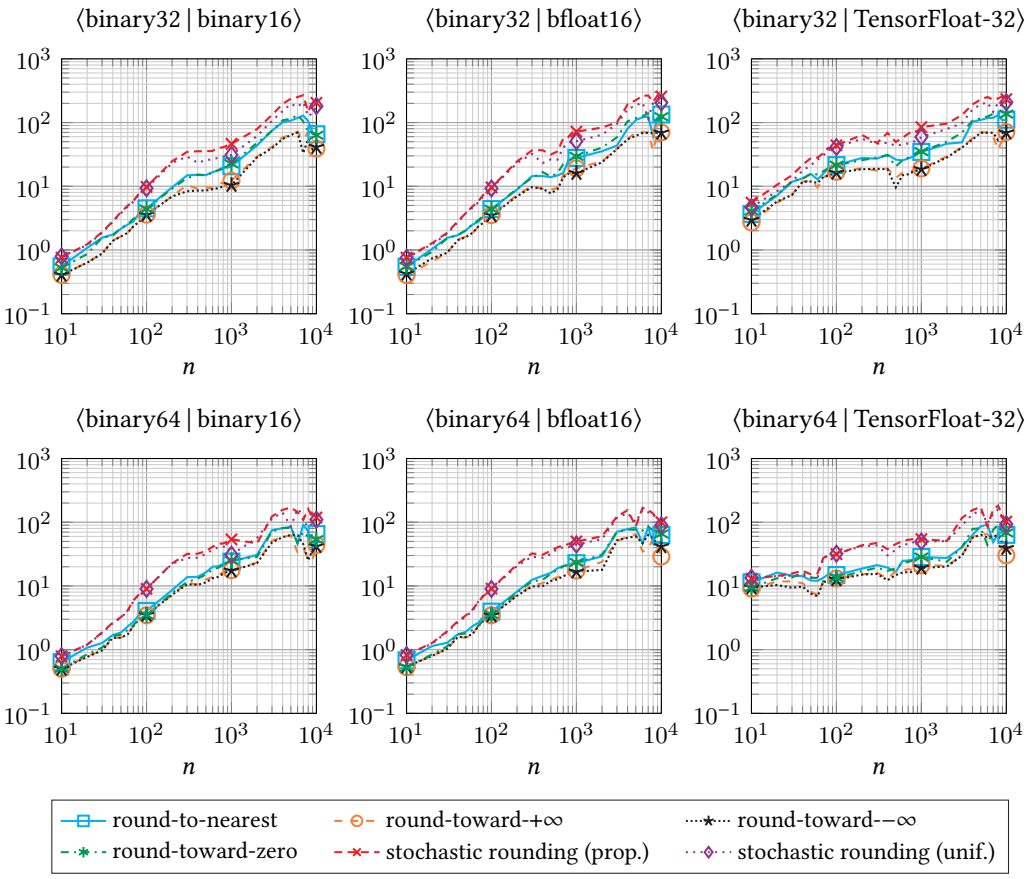

Fig. 2: Ratio of the execution time of `chop_ml` to that of `cpfloat_ml` on $n \times n$ matrices of normal floating-point numbers stored in binary32 (top row) and binary64 (bottom row).

uniformly distributed in $(0, 1)$, and then adding to each entry of X the constant value $x_{\min}^{(\ell)}$, which guarantees that x_{ij} is distributed uniformly in the interval $\left(x_{\min}^{(\ell)}, 1 + x_{\min}^{(\ell)}\right)$ for $i, j = 1, \ldots, n$.

In all six cases, the speedup is greater than one for matrices of order 40 or more, and increases with the size of the input matrix. The two rounding modes for which the new algorithms bring the most significant gains are the two flavors of stochastic rounding. This is expected, as for this rounding mode a large fraction of the computation is devoted to the generation of pseudo-random numbers, an operation for which the new algorithms have a great advantage over `chop_ml`, as they use a more efficient pseudo-random number generator. Round-to-nearest and round-toward-zero show a very similar speedup, just below that of stochastic rounding. The performance gain is somewhat lower for round-toward-$-\infty$ and round-toward-$+\infty$, and the two rounding modes typically achieve very similar results, as the underlying algorithms are essentially equivalent in terms of the operations they perform.

