Low-precision climate computing: Preserving information despite fewer bits

Milan Klöwer
Atmospheric, Oceanic and Planetary Physics
Jesus College
University of Oxford

supervised by

Prof. Tim Palmer, University of Oxford
Dr. Peter Düben, European Centre for Medium-Range Weather Forecasts

A thesis submitted for the degree Doctor of Philosophy

Oxford, September 2021

DON’T PANIC.
NOBODY NEEDED THOSE BITS ANYWAY.

Abstract

Progress towards more reliable weather and climate forecasts is limited by the resolution of numerical models and the complexity of simulated processes. Performance is therefore a major bottleneck and current models are not computationally efficient. High precision calculations are unnecessary, despite being the standard, given the uncertainties in the climate system and the errors from discretisation, data assimilation and unresolved climate processes. In this thesis, we advance several aspects of low-precision climate computing to preserve information despite fewer bits: An information-preserving compression is developed that distinguishes between real and false information to reduce the very large volume of climate data produced by numerical models, while minimising information loss. The bitwise real information content estimates the minimum required precision in climate data, which depends on the variable and is lower than the standard precision-levels of floating-point numbers. The impact of rounding errors introduced by different low-precision arithmetics with deterministic or stochastic rounding modes is analysed in chaotic dynamical systems. Standard floating-point numbers are not the best number format for weather and climate simulations. However, alternatives, such as posits, exist, but it is unclear whether the large effort needed to develop the respective hardware for future supercomputers is justified given the moderate advantage they provide in our applications. A much more central issue towards 16-bit climate models is the design of low precision-resilient algorithms. A naive transition to 16 bits either fails or was found to cause issues like amplified gravity waves, a change in geostrophy or rounding errors that grow as quickly as discretisation errors. However, many of these issues are found to be preventable with techniques such as scaling or a compensated time integration. Combining techniques, we develop a 16-bit fluid circulation model that approaches 4x speedups on Fujitsu's A64FX processor compared to 64 bits, despite minimal rounding errors. The result of this thesis show that there is little reason to assume that 16-bit weather and climate models are not possible. While the design of models to compute and output only the bitwise real information is challenging, it will be a major step towards computationally efficient digital twins of the Earth's climate system.

Contents

1 Introduction

Numerical models of weather and climate use binary numbers to calculate and store information. All information in a climate model, or its data output, is encoded into bits. Representing real numbers with bits bisects for every bit the real axis into different sections, similar to a binary tree. While there are various ways to encode a real number into bits, some bits will be more significant, others less, to encode a good approximation. For example, flipping the sign bit will generally result in the most significant change of the represented number. Flipping other bits will cause a much less significant change. Depending on the encoding and the number of bits used, some bits will be more essential than others in a climate model or its data to obtain a meaningful simulation. Especially the least significant bits may provide a precision that is more precise than necessary to resolve the uncertainty inherent in most numbers in climate data or their computation.

The uncertainty of weather or climate data, whether simulated or measured, depends on many different factors and use cases. While it may be crucial to distinguish between 0°C and slightly warmer to measure frost on a field, the exact temperature on a single field will have negligible impact on the global mean surface temperature averaged over decades. On the other hand, a weather forecast for a given location is rarely more than 1°C accurate, but to quantify global warming from year to year, a higher precision of 0.1°C or even 0.01°C is needed [Haustein *et al.*, 2017]. In general every number will come with a different uncertainty, some of which will amplify during calculation or cancel out.

While the uncertainty in climate computations and data is variable, the precision of binary number formats has been standardised since the 1980s [IEEE, 1985, 2008]. Consequently, only two levels of precision are widely available (single and double precision) for all areas of computing, regardless of the varying levels of required precision per application. So-called single-precision floating-point numbers use 32 bits to encode a real number, providing at least 7 decimal places of precision in the representable range between 10^{-38} and 10^{38}. Double-precision floating-point numbers use 64 bits instead, resolving numbers over more than 600 orders of magnitude at a precision of more than 16 decimal places.

64-bit computations are the de facto standard for scientific computing. Most programming languages use it as the default precision, which enabled decades of scientific calculations while ignoring the remaining, in most cases indeed negligible, rounding errors. Applications where computational performance is not essential, or where optimi-

sations for lower precision have not been applied, are generally advised to use 64 bits. However, many areas of high-performance computing, including the data compression and storage, can benefit from low-precision computations [Palmer, 2015]. Requiring the world's largest supercomputers, numerical weather prediction and climate projections are such high-performance computing applications [Bauer *et al.*, 2021a].

On many processors two 32-bit computations can be vectorised into a single 64-bit operation, effectively making computations with 32 bits twice as fast as with 64 bits. Similarly, the vectorisation of 16 or even 8-bit computations, if supported, allow for 4 or 8 times as many operations within the same time [Sato *et al.*, 2020]. But vectorisation is not the only advantage of low-precision computing. The performance bottleneck of many applications is the speed at which data can be loaded from memory [Müller *et al.*, 2019]. Decreasing the time it takes for the computation alone will not increase performance if the processor has to wait for data to be loaded from memory. However, such a memory-bound application would greatly benefit if every number in memory is stored in only 32, 16 or even 8 bit instead of 64, which reduces the data volume transferred between memory and processor. An array of 16-bit numbers is loaded four times as fast from memory, compared to an array of the same size but with 64 bits per entry.

While most processors only support 32 and 64-bit computations, more freedom is available for data compression and storage: In principle, numbers encoded with an arbitrary number of bits can be packed into an array, such that also 10, 17 or 31 bits per number are possible [WMO, 2003]. Although those bits will need to be unpacked into 32 or 64 bits for post-processing, the data volume would be greatly reduced, lowering storage requirements in data archives. Consequently, the bits per number can directly reflect the uncertainty in data, provided such an uncertainty is known. Storing more bits to preserve a higher precision therefore corresponds to storing *false information* as the uncertainty masks the unnecessarily high precision [Jeffress *et al.*, 2017]. In general, it is an open question how to distinguish between real and false information in data, as the uncertainties may be unknown. This question directly translates to the bits per number, of which only as few as essential should be used to preserve the real information.

Modern hardware has started to support also 16-bit computations next to the traditional 32 and 64 bits, adding so-called half precision to the set of available levels of precision for computing [Jouppi *et al.*, 2018b; Odajima *et al.*, 2020]. Fugaku, the fastest supercomputer as of June 2021 (TOP500, Dongarra & Luszczek [2011]), consists of many million processors all of which are capable to compute 16-bit numbers four times as fast as 64 bits. However, these 16-bit half-precision floating-point numbers have a reduced precision of less than 4 decimal places over a limited range from $6 \cdot 10^{-8}$ to 65504.

Numbers with an absolute value smaller or larger cannot be represented. Using 16-bit computations in complex simulations is therefore challenging [Klöwer *et al.*, 2020]: The range of numbers occurring in simulation has to be controlled, which requires scaling of algorithms to avoid very small and very large numbers outside the representable range. Rounding errors arising from computations at lower precision should not exceed other errors, which may require some algorithms to be resilient to low precision such that rounding errors do not accumulate over time [Higham, 2002]. For complex simulations it is often difficult to understand in which computations precision or range issues arise as most error analysis is based on the data output and not on intermediate results within a simulation [Fevotte & Lathuilière, 2019; Jézéquel & Chesneaux, 2008].

Simulations with 64 bits are not rounding error-free, but the error analysis is challenging or impossible in situations where analytical solutions are unknown. Even higher precision can be used as a reference, assumed to be error-free, but simulated systems can differ from their analytic counterparts at any finite precision [Boghosian *et al.*, 2019]. As an alternative, interval arithmetic simultaneously rounds up and down to contain the exact result and therefore provides an upper and lower bound on the rounding error [Gustafson, 2015]. However, in many chaotic systems these bounds diverge quickly and the interval becomes meaningless as the effective error is highly overestimated.

A weather or climate model simulates a chaotic system such that every rounding error grows and eventually causes the simulated weather to change globally [Lorenz, 1963; Palmer *et al.*, 2014]. In practice, the statistics of any low precision simulation have to be compared to the statistics from a high precision reference. A transition to lower precision can then be considered if no significant deviation of the simulated statistics with a changing precision is found. Such a comparison can be understood as a Turing test [Baker *et al.*, 2019; Turing, 1950] in which a low precision simulation or data set has to pass in comparison to a high precision reference with respect to any metrics that are considered important for a simulation's use cases. For weather forecast models higher rounding errors may be acceptable if the skill in predicting the weather at a given location within a week is not degraded [Düben *et al.*, 2014]. However, the impacts of rounding errors on such complex systems can be diverse and not obvious. While a change in precision impacts the simulation of every chaotic dynamical system, the usefulness of a simulation can be unaffected even over a large range of precision [Palmer, 2015].

In this thesis, several open questions of low-precision climate computing are addressed. Chapter 2 introduces the methods that are used throughout several following chapters. Various binary number formats are presented that can be used to approximate real numbers. Some formats are widely available standards, others are recently

proposed alternatives. The concepts of how different number formats encode information into bits are essential to understanding how changing precision affects climate simulations and data. How non-standard number formats can be used flexibly in simulations is explained with code composability. Error norms are discussed which quantify the rounding errors arising from different available rounding modes. The last section in this chapter introduces core concepts of information theory to analyse the information contained within the bits of binary numbers.

Chapter 3 develops the concept of information-preserving compression for climate data. The bitwise real information content is defined and proposed as a metric to distinguish between the real and false information in climate data. Using a large set of atmospheric data, the real information content leads to an individual precision for every variable like temperature or humidity, objectively representing its uncertainty. Consequently, only the real information is subject to data compression while discarding bits without information.

Chapter 4 investigates the impact of rounding errors from various numbers format in representing simple chaotic dynamical systems. Bitwise periodic orbits are systematically analysed to better understand how the structure of attractors is affected with lower precision. Invariant measures quantify the statistics of a simulated system, which are compared as a function of the number format and its precision.

Chapter 5 explores the scope of various 16-bit formats when simulating the shallow water model, a medium-complexity fluid circulation model. The impact of rounding errors on the simulated dynamics is analysed, including geostrophy and gravity waves. Short-term forecasts are used to assess the growth of rounding errors arising from 16-bit calculations and when mixing 16 and 32 bits of precision used for different calculations within the simulation.

Chapter 6 implements the fluid circulation model ShallowWaters.jl on 16-bit hardware. The Fujitsu processor A64FX is used to investigate the potential of accelerating weather and climate models with 16-bit calculations. The simulated shallow water equations are systematically rescaled to fit into the limited range of representable numbers with 16 bit. A compensated time integration is presented that reduces precision issues with 16 bits. The chapter concludes with an analysis of the performance gain between 64, 32 and 16 bits on A64FX.

Chapter 7 summarises all previous chapters and discusses an outlook of low-precision climate computing for improved forecasts of the global weather and the Earth's climate system in the future.

2 Methods

Contributions Parts of this chapter are based on the following publication [*]

M Klöwer, PD Düben, and TN Palmer, 2020. *Number Formats, Error Mitigation, and Scope for 16-Bit Arithmetics in Weather and Climate Modeling Analyzed With a Shallow Water Model*, **Journal of Advances in Modeling Earth Systems**, 10.1029/2020MS002246.

This chapter presents the methods that are used in several of the following chapters. We introduce various binary number formats, their rounding modes, and present error norms to quantify the resulting rounding errors. It follows a section focusing on programming aspects around type flexibility and code composability to illustrate how custom number formats can be used within the Julia programming language. The chapter closes with an introduction to information theory to explain the essential concepts to quantify information in binary number formats.

2.1 Binary number formats

This section describes encodings for binary integers (section 2.1.1), and different formats that approximate real numbers: Fixed-point numbers (section 2.1.2), floating-point numbers (section 2.1.3), logarithmic fixed-point numbers (section 2.1.4), and posits (section 2.1.5). A summary on the number formats is given in section 2.1.6.

2.1.1 Integers

Unsigned integers

Non-negative integers are represented in binary as unsigned integers. An n-bit unsigned integer is a sequence of n bits $b_1, \ldots, b_n$ which are decoded into the integer u as a sum of powers of two with exponents $n-1, n-2, \ldots, 1, 0$

$$u = \sum_{i=1}^{n} b_i 2^{n-i} \tag{2.1}$$

We hereby use the big-endian notation [Cohen, 1981; James, 1990], such that the first bit b_1 represents the most significant figure and the last bit b_n the least significant. For example, the

[*] with the following author contributions. Conceptualisation: MK, PDD. Data curation: MK. Formal Analysis: MK. Methodology: MK. Visualisation: MK. Writing – original draft: MK. Writing – review and editing: MK, PDD, TNP

$n = 4$ bit unsigned integer `0101` is decoded as $0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$ and therefore represents the integer 5 in decimal.

n-bit unsigned integers will be referred to as UInt*n*, such that the 8-bit version is called UInt8, and so on. It is often convenient to write the hexadecimal format to represent unsigned integers or any bit sequence in general. We prefix a hexadecimal with `0x`, such that `1100010010100101` in binary is equivalent to `0xc4a5` in hexadecimal. Always 4 bits are hereby grouped and represented with one of the 16 symbols from 0-9 and a-f for the 16 possible states from `0000` (=`0x0`) to `1111` (=`0xf`). For readability hexadecimal numbers may be intercepted with an underscore, which has no other meaning, e.g. `0x5a19_ab7b = 0x5a19ab7b`.

The range of representable integers with UInt*n* is 0 to $2^n - 1$. Addition, subtraction and multiplication with a result within this range is exact so that no rounding has to be applied. A result outside this range is in most implementations subject to a *wrap-around* behaviour which is equivalent to subsequent modulo operation with 2^n. For example, for UInt16 we have `0xffff + 0x1 = 0x0`, i.e. adding 1 to 65536 would return 0; or `0x0000 - 0x1 = 0xffff`, i.e. subtracting 1 from 0 returns 65536. Such an example of modulo arithmetic avoids special cases (like NaN for floats, see section 2.1.3), but can easily yield catastrophic results when integers are used to approximate real numbers for scientific simulations.

Signed integers

While unsigned integers cannot represent negative numbers, signed integers use the first bit b_1 to indicate the sign ($b_1 = 0$ is positive and $b_1 = 1$ negative). One way to decode an *n*-bit signed integer is the sign and magnitude representation

$$s = (-1)^{b_1} \sum_{i=2}^{n} b_i 2^{n-i-1} \tag{2.2}$$

which covers the range $[-2^{n-1} - 1, 2^{n-1} - 1]$ but uses two bitpatterns to encode 0 (`0x00...00` = +0 and `0x10...00` = -0). However, to avoid multiple representations of zero and to simplify hardware implementations, the sign and magnitude representation is rarely used. The most common representation uses the two's complement, which flips all bits in a sequence (i.e. `0` $\rightarrow$ `1` and `1` $\rightarrow$ `0`) and adds `1`. For example, the two's complement of `0101` is `1011`. The two's complement representation of negative integers, i.e. those with a sign bit of 1, first flips all magnitude bits and then adds 1 before using Eq. 2.2 for decoding. For example, in the 4-bit signed integer format Int4 a value is stored as `1110` (the sign bit is marked in red). The magnitude bits of `1110` are then converted using the two's complement to `1010` (`110` is bit flipped into `001` and then adding `1` yields `010`), which is the sign and magnitude representation (SignMag) that can be decoded using Eq. 2.2 into -2 in decimal: $1110_{\text{Int4}} \rightarrow 1010_{\text{SignMag}} \rightarrow -2$.

Signed integers with the two's complement representation are in $[-2^{n-1}, 2^{n-1} - 1]$, which is subject to a similar wrap-around behaviour for arithmetic results beyond the representable range as for unsigned integers. The two's complement representation is favoured in hardware

implementations as no distinction has to be made between signed and unsigned types for the arithmetic operations addition, subtraction and multiplication [Choo *et al.*, 2003].

2.1.2 Fixed-point numbers

Fixed-point numbers extend the integer format with fraction bits in order to better approximate real numbers. Next to n_i signed integer bits, an n-bit fixed-point number format also uses n_f fraction bits to encode an additional sum of powers of two with negative exponents $-1, -2, ..., -n_f$. Note that $n = n_i + n_f$, so every additional fraction bit reduces the number of integer bits. A common notation is, for example, Q6.10, which is the 16-bit fixed-point format with 6 signed integer bits and 10 fraction bits. With the sign and magnitude representation, a fixed-point number is decoded into x as

$$x = (-1)^{b_1} \left(\sum_{i=2}^{n_i} b_i 2^{n-i-1} + \sum_{i=1}^{n_f} b_{n_i+i} 2^{-i} \right) \tag{2.3}$$

The first summation is identical to the signed integers (Eq. 2.2), and therefore one might use the two's complement alternatively. Using $n_f = 1$ can only encode real numbers at a distance of $\frac{1}{2}$ apart, so the representable numbers are $..., -\frac{1}{2}, 0, \frac{1}{2}, 1, \frac{3}{2}, 2, ...$, but using more fraction bits allows a higher precision. For $n_f = 10$ fraction bits, numbers with a spacing of $2^{-10} \approx 0.001$ are representable, which is also the smallest representable positive number called *minpos*. However, this comes at the cost of a reduced largest representable number *maxpos* and therefore a smaller range $maxpos - minpos$ (Table 2.1). For the format Q6.10 with 6 integer bits and 10 fraction bits the largest representable number is $maxpos = \texttt{0x7fff} = 2^5 - 1 + \sum_1^{10} 2^{-i} \approx 31.999$. Results beyond *maxpos* will trigger in many implementations also a wrap-around behaviour as for integers.

In the following we consider the range of a number format to be $maxpos - minpos$, and the dynamic range as $\frac{maxpos}{minpos}$, usually given in $\log_{10}(\frac{maxpos}{minpos})$ orders of magnitude. For a 16-bit fixed-point number format the range is approximately $2^{n_i - 1}$ and therefore decreases with fewer integer bits. However, the dynamic range is always 2^{n-1} and therefore only increases for more bits in an n-bit format. For Q6.10 the range is less than 32 and the dynamic range is 4.5 orders of magnitude.

Choosing the number of integer bits n_i and fraction bits n_f is equivalent to interpreting the bits of a fixed-point number as an integer with the scaling factor 2^{-n_f}. Flexibility regarding the range can therefore be achieved with integer arithmetic if fixed-point numbers are used [Russell *et al.*, 2017]. Unfortunately, the very limited range at the cost of precision is unsuited for many applications. Nevertheless, they provide exact addition, subtraction and multiplication (as with integers) unless an overflow or underflow to 0 occurs. They can be suited to represent monetary values for example. However, fixed-point numbers are rarely used since standardised floating-point numbers became widely available in the 1980s and 1990s.

2.1.3 Floating-point numbers

While fixed-point numbers have constant spacing, meaning a fixed number of digits *after* the decimal point, floating-point numbers (or simply floats) were introduced for a *floating* decimal point. In consequence, floats have an approximately constant number of significant digits (before and after the decimal point) across a wide range of representable numbers. Floats achieve this by encoding an exponent which is multiplied to the significand (also called mantissa).