In order to investigate whether the use of subnormal numbers has any impact on the difference in performance between the two implementations, we repeated the experiments above with matrices containing only subnormal entries in the target format. In our experiments, we generated a matrix with these characteristics by first constructing an $n \times n$ matrix X with entries sampled from the uniform distribution over $(0, 1)$, scaling

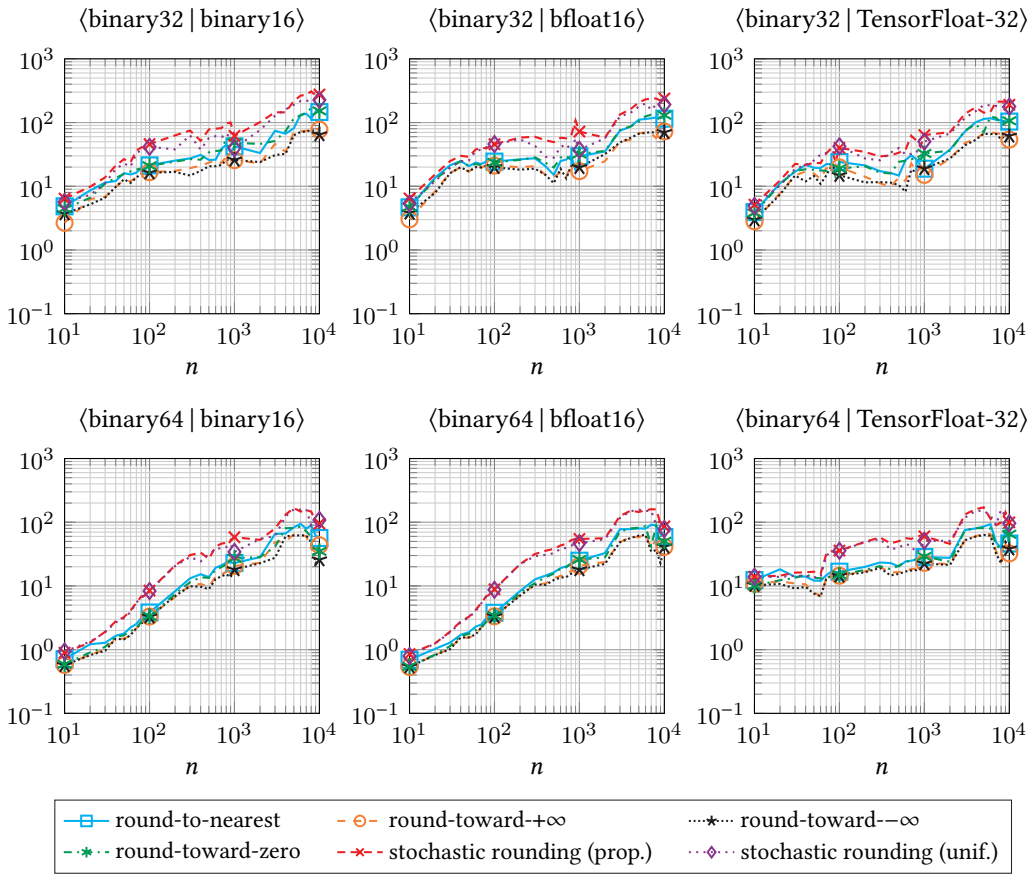

Fig. 3: Ratio of the execution time of `chop_ml` to that of `cpfloat_ml` on $n \times n$ matrices of subnormal floating-point numbers stored in binary32 (top row) and binary64 (bottom row).

its entries by $x^{(\ell)}_{\min} - x^{(\ell)}_{\min\text{sub}}$, and then adding $x^{(\ell)}_{\min\text{sub}}$ to the result. This guarantees that all the entries of X belong to the interval $(x^{(\ell)}_{\min\text{sub}}, x^{(\ell)}_{\min})$, and are thus subnormal in the target precision. According to the results reported in Figure 3, the speedups are largely unaffected when the storage format is binary64, but tend to be larger for n up to 1,000 if the storage format is binary32 and the target format is binary16 or bfloat16.

The result of a similar comparison between `cpfloat` and `floatp_ml` are given in Figure 4. In this case, we limit n to 100, because of the large execution time of the former code on this test set. As with `chop_ml`, the speedup grows linearly with the number of elements in the matrix being converted. However, `floatp_ml` seems to be less efficient than `chop_ml` at the task we examine: the speedup is typically above 100, and is always above 1,000 for the TensorFloat-32 format.