The floating-point format is standardised in IEEE [1985, 2008] which defines how the bits $b_1, ..., b_n$ of an n-bit float are converted into a real number x in terms of a sign bit, n_e exponent and n_m mantissa bits. Most floats (hence they are called *normal floats*) are decoded as

$$x = (-1)^{sign\ bit} \cdot 2^{e-bias} \cdot (1+f) \tag{2.4}$$

and exceptions are discussed in Eq. 2.6. The exponent bits are interpreted as an unsigned integer e, such that $e - bias$ converts them effectively to signed integers, with $bias = 2^{n_e-1} - 1$. Consequently, both positive and negative exponents can be represented, for absolute values of x larger than 1 and in between 0 and 1, respectively. The bias changes with the number of exponent bits n_e to have logarithmically a similar range for very large and very small numbers: $e - bias$ is any power of two in $[-2^{n_e-1}+2, 2^{n_e-1}-1]$ for $e = 1, ..., 2^{n_e}-2$. Note, $e = 0$ and $e = 2^{n_e}-1$ are reserved for special cases such as 0 and infinity, which will be introduced in Eq. 2.6.

The mantissa bits $m_1, ..., m_{n_m}$ are interpreted as the fraction $f = \sum_{i=1}^{n_m} m_i 2^{-i}$, identical to the fraction bits of fixed-point numbers (see Eq. 2.3). Including the so-called hidden bit (which represents the addition with $2^0 = 1$ in Eq. 2.4) the term $(1+f)$ is in the bounds $[1, 2)$, and therefore encodes the significant digits behind the decimal (or binary) point.

An 8-bit float, for example, encodes the closest float to $\pi = 3.14159...$ with a sign bit (red), $n_e = 3$ exponent bits (blue) and $n_m = 4$ mantissa bits (black) as

$$0\,1001001_{\text{Float8}} = (-1)^0 \cdot 2^{4-bias} \cdot (1 + 2^{-1} + 2^{-4}) = 3.125 \approx \pi \tag{2.5}$$

with $bias = 2^{n_e-1} - 1 = 3$ here.

In order to also encode 0, plus and minus infinity ($\pm$Inf), and Not-A-Number (NaN), there are exceptions to Eq. 2.4 that are discussed in the following: When all exponent bits and mantissa bits are 0 a float is decoded as 0. If the exponent bits are all 0 but the mantissa bits are not ($f > 0$) a float is decoded as one of the subnormal numbers, which were introduced to uniformly fill the gap between 0 and the smallest normal number $2^{-2^{n_e-1}+2}$ (also called *floatmin*) that can be obtained from Eq. 2.4. Another exception to Eq. 2.4 occurs when all exponent bits are 1, in which case the float is decoded as $\pm\infty$ (depending on the sign bit) if the mantissa bits are all 0.

However, for any non-zero mantissa bits ($f > 0$) the float is decoded as NaN. In summary,

$$x = \begin{cases} (-1)^s \cdot 2^{e-bias} \cdot (1+f) & \text{if } e > 0, & & \text{(normals)} \\ (-1)^s \cdot 2^{1-bias} \cdot f & \text{if } e = 0 & \text{and } f > 0, & \text{(subnormals)} \\ (-1)^s \cdot 0 & \text{if } e = 0 & \text{and } f = 0, & (\pm 0) \\ (-1)^s \cdot \infty & \text{if } e = 2^{n_e} - 1 & \text{and } f = 0, & (\pm \text{Inf}) \\ \text{NaN} & \text{if } e = 2^{n_e} - 1 & \text{and } f > 0, & \text{(Not-A-Number).} \end{cases} \tag{2.6}$$

Floating-point formats cover a wide range of representable numbers. The smallest positive float is called *minpos*, which is the smallest subnormal number encoded as `0x00...01`, $minpos = 2^{2-2^{n_e-1}-n_m}$. The largest positive float is called *floatmax* or *maxpos* which occurs for $e = 2^{n_e} - 2$ (the exponent bits are `0x11...10`) and $f = 2 - 2^{-n_m}$ (the mantissa bits are `0x11...11`), hence $maxpos = 2^{2^{n_e}-1}\left(1 - 2^{-n_m-1}\right)$. For an illustration of the dynamic range covered by some common floating-point formats see Fig. 2.4. More exponent bits allow for a wider dynamic range as both *minpos* and *maxpos* effectively scale with $2^{\pm 2^{n_e}}$ (positive sign for *maxpos*, negative sign for *minpos*). However, more exponent bits also mean less mantissa bits which reduces the precision. Consequently, for a given number of total bits per float there is a trade-off between range and precision, similar to the trade-off for fixed-point numbers. For more details on the precision of floats see section 2.3.3.

Due to the exponent bits, floats are approximately logarithmically distributed on the real number line with about half of the representable numbers packed between -1 and 1. However, as the mantissa bits are a sum of powers of two, floats are also piecewise uniformly (or linearly) distributed. Within a given range of two powers of two $[2^i, 2^{i+1})$, e.g. $[1, 2)$ or $[\frac{1}{8}, \frac{1}{4})$ only the mantissa bits change, such that the distance δ between two subsequent floats is always equal and $\delta = 2^{i-n_m}$. But for the next larger power of two, this distance doubles, whereby this doubling accounts for the approximately logarithmic distribution of floats throughout the dynamic range. The distance δ is closely related to the rounding error and discussed in more detail in section 2.2.2.

The two most common floating-point formats are Float32 and Float64. Float32 is a 32-bit format with $n_e = 8$ exponent bits and $n_m = 23$ mantissa bits. Float64 is a 64-bit format with $n_e = 11$ exponent bits and $n_m = 52$ mantissa bits. Both have been standardised in IEEE [1985]. Recently more widely available on specialised hardware are the two 16-bit low precision formats Float16 (5 exponent bits and 10 mantissa bits) and BFloat16 (8 exponent bits and 7 mantissa bits, a truncated version of Float32). The number $\pi = 3.14159\ldots$ is encoded in these formats as (sign

bit in red, exponent bits in blue, mantissa bits in black)

$$\begin{aligned}
\text{Float64:} &\quad 0100000000001001001000011111011 \\
&\quad 0101010001000100001010110100011000 = 3.141592653589793 \\
\text{Float32:} &\quad 0100000001001001000011111011011 = 3.1415927 \\
\text{Float16:} &\quad 0100001001001000 = 3.140625 \\
\text{BFloat16:} &\quad 0100000001001001 = 3.140625
\end{aligned} \tag{2.7}$$

While both Float16 and BFloat16 represent π as 3.140625 note that Float16 is three bits more precise, which just happen to be all 0.

Floating-point formats have many ways to encode a NaN (Eq. 2.6). In the case of the exponent bits being all 1 any non-zero mantissa encodes a NaN, which amounts to $2^{n_m+1} - 2$ possibilities (2^{n_m+1} for all combinations of mantissa bits and the sign, minus two for the mantissa being zero, which encodes $\pm$Inf). Hence, for Float64 there are more than $1.8 \cdot 10^{16}$ bitpatterns that encode NaN, which are, however, only $2^{-n_e} \approx 0.05\%$ of all bitpatterns. Despite fewer mantissa bits, for Float16 there are still 2046 ways to encode a NaN, about 3.1% of all available bitpatterns (see also Fig. 6.2).

A summary of the characteristic of different floating-point formats is found in Table 2.1. For a more in-depth discussion of the precision of these formats see section 2.3.3.

2.1.4 Logarithmic fixed-point numbers

The major difference between floats and integers (or fixed-point numbers) is their ability to represent very large and very small numbers simultaneously at similar precision. The floating-point format achieves this with exponent bits such that the floats are approximately logarithmically spaced out on the positive and negative part of the real number line. This motivates us to define a logarithmic fixed-point number format (in short logfix), which is not a hybrid format of linear and logarithmic spacing as floats are, but follows a perfect logarithmic spacing. We implement the logfix formats presented here as the software emulator LogFixPoint16s.jl and further technical details can be found therein.

Logfixs are an appealing number format as multiplication, division, square root and power are exact and do not include any rounding errors unless the result is outside the representable range. This property is equivalent to fixed-point numbers being exact on addition and subtraction, which, due to the logarithm, translates to multiplication and division for logfixs. We design an n-bit logfix format with one sign bit s, and in the exponent n_i signed integer bits followed by n_f fraction bits, such that $n = 1 + n_i + n_f$. The decoding of a logfix number is defined as

$$x = (-1)^{b_1} \cdot 2^k \tag{2.8}$$

with the exponent $k = i + f$ being a fixed-point number with n_i integer bits and n_f fraction bits.

The signed integer i is in $[-2^{n_i-1}, 2^{n_i-1}-1]$ (one bit is effectively used for the sign of the exponent) and uses a biased representation similar to floats: The integer bits are first interpreted as an unsigned integer u and converted to the signed integer i by $i = u - bias$, where $bias = 2^{n_i-1}$. For example, with $n_i = 5$ integer bits the bitpattern $u = 10001 = 17$ is converted as $i = u - bias = 17 - 2^4 = 1$, such that for $f = 0$ the logfix 01000100... represents the number 2. The fraction f is in $[0, 1)$ and identical to the mantissa encoding in floating-point numbers (but without the hidden bit) $f = \sum_{i=1}^{n_f} b_{1+n_i+i} 2^{-i}$.

Eq. 2.8 has only two exceptions: The bit pattern 00...00 is reserved to encode zero and 100...0 to encode Not-A-Real (NaR, which combines NaN with $\pm\infty$ into a single bitpattern and is borrowed from the design of posits, see section 2.1.5). Hence, $x = 0$ if $b_1 = k = 0$ and $x = \text{NaR}$ if $b_1 = 1$ but $k = 0$.

The trade-off between precision and range is for logfixs similar to floats when choosing a corresponding number of bits: The logfix format with $n_i = 5$ integer bits and $n_f = 10$ fraction bits is similar to Float16 with 5 exponent bits and 10 mantissa bits. Using $n_i = 8$ integer bits and $n_f = 7$ fraction bits is similar to BFloat16, such that we call these two logfix formats LogFixPoint16 and BLogFixPoint16 in analogy. The smallest representable positive number for logfixs is $minpos = 2^{-2^{n_i-1}+2^{-n_f}}$ which is encoded as 00...01. The largest representable number for logfixs is $maxpos = 2^{2^{n_i-1}-2^{-n_f}}$ encoded as 011...1, such that for our design of logfixs $maxpos = \frac{1}{minpos}$, meaning that the dynamic range is symmetric around 1. This is not the case for floats, as special cases like NaNs occupy part of the theoretically available dynamic range (see Eq. 2.6). Logfixs are therefore symmetric with respect to the multiplicative inverse around 1 and symmetric with respect to addition around 0. Floats are only symmetric with respect to addition.

Multiplication, division, square root and power are easily implemented for a logarithmic number format with integer arithmetic. Multiplication of two numbers x, y, for example, is equivalent to the addition of the exponents

$$xy = (-1)^{s_x \veebar s_y} \cdot 2^{k_x + k_y} \tag{2.9}$$

with $\veebar$ being the logical xor-operation on the sign bits s_x, s_y. Exceptions to Eq. 2.9 occur for $x = \text{NaR}$ or $y = \text{NaR}$ in which case $xy = \text{NaR}$, or if no NaRs are present but if $x = 0$ or $y = 0$ then $xy = 0$. The multiplication is then followed by checking for underflows ($xy < \frac{1}{2} minpos$) or overflows ($xy > maxpos$), in which case 0 or NaR is returned, respectively.

Addition and subtraction for logfixs is based on the Gaussian logarithms G^+, G^- which are functions of the argument a

$$\begin{aligned} G^+(a) &= -a + \log_2(1 + 2^a) \\ G^-(a) &= -a + \log_2(1 - 2^a) \end{aligned} \tag{2.10}$$

The evaluation of the Gaussian logarithms makes addition and subtraction for logfixs much more complicated than the other arithmetic operations. The result $z = (-1)^{s_z} \cdot 2^{k_z}$ of an ad-

dition $x + y = z$ is given by

$$\begin{aligned} k_z &= \log_2(|x| + |y|) \\ &= k_y - (k_y - k_x) + \log_2(1 + 2^{k_y - k_x}) = k_y + G^+(k_y - k_x) \end{aligned} \tag{2.11}$$

and similar for subtraction $y - x = z$

$$\begin{aligned} k_z &= \log_2(|y| - |x|) \\ &= k_y - (k_y - k_x) + \log_2(1 - 2^{k_y - k_x}) = k_y + G^-(k_y - k_x) \end{aligned} \tag{2.12}$$

for $y \geq x \geq 0$, which can be generalised to all x, y by rewriting $x + y$ as $y + x$ or $x - y = -(y - x)$, etc.

The result of the base-2 logarithm $\log_2$ inside the Gaussian logarithms has to be rounded to a fixed-point number like k_x and k_y for the subsequent addition. Consequently, for any finite-precision logfix format the Gaussian logarithms become

$$\begin{aligned} G^+(a) &= -a + \text{round}(\log_2(1 + 2^a)) \\ G^-(a) &= -a + \text{round}(\log_2(1 - 2^a)) \end{aligned} \tag{2.13}$$

which introduces a rounding error in general, meaning that addition and subtraction with logfixs are not exact. We will discuss rounding for logfixs in section 2.2.3, and define the specific rounding function round() therein.

Instead of evaluation the Gaussian logarithms on every arithmetic operation, they can be precomputed as only a function of the single argument $k_y - k_x$. For an n-bit logfix format there are only $2^{n-1}-1$ different values for $k_y - k_x$, such that in 16 bit the two precomputed table lookups have an upper bound of 32767 entries each. In practice, such table lookups are considerably smaller as for $k_y \gg k_x$ the functions G^+, G^- approach zero, as their increment on k_y is eventually round back to k_y. Consequently, there will be a maximum resolvable difference $k_y - k_x$ beyond which G^+ and G^- do not have to be computed. For the $n = 16$-bit format LogFixPoint16 with $n_f = 10$ fraction bits the two table lookups are each 24KB large (11805 entries at 16 bit per entry). For BLogFixPoint16 ($n_f = 7$ fraction bits) the two table lookups are even smaller at 2KB each (1092 entries at 16 bit per entry). While the small sizes of these table lookups are promising for low-precision hardware implementations, addition and subtraction remain the performance and precision bottleneck for logfix formats. The size of these table lookups increases exponentially with the number of bits, meaning that implementations for 32-bit or 64-bit logfixs cannot be based on table lookups. Such high-precision logfix formats will only be possible if the logarithms can be computed directly. Although the design of floats is mathematically more complicated (compare Eq. 2.6 to Eq. 2.8), they enjoy full hardware support, such that there's currently no competitive hardware based on logfix arithmetic available that could be used as an alternative to floats.

For a summary on the logfix formats LogFixPoint16 and BLogFixPoint16 see section 2.1.6

and Table 2.1.

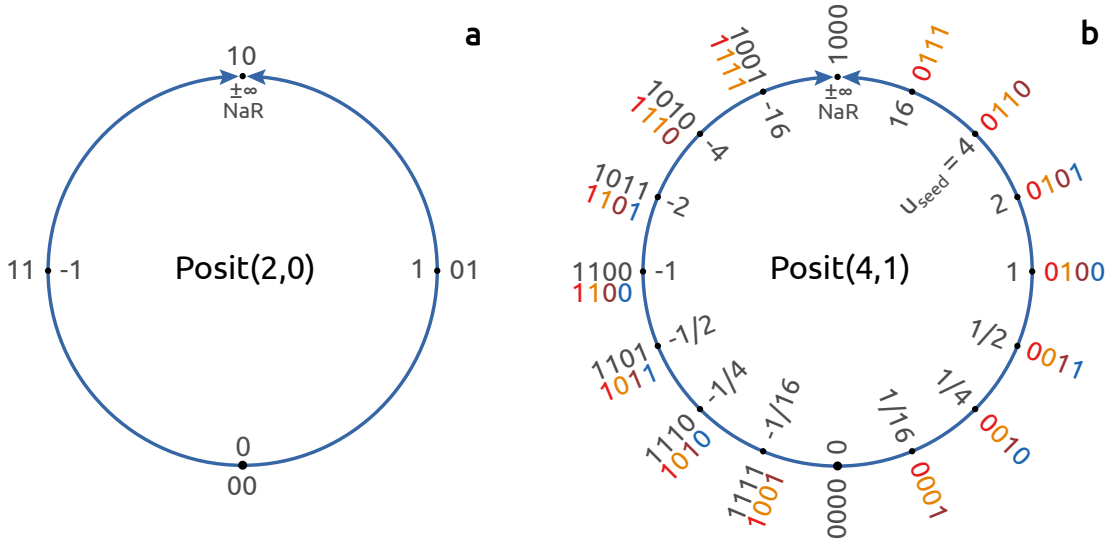

Figure 2.1 | Two posit number formats obtained by projecting the real axis onto a circle. **a** The 2-bit Posit(2,0) format and **b** the 4-bit Posit(4,1) format. The bitpatterns are marked on the outside and the respective values on the inside of each circle. Bit patterns of negative numbers (black) have to be converted to their two's complement (colours) first. At the top (12 o'clock) of every posit circle is Not-A-Real (NaR), also called complex infinity ($\pm\infty$), at the bottom (6 o'clock) is always 0, at 3 o'clock always 1 and at 9 o'clock always -1. After Gustafson & Yonemoto [2017].

2.1.5 Posit numbers

Posit numbers were introduced by Gustafson & Yonemoto [2017] as one of the universal number formats (Unums, Gustafson [2015]) and arise from a projection of the real axis onto a circle (Fig. 2.1). With only one bitpattern for 0 and one for Not-a-Real (NaR, or complex infinity $\pm\infty$, a replacement for Not-a-Number NaN), posits have a bijective mapping between all available bitpatterns and representable numbers. The circle is split into *regimes*, determined by a constant *useed*, which always marks the north-east on the posit circle (Fig. 2.1b). Regimes are defined by $useed^{\pm 1}$, $useed^{\pm 2}$, $useed^{\pm 3}$, etc. To encode these regimes into bits, posit numbers extend floating-point arithmetic by introducing regime bits that are responsible for the dynamic range of representable numbers. Instead of having a fixed length, regime bits are defined as the sequence of identical bits after the sign bit, which are eventually terminated by an opposite bit, e.g. 000001 or 1110, the last bit is the terminating bit respectively.

The flexible length allows the mantissa (or significand) to occupy more bits when fewer regime bits are needed, which is the case for numbers around 1 and -1 (in a logarithmic sense). More mantissa bits mean that the representable numbers are denser packed, resulting in a higher precision around ± 1. On the other hand, the regime bits occupy more bit positions for very small or very large numbers in an absolute sense, e.g. -10^{-10} or 10^{10}, which reduces the

number of available mantissa bits for those. Therefore, the higher precision around ± 1 for posits is traded against a gradually lower precision for very small or very large numbers. While floats have a constant number of mantissa bits such that the result of a calculation is always precise to a similar amount of significant digits, calculations with posits will be more precise when the result is around ± 1. Depending on the application, this can reduce rounding errors. For a quantitative comparison of the precision of the number formats see section 2.3.3.

A number is encoded into an n-bit posit with a sign bit s, n_r regime bits including the terminating bit, up to n_e exponent bits and n_m mantissa bits, such that $n = 1 + n_r + n_e + n_m$. A posit format is uniquely defined by the number of bits n and the number of exponent bits n_e and we call this format Posit(n,n_e). However, as illustrated in Fig. 2.1 for the format Posit(4,1) with one exponent bit, the regime bits can occupy as many bit positions such that the exponent bits are pushed beyond the available bit positions. In this case there are actually less than n_e exponent bits explicitly encoded, and the hidden exponent bits are implicitly assumed to be 0. The same holds for the mantissa bits, which are all implicitly 0 if no bit positions are available for the mantissa.

For positive numbers, decoding from posit bits into the number p is defined as [Chen *et al.*, 2018; Gustafson & Yonemoto, 2017; Klöwer *et al.*, 2019] (negative posits are converted first to their two's complement, see Eq. 2.16)

$$p = (-1)^s \cdot useed^k \cdot 2^e \cdot (1 + f) \tag{2.14}$$

where k is the number of regime bits, excluding the terminating bit. The exponent bits are interpreted as the unsigned integer e and f is the fraction which is encoded in the fraction (or mantissa) bits. The base $useed = 2^{2^{n_e}}$ is determined by the number of exponent bits n_e in a given format; e.g. for Posit(4,1) with $n_e = 1$ exponent bit we have $useed = 4$. More exponent bits increase - by increasing $useed$ - the dynamic range of representable numbers at the cost of precision, as increasing $useed$ also increases the spacing between representable numbers. The exponent bits themselves do not affect the dynamic range by changing the value of 2^e in Eq. 2.14. Instead, they fill gaps of powers of two spanned by $useed = 4, 16, 256, \ldots$ for $n_e = 1, 2, 3, \ldots$, such that the regime and exponent combined encodes every power of two over the range of representable numbers. Hence, every posit number can be written as $p = \pm 2^n \cdot (1 + f)$ with a given integer n [Chen *et al.*, 2018; Gustafson & Yonemoto, 2017]. The mantissa bits encode the fraction f following the floating-point standard [IEEE, 1985], see section 2.1.3, with the difference that the number of mantissa bits n_m can change.

An example for posit decoding is provided in the Posit(8,1)-system (i.e. $useed = 4$)

$$57 \approx 01110111_{\text{Posit(8,1)}} = (-1)^0 \cdot 4^2 \cdot 2^1 \cdot (1 + 2^{-1} + 2^{-2}) = 56 \tag{2.15}$$

The sign bit is given in red, regime bits in orange, the terminating regime bit in brown, the exponent bit in blue and the fraction bits in black. The k-value is inferred from the number of regime bits, that are counted as negative for the regime bits being 0 (which encodes numbers $|p| < 1$),

e.g. 0001 yields $k = -3$. For the regime bits being 1 they are counted as positive but subtract 1 (which encodes numbers $|p| \geq 1$), e.g. 1110 yields $k = 2$.

For negative numbers, i.e. the sign bit being $s = 1$, all other bits are first converted to their two's complement (flipping all bits and adding 1, see section 2.1.1 and Choo *et al.* [2003]), denoted here with an underscore subscript

$$\begin{aligned} -0.28 \approx 11011110_{\text{Posit(8,1)}} &= 10100010\_ \\ &= (-1)^1 \cdot 4^{-1} \cdot 2^0 \cdot (1 + 2^{-3}) = -0.28125. \end{aligned} \tag{2.16}$$

After the conversion to the two's complement, the bits are interpreted in the same way as in Eq. 2.15. Posits have two exceptions. The bitpattern 00...00 is reserved to represent 0 and the bitpattern 100...0 represents Not-a-Real (NaR). Many functions are simplified for posits, as only these two exceptions cases have to be handled. Conversely, Float64 has more than 10^{15} bitpatterns reserved for NaN. Although they only make up $< 0.05\%$ of all available bit patterns, the percentage of redundant bitpatterns for NaN increases for floats with fewer exponent bits (Table 2.1), and only poses a noticeable issue for Float16 and Float8.

The smallest representable number *minpos* for posits is encoded in the bitpattern 000...01 (Fig. 2.1b), with $n-2$ regime bits, such that $k = -(n-2)$ and $minpos = useed^k = 2^{-2^{n_e}(n-2)}$. Posits are symmetric with respect to the multiplicative inverse around 1 and therefore the largest representable number $maxpos = \frac{1}{minpos}$, which occurs for the bitpattern 011...1, with $n-1$ regime bits, such that $k = n-2$ and $maxpos = useed^k = 2^{2^{n_e}(n-2)}$, as required for symmetry. Posits also come with a no-overflow rounding mode, which will be discussed in more detail in section 2.2.2.

The proposed posit standard defines several posit number formats as a drop-in replacement for floats. Posit8 is the 8-bit posit format with 0 exponent bits (hence also called Posit(8,0)). Although 8-bit floats are not officially defined, the Float8 format presented in Table 2.1 has 3 exponent bits and has a similar range-precision trade-off as Posit8. Posit16 is the 16-bit format with 1 exponent bit, hence also called Posit(16,1) and is thought to be the alternative to Float16. Posit32 is the 32-bit posit format with 2 exponent bits, hence shortened from Posit(32,2), which is proposed to be the alternative to Float32. While a Posit64 format with 64 bits including 3 exponent bits is defined, we will not consider this format here. The following chapters will mostly investigate Posit8,Posit16 and Posit32, as in the proposed posit standard, but chapter 5 will additionally discuss Posit(16,2).

The posit number framework also highly recommends *quires*, an additional register on hardware to store intermediate results. Dot-product operations like $\mathbf{a} \cdot \mathbf{b} = a_1 b_1 + a_2 b_2 + ... + a_n b_n$ are fused with quire arithmetic and can therefore be executed with a single rounding error, which is only applied when converting back to posits. The quire concept could also be applied to floating-point arithmetic (fused multiply-add $ab + c$ is available on some processors), but is technically difficult to implement on hardware for a general dot-product as the required registers would need to be much larger in size. For fair comparison we do not take quires into account.

In order to use posits on a conventional CPU we developed for the programming language

Julia [Bezanson *et al.*, 2017] the posit emulator *SoftPosit.jl* [Klöwer & Giordano, 2019], which is a wrapper for the C-based library SoftPosit [Leong, 2020]. More details and examples on how to use different number formats interchangeably within the Julia programming language are provided in section 2.4.

2.1.6 Summary on number formats

Format	bits	exp bits	*minpos*	*maxpos*	dyn range	ϵ	% NaR
Float64	64	11	$5.0 \cdot 10^{-324}$	$1.8 \cdot 10^{308}$	631.6	16.3	0.05
Float32	32	8	$1.0 \cdot 10^{-45}$	$3.4 \cdot 10^{38}$	83.4	7.6	0.39
Float16	16	5	$6.0 \cdot 10^{-8}$	65504	12.0	3.7	3.13
BFloat16	16	8	$9.2 \cdot 10^{-41}$	$3.4 \cdot 10^{38}$	78.6	2.8	0.39
Float8	8	3	$1.5 \cdot 10^{-2}$	15.5	3.0	1.9	12.50
Posit32	32	2	$7.5 \cdot 10^{-37}$	$1.3 \cdot 10^{36}$	72.2	8.8	10^{-8}
Posit16	16	1	$3.7 \cdot 10^{-9}$	$2.7 \cdot 10^{8}$	16.9	4.3	10^{-3}
Posit(16,2)	16	2	$1.4 \cdot 10^{-17}$	$7.2 \cdot 10^{16}$	33.7	4.0	10^{-3}
Posit8	8	0	$1.6 \cdot 10^{-2}$	64	3.6	2.2	0.39
Int64	64	0	1	$9.2 \cdot 10^{18}$	19.0	0.8	0
Int16	16	0	1	32767	4.5	0.8	0
Q6.10	16	0	$9.8 \cdot 10^{-4}$	32.0	4.5	3.7	0
LogFixPoint16	16	5	$1.5 \cdot 10^{-5}$	65491.7	9.6	3.8	10^{-3}
BLogFixPoint16	16	8	$5.4 \cdot 10^{-20}$	$9.1 \cdot 10^{18}$	77.1	2.9	10^{-3}

Table 2.1 | Some characteristics of various number formats. The number of exponent bits is for logfixs the number of integer bits in the exponent (see section 2.1.4). *minpos* is the smallest representable positive number, *maxpos* the largest. The dynamic range is $\log_{10}(\frac{maxpos}{minpos})$. The machine precision ϵ is the decimal precision at 1 (see section 2.3.3), equivalent to the decimal places at least correct after rounding. % NaR denotes the approximate percentage of bit patterns that represent Not-A-Number (NaN), infinity or Not-A-Real (NaR).

Some characteristics of the presented binary number formats are summarized in Table 2.1. Integer formats are usually unsuited for scientific computing as no numbers between 0 and 1 can be represented. While fixed-point numbers can act as scaled integers, this lowers *maxpos* as the dynamic range $\log_{10}(\frac{maxpos}{minpos})$ does not change between signed integers and fixed-point numbers, and is comparably small. Floating-point numbers are therefore in general a much better choice for scientific computing as they provide a large dynamic range as a sufficiently high precision around 1. Virtually equivalent to floats in these characteristics are logarithmic fixed-point numbers, which also avoid a large share of NaRs in their design. Following these characteristics, posits are slightly superior to floats and logfixs with the same number of bits. A more in-depth comparison of these number formats is part of chapter 4 and 5.

2.2 Rounding modes

Representing a real number x in any finite-precision number format introduces, in general, a rounding error (or round-off error) as only an approximation to x can be represented in a given format. Depending on the distribution of the representable numbers on the real axis the rounding error will differ and in the case where x is exactly representable no rounding error occurs. Such rounding errors occur in the conversion between formats, but in general also after arithmetic operations within one format, including two argument operations such as addition and multiplication but also single-argument operation such as the square root. Rounding errors arise when the arithmetic result does not map again onto a representable number. For example, $1 + 1 = 2$ results in another integer that is again representable within the format, hence no rounding is applied. However, $5/4 = 1.2$ and the result is not representable as integer. Therefore rounding has to be applied to make the result representable again, introducing a rounding error. The set of rules that decide which representable number serves as approximation to the exact result is called the rounding mode.

Rounding a number in a given format usually makes its representation simpler by some measure. For example, rounding $\pi = 3.14159...$ to 3.1 reduces the number of decimal places required to represent an approximation to π. However, the representation is usually only simplified in the format where the rounding is applied in and does not translate to other formats: While the approximation 3.1 of π requires fewer decimal places (meaning that trailing decimal places are set to zero) its representation in binary does not set any number of trailing bits to zero, as seen in the following

$$\begin{aligned}
\text{Float32:} &\quad 01000000010010010000111111011011 = 3.1415927 \\
\text{Float32:} &\quad 01000000010001100110011001100110 = 3.1 \\
\text{BFloat16:} &\quad 0100000001001001 = 3.140625
\end{aligned} \tag{2.17}$$

As shown in this example, rounding in decimal does not round in Float32. However, rounding in binary from Float32 to BFloat16 sets trailing bits to zero, but does not considerably reduce the decimal places. In the following we will present several *binary* rounding modes, meaning that the binary encoding of an exact value within a given format is replaced by shorter (or at least simpler due to more trailing zeros) bit sequence, representing an approximation to the exact value.

2.2.1 Bitshaving, bitsetting and halfshave

Bitshave is one of the simplest binary rounding modes that *shaves* off a given amount of trailing bits and sets them to 0 [Kouznetsov, 2020; Zender, 2016]. This is the binary version of *round-to-zero*, which rounds down for positive values, but rounds up for negative values. For an exact

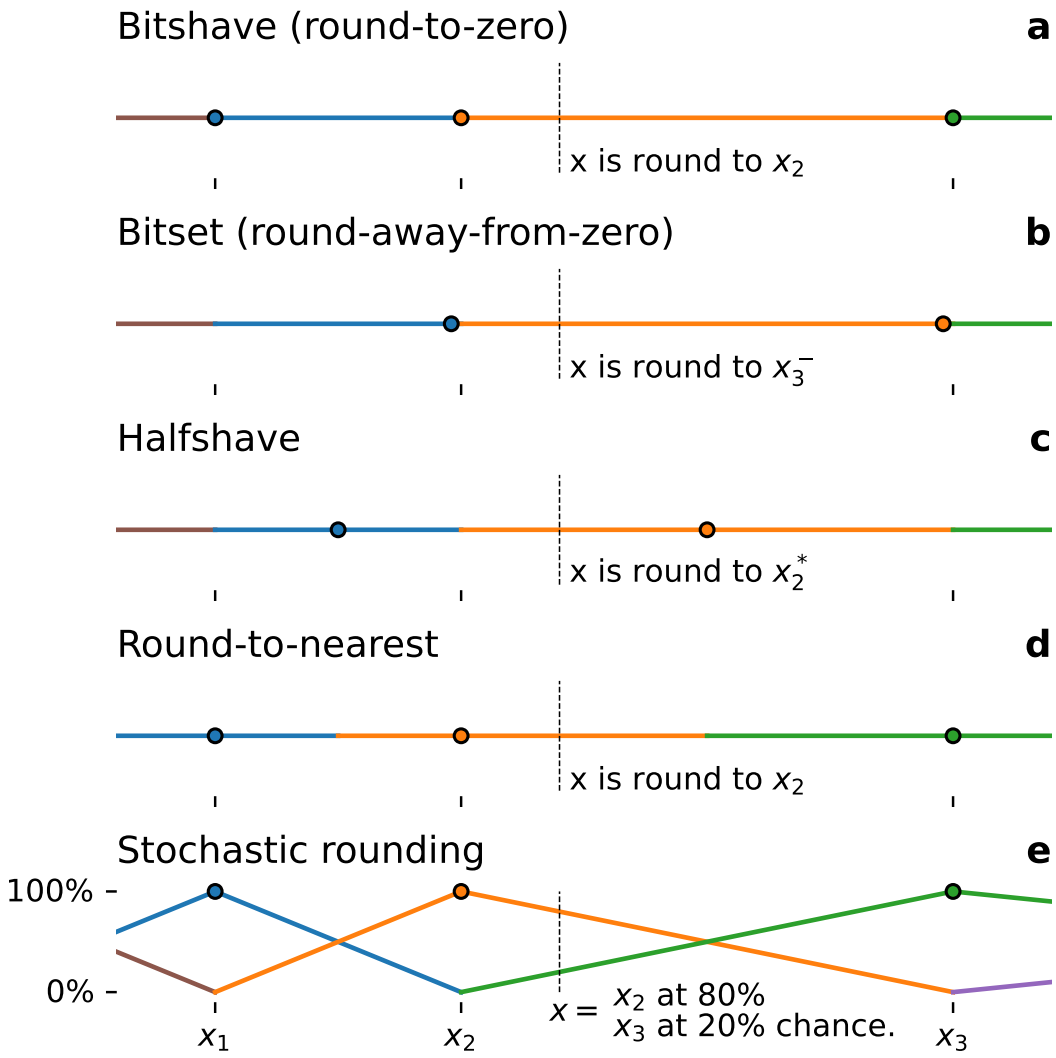

Figure 2.2 | Schematic to illustrate some rounding modes. **a** Bitshave rounds the exact value x, indicated by a dashed line, down to the next smaller or equal representable number x_2 (all assumed positive). **b** Bitset rounds x up to $x_3^- = x_3 - \epsilon$, i.e. just smaller than x_3 by machine precision ϵ of the exact representation (see Eq. 2.20). **c** Halfshave rounds x to $x_2^* = \frac{x_2+x_3}{2}$ which is half-way between the representable numbers x_2, x_3. **d** Round-to-nearest rounds x to the nearest representable number, x_2 here. Nearest is hereby defined as the smallest linear distance $|x - x_i|$ for all representable numbers x_i. **e** Stochastic rounding rounds x up or down at a chance that is proportional to the distance of x from its adjacent representable numbers. Here, x is located one fifth the way between x_2 and x_3, hence it is rounded down to x_2 in $1 - \frac{1}{5} = 80\%$ of the cases, but rounded up to x_3 at $\frac{1}{5} = 20\%$ chance. The rounding modes **a**-**d** are deterministic. Line segments and representable numbers (circles) are colour-coded, such that all numbers on a line segment of a given colour are rounded to the representable number with the same colour.

value x in between two representable numbers x_1, x_2 round-to-zero is

$$\text{round}_{\text{to-zero}}(x) = \begin{cases} x_1, & \text{if } x \geq 0 \text{ and } x_1 \leq x < x_2 \\ x_2, & \text{if } x < 0 \text{ and } x_1 < x \leq x_2 \end{cases} \tag{2.18}$$

For an illustration of the round-to-zero see Fig. 2.2a. Bitshave achieves this behaviour by setting trailing mantissa bits to 0, which reduces the fraction f in the mantissa (Eq. 2.4). For example, bitshaving π in Float32 to 7 mantissa bits yields (rounded bits in grey)

$$\begin{aligned} \text{Float32:} \quad & 01000000010010010000111111011011 = 3.1415927 \\ \text{Float32, bitshaved:} \quad & 01000000010010010000000000000000 = 3.140625 \end{aligned} \tag{2.19}$$

This rounding mode therefore introduces a worst-case error that is equivalent to $\delta = x_2 - x_1$, the distance between the two representable numbers. The worst-case error occurs if the trailing bits that are subject to rounding are all 1, which means that the exact value x is just smaller (for positive x) than a representable number. For example, bitshaving $x = 0111$ to one leading bit yields $x_1 = 0000$, although the next larger representable number $x_2 = 1000$ is closer to x.

Bitshave is a rounding mode with a bias towards zero: For positive numbers, it will always round to values that are smaller or the same, therefore it is expected to reduce the mean. For numbers of either sign, bitshave will reduce the variance, as it will always round to values that are closer to zero or the same.

Bitset

An alternative to bitshaving is bitsetting [Zender, 2016], which sets the trailing bits to 1 instead. Following the same example as above (Fig. 2.2b)

$$\begin{aligned} \text{Float32:} \quad & 01000000010010010000111111011011 = 3.1415927 \\ \text{Float32, bitset:} \quad & 01000000010010011111111111111111 = 3.1562498 \end{aligned} \tag{2.20}$$

Different to bitshave, bitsetting has a bias away from zero and will increase the variance following a similar argumentation. Furthermore, although the trailing bits are simpler because they are all 1, they still have to be explicitly stored to preserve the rounded value. So while bitsetting is a valid binary rounding mode, it does not provide any storage benefits and will need to be combined with methods like lossless compression (see section **??**) to actually yield rounded values that require fewer bits in memory.

Halfshave

To remove the biases from bitshaving and bitsetting, the binary rounding mode halfshave is proposed [Kouznetsov, 2020; Zender, 2016]. This rounding mode is similar to bitshaving, but

replaces the shaved bits with 1000... instead of 0000.... Halfshave therefore replaces the representable numbers $x_1, x_2, x_3, ...$ that could occur from bitshaving with the respective mid-points. Halfshaved representable numbers are therefore $x_1^* = \frac{x_1+x_2}{2}, x_2^* = \frac{x_2+x_3}{2}, ...$ and require one more bit (which is always 1) to be stored explicitly (which can be subject to lossless compression due to redundancy, see section 3.3.10). Following the same example as above

$$\begin{aligned} \text{Float32:} \quad & 01000000010010010000111111011011 = 3.1415927 \\ \text{Float32, bitset:} \quad & 01000000010010011000000000000000 = 3.1484375 \end{aligned} \tag{2.21}$$

where the trailing bit that is set to 1 is marked in green.