8.3. Overhead of the MATLAB interface. As a final test, we consider the overhead introduced by MATLAB when calling the underlying C implementation of the rounding algorithms. In Figure 5 we compare the execution time required to convert a matrix to binary16 by means of a direct call to the C code (left column) with that required by a call to the MEX interface from MATLAB (right column). As the performance of the two algorithms is very similar and the data in the two series is hard to compare directly, we provide the speedup in the third column. As done in previous experiments,

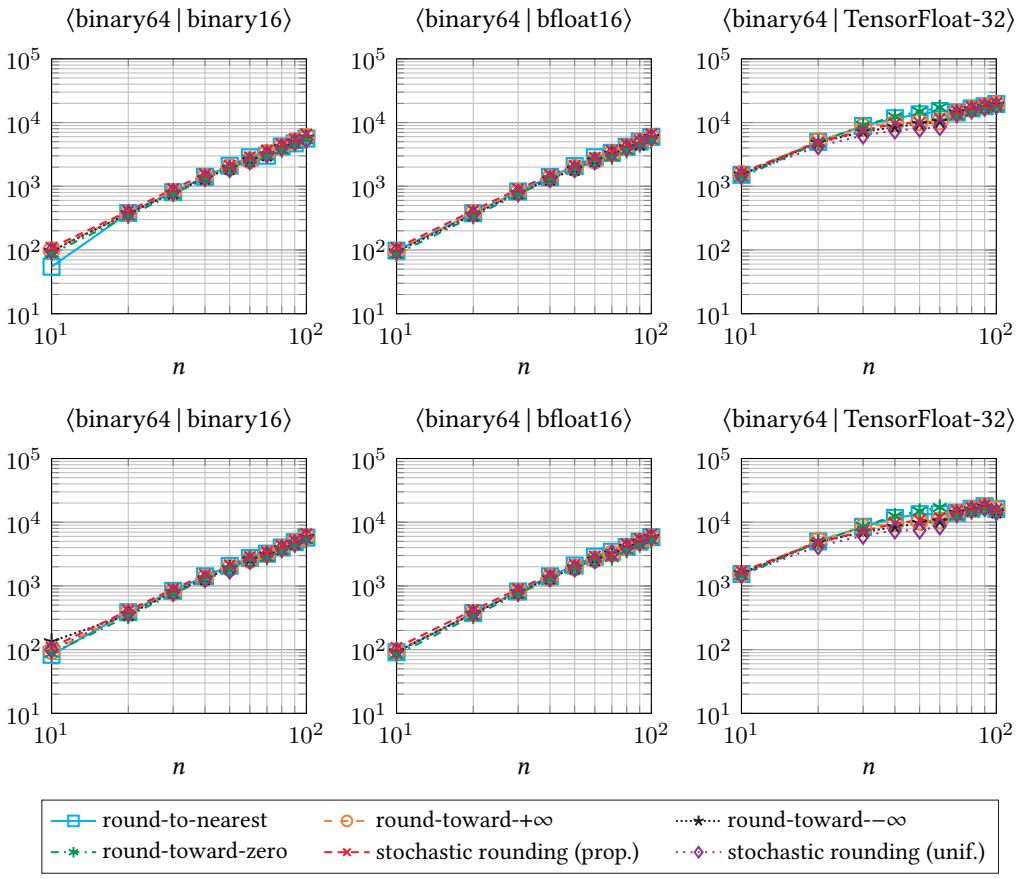

Fig. 4: Ratio of the execution time of `floatp_ml` to that of `cpfloat_ml` on $n \times n$ matrices of normal (top row) and subnormal (bottom row).

we repeat the experiment for both binary32 (top row) and binary64 (bottom row). We remark that the C function was tuned by using the `make autotune` command, whereas for the MEX interface we used `cpfloat_autotune`, a MATLAB function included in the software package.

By looking at the raw execution time, we can see that in our implementation stochastic rounding is the slowest rounding mode, whereas the performance of the other rounding modes is so similar that the lines are hard to distinguish for both the C interface and the MEX interface. The data in the right-most column shows that for both storage formats we consider, the overhead of the MEX interface is significant for small matrices, but becomes negligible for matrices of order 3,000 or larger.