Halfshave can either round up or down (Fig. 2.2c) and in most cases rounds to the nearest representable number, which largely removes the positive/negative and to/away-from-zero bias. However, the bias is not fully removed as halfshave does not always round to the nearest representable number: For non-equidistantly spaced representable numbers $x_1, x_2, x_3, ...$ the ties, i.e. those numbers that lie exactly in between two representable numbers, of the representable numbers with halfshave $x_1^*, x_2^*, x_3^*, ...$ are again $x_1, x_2, x_3, ...$ (see Fig. 2.2c) but not the mid-points of $x_1^*, x_2^*, x_3^*, ...$ Hence, there will be some numbers that are rounded away from their nearest representable number. For example, if $x_1 = 0.5, x_2 = 1, x_3 = 2, x_4 = 3$ then $x_1^* = 0.75, x_2^* = 1.5, x_3^* = 2.5$ and the number $x = 1 = 00111111100...0$ in Float32 will be rounded up to $x_2^* = 1.5 = 001111111100...0$ with halfshave although x_1^* is closer ($|x - x_1^*| = 0.25$ versus $|x - x_2^*| = 0.5$).

Furthermore, even for equidistant spacing, ties are always rounded away from zero with halfshave: $x = 2$ is exactly in between $x_2^* = 1.5$ and $x_3^* = 2.5$, however, its binary representation is even (it ends in a 0 bit) as all x_i^* end in a 1 bit, which means they are odd. Due to the ties being even, halfshave will always round them away from zero. Hence, there is an away-from-zero bias for the ties with halfshave. Such a bias is much smaller than the biases discussed for bitshave and bitset, but depending on the frequency of ties being round can be significant. One possibility to overcome the remaining biases with halfshave is round-to-nearest, which is discussed in the next section.

2.2.2 Round-to-nearest

The default rounding mode for floats and posits is round-to-nearest tie-to-even [IEEE, 1985]. In this rounding mode an exact result x is rounded to the nearest representable number x_i. Nearest is hereby defined as the L_1 distance $|x - x_i|$ (alternatives, such as round-to-nearest in logarithmic space are discussed in section 2.2.3). In case x is a tie, i.e. exactly half-way between two representable numbers, it will be rounded to the nearest even. A floating-point number x_i is considered to be even, if its mantissa ends in a zero bit, e.g. 0110 is even, while 0011 is odd. This rule will round one tie down, but the next one up, which removes a bias that otherwise persists (see halfshave, section 2.2.1).

Let x_1 and x_2 be the closest two representable numbers to $x \geq 0$ and $x_1 \leq x < x_2$ (for $x < 0$

this changes to $x_1 < x \leq x_2$) then

$$\text{round}_{\text{to-nearest}}(x) = \begin{cases} x_1 & \text{if } x - x_1 < x_2 - x, \\ x_1 & \text{if } x - x_1 = x_2 - x \text{ and } x1 \text{ even}, \\ x_2 & \text{else}. \end{cases} \tag{2.22}$$

and an example can be seen in Fig. 2.2d.

In binary this rounding mode always sets all trailing bits to 0 (as with bitshave), but depending on the case in Eq. 2.22, may introduce a carry bit that carries into the unrounded bits for round up to always obtain the nearest representable number. Some examples follow using 8 mantissa bits that are rounded to 4, i.e. the representable numbers are `0000`, `0001`, ..., `1111` with lesser significant mantissa bits implied to be 0. The function round(x) implements round-to-nearest tie-to-even in these examples

$$\begin{aligned}
&\text{Example 1: } \text{round}(01000110) \\
&\qquad\qquad = 01000000 \qquad \text{(round down)} \\
&\text{Example 2: } \text{round}(01001110) \\
&\qquad\qquad = 01010000 \qquad \text{(round up)} \\
&\text{Example 3: } \text{round}(01111010) \\
&\qquad\qquad = 10000000 \qquad \text{(round up)} \\
&\text{Example 4: } \text{round}(11101000) \\
&\qquad\qquad = 11100000 \qquad \text{(round down, tie to even)} \\
&\text{Example 5: } \text{round}(00011000) \\
&\qquad\qquad = 00100000 \qquad \text{(round up, tie to even)}
\end{aligned} \tag{2.23}$$

In the example 1 $x =$ `01000110` is closer to `01000000` than to `01010000`, the next larger representable number, hence round down is applied. In example 2 we round up and the carry bit flips the least significant mantissa that is left after rounding, marked in orange. In example 3, the carry bit can affect several otherwise unrounded bits; flipping them all to obtain the nearest representable number. In example 4, $x =$ `11101000` is a tie, as it is exactly half-way between `11100000` and `11110000`. However, x is rounded to the former which is the even representable number of the two. In example 5, $x =$ `00011000` is also a tie, but this time the nearest even representable number is larger, such that x is rounded up.

Round-to-nearest is considered to be a bias-free rounding mode, as it does not have a positive/negative bias, nor a towards-zero/away-from zero bias. These properties also hold for the ties (which is not the case for halfshave, see section 2.2.1). However, some issues remain, which will be discussed in comparison to stochastic rounding (section 2.2.4).

Of interest is also the behaviour of rounding modes with respect to underflow (i.e. the absolute value of the result x is smaller than *minpos*, $|x| <$ *minpos*) and overflow (larger than *maxpos*,

$|x| > maxpos$). Floats underflow if the exact result x is closer to 0 than to $minpos$, i.e. $\text{round}(x) = 0$ if $0 \leq |x| \leq \frac{minpos}{2}$, which is no exception from Eq. 2.22 and the tie-to-even rule applies. Floats overflow and return infinity if $|x| > maxpos$ (and similar for negative numbers, $\text{round}(x) = -\infty$ if $x < -maxpos$) which is an exception to Eq. 2.22. The overflow is intended to flag arithmetic operations with results that are beyond representable and therefore likely include a large rounding error. Infinities propagate through arithmetic operations and algorithms return either infinity or NaN (e.g. $\frac{\infty}{\infty} = \text{NaN}$ in float arithmetic).

While posit arithmetic also implements the same round-to-nearest rounding mode, posits come with a no-overflow rounding mode, such that $\text{round}(x) = maxpos$ if $x > maxpos$ (and similar for $x < -maxpos$ with $\text{round}(x) = -maxpos$). This is motivated as rounding to infinity returns a result that is infinitely less correct than $maxpos$. Although an overflow is therefore not easily signalled within an algorithm, it is proposed to perform overflow-like checks on the software level to simplify exception handling on hardware [Gustafson & Yonemoto, 2017]. Whether this leads to increased performance in hardware implementations is unclear, but circuits will be simplified.

2.2.3 Round-to-nearest for logarithmic fixed-point numbers

For a logarithmic-fixed point number format rounding to the nearest representable number is applied to the fraction bits in logarithmic space (see Eq. 2.13) and not in linear space, which is the default for floating-point numbers. The tie between round down and round up is therefore not the midpoint $\frac{x_i + x_{i+1}}{2}$ with two representable numbers x_i, x_{i+1}, but the midpoint in logarithmic space

$$2^{\frac{\log_2(x_i)+\log_2(x_{i+1})}{2}} = \sqrt{x_i x_{i+1}} \tag{2.24}$$

Round-to-nearest in linear space for logfixs therefore has to be derived.

Let a be a positive logarithmic fixed-point number and $r = 2^{n_f}$ the scale that, when multiplied to the exponent k of a, yields an integer $m = rk = r \log_2(a)$. The next larger representable logfix b is then $m + 1 = r \log_2(b)$. The function round() in Eq. 2.13 should be chosen such that the midpoint $\frac{a+b}{2}$ lies exactly at $m + \frac{1}{2}$ in logarithmic space to serve as the tie and satisfy round-to-nearest in linear space. Adding a constant c_r before rounding achieves this with

$$c_r = \frac{1}{2} - r(\log_2(\sqrt[r]{2} + 1) - 1) \tag{2.25}$$

The value of c_r only changes with the number of fraction bits n_f and is therefore constant for a specific logfix format. The conversion of an exact number x to its nearest representable logfix $x_1 = 2^k$, described by the scaled-to-integer exponent rk, is then

$$rk = \text{round}_{\text{Int}}(c_r + r \log_2(|x|)) \tag{2.26}$$

Where $\text{round}_{\text{Int}}()$ rounds to the nearest integer including tie-to-even, which therefore also preserves this tie rule for logfixs. Similarly, c_r is included in the Gaussian logarithms G^+, G^- (Eq. 2.13), now written with the scaled-to-integer difference $m = r(k_y - k_x)$

$$\begin{aligned} G^+(m) &= -m + \text{round}_{\text{Int}}(c_r + r \log 2(1 + 2^{\frac{m}{r}})) \\ G^-(m) &= -m + \text{round}_{\text{Int}}(c_r + r \log 2(1 - 2^{\frac{m}{r}})) \end{aligned} \tag{2.27}$$

In case round-to-nearest in logarithmic space is preferred, one simply sets $c_r = 0$ as $\text{round}_{\text{Int}}()$ rounds to the nearest integer within the logarithm. The difference between rounding in linear and logarithmic space is illustrated in Fig. 2.3.

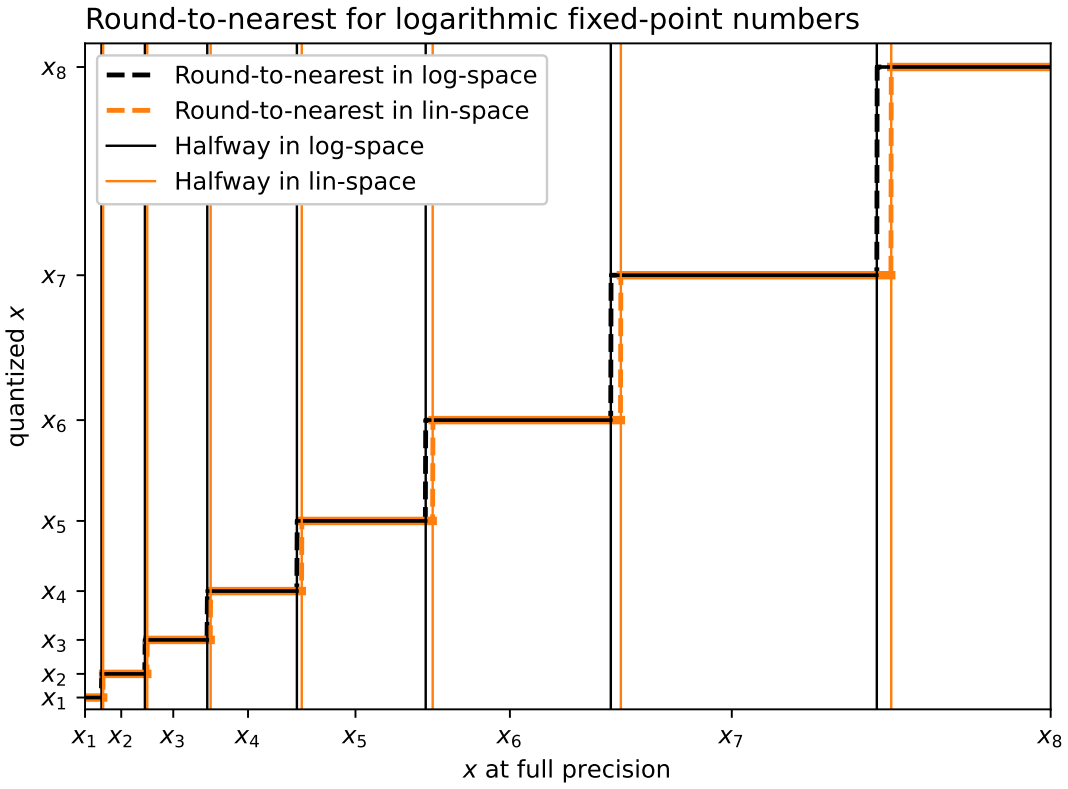

Figure 2.3 | Schematic to illustrate round-to-nearest in linear and logarithmic space for logarithmic fixed-point numbers. Exact values x at full precision are presented on the x-axis and their rounding to the logarithmically spaced representable numbers $x_1, \ldots, x_8$ on the y-axis. Round-to-nearest in linear space will use the midpoints $\frac{x_i + x_{i+1}}{2}$ (arithmetic mean) as ties, round-to-nearest in logarithmic space uses the geometric mean $\sqrt{x_i x_{i+1}}$.

The rounding mode for logfixs is completed with an underflow at half the smallest representable number, as for floats and posits (see section 2.2.2) and no overflow, where $\pm maxpos$ is returned instead as for posits.

2.2.4 Stochastic rounding

The previously presented rounding modes are deterministic, such that a given exact value x they always return the same rounded value. This discards information contained in x and two distinct exact values x, y rounded to the same value cannot be distinguished anymore. In the following we will present one stochastic rounding mode that can preserve some of this information at least in a statistical sense.

For stochastic rounding, rounding of x down to a representable number x_1 or up to x_2 occurs at probabilities that are proportional to the distance between x and x_1, x_2, respectively [Croci & Giles, 2020; Gupta *et al.*, 2015; Höhfeld & Fahlman, 1992; Mikaitis, 2020; Muller & Indiveri, 2015]. Let δ be the distance between x_1, x_2, $\delta = |x_2 - x_1|$ then

$$\text{round}_{\text{stoch}}(x) = \begin{cases} x_1 & \text{with probability} \quad 1 - \delta^{-1}(x - x_1) \\ x_2 & \text{with probability} \quad \delta^{-1}(x - x_1). \end{cases} \tag{2.28}$$

This behaviour is illustrated in Fig. 2.2. In case that x is already identical with a representable number no rounding is applied and the chance to obtain another representable number is zero. In that sense stochastic rounding preserves the idempotence of the other deterministic rounding modes, i.e. $\text{round}_{\text{stoch}}(\text{round}_{\text{stoch}}(x)) = \text{round}_{\text{stoch}}(x)$.

For x being half way between two representable numbers, the chance of round up or round down is 50%. The introduced absolute rounding error for stochastic rounding is always at least as large as for round-to-nearest and is larger when rounding away from the nearest representable number occurs. For x just larger than x_1, the probability that x is rounded to x_2 is non-zero and the absolute rounding error can then be up to $\pm\delta$, whereas for round-to-nearest the error is bounded by $\pm\frac{\delta}{2}$. Although the average absolute rounding error is therefore larger for stochastic rounding, the expected rounding error decreases towards zero for repeated roundings

$$\lim_{N\to\infty} \frac{1}{N} \sum_{i}^{N} \text{round}_{\text{stoch}}(x) = x \tag{2.29}$$

as follows by inserting Eq. 2.28. Stochastic rounding is therefore exact in expectation. There are other stochastic rounding modes, e.g. where the probability in Eq. 2.28 is $\frac{1}{2}$ in both cases, which do not have this property. We will only discuss exact-in-expectation stochastic rounding in the following.

2.2.5 Efficient bitwise implementations

The rounding modes from section 2.2.1, 2.2.2 and 2.2.4 are implemented in binary using bitwise operations. The bitwise logical operations used are logical and (&), or (|), and exclusive or (xor, $\veebar$). Additionally, we will use the *logical* bitshift operators $<< n$ and $>> n$ that shift the bits by n bit

positions, push in 0s and the bits pushed out are lost. For example, $01001011 >> 3 = 00001001$ and $11011010 << 1 = 10110100$. These are the default bitshift operations for unsigned integers, however, for signed integers the right-bitshift operation $>>$ is an *arithmetic* bitshift, which pushes in bits according to the bit in the first bit position. For example, $1000 >> 1 = 1100$ and $0111 >> 1 = 0011$. This is motivated to preserve the sign of negative integers (see section 2.1.1, e.g. -8 >> 1 = -4 and 8 >> 1 = 4) and will be made use of for the bitwise implementation of stochastic rounding. The algorithms presented here are implemented in a more general form in the open-source packages BitInformation.jl and StochasticRounding.jl.

```
function bitshave(x::Float32)
    ui = reinterpret(UInt32,x)
    ui &= 0xffff_0000       # set last 16 bits to 0
    return reinterpret(Float32,ui)
end

function bitset(x::Float32)
    ui = reinterpret(UInt32,x)
    ui |= 0x0000_ffff       # set last 16 bits to 1
    return reinterpret(Float32,ui)
end

function halfshave(x::Float32)
    ui = reinterpret(UInt32,x)
    ui &= 0xffff_0000       # 1. shave last 16 bits to 1
    ui |= 0x0000_8000       # 2. set most significant trailbit to 1
    return reinterpret(Float32,ui)
end
```

Listing 2.1 | Bitwise implementation of bitshave, bitset and halfshave for rounding to 7 mantissa bits. Bitshave rounds a Float32 x to $n_m = 7$ mantissa bits (equivalent to a BFloat16, but the trailing bits remain stored) using the bitwise logical-and operation &. For other numbers of mantissa bits to be retained the mask `0xffff_0000` has to be changed, e.g. `0xffff_e000` will round to $n_m = 10$ mantissa bits. The function `reinterpret` leaves the bits unchanged but changes the associated type. Bitset is implemented similar to bitshave but with the logical-or operation | which sets the trailing bits to 1. Halfshave is similar to bitshave but the trailing bits are replaced by 100...0.

Bitshave, bitset and halfshave are easily implemented with the bitwise logical operations, given here as an example for rounding from Float32 to BFloat16 (but preserving the trailing bits, which would not be explicitly stored in most cases, Listing 2.1). Round-to-nearest is more complicated to implement as the cases from Eq. 2.22 are implemented without if-clauses, efficiently using bitwise operations (Listing 2.2). This algorithm is adopted from the open-source package BFloat16s.jl.

The general idea of implementing round-to-nearest is to add $\frac{\delta}{2}$ (half the distance δ between two representable numbers) before round-to-zero. Ignoring the tie-to-even rule, Eq. 2.22 can also be written as $\text{round}_{\text{nearest}}(x) = \text{round}_{\text{to-zero}}(x + \frac{\delta}{2})$. To account for the tie-to-even rule, one has

```
1  function round_nearest(x::Float32)
2      isnan(x) && return NaN32   # do not round NaNs
3      ui = reinterpret(UInt32, x)
4      # 1. check for even/odd of bit 16 with >> 16 & 1
5      # 2. add either 0x7fff (even) or 0x8000 (odd) = +eps/2 with tie-to-even
6      ui += 0x0000_7fff + ((ui >> 16) & 0x0000_0001)
7      # 3. set trailing bits to 0 for rounding
8      ui &= 0xffff_0000
9      return reinterpret(Float32,ui)
10 end
```

Listing 2.2 | Efficient round-to-nearest implemented with integer arithmetic. Example algorithm rounding Float32 to 7 mantissa bits as in Listing 2.1. Algorithm adapted from the open-source package BFloat16s.jl

to add just less than $\frac{\delta}{2}$ if $\text{round}_{\text{to-zero}}(x)$ even. Otherwise, adding $\frac{\delta}{2}$ to the tie will round up to the odd representable number, violating tie-to-even. In binary, using rounding Float32 to 7 mantissa bits as example, this is achieved by checking the state of the least significant preserved bit. If $b_{16} = 1$ (bit 16, the 7th mantissa bit for Float32) then add `0x0000_8000`, which corresponds to $\frac{\delta}{2}$, otherwise add `0x0000_7fff`, which is just less than $\frac{\delta}{2}$ (see line 6 in Listing 2.2). Round-to-zero is then achieved as with bitshave in Listing 2.1, line 3). Note that the algorithm presented here uses integer arithmetic, such that floating-point numbers are rounded without using floating-point arithmetic. We will take a similar approach for the implementation of stochastic rounding, which is greatly simplified using integer arithmetic as discussed next.