We conclude with an important observation. Our results indicate that MATLAB code that requires the functionalities of `cpfloat_ml` should be translated into C in order to obtain the maximum efficiency. We would like to stress, however, that this translation might bring only a minor performance gain, and in fact not be worth the effort unless the `cpfloat` function is used extensively on matrices of small size. In fact, the overhead of the MEX interface is small in an absolute sense, and for the small matrices where it is noticeable, the overall execution time of both `cpfloat` and `cpfloat_ml` is below 5 milliseconds. This suggests that switching to a pure C implementation would bring only a marginal benefit in most cases.

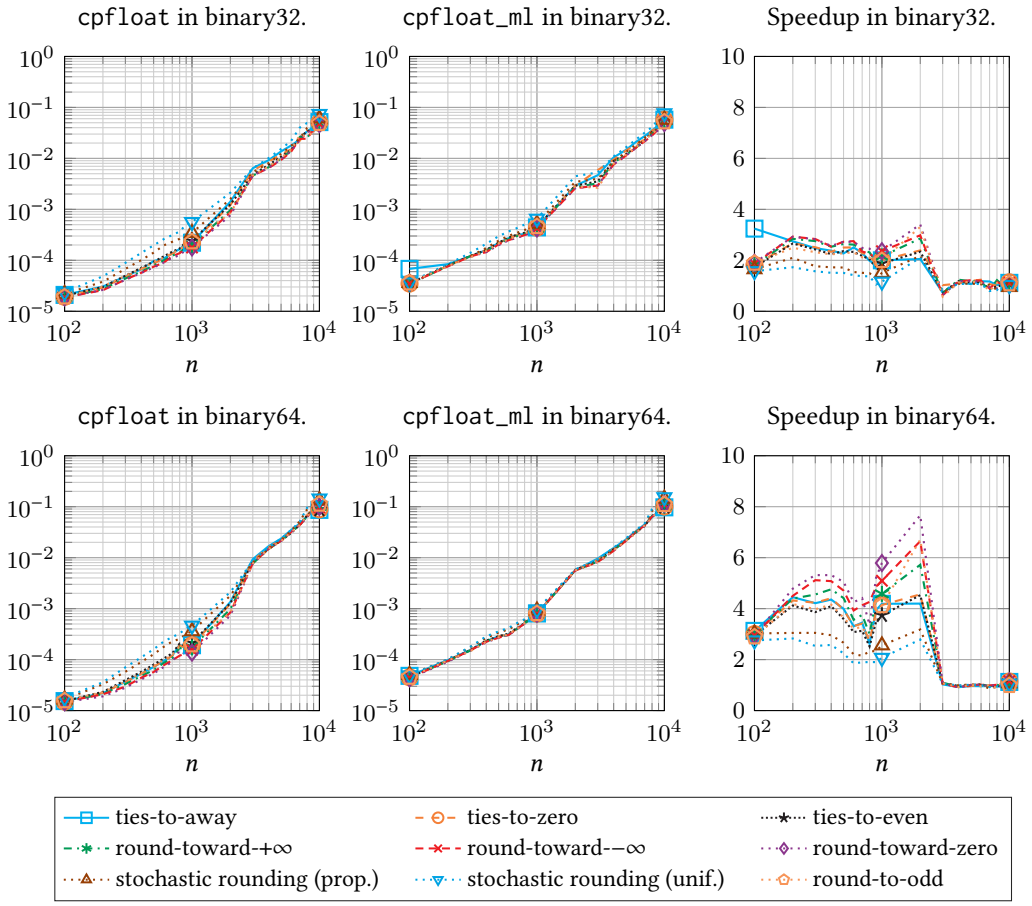

Fig. 5: Execution time, in seconds, of `cpfloat` (first column) and `cpfloat_ml` (second column) on matrices of increasing order n and target format binary16. The third column represents the ratio of the execution time in the first column to that in the second.