Stochastic rounding

Implementing stochastic rounding directly from Eq. 2.28 requires the calculation of probabilities. It is more efficient however, to stochastically perturb the exact value x by adding a random number to it and then to deterministically round the result. However, implementing this idea with floating-point arithmetic requires the random perturbation to be correctly scaled depending on the value x itself. This involves several steps using floating-point arithmetic, which will slow down the stochastic rounding compared to integer arithmetic-based implementations. We present an algorithm that follows the same idea of randomly perturbing the exact value x, but in integer arithmetic which greatly simplifies stochastic rounding and avoids if-branching.

While floats have a hybrid logarithmic-uniform distribution on the real axis, floats are uniformly distributed when interpreted as (signed) integers. Hence, in line 3 of Listing 2.3 it is only necessary to arithmetic add random bits using integer arithmetic, no further scaling is required. This creates a random perturbation in $[0, \delta)$ followed by a round to zero, implemented using bitshave through a bitshift.

For example, an exactly representable value x is perturbed by less than δ, such that it cannot be larger than the next larger representable number. Bitshave then removes all random bits and

```julia
function stochastic_round(x::Float32)
    ui = reinterpret(UInt32,x)
    ui += rand(UInt16)    # add random perturbation in [0,delta)
    return reinterpret(BFloat16,(ui >> 16) % UInt16)    # bitshave
end
```

Listing 2.3 | Efficient bitwise stochastic rounding with integer arithmetic as implemented by StochasticRounding.jl. Example algorithm rounding Float32 to $n_f = 7$ mantissa bits as in Listing 2.1. `rand(UInt16)` creates 16 random bits with equal probabilities that are interpreted as 16-bit unsigned integer. Any pseudo random number generator can be used. To round the stochastically perturbed value, the bitshave operation is applied using $>> 16$, which shifts the trailing bits off the register.

x is always rounded to x as the probability to round away is 0. For x being a tie, i.e. half-way between two representable numbers, the perturbation returns at probability $\frac{1}{2}$ a value that is smaller than the next larger representable number, in which case bitshave rounds down. If the perturbation is larger, bitshave will round to the next larger representable number, such that the probability of x being round up or down is equal and $\frac{1}{2}$ in both cases.

2.3 Error norms

To assess the impact of rounding errors, the absolute, mean, relative, and decimal error are presented in the following. The decimal error can be translated to the decimal precision, which is used to quantify and compare the precision of the number formats from section 2.1.

2.3.1 Absolute and mean error

The absolute, or linear, error of an exact value x and its approximation $\hat{x}$ (which is for example a representable number of a finite-precision number format) is

$$\text{absolute error} = |x - \hat{x}| \tag{2.30}$$

and therefore the L_1 distance between x and $\hat{x}$. The absolute error has the unit of $x, \hat{x}$. The absolute error is invariant under addition, $|(x+c)-(\hat{x}+c)| = |x-\hat{x}|$, such that using either Kelvin or degree Celsius yields the same absolute error. However, the absolute error is not invariant under scaling, such that using millimeter or meter will change also the unit of the absolute error, $s^{-1}|x - \hat{x}|$ with scaling s, $x_s = sx, \hat{x}_s = s\hat{x}$. Assessing the absolute error of two arrays of numbers the exact ones $X = (x_1, x_2, ..., x_n)$ and the respective approximations $\hat{X} = (\hat{x}_1, \hat{x}_2, ..., \hat{x}_n)$ yields n absolute errors, hence an error distribution. The mean of such a distribution is the mean absolute error

$$\text{mean absolute error} = \frac{1}{n}\sum_{i=1}^{n} |x_i - \hat{x}_i| \tag{2.31}$$

and similarly the median, maximum or any other statistical property can be calculated.

In contrast, the mean error assesses the error in the respective means of the two arrays

$$\text{mean error} = \frac{1}{n}\sum_{i=1}^{n}\hat{x}_i - \frac{1}{n}\sum_{i=1}^{n}x_i = \frac{1}{n}\sum_{i=1}^{n}\hat{x}_i - x_i \tag{2.32}$$

which can be both positive or negative. For example with $x_i \geq 0$ and $\hat{x}_i$ obtained via round-to-zero there is a negative mean error (i.e. bias) as the mean after rounding is smaller than before. For round-away-from-zero the bias is positive.

2.3.2 Relative error

The relative error assesses the absolute error relative to the exact value. It is defined as

$$\text{relative error} = \left|\frac{x-\hat{x}}{x}\right| \tag{2.33}$$

and therefore dimensionless and invariant under scaling. The relative error does not discriminate sign changes, for example the relative error for $x = 1$ and $\hat{x} = 3$ or $\hat{x} = -1$ is identical: $|1-3| = 2 = |1-(-1)|$. It is therefore a less suitable error metric in situations where non-negative variables could be subject to sign changes in approximation. For example when approximating non-negative variables like concentrations or precipitation with a negative value, an error metric that returns infinity for sign changes might be more suitable. This issue is better addressed with the decimal error, which is discussed next.

2.3.3 Decimal error and precision

The decimal error is here defined as

$$\text{decimal error} = \begin{cases} 0 & \text{for } x = \hat{x} = 0 \\ \infty & \text{for } \operatorname{sign}(x) \neq \operatorname{sign}(\hat{x}) \\ \left|\log_{10}\left(\frac{x}{\hat{x}}\right)\right| & \text{else.} \end{cases} \tag{2.34}$$

and therefore is the absolute error in logarithmic space $|\log_{10}(\frac{x}{\hat{x}})| = |\log_{10}(x) - \log_{10}(\hat{x})|$ unless $x = \hat{x} = 0$. The decimal error quantifies the orders of magnitude an approximation $\hat{x}$ is away from the exact value x, for example $x = 1$ and $\hat{x} = 10$ yields a decimal error of 1, and so does $\hat{x} = \frac{1}{10}$. Based on the decimal error the decimal precision is defined for $\operatorname{sign}(x) = \operatorname{sign}(\hat{x})$ and $x \neq 0, \hat{x} \neq 0$ as [Gustafson & Yonemoto, 2017; Klöwer *et al.*, 2019]

$$\text{decimal precision} = -\log_{10}\left|\log_{10}\left(\frac{x}{\hat{x}}\right)\right| \tag{2.35}$$

The decimal precision quantifies the number of decimal places that are correct when using $\hat{x}$ as an approximation to x. For example, $x = \pi$ and $\hat{x} = 3.14$ yields a decimal precision of 3.66..., as

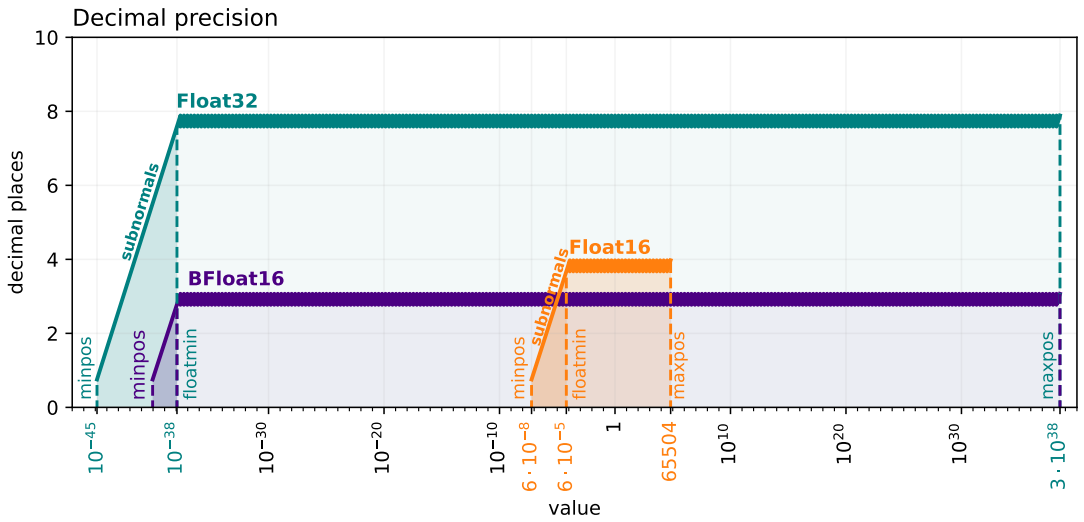

Figure 2.4 | Decimal precision of Float16, BFloat16 and Float32 over the range of representable numbers. The decimal precision is worst-case, i.e. given in terms of decimal places that are at least correct after rounding (see section 2.3.3). The smallest representable number (*minpos*), the smallest normal number (*floatmin*) and the largest representable number (*maxpos*) are denoted with vertical dashed lines (see section 2.1.3). The subnormal range is between *minpos* and *floatmin*.

the three explicit decimal digits 3, 1, 4 agree and the following implicit 0 digits in $\hat{x} = 3.1400...$ are close to the exact digits 159... in 3.14159...

The decimal precision approaches infinity when the approximation $\hat{x}$ approaches the exact result x (where the decimal error drops to 0). For the round-to-nearest rounding mode, the decimal precision has a minimum at the geometric mean $\sqrt{\hat{x}_1 \hat{x}_2}$ in between two representable numbers $\hat{x}_1, \hat{x}_2$. This minimum defines the *worst-case* decimal precision, i.e. the decimal precision when the rounding error is maximised. The worst-case decimal precision is the number of decimal places that are at least correct after rounding [Gustafson & Yonemoto, 2017]. So while the decimal precision of a given number format over the whole range of representable numbers is of interest, it is dependent on the exact location of representable numbers, not just the spacing in between. In contrast, the worst-case decimal precision over the whole range of representable numbers is a more useful metric to assess the provided precision of a number format. In the following we will refer to the worst-case decimal precision simply as decimal precision.

Fig. 2.4 compares the decimal precision for the three floating-point formats Float32, Float16 and BFloat16. Due to the exponent bits, the decimal precision is approximately constant over the normal range of floats for each format. It is not exactly constant as the mantissa bits are uniformly distributed between two powers of 2 (Eq. 2.4), hence the decimal precision for these increases towards the larger of the two powers of 2. This increase causes a sawtooth wave with a wavelength that is exactly the distance between two consecutive powers of two. The minimum of this sawtooth wave is reached when the spacing doubles between floats larger than the next

power of 2. A zoomed-in version of Fig. 2.4 is Fig. 2.5, which will later discuss the decimal precision of other formats.

Subnormal floats are uniformly distributed, such that their decimal precision drops towards 0 between *floatmin* and *minpos*. Float32 has a decimal precision of at least 7.6 decimal places over more than 80 orders of magnitude (Table 2.1). Due to only 7 mantissa bits with BFloat16, this format has a much lower decimal precision of 2.8 decimal places. One can therefore not expect more than 3 decimal digits to be correct when using BFloat16 for calculations. While BFloat16 largely preserves the dynamic range of Float32, Float16 uses 10 mantissa bits, reducing the dynamic range to 12 orders of magnitude. Using 3 bits for the mantissa instead of the exponent, Float16 reaches a decimal precision of 3.7.

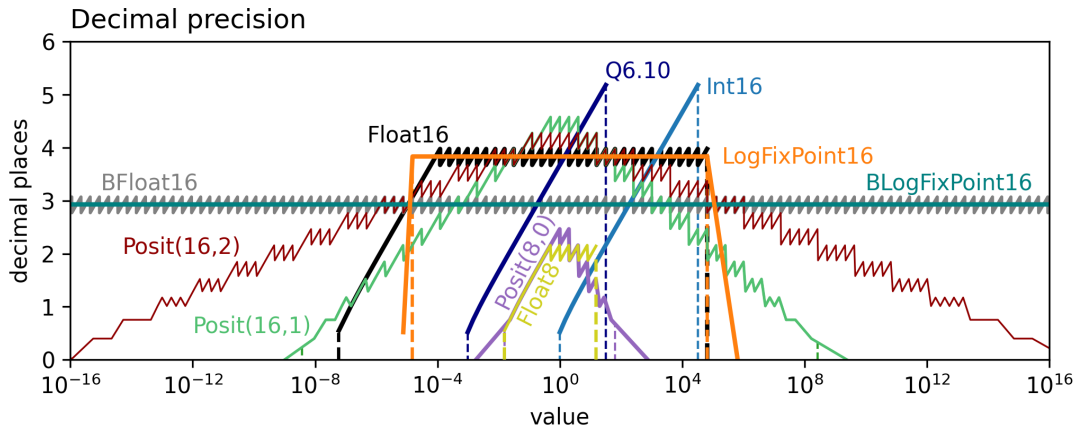

Figure 2.5 | Decimal precision of various 8 and 16-bit number formats. The number formats are described in more detail in Table 2.1. The decimal precision is worst-case, i.e. given in terms of decimal places that are at least correct after rounding (see section 2.3.3). Vertical dashed lines indicate the range of representable numbers (*minpos*, *maxpos*), respectively. BFloat16 and BLogFixPoint16 have representable range outside the figure limits, see Fig. 2.4.

The decimal precision of posits consists of a similar sawtooth wave as for floats as they share the same definition for the mantissa/fraction bits. However, for every power of *useed* away from 1 there is one bitposition less available for the mantissa. Consequently, posits have a pyramid-like shape in the decimal precision, whereby highest precisions are reached for the power of *useed* (e.g. $useed = 4$ for 1 exponent bit) centred on 1. The gradually lower precision away from 1, allows posits to have both a high precision at 1 and at the same time a wide dynamic range. The no-overflow rounding mode can be seen as the precision does not immediately drop to negative infinity outside the dynamic range for posits.

The logarithmic fixed-point number formats LogFixPoint16 and BLogFixPoint16 have a perfectly flat decimal precision of about 4 and 3 decimal places, respectively, due to the logarithmic spacing of the representable numbers. The precision drops quickly, but not immediately, to zero outside of the dynamic range due to the no-overflow rounding mode adopted from posits.

However, the decimal precision for LogFixPoint16 and BLogFixPoint16 only applies to operations such as conversion, addition and subtraction. For multiplication, division, power of 2 and square root the decimal precision is infinity, as these arithmetic operations map perfectly onto another logfix number for all arguments, unless an underflow or overflow occurs (section 2.1.4).

The decimal precision of 16-bit signed integers Int16 is negative infinity for any number below 0.5 (round to 0) and increases to the largest representable integer $maxpos = 2^{15} - 1 = 32767$. In that sense, using signed integers for scientific computing is tricky as most calculations should be placed near *maxpos* for increased precision, but a result beyond *maxpos* will yield desastrous results to the the wrap-around behaviour of integers (section 2.1.1). Similar conclusions hold for the fixed-point format Q6.10, as the decimal precision is simply shifted towards smaller numbers by a factor of $\frac{1}{2}$ for each additional fraction bit.

2.4 Code composability through type flexibility

The various number formats of the previous sections have to be emulated in software to use them in simulations, as only Float32 and Float64 are widely supported on hardware. Float16 is an exception as it is emulated in all chapters but chapter 6, which explicitly describes the usage of hardware-accelerated Float16. We therefore describe in the following the necessary programming paradigms to efficiently implement algorithms in a way that any (custom) number format can be used, whether hardware-accelerated or not. This type flexibility can also be exploited for number formats that analyse the code during execution, which is described in section 2.4.2.

2.4.1 A type-flexible programming paradigm

The Julia programming language provides several programming paradigms that facilitate the use of arbitrary number formats without the need to rewrite an algorithm [Bezanson *et al.*, 2017]. As this is an essential feature of Julia and extensively made use of in several chapters, we briefly outline the benefits of Julia by computing the harmonic sum $\sum_{i=1}^{\infty} \frac{1}{i}$ with various number formats as an example. Analytically the harmonic sum diverges, but with finite-precision arithmetic several issues arise: With an increasing sum the precision is eventually lower than required to represent the increment of the next summand. The integer i as well as its inverse $\frac{1}{i}$ have to be representable in a given number format, and are also subject to rounding errors.

The function `harmonic_sum` takes 2 positional arguments, whereby the second is optional and just prevents an infinite loop (Listing 2.4). As the first argument the function takes a type `T`, which can be a number format or any other, possibly custom, type in Julia. The syntax `where T` triggers a separate compilation for every type `T` that is provided as the first argument. Julia hereby makes use of its just-in-time (JIT) compiler, which automatically compiles a function at first execution. It relies thereby on *multiple-dispatch* which is a programming paradigm which assesses the types of all arguments to determine which (possibly already compiled) version of a function to execute. With multiple-dispatch several, conceptually similar functions of the same

```
1  function harmonic_sum(::Type{T},steps::Int=2000) where T
2
3      s = zero(T)      # partial sum, start with zero-element of type T
4      oone = one(T)    # numerator of every fraction, one-element of type T
5
6      for i in 1:steps
7          s_old = s
8          s += oone/convert(T,i)  # s = s + 1/i, harmonic sum
9
10         if s == s_old           # check for convergence
11             print((Float64(s),i))
12             break
13         end
14     end
15 end
```

Listing 2.4 | A type-flexible function calculating the harmonic sum in Julia. The number format is passed on as type in the first argument. The syntax `where T` triggers a separate compilation for every number format. The function `harmonic_sum` is type-stable as all types inside the function are declared and therefore unambiguous to the compiler.

name can exist in parallel, as long as they are defined for different types, and the most type-specific version is chosen in case functions are defined for super and subtypes.

The function `harmonic_sum` is type-stable, as the types of all variables are either internally declared (e.g. `i`, the counter of the loop is always an integer) or follow directly from the provided types in the arguments. Therefore, writing abstractly `s = zero(T)` to obtain the zero-element of type `T` instead of `s = 0` is important to avoid implicit conversions between types. The function `harmonic_sum` is also type-flexible, as it can be executed with every type `T` that has the following functions defined: `zero(T)`, `one(T)`, addition, division, comparison (== in line 10) conversion from integer (`convert(T,i)` in line 8) and to Float64. For example, `zero(::Type{Posit8}) = reinterpret(Posit8,0x00)` to define the zero-element for 8-bit posits, or `+(x::LogFixPoint16, y::LogFixPoint16) = ...` to define the addition between LogFixPoint16s.

These features of Julia allow for *code composability*. As the function `harmonic_sum` is abstractly written to allow for type flexibility it does not know about any (possibly custom) number format, such as posits or logfixs. On the other hand, we can define a custom number format including its arithmetic and other functions (e.g. `zero(T)`) that does not know about the function `harmonic_sum`. In essence, while both codes are completely independent (and possible defined in separate packages), they are composable, meaning they can be executed together. The function `harmonic_sum` therefore has to be defined only once, and does not have to be changed or rewritten for other formats.

The harmonic sum converges after 513 elements when using Float16 (Listing 2.5). The precision of BFloat16 is so low that the sum already converges after 65 elements, as the addition of the next term $\frac{1}{66}$ is rounded back to 5.0625. The addition of small terms to values of size $\mathcal{O}(1)$

```
julia> using SoftPosit, BFloat16s, LogFixPoint16s   # load number formats
julia> LogFixPoint16s.set_nfrac(10)                 # set to 10 fraction bits
julia> harmonic_sum(Float16)
(7.0859375, 513)            # converged to 7.09... after 513 elements

julia> harmonic_sum(BFloat16)
(5.0625, 65)                # converged to 5.06... after 65 elements

julia> harmonic_sum(Posit16)
(7.77734375, 1024)          # converged to 7.78... after 1024 elements

julia> harmonic_sum(LogFixPoint16)
(6.851249694824219, 432)    # converged to 6.85... after 432 elements
```

Listing 2.5 | Executing `harmonic_sum` with different number formats in the Julia shell. The number format is passed on as an argument, causing the function `harmonic_sum` to be compiled and executed with that format. Depending on the precision of the number format, the harmonic sum converges after 65 (BFloat16) to 1024 elements (Posit16).

is one of the major challenges with low-precision arithmetic, (section 5.2.4 and 6.2.3). Using Posit16, the sum converges only after 1024 terms, due to the higher decimal precision of posits between 1 and 8. The precision of LogFixPoint16 is similar to Float16, and the sum converges after 432 elements.

2.4.2 Analysis number formats

The type flexibility and code composability in Julia can be further extended by designing a custom number format that executes additional code than just returning the result when an arithmetic operations is called. If we are interested to understand why and how the harmonic sum converges with low precision number formats, we could alter `harmonic_sum` directly, however, for many larger algorithms, nested in a large amount of functions, this might be unfeasible or inefficient. Julia's code composability offers another high-level way that is illustrated in the following.

We create a new custom number format by defining a new type Float16print (Listing 2.6, line 1-3), which behaves as Float16 but we explicitly alter the definition of the addition for two Float16print values to execute additional code that helps to analyse the function `harmonic_sum`. Here, we add a print statement to obtain information on the numbers that are calculated within `harmonic_sum`, without changing `harmonic_sum` itself. In general, the arithmetic operations can be altered in many ways, and more specific implementations of analysis number formats are presented in section 6.2.2.

While the application here is rather simple, it shows how an analysis number format will change the compilation of a function. Here, Julia's compiler will create a compiled version of `harmonic_sum` that contains a `print` statement after every addition. Hence, the two codes of Listing 2.4 and 2.6 are composed allowing the user to interfere with pre-existing algorithms on

```julia
struct Float16print      # define a custom type called Float16print
    float16::Float16     # that contains only a Float16
end

# extend the zero and one functions for Float16print (same as for Float16)
Base.zero(::Type{Float16print}) = Float16print(zero(Float16))
Base.one(::Type{Float16print}) = Float16print(one(Float16))

# define division (/) for Float16print as for Float16
Base.:(/)(x::Float16print,y::Float16print) =
                    Float16print(x.float16 / y.float16)

# define addition (+) for Float16print as for Float16 but also print result
function Base.:(+)(x::Float16print,y::Float16print)
     z = x.float16 + y.float16     # unpack into float16, perform addition
     println(z)                    # add any aditional analysis code here
     return Float16print(z)
end

# define conversion from Integer and to Float64 as for Float16
Base.convert(::Type{Float16print},i::Int64) = Float16print(Float16(i))
Base.Float64(x::Float16print) = Float64(x.float16)
```

Listing 2.6 | Minimal example of the definition of a new custom analysis number format `Float16print` in Julia. The number format behaves like Float16 and defines all operations needed for `harmonic_sum` but the addition is extended with a `print` statement that helps to analyse code without changing `harmonic_sum` itself.

a high-level basis; e.g. no compiler knowledge is necessary.

As a result, Listing 2.7 shows how the first elements of the harmonic sum $1 + \frac{1}{2} + \frac{1}{3} + ...$ are added, and how the sum stagnates at 7.086. The next term $\frac{1}{514}$ cannot change 7.086 in addition as it is less than half the distance to the next representable number (round-to-nearest, section 2.2.2) and the result is rounded back. In Float16 arithmetic we therefore have $7.086 + \frac{1}{514} = 7.086$.

```
julia> harmonic_sum(Float16print)
1.0
1.5
1.833
...
7.082
7.086
7.086
(7.0859375, 513)
```

Listing 2.7 | Analysing the function `harmonic_sum` with the new custom number format `Float16print`. Executing `harmonic_sum` with `Float16print` triggers a `print` statement on every addition, i.e. after every term of the harmonic sum, but any other additional code can be executed too. Analysis number formats can be used to investigate even complicated algorithms without explicitly changing them.

2.5 Information theory

The following sections introduce information theory and how it can be used to analyse the information contained in binary number formats, which are used for weather and climate computing. A more specific discussion on the bitwise real information content analysed for atmospheric data in chapter 3 is found in section 3.3.1.

2.5.1 Entropy

Information entropy is a quantity that measures the amount of information contained in a random variable [MacKay, 2003; Shannon, 1948]. The information entropy H for a discrete random variable X with n outcomes $x_1, x_2, ..., x_n$ is defined as

$$H(X) = -\sum_{i}^{n} p_i \log_b(p_i), \tag{2.36}$$

where the p_i are the values of the probability mass function of X, i.e. the probabilities associated with the i-th outcome, such that $p_i \geq 0, i = 1, ..., n$, $\sum_{i=1}^{n} p_i = 1$. The base b of the logarithm defines the unit of entropy. In the following chapters we will exclusively use $b = 2$ such that the unit of information is given in bits. Furthermore, the entropy of impossible outcomes is defined as 0, i.e. for $p_j = 0$ we set $p_j \log_b(p_j) = 0$.

For example, when flipping a fair coin each side of the coin has the probability $p_1 = p_2 = \frac{1}{2}$. Here, for $i = 1$ the outcome is *tail*, and for $i = 2$ it is *head*, so each of the two outcomes is equally likely. However, if we assume that the coin is bent, its probabilities are, for example, changed to $p_1 = \frac{1}{10}$ and $p_2 = \frac{9}{10}$. Now in 9 out of 10 cases we would hardly be surprised as the outcome is head as we should expect, but in 1 out of 10 cases we would be surprised — the rarer the event the larger the surprise. Applying Eq. 2.36 yields an information entropy of 1 bit for the fair coin, but an entropy of 0.47 for the bent coin. Intuitively, information entropy can be interpreted as the average surprise when observing a random variable. When flipping the fair coin, each of the two outcomes are equally likely and therefore the uncertainty, i.e. the surprise, of the outcome is maximised. However, when observing the bent coin, in most cases the outcome corresponds to our expectation, which is the more certain event, yielding an information entropy that is less than half of the fair coin. The average surprise of the bent coin is lower than the average surprise of the fair coin.

The entropy of a discrete random variable defined over a closed interval, is maximised by a uniformly distributed probability mass function, i.e. all outcomes are equally likely (Eq. 2.36). For n possible outcomes the maximum entropy is then $\log_2(n)$ bits. Therefore, the fair coin flip with equal probabilities has maximum entropy, meaning that the prior knowledge of the probability function does not provide a means to predict the outcome. In that sense, observing the outcome provides exactly 1 bit of information. In contrast, the flip of the bent coin does not have maximum entropy but observing the event only conveys 0.47 bit of information, as in most cases we can correctly predict its outcome.

The outcomes of several coin flips can be represented with head and tail or equivalently `0` and `1` such that a sequence of 8 flips with the fair coin can be recorded as, for example, `10110100`. Every element of that sequence is a bit, forming an 8-bit sequence or 1 byte together. Given the entropy of 1 bit for every flip of the fair coin, the observation of the outcome adds another bit to the sequence. In the case of the bent coin, such a sequence will contain on average nine times as many `0`s than `1`s, for example 14 flips of the bent coin could result in `00001000000001`. When taking independent samples from a random variable and encoding them as such a sequence, Shannon's source coding theorem [Shannon, 1948] states that the information entropy is the lower limit on the average number of bits required to encode that sequence. For example, for the bent coin one may use an encoding which only stores the number of flips until the next 1 (e.g. 5,9 for `00001000000001`) instead of the outcome of each individual throw. Finding an encoding for a sequence that approaches maximum entropy and therefore requires less storage without loss of information is the subject of lossless compression algorithms [MacKay, 2003].

2.5.2 Bitpattern entropy

An n-bit number format has 2^n bitpatterns available to encode a real number. The bitpattern entropy is the Shannon information entropy H, in units of bits [Shannon, 1948], calculated from

the probability p_i, $i = 1, ..., 2^n$ of each bitpattern

$$H = -\sum_{i=1}^{2^n} p_i \log_2(p_i) \tag{2.37}$$

For most data arrays, the bitpatterns are not used at uniform probability. For a uniform distribution the probabilities are all equal and $p_i = 2^{-n}$ and the entropy is maximised to $\log_2(2^n) = n$. The difference $n - H$ between maximum entropy n and the actual entropy H is denoted as free entropy. With Shannon's source coding theorem, the free entropy and can be interpreted as the number of bits that are effectively unused in the encoding of a data array into the respective number format. For example, a data array of signed integers that is uniform in the positive range, but does not contain negative integers will have a free entropy of 1 bit, as the sign bit is unused.

Floating-point numbers can never have a maximum entropy as their encoding is not a bijective function. While one could argue that encoding both +0 and -0 with distinct bitpatterns is bijective ($\frac{1}{-0} = -\infty \neq \infty = \frac{1}{0}$ in float arithmetic), many bitpatterns are used to encode NaN (see Eq. 2.6), violating bijection. In practice, the free entropy resulting from this redundancy is negligible. Even for Float8 with 12.5% of bitpatterns representing NaR (Table 2.1) the resulting free entropy is 0.2 bit, reducing the maximum entropy of 8 bit to 7.8 bit. In contrast, using Float64 to represent numbers in the range of [256,512) (similar to, for example, air or water temperature in Kelvin on Earth) has an entropy of at most 52 bits compared to the 64 available bits as the sign and exponent bits are unused. Furthermore, such a data array would need to have at least $2^{52} \approx 10^{15.6}$ entries to reach maximum entropy in 52 bits. This corresponds to a theoretical array size of 36 Petabyte, which is far beyond typical array sizes. From an entropy perspective, using 64-bit number formats are therefore a very inefficient way to encode real numbers into bits. Nevertheless, this entropy inefficiency allows Float64 calculations to have minimal rounding errors.

2.5.3 Conditional information entropy

In the previous section, the information entropy is calculated from *unconditional* probabilities, either known or estimated from the frequency of occurrences in data. Now, we will extend the concept of entropy to *conditional* probabilities, such that the conditional entropy measures a statistical relation between two random variables X, Y with outcomes x_i, y_j, $i = 1, ..., n, j = 1, ..., m$. The conditional entropy $H(Y|X)$ of Y conditioned on X is defined as

$$H(Y|X) = -\sum_{i,j} p_{i,j} \log_b \left(\frac{p_{i,j}}{p_i} \right) \tag{2.38}$$

with $p_{i,j}$ being an entry from the joint probability matrix p_{XY} of the outcomes in X, Y and p_i the marginal probability of X.

Assume the situation of two sport teams in a final where no draw is possible. If team 1 wins then team 2 loses and vice versa, but assume that team 1 is twice as likely to win than team 2. Hence, $P(\text{Team 1 wins}) = \frac{2}{3}$ and $P(\text{Team 2 wins}) = \frac{1}{3}$, which are the marginal probabilities p_i, p_j, but the conditional probabilities are 1

$$P(\text{Team 1 wins}|\text{Team 2 lost}) = P(\text{Team 2 wins}|\text{Team 1 lost}) = 1 \tag{2.39}$$

The joint probability matrix is then

$$p_{XY} = \begin{pmatrix} 0 & \frac{1}{3} \\ \frac{2}{3} & 0 \end{pmatrix} \tag{2.40}$$

Substituting into Eq. 2.38 yields a conditional entropy of $H(\text{Team 2}|\text{Team 1}) = 0$ which can be interpreted as follows. While it is unclear which team will win, once one team is known to have won no further information is provided when learning that the other team has lost — this is implicitly given, given the knowledge of whether one team has won or lost. However, the unconditional entropy, calculated from the unconditional probabilities of either team winning, $p_i = \frac{1}{3}, p_j = \frac{2}{3}$, is $H \approx 0.92$ bit. The (unconditional) information entropy is larger than 0 as it is still uncertain which team will win, however, once one team wins, it implicitly follows that the other team has lost, which does not provide any further information.

Consider now the following joint probability that represents the wind direction for X with *East, no wind,* and *West* and the number of sailing boats for Y with some *sailing boats* and *no sailing boats*.

$$p_{XY} = \begin{pmatrix} \frac{3}{8} & 0 \\ 0 & \frac{1}{4} \\ \frac{3}{8} & 0 \end{pmatrix} \tag{2.41}$$

In 25% of the cases no wind was observed, $P(\text{no wind}) = \frac{1}{4}$. For no wind, no sailing boats were seen, $P(\text{no sailing boats}|\text{no wind}) = 1$. In 75% the wind came either from the East or from the West at equal probabilities, $P(\text{East}) = P(\text{West}) = \frac{3}{8}$. In either of those cases some sailing boats were seen, $P(\text{sailing boats}|\text{East}) = P(\text{sailing boats}|\text{West}) = 1$. The conditional entropy $H(Y|X)$, i.e. the number of sailing boats conditioned on the wind direction, is calculated from p_{XY} as $H(Y|X) = 0$. Once we know there is wind, either from East or West, we know there are sailing boats. Observing sailing boats therefore does not add any information to the already acquired knowledge that the wind comes either from East or West (but it is certainly not no wind).

However, we may not be able to measure the wind directly but are only able to observe the number of sailing boats. In that situation we are interested in the conditional entropy of the wind direction X given the number of sailing boats Y, i.e. $H(X|Y)$. This interchanges the predictor and predictand from before. Transposing p_{XY} and interchanging the marginal probabilities p_i, p_j in Eq. 2.38 we calculate the conditional entropy of the wind direction conditioned on the number of sailing boats as $H(X|Y) = 0.75$ bit. This can be interpreted as knowing there are sailing boats

tells us there is wind too, but we cannot infer the wind direction. Hence, there is a remaining uncertainty which yields a non-zero (conditional) entropy.

2.5.4 Mutual information

In general, we may ask about the *mutual information* of two random variables. In the example from before, we would like to know the mutual information between wind direction and the number of sailing boats. The mutual information $M(X, Y)$ for two random variables X, Y from information theory is defined as

$$M(X, Y) = \sum_{i,j} p_{i,j} \log_b \left(\frac{p_{i,j}}{p_i p_j} \right) \tag{2.42}$$

which is therefore a symmetric quantity, $M(X, Y) = M(Y, X)$. The mutual information quantifies the information between X, Y that is already known about either X or Y from knowing the other. Two statistically independent variables therefore have 0 mutual information, but for fully dependent variables X, Y the mutual information is equal the unconditional entropy $H(X) = H(Y)$. This is in contrast to the conditional entropy which quantifies the information that is *still gained* from observing Y given that X is already known, not the information that we *already have* about Y. In that sense, the conditional entropy $H(Y|X)$ of two independent variables is the unconditional entropy $H(Y)$, but the conditional entropy of two fully dependent variables X, Y is zero.

The mutual information between wind direction and the number of sailing boats is $M \approx 0.81$ bit. Although there is no correlation between the wind direction and the number of sailing boats, there is a nonlinear relationship between the two random variables that is quantified by the mutual information. The mutual information cannot reveal whether the statistical dependence between two random variables is linear or nonlinear. However, any change of the probability distribution of one random variable given the state of another is detected and yields a mutual information larger than zero.

We will now discuss how the mutual information is applied to bits as they occur in binary number formats. The joint probability is for two random variables X, Y that both can both take either of two states, 0 or 1, a 2x2 matrix. In that case we can describe X, Y as bits being either 0 or 1 and hence binary. Such a joint probability matrix is then in general

$$p_{XY} = \begin{pmatrix} p_{00} & p_{01} \\ p_{10} & p_{11} \end{pmatrix} \tag{2.43}$$

with p_{ij} the probability that $X = i$ and $Y = j$ simultaneously. In the case of

$$p_{XY} = \begin{pmatrix} p_{00} & 0 \\ 0 & p_{11} \end{pmatrix} \tag{2.44}$$

the state of X is entirely determined by the state of Y and hence the conditional entropy is zero, while the mutual information is $M(X,Y) = H(X) = H(Y)$. The correlation between X and Y is 1 in this case and if the states `0,1` are also identified with their respective numerical values $0, 1$ (the latter requires a mapping of outcomes in X, Y to the real numbers, which is not always possible). Similarly, for

$$p_{XY} = \begin{pmatrix} 0 & p_{01} \\ p_{10} & 0 \end{pmatrix} \tag{2.45}$$

the same conclusions in terms of information hold, but the correlation is -1. Statistically independent X, Y translate in the joint probability matrix to

$$p_{XY} = \begin{pmatrix} p_0 & p_0 \\ p_1 & p_1 \end{pmatrix}, \quad \text{or} \quad p_{XY} = \begin{pmatrix} p_0 & p_1 \\ p_0 & p_1 \end{pmatrix} \tag{2.46}$$

such that the probabilities in X are independent of those in Y and vice versa.

Bitwise mutual information is a central concept to distinguish real and false information in data, which is the aim of the next chapter to develop an information-preserving compression for climate data.

3 Information-preserving compression for climate data

Contributions This chapter is largely based on the following publication*

M Klöwer, M Razinger, JJ Dominguez, PD Düben and TN Palmer, 2021. *Compressing atmospheric data into its real information content*, **Nature Computational Science**, accepted. Preprint 10.21203/rs.3.rs-590601/v1

Abstract. Hundreds of petabytes of data are produced annually at weather and climate forecast centres worldwide. Compression is inevitable to reduce storage and to facilitate data sharing. Current techniques do not distinguish the real from the false information in data. We define the bitwise real information content from information theory for data from the Copernicus Atmospheric Monitoring Service (CAMS). Most variables contain less than 7 bits of real information per value, which are also highly compressible due to spatio-temporal correlation. Rounding bits without real information to zero facilitates lossless compression algorithms and encodes the uncertainty within the data itself. The entire CAMS data is compressed by a factor of 17x, relative to 64-bit floats, while preserving 99% of real information. Combined with 4-dimensional compression to exploit the spatio-temporal correlation, factors beyond 60x are achieved without an increase in forecast errors. A data compression Turing test is proposed to optimise compressibility while minimising information loss for the end use of weather and climate forecast data.

3.1 Introduction

Many supercomputing centres in the world perform operational weather and climate simulations several times per day [Bauer *et al.*, 2015]. The European Centre for Medium-Range Weather Forecasts (ECMWF) produces 230TB of data on a typical day and most of the data is stored on magnetic tapes in its archive. The data production is predicted

*with the following author contributions. Conceptualisation: MK, MR, JJD. Data curation: MR, JJD, MK. Formal Analysis: MK. Methodology: MK. Visualisation: MK. Writing – original draft: MK. Writing – review & editing: MK, PDD, MR, JJD, TNP.

to quadruple within the next decade due to an increased spatial resolution of the forecast model [Bauer *et al.*, 2020; Schär *et al.*, 2020; Voosen, 2020]. Initiatives towards operational predictions with global storm-resolving simulations, such as Destination Earth [Bauer *et al.*, 2021a,b] or DYAMOND [Stevens *et al.*, 2019], with a grid spacing of a couple of kilometres will further increase data volume. This data describes physical and chemical variables of atmosphere, ocean and land in up to 6 dimensions: three in space, time, forecast lead time, and the ensemble dimension. The latter results from calculating an ensemble of forecasts to estimate the uncertainty of predictions [Molteni *et al.*, 1996; Palmer, 2019b]. Most geophysical and geochemical variables are highly correlated in all of those dimensions, a property that is rarely exploited for climate data compression, although multidimensional compressors are being developed [Ballester-Ripoll *et al.*, 2020; Di & Cappello, 2016; Lindstrom, 2014; von Larcher & Klein, 2019; Zhao *et al.*, 2020].