9. Summary and future work. Motivated by the growing number of tools and libraries for simulating low-precision arithmetic, we considered the problem of rounding floating-point numbers to low precision in software. We developed low-level algorithms for a number of rounding modes, explained how to implement them efficiently using bit manipulation, and how to validate their behavior experimentally by means of exhaustive testing. We developed `CPFloat`, an efficient C library that implements all the algorithms discussed here and can be used from within MATLAB and Octave by means of a MEX interface we provide. Our experimental results showed that the new implementations outperform existing alternatives in C, C++, and MATLAB.

Traditionally, floating-point arithmetic has been the most widely adopted technique for working with non-integer numbers in high-performance scientific computing, but alternative methods have recently begun to gain popularity. In particular, we believe that the techniques we developed here could be adapted to posit arithmetic [19, 9], a generalization of the IEEE 754 floating-point number format, and to fixed-precision arithmetic, the de facto standard technique for working with reals on systems that are not equipped with a floating-point unit. This will be the subject of future work.

Acknowledgments. The authors thank Nicholas J. Higham for useful discussions about the MATLAB function `chop` and for insightful feedback on early drafts of this manuscript. The authors are also grateful to Theo Mary for testing the MEX interface to CPFloat and for reporting bugs affecting the software.

References.

[1] Arm Limited, *Arm architecture reference manual*, Tech. Report ARM DDI 0487F.c (ID072120), Mar. 2020.

[2] S. Boldo and G. Melquiond, *Emulation of a FMA and correctly rounded sums: Proved algorithms using rounding to odd*, IEEE Trans. Comput., 57 (2008), p. 462–471.

[3] J. M. Bull, *Measuring synchronisation and scheduling overheads in OpenMP*, in Proceedings of First European Workshop on OpenMP, Lund, Sweden, Sept. 1999, pp. 99–105.

[4] J. M. Bull, F. Reid, and N. McDonnell, *A microbenchmark suite for OpenMP tasks*, Lecture Notes in Computer Science, (2012), p. 271–274.

[5] N. Burgess, J. Milanovic, N. Stephens, K. Monachopoulos, and D. Mansell, *Bfloat16 processing for neural networks*, Proceedings of the 26th IEEE Symposium on Computer Arithmetic, (2019), pp. 88–91.

[6] M. P. Connolly, N. J. Higham, and T. Mary, *Stochastic rounding and its probabilistic backward error analysis*, MIMS EPrint 2020.12, Manchester Institute for Mathematical Sciences, The University of Manchester, UK, Apr. 2020. Revised August 2020.

[7] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C.-K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y.-H. Weng, A. Wild, Y. Yang, and H. Wang, *Loihi: A neuromorphic manycore processor with on-chip learning*, IEEE Micro, 38 (2018), pp. 82–99.

[8] A. Dawson and P. D. Düben, *rpe v5: An emulator for reduced floating-point precision in large numerical simulations*, Geosci. Model Dev., 10 (2017), pp. 2221–2230.

[9] F. de Dinechin, L. Forget, J.-M. Muller, and Y. Uguen, *Posits: The good, the bad and the ugly*, Proceedings of the Conference for Next Generation Arithmetic, (2019), pp. 1–10.

[10] M. Fasi and M. Mikaitis, *Algorithms for stochastically rounded elementary arithmetic operations in IEEE 754 floating-point arithmetic*, MIMS EPrint 2020.9, Manchester Institute for Mathematical Sciences, The University of Manchester, UK, Mar 2020.

[11] G. Flegar, F. Scheidegger, V. Novaković, G. Mariani, A. E. Tomás, A. C. I. Malossi, and E. S. Quintana-Ortí, *FloatX: A C++ library for customized floating-point arithmetic*, ACM Trans. Math. Software, 45 (2019), p. 1–23.

[12] G. E. Forsythe, *Round-off errors in numerical integration on automatic machinery*, Bull. Amer. Math. Soc., 56 (1950), pp. 55–64.

[13] ——, *Reprint of a note on rounding-off errors*, SIAM Rev., 1 (1959), p. 66–67.

[14] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, *MPFR: A multiple-precision binary floating-point library with correct rounding*, ACM Trans. Math. Software, 33 (2007), pp. 13:1–13:15.

[15] D. Goldberg, *What every computer scientist should know about floating-point arithmetic*, ACM Comp. Surv., 23 (1991), p. 5–48.

[16] F. Goualard, *Generating random floating-point numbers by dividing integers: A case study*, in Computational Science – ICCS 2020, V. V. Krzhizhanovskaya, G. Závodszky, M. H. Lees, J. J. Dongarra, P. M. A. Sloot, S. Brissos, and J. Teixeira, eds., Lecture Notes in Computer Science, Cham, Switzerland, 2020, Springer-Verlag, p. 15–28.

[17] Graphcore Limited, *IPU programmer's guide*, 2020.

[18] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, *Deep learning with limited numerical precision*, in Proceedings of the 32nd International Conference on Machine Learning, F. Bach and D. Blei, eds., vol. 37 of Proceedings of Machine Learning Research, Lille, France, July 2015, PMLR, pp. 1737–1746.

[19] J. L. Gustafson and I. T. Yonemoto, *Beating floating point at its own game: Posit arithmetic*, Supercomputing Frontiers and Innovations, 4 (2017), pp. 71–86.

[20] N. J. Higham and S. Pranesh, *Simulating low precision floating-point arithmetic*, SIAM J. Sci. Comput., 41 (2019), pp. C585–C602.

[21] M. Hopkins, M. Mikaitis, D. R. Lester, and S. Furber, *Stochastic rounding and reduced-precision fixed-point arithmetic for solving neural ordinary differential equations*, Philos. Trans. R. Soc. A, 378 (2020).

[22] Intel Corporation, *BFLOAT16—hardware numerics definition*. White paper. Document number 338302-001US, Nov. 2018.

[23] ——, *Intel architecture instruction set extensions and future features programming reference*, Mar. 2020.

[24] International Business Machines Corporation, *Power ISA version 3.0 B*, 2017.

[25] G. Meurant, *FLOATP_Toolbox, Matlab software, variable precision floating point arithmetic*, 2020.

[26] C. B. Moler, *"Half precision" 16-bit floating point arithmetic*. Blog post, Dec. 2017.

[27] J.-M. Muller, *On the definition of* ulp(x), Research Report RR-5504, LIP RR-2005-09, INRIA, LIP, May 2005.

[28] NVIDIA Corporation, *NVIDIA Tesla P100 GPU architecture*, Tech. Report WP-08019-001_v01.1, 2016.

[29] ——, *NVIDIA A100 tensor core GPU architecture*, 2020. NVIDIA whitepaper v1.0.

[30] I. of Electrical and E. Engineers, *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*, Institute of Electrical and Electronics Engineers, Piscataway, NJ, USA, Oct. 1985. Reprinted in SIGPLAN Notices, 22(2):9–25, 1987.

[31] ——, *IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2008 (revision of IEEE Std 754-1985)*, Institute of Electrical and Electronics Engineers, Piscataway, NJ, USA, Aug. 2008.

[32] ——, *IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2019 (revision of IEEE Std 754-2008)*, Institute of Electrical and Electronics Engineers, Piscataway, NJ, USA, July 2019.

[33] M. E. O'Neill, *PCG: A family of simple fast space-efficient statistically good algorithms for random number generation*, Tech. Report HMC-CS-2014-0905, Harvey Mudd College, Claremont, CA, Sept. 2014.

[34] M. L. Overton, *Numerical Computing with IEEE Floating Point Arithmetic*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001.

[35] Radeon Technologies Group, *Radeon's next-generation Vega architecture*, tech. report, Advanced Micro Devices, 2017. File no longer available on the AMD website. Archived version at https://en.wikichip.org/w/images/a/a1/vega-whitepaper.pdf.

[36] P. Roux, *Innocuous double rounding of basic arithmetic operations*, J. Formaliz. Reason., 7 (2014), pp. 131–142.

[37] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, *Training deep neural networks with 8-bit floating point numbers*, in Advances in Neural Information Processing Systems 31, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, eds., Curran Associates, Inc., Montreal, Canada, Dec. 2018, pp. 7675–7684.

[38] J. H. Wilkinson, *Rounding Errors in Algebraic Processes*, Notes on Applied Science No. 32, Her Majesty's Stationery Office, London, 1963. Also published by Prentice-Hall, Englewood Cliffs, NJ, USA. Reprinted by Dover, New York, 1994.

[39] T. Zhang, Z. Lin, G. Yang, and C. De Sa, *QPyTorch: A low-precision arithmetic simulation framework*. arXiv:1910.04540 [cs.LG], Oct. 2019.