Floating-point numbers are the standard to represent real numbers in binary form. 64-bit double precision floating-point numbers (Float64) consist of a sign bit, 11 exponent bits representing a power of two, and 52 mantissa bits allowing for 16 decimal places of precision across more than 600 orders of magnitude (IEEE [1985] and section 2.1.3). Most weather and climate models are based on Float64 arithmetics, which has been questioned as the transition to 32-bit single precision floats (Float32) does not necessarily decrease the quality of forecasts [Tintó Prims *et al.*, 2019; Váňa *et al.*, 2017]. Many bits in Float32 only contain a limited amount of information as even 16-bit arithmetic has been shown to be sufficient for parts of weather and climate applications [Ackmann *et al.*, 2021; Dawson *et al.*, 2018; Fan *et al.*, 2019; Hatfield *et al.*, 2019; Klöwer *et al.*, 2020]. The information, as defined by Shannon information theory [Kleeman, 2011; MacKay, 2003; Shannon, 1948], for simple chaotic dynamical systems is often zero for many of the 32 bits in Float32 [Jeffress *et al.*, 2017]. This supports the general concept of low-precision climate modelling for calculations and data storage, as, at least in theory, many rounding errors are entirely masked by other uncertainties in the chaotic climate system [Palmer, 2014, 2015].

The bitwise information content has been formulated for predictability in dynamical systems [Jeffress *et al.*, 2017]. It quantifies how much individual bits in the floating-point representation contribute to the information necessary to predict the state of a chaotic system at a later point in time. This technique has been used to optimise the simulation of simple chaotic systems on inexact hardware to reduce the precision as much as possible. Here, we extend the bitwise information content to distinguish between bits with real and false information in data and to quantify the preserved real information in data compression.

Data compression for floating-point numbers often poses a trade-off in size, precision and speed [Hübbe *et al.*, 2013; Kuhn *et al.*, 2016; Silver & Zender, 2017]: Higher compression factors for smaller file sizes can be achieved with lossy compression, which reduces the precision and introduces rounding errors. Additionally, higher compression requires more sophisticated compression algorithms, which can decrease compression and/or decompression speed. A reduction in precision is not necessarily a loss of real information, as occurring rounding errors are relative to a reference that itself comes with uncertainty. Here, we calculate the bitwise real information content [Jeffress *et al.*, 2017; Kleeman, 2011; Shannon, 1948] of atmospheric data to discard bits that contain no information [Kouznetsov, 2020; Zender, 2016] and only compress the real information content. Combined with modern compression algorithms [Lindstrom, 2014; Lindstrom & Isenburg, 2006] the multi-dimensional correlation of climate data is exploited for higher compression efficiency [Baker *et al.*, 2016, 2019; Woodring *et al.*, 2011].

3.2 Data

3.2.1 Copernicus Atmospheric Monitoring Service

CAMS data is analysed for one time step on 01/12/2019 12:00 UTC, bilinearly regridded onto a regular 0.4°x0.4° longitude-latitude grid using climate data operators (cdo) v1.9. All 137 vertical model levels are included. Furthermore, global fields of temperature from ECMWF's ensemble prediction system with 91 vertical levels are used from the first 25 members of a 50-member 15-day ensemble forecast starting on 24 Sept 2020 00:00 UTC. Bilinear regridding onto a regular 0.2°x0.2° longitude-latitude grid, similar as for the CAMS data, is applied. All compression methods here include the conversion from Float64 to Float32.

Only longitude-latitude grids are considered in this paper. However, the methodology can be applied to other grids too. For example, ECMWF's octahedral grid collapses the two horizontal dimensions into a single horizontal dimension which circles on latitude bands around the globe starting at the South Pole till reaching the North Pole [Malardel *et al.*, 2016]. Fewer grid points of the octahedral grid reduce the size, but the correlation in latitudinal direction cannot be exploited.

3.2.2 Grid definitions

The compression methods described here are applied to gridded binary data. Data on structured grids can be represented as a tensor, such that for two dimensions that data can be arranged in a matrix A with elements a_{ij} and i, j indices. Adjacent elements in A, e.g. a_{ij} and $a_{i+1,j}$, are also adjacent grid points. Every element a_{ij} is a floating-point number, or in general a number represented in any binary format. The n bits in a_{ij} are described as bit positions, including sign, exponent and mantissa bits. In the following we will consider sequences of bits that arise from incrementing the indices i or j while holding the bit position fixed. For example, the sequence of bits consisting of the first mantissa bit in a_{ij}, then the first mantissa bit in $a_{i+1,j}$, and so on. We may refer to these bits as bits from adjacent grid points. Every bit position in elements of A is itself a matrix, e.g. the matrix of sign bits across all grid points.

3.3 Methods

3.3.1 Real information content

The Shannon information entropy H in units of bits [Shannon, 1948] takes for a bitstream $b = b_1 b_2 ... b_k ... b_l$, i.e. a sequence of bits of length l, the form

$$H = -p_0 \log_2(p_0) - p_1 \log_2(p_1) \tag{3.1}$$

with p_0, p_1 being the probability of a bit b_k in b being 0 or 1 (see section 2.5.1). The entropy is maximised to 1 bit for equal probabilities $p_0 = p_1 = \frac{1}{2}$ in b. We derive the mutual information [Kraskov *et al.*, 2004; Pothapakula *et al.*, 2019; Schreiber, 2000] of two bitstreams $r = r_1 r_2 ... r_k ... r_l$ and $s = s_1 s_2 ... s_k ... s_l$ (see section 2.5.4). The mutual information is defined via the joint probability mass function p_{rs}, which here takes the form of a 2x2 matrix

$$p_{rs} = \begin{pmatrix} p_{00} & p_{01} \\ p_{10} & p_{11} \end{pmatrix} \tag{3.2}$$

with p_{ij} being the probability that the bits are in the state $r_k = i$ and $s_k = j$ simultaneously and $p_{00} + p_{01} + p_{10} + p_{11} = 1$. The marginal probabilities follow as column or row-wise additions in p_{rs}, e.g. the probability that $r_k = 0$ is $p_{r=0} = p_{00} + p_{01}$. The mutual

information $M(r, s)$ of the two bitstreams r, s is then

$$M(r,s) = \sum_{r=0}^{1}\sum_{s=0}^{1} p_{rs} \log_2\left(\frac{p_{rs}}{p_{r=r}p_{s=s}}\right) \tag{3.3}$$

We now consider the two bitstreams r, s being the preceding and succeeding bits (for example in space or time) in a single bitstream b, i.e. $r = b_1 b_2 ... b_{l-1}$ and $s = b_2 b_3 ... b_l$. As explained in section 3.2.2, this can for example be the bitstream of all first mantissa bits in the gridded data. Considering r, s as the preceding and succeeding bits is equivalent to the bitwise mutual information in adjacent grid points. The (unconditional) entropy is then effectively $H = H(r) = H(s)$ as in Eq. 3.1 and for l being very large. The conditional entropies H_0, H_1 (see section 2.5.3) are conditioned on the state of the preceding bit b_{k-1} being 0 or 1, respectively

$$H_0 = -p_{00}\log_2(p_{00}) - p_{01}\log_2(p_0 1) \tag{3.4}$$

$$H_1 = -p_{10}\log_2(p_{10}) - p_{11}\log_2(p_1 1) \tag{3.5}$$

The conditional entropy is maximised to 1 bit for bitstreams where the probability of a bit being 0 or 1 does not depend on the state of the preceding bit, which is here defined as *false information*. With the conditional and unconditional entropies and p_0, p_1 as in Eq. 3.1 the mutual information M of succeeding bits can be written as

$$I = H - p_0 H_0 - p_1 H_1 \tag{3.6}$$

which is the real information content I. This definition is similar to Jeffress *et al.* [2017] but avoids an additional assumption of an uncertainty measure. Their formulation similarly uses the state of bits as predictors but assesses the conditional probability mass functions of a dynamical system as predictands. The binwidth of the probability mass function is chosen to represent the uncertainty in the system, which the bitwise real information strongly depends on. The formulation here avoids such an additional assumption of uncertainty as bits are used as both predictors and predictands in the conditional entropy. Consequently, the uncertainty is obtained from the data itself solely based on the mutual information between bits in adjacent grid points.

Eq. 3.6 defines the real information as the entropy minus the false information. For bitstreams with either $p_0 = 1$ or $p_1 = 1$, i.e. all bits are either 0 or 1, the entropies are zero $H = H_0 = H_1 = 0$ and we may refer to the bits in the bitstream as being unused. In the case where $H > p_0 H_0 + p_1 H_1$, the preceding bit is a predictor for the succeeding

bit which means that the bitstream contains real information ($I > 0$).

3.3.2 The multidimensional real information content

The real information content I_m for an m-dimensional array A is the sum of the real information along the m dimensions. Let b_j be a bitstream obtained by unravelling a given bitposition in A along its j-th dimension. Although the unconditional entropy H is unchanged along the m-dimensions, the conditional entropies H_0, H_1 change as the preceding and succeeding bit is found in another dimension, e.g. b_2 is obtained by re-ordering b_1. $H_0(b_j)$ and $H_1(b_j)$ are the respective conditional entropies calculated from bitstream b_j. Normalisation by $\frac{1}{m}$ is applied to I_m such that the maximum information is 1 bit in I_m^*

$$I_m^* = H - \frac{p_0}{m}\sum_{j=1}^{m} H_0(b_j) - \frac{p_1}{m}\sum_{j=1}^{m} H_1(b_j) \tag{3.7}$$

Due to the presence of periodic boundary conditions for longitude a succeeding bit might be found across the bounds of A. This simplifies the calculation as the bitstreams are obtained from permuting the dimensions of and subsequent unravelling into a vector.

3.3.3 Preserved information

We define the preserved information $P(r, s)$ in a bitstream s when approximating r (e.g. after a lossy compression) via the symmetric normalised mutual information

$$R(r, s) = \frac{2M(r, s)}{H(r) + H(s)} \tag{3.8}$$

R is the redundancy of information of r in s. The preserved information P in units of bits is then the redundancy-weighted real information I in r

$$P(r, s) = R(r, s)I(r) \tag{3.9}$$

The information loss L is $1 - P$ and represents the unpreserved information of r in s. In most cases we are interested in the preserved information of an array $X = (x_1, x_2, ...x_q, ...x_n)$ of bitstreams x when approximated by a previously compressed array $Y = (y_1, y_2, ...y_q, ...y_n)$. For an array A of floats with $n = 32$ bit, for example, x_1 is the bitstream of all sign bits unravelled along a given dimension (e.g. longitudes) and x_{32} is the bitstream of the last

mantissa bits. The redundancy $R(X, Y)$ and the real information $I(X)$ are then calculated for each bit position q individually, and the fraction of preserved information P is the redundancy-weighted mean of the real information in X

$$P(X, Y) = \frac{\sum_{q=1}^{n} R(x_q, y_q) I(x_q)}{\sum_{q=1}^{n} I(x_q)} \tag{3.10}$$

The quantity $\sum_{q=1}^{n} I(x_q)$ is the total information in X and therefore also in A. The redundancy is $R = 1$ for bits that are unchanged during rounding and $R = 0$ for bits that are rounded to zero. The preserved information with bitshave or halfshave [Kouznetsov, 2020; Zender, 2016] (i.e. replacing mantissa bits without real information with either 00...00 or 10...00, respectively) is therefore equivalent to truncating the bitwise real information for the (half)shaved bits. For round-to-nearest, however, the carry bit depends on the state of bits across several bit positions. To account for interdependency of bit positions the mutual information has to be extended to include more bit positions in the joint probability p_{rs}, which will then be a $m \times 2$ matrix. For computational simplicity, we truncate the real information as the rounding errors of round-to-nearest and halfshave are equivalent.

3.3.4 Significance of real information

In the analysis of real information it is important to distinguish between bits with very little but significant information and those with information that is insignificantly different from zero. While the former have to be retained, the latter should be discarded to increase compressibility. A significance test for real information is therefore presented.

For an entirely independent and approximately equal occurrence of bits in a bitstream of length I, the probabilities p_0, p_1 of a bit being 0 or 1 approach $p_0 \approx p_1 \approx \frac{1}{2}$, but they are in practice not equal for $I < \infty$. Consequently, the entropy is smaller than 1, but only insignificantly. The probability p_1 of successes in the binomial distribution (with parameter $\frac{1}{2}$) with I trials (using the normal approximation for large I) is

$$p_1 = \frac{1}{2} + \frac{z}{2\sqrt{2}} \tag{3.11}$$

where z is the $1 - \frac{1}{2}(1 - c)$ quantile at confidence level c of the standard normal distribution. For $c = 0.99$ corresponding to a 99%-confidence level which is used as default here, $z = 2.58$ and for $I = 5.5 \cdot 10^7$ (the size of a 3D array from CAMS) a probability $\frac{1}{2} \leq p \leq p_1 = 0.5002$ is considered insignificantly different from equal occurrence

$p_0 = p_1$. The associated free entropy H_f in units of bits follows as

$$H_f = 1 - p_1 \log_2(p_1) - (1 - p_1) \log_2(1 - p_1) \tag{3.12}$$

We consider real information below H_f as insignificantly different from 0 and set the real information $I = 0$.

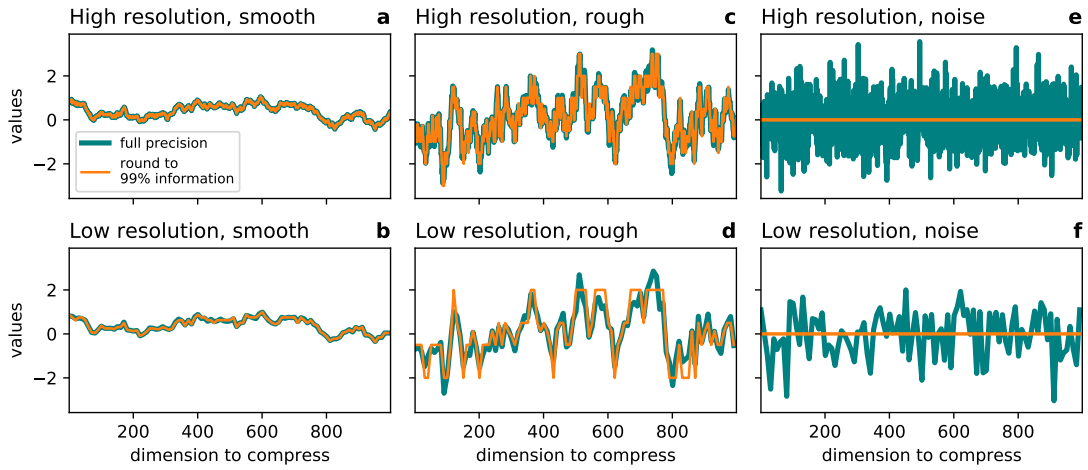

Figure 3.1 | Resolution and smoothness dependence of the information-preserving compression. **a,b** Highly autocorrelated data (1st order auto-regressive process with correlation $r = 0.999$) will have many mantissa bits preserved, at high and low resolution. **c,d** Many mantissa bits in data with less autocorrelation ($r = 0.95$) will be independent at low resolution and therefore rounded to zero. **e,f** All bits in random data ($r = 0$) drawn from a standard normal distribution are fully independent so that removing the false information rounds this data to zero. Low resolution data (**b,d,f**) is obtained from high resolution (**a,c,e**) by subsampling every 10th data point.

3.3.5 Dependency of the bitwise real information on correlation

The real information as defined here depends on the mutual information of bits in adjacent grid points. Higher autocorrelation in data (meaning a higher correlation between adjacent grid points) increases the mutual information in the mantissa bits. With higher correlation the adjacent grid values are closer, increasing the statistical dependence of mantissa bits that would otherwise be independent at lower correlation. Consequently, the real bitwise information content is increased and more mantissa bits have to be retained to preserve 99% of real information (Fig. 3.2a and b).

The increasing number of retained mantissa bits with higher autocorrelation in data will decrease compression factors, as it is easier to compress bits that are rounded to zero. However, a higher correlation also increases the redundancy in bits of adjacent

grid points, which favours a more efficient lossless compression. These two effects counteract and compression factors only increase piecewise over a small range of correlations while the retained mantissa bits are constant (Fig. 3.2c and d). Once an additional mantissa bit has to be retained to preserve 99% of real information, compression factors jump back down again, resulting in a sawtooth wave. Over a wide range of typical correlation coefficients (0.5 - 0.9999) the compression factors are otherwise constant and no higher compressibility is found with increased correlation.

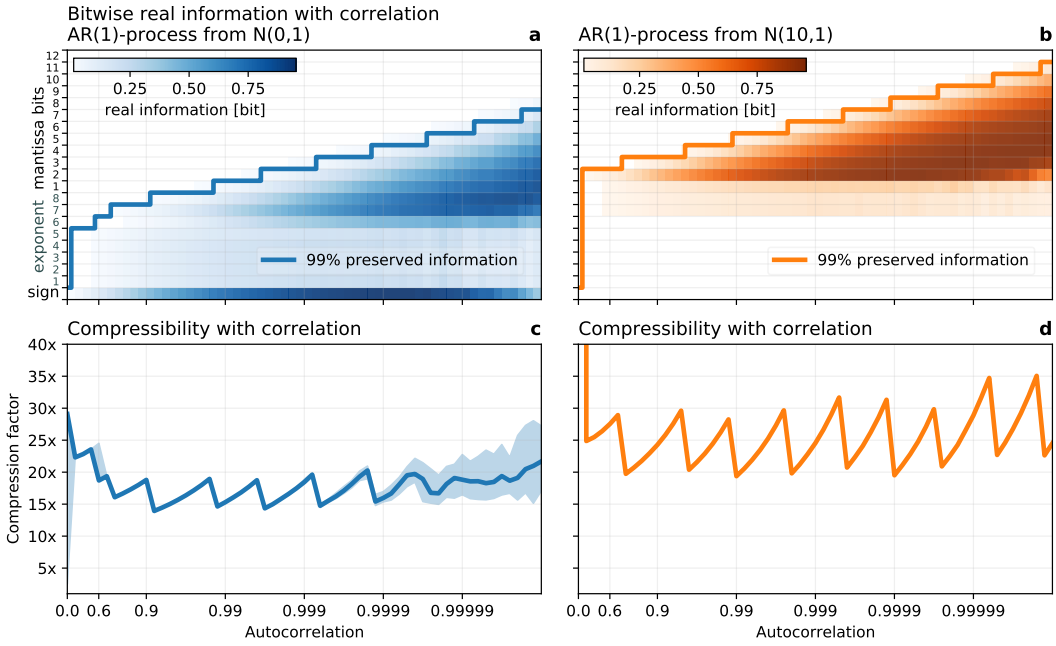

Figure 3.2 | Dependency of the bitwise real information and compressibility on correlation. a The bitwise real information content of a first-order autoregressive process (AR(1) with Gaussian distribution N(0,1), i.e. with zero mean and unit variance) with varying lag-1 autocorrelation. The bits that have to be retained to preserve 99% of information are enclosed with a solid line. b as **a** but the AR(1) process follows a Gaussian distribution with a mean of 10. **c,d** Compression factors for **a,b** when preserving 99% of information. Shading denotes the interdecile range.

The compression factors can, however, depend on the range of values represented in binary: A shift in the mean to have positive or negative values only means that the sign bit is unused, which increases compression factors (compare Fig. 3.2a with b), despite identical correlation coefficients. Although the correlation is invariant under multiplicative scaling and addition, the bitwise information changes under addition: When the range of values in data fits into a power of two its real information is shifted across bit positions into the mantissa bits, such that the exponent bits are unused. This can be observed for atmospheric temperatures stored in Kelvin (within 200-330K) where only

the last exponent bit and mantissa bits contain information (Fig. 3.18). Using Celsius instead shifts information from the mantissa bits into the exponent and sign bits.

3.3.6 Limitations of the information-preserving compression

The definition of real information presented here is based on the mutual information in adjacent grid points. We therefore assume a spatial and temporal coherence of data that will come with some autocorrelation. For vanishing autocorrelation in the data the real information content will drop to zero, as the mutual information between bits in adjacent but independent grid points approaches zero. In this case the entire dataset is identified as false information and consequently rounded to zero. In practice, this only occurs with data having autocorrelation coefficients of less than 0.2 (Fig. 3.2). If there is valuable scientific information in such seemingly random data, then the underlying assumption that real information is identified by the mutual information in adjacent grid points does not hold.

Issues with the bitwise real information content can arise in data that was previously subject to lossy compression: Linear or logarithmic quantization, for example, rounds data in linear or logarithmic space, respectively, which is not equivalent to binary rounding in the floating-point format. Consequently, such a quantization will generally introduce non-zero bits in the mantissa of floats when decompressed. These bits can have some statistical dependence, appearing as artificial information induced by the quantization. Such artificial information can be observed as a small background information (i.e. the bitwise real information is always significantly different from 0) or a reemerging information in the last mantissa bits. In this case, the information distribution across bit positions deviates clearly from the typical, where the information drops monotonically in the mantissa bits and becomes insignificantly different from 0 (see the examples in Fig. 3.10, 3.2, 3.4 or 3.18). A solution to this quantization-induced artificial information is to apply the bitwise real information analysis not on the decompressed floats but on the rounded integers representing the compressed data in linear or logarithmic quantization. The bitwise real information content as defined here is independent of the binary format and therefore can be applied to floats as well as the unsigned integers from quantization. Note that this issue does not arise when rounding is applied in the same encoding which is also used to analyse the real information content. In our case, using the floating-point representation for the rounding (i.e. removing the false information) the rounded mantissa bits are always 0 which also means zero entropy when analysing the information. Applying the rounding for floats repeatedly will have no effect beyond

the first application (idempotence).

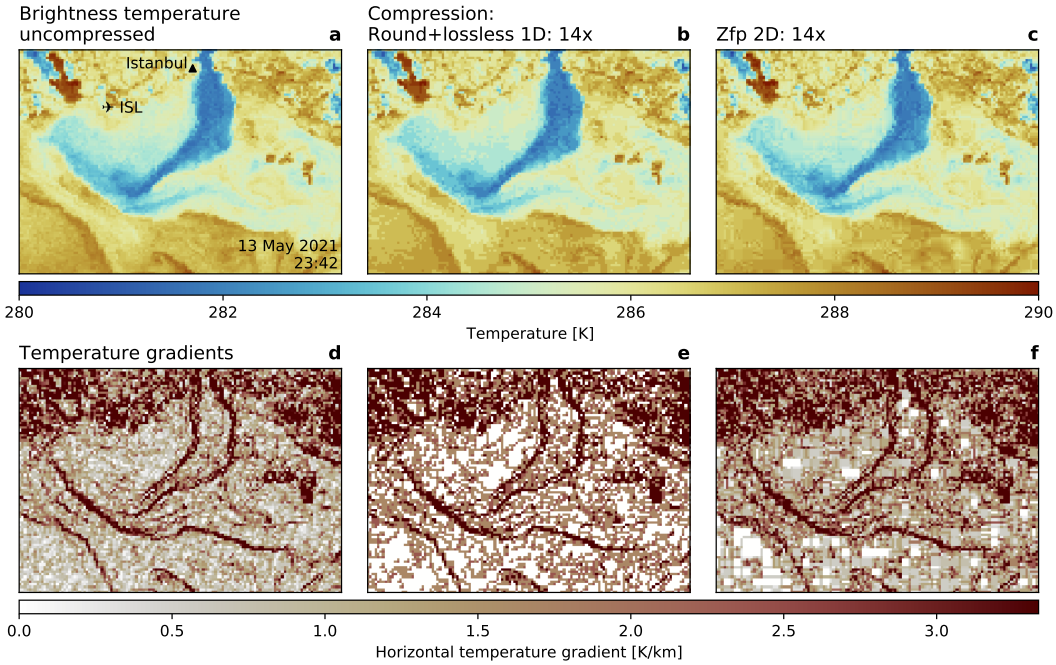

Figure 3.3 | Preservation of gradients during compression. Compressing the brightness temperature of Fig. 3.14 (VIIRS sensor aboard the satellite Suomi NPP) south of Istanbul where the Black Sea outflows into the Marmara Sea. Oceanic fronts with strong horizontal gradients in sea surface temperature are visible. **a** Brightness temperature uncompressed. **b** as **a** but compressed using round+lossless preserving 99% of real information. **c** as **a** but using Zfp compression in the two horizontal dimensions. **d** Horizontal temperature gradient uncompressed highlighting the oceanic fronts from **a**. **e** as **a** but the horizontal gradient is calculated from the round+lossless compressed dataset as shown in **b**. **f** as **e** but using Zfp compression as shown in **c**. The coarseness of the visualisation represents the resolution of the data. Istanbul (Hagia Sophia) and Atatürk Airport (ISL) are marked for orientation.

3.3.7 Preservation of gradients

The preservation of gradients and other higher-order derivatives in data is a challenging aspect of compression. Removing false information in data via rounding can result in identical values in adjacent grid points. Even if these values were not identical before rounding, they may not be significantly different from each other in the sense of real and false information. In this case, a previously weak but non-zero gradient will be rounded to zero. In other cases the rounding error is small compared to the standard deviation of the data, such that rounding has a negligible impact on the variance as values are independently equally likely to be round up or down.

This can be illustrated in the example of analysing oceanic fronts obtained from satel-

lite measurements of sea surface temperatures (Fig. 3.3). Identified by large horizontal gradients in temperature, the location and strength of oceanic fronts is well preserved using compressed data. However, areas of very weak gradients can largely vanish with round+lossless (Fig. 3.3e). In this case the temperature in adjacent grid points is insignificantly different from each other and therefore the gradient zero after the removal of false information. Weak gradients are better preserved with Zfp compression at similar compression factors, but its block structure becomes visible (Fig. 3.3f).

3.3.8 Structural similarity

A metric to assess the quality of lossy compression in image processing is the structural similarity index measure (SSIM, Wang *et al.* [2004]). For images it is based on comparisons of luminance, contrast and structure. For floating-point arrays the luminance contributions to SSIM can be interpreted as the preservation of the mean; the contrast compares the variances and the structure compares the correlation. The SSIM of two arrays A, B of same size is defined as

$$\mathrm{SSIM}(A,B) = \frac{(2\mu_A\mu_B + c_1)(2\sigma_{AB} + c_2)}{(\mu_A^2 + \mu_B^2 + c_1)(\sigma_A^2 + \sigma_B^2 + c_2)} \tag{3.13}$$

With μ_A, μ_B the respective means, σ_A^2, σ_B^2 the respective variances and σ_{AB} the covariance. $c_1 = (k_1 L)^2$ and $c_2 = (k_2 L)^2$ are introduced to increase stability with a small denominator and $k_1 = 0.01$, $k_2 = 0.03$. The dynamic range is $L = \max(\max(A), \max(B)) - \min(\min(A), \min(B))$. The SSIM is a value in $[0, 1]$ where the best possible similarity $\mathrm{SSIM} = 1$ is only achieved for identical arrays.

For rounded floating-point arrays the decimal error is proportional to the square root of the dissimilarity $1 - \mathrm{SSIM}$ (Fig. 3.4c). The SSIM in this case is approximately equal to the correlation, as round-to-nearest is bias-free (i.e. $\mu_A \approx \mu_B$) and as the rounding error is typically much smaller than the standard deviation of the data (i.e. $\sigma_A \approx \sigma_B$). Here, we use the logarithmic SSIM, $\mathrm{SSIM}_{\log}(A,B) = \mathrm{SSIM}(\log(A), \log(B))$, which is the SSIM applied to log-preprocessed data (the logarithm is applied element-wise). The usage of $\mathrm{SSIM}_{\log}$ is motivated due to the rather logarithmic data distribution for most variables (Fig. 3.7), but similar results are obtained for SSIM. The proportionality to the decimal error is unchanged when using $\mathrm{SSIM}_{\log}$.

Baker *et al.* [2019] propose the SSIM as a quality metric for lossy compression of climate data52. While for image processing SSIM > 0.98 is considered good quality, Baker *et al.* [2019] suggest a higher threshold of SSIM = 0.99995 for climate data compression.

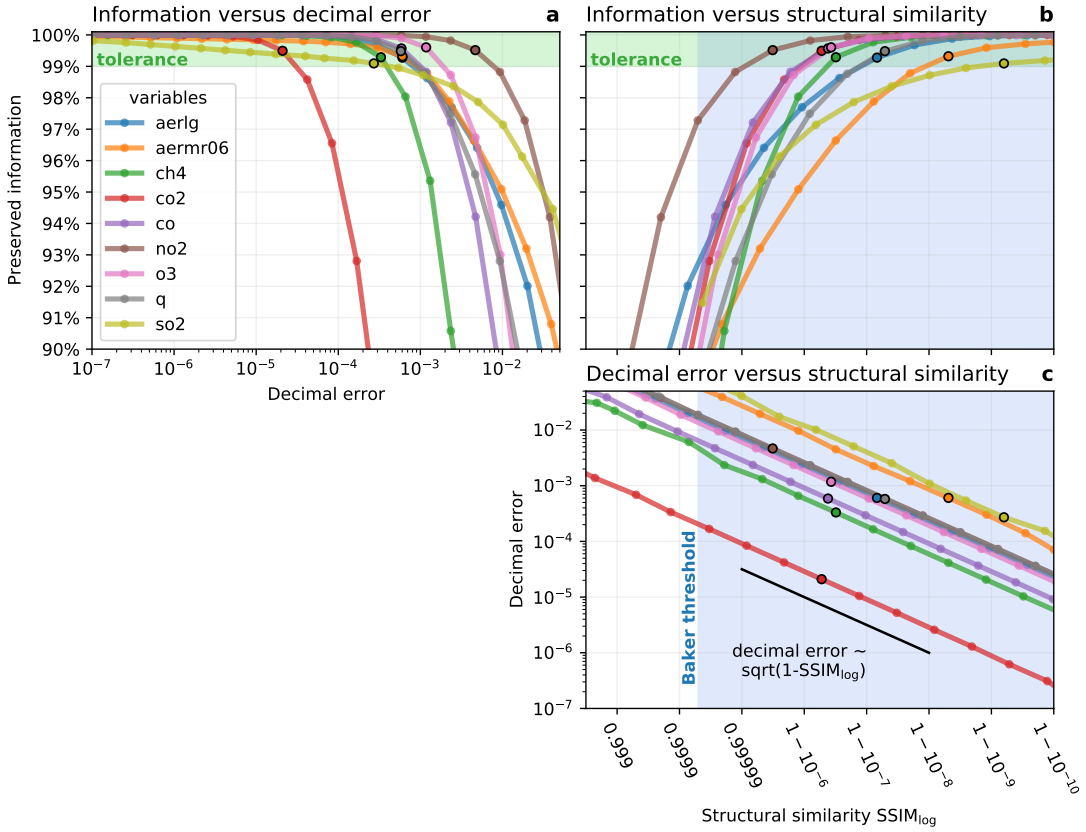

Figure 3.4 | The relationships between preserved information, decimal error and structural similarity for rounding within the information-preserving compression. **a** The last 1% of information tends to be distributed across many mantissa bits such that a trade-off arises where a large increase in compressibility is achieved for a small tolerance in information loss. The preserved information is presented as a function of the decimal error, which itself increases exponentially for every additional bit (small circles) that is discarded due to rounding. Denoted circles present the number of mantissa bits that have to be retained during compression to preserve at least 99% of information. **b** The preserved information increases as a function of the structural similarity (SSIM, Wang *et al.* [2004]). The proposed threshold for climate data of SSIM=0.99995 by Baker *et al.* [2019] is shaded. All variables are very close or above the Baker threshold when preserving 99% of information. **c** The decimal error is proportional to the square root of the structural dissimilarity 1-SSIM for binary rounding within the information-preserving compression

The preserved information as defined here can be used as a compression quality metric similar to the SSIM. When preserving 99% of real information the $\mathrm{SSIM}_{\log}$ is also above the Baker threshold (Fig. 3.4b), reassuring that our threshold of 99% preserved real information is reasonable. In general, the preserved information is a monotonic function of the structural similarity SSIM (or $\mathrm{SSIM}_{\log}$) for rounded floating-point arrays, further supporting the usage of preserved information as a metric for data compression.

3.3.9 Linear and logarithmic quantization

The n-bit linear quantization compression for each element a in an array A is

$$\tilde{a} = \mathrm{round}\left(2^{n-1}\frac{a - \min(A)}{\max(A) - \min(A)}\right) \tag{3.14}$$

with round a function that rounds to the nearest integer in $0, \ldots, 2^{n-1}$. Consequently, every compressed element $\tilde{a}$ can be stored with n bits. The n-bit logarithmic quantization compression for every element $a \geq 0$ in A is

$$\tilde{a} = \begin{cases} 0 & \text{if } a = 0, \\ \mathrm{round}\left(c + \Delta^{-1}\log(a)\right) + 1 & \text{else.} \end{cases} \tag{3.15}$$

which reserves the bit pattern zero to encode 0. The logarithmic spacing is

$$\Delta = \frac{\log(\max(A)) - \log(\min^{+}(A))}{2^n - 2} \tag{3.16}$$

The constant $c = 1/2 - \Delta^{-1}\log(\min^{+}(A)(\exp(\Delta)+1)/2)$ is chosen to implement round-to-nearest in linear space instead of in logarithmic space, for which $c = -\Delta^{-1}\log(\min^{+}(A))$. The function $\min^{+}(A)$ is the minimum of all positive elements in A. For a more general discussion on round-to-nearest for logarithmic numbers formats see section 2.2.3.

3.3.10 Lossless compression

We use Zstandard as a default lossless algorithm for the round+lossless method. Zstandard is a modern compression algorithm that combines many techniques to form a single compressor with tunable 22 compression levels that allow large trade-offs between compression speed and factors [Collet, 2020; Skibinski, 2020]. Here, we use compression level 10, as it presents a reasonable compromise between speed and size. Zstandard outperforms other tested algorithms (deflate, LZ4, LZ4HC and Blosc) in our appli-

cations and is also found to be among the best in the lzbench compression benchmark [Skibinski, 2020] and other studies have focused on comparisons [Delaunay *et al.*, 2019]. Lossless compressors are often combined with reversible transformations that preprocess the data. The so-called bitshuffle transposes an array on the bit-level, such that bit positions (e.g. the sign bit) of floating-point numbers are stored next to each other in memory. Another example is the bitwise XOR-operation [Pelkonen *et al.*, 2015] with the preceding floating-point value, which sets subsequent bits that are identical to 0. Neither bitshuffle nor XOR significantly increased the compression factors in our applications.

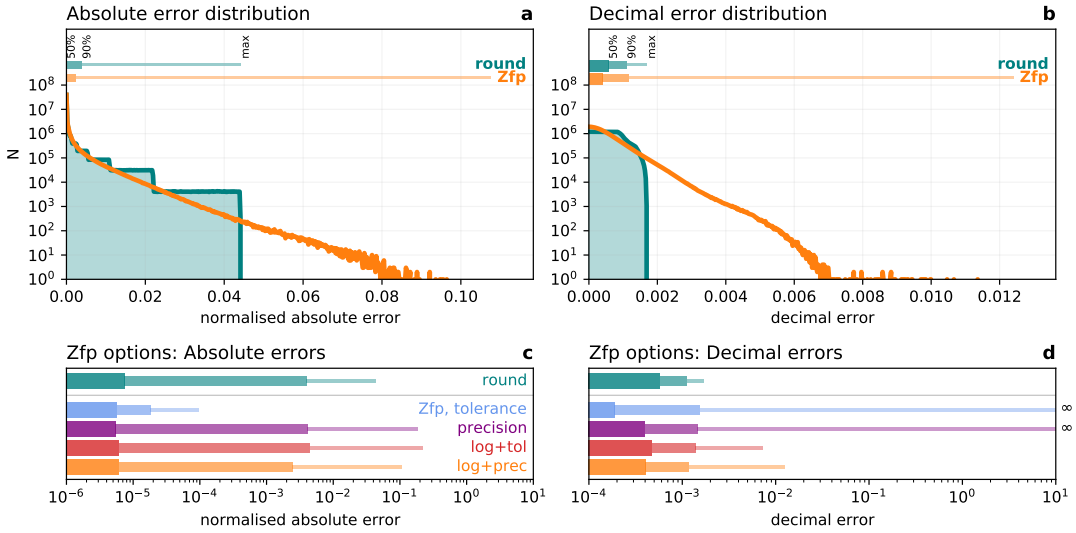

Figure 3.5 | Error distribution of binary rounding compared to Zfp compression. IEEE round-to-nearest and Zfp compression of water vapour (specific humidity) in the three spatial dimensions. **a, c** normalised absolute errors **b, d** decimal errors. 7 mantissa bits are retained for rounding corresponding to 99% preserved information. The precision parameter of Zfp is chosen to yield median errors that are at least as small as those obtained by rounding. **c, d** Zfp via specifying tolerance (tol) or precision (prec) with and without log-preprocessing. Maximum decimal errors that reached infinity in **d** due to sign changes are marked.

3.3.11 Matching retained bits to Zfp's precision

The Zfp compression algorithm divides a d-dimensional array into blocks of size 4^d to exploit correlation in every dimension of the data. Within each block a transformation of the data is applied with specified absolute error tolerance or precision, which bounds a local relative error. We use Zfp in its precision mode, which offers discrete levels to manually adjust the retained precision. Due to the rather logarithmic distribution of

CAMS data (Fig. 3.7), a log-preprocessing of the data is applied to prevent sign changes (including a flushing to zero) within the compression. The error introduced by Zfp is approximately normally distributed and therefore usually yields higher maximum errors compared to round-to-nearest in float arithmetic, although median errors are comparable. In order to find an equivalent error level between the two methods, we therefore choose the precision level of Zfp to yield median absolute and decimal errors that are at least as small as those from rounding.

This method is illustrated in Fig. 3.5 in more detail: Errors introduced from round-to-nearest for floats have very rigid error bounds, the majority of errors from Zfp compression are within these bounds when matching median errors. However, given the normal distribution of errors with Zfp, there will be a small share of errors that is beyond the bounds from round-to-nearest. Using the precision mode of Zfp and log-preprocessed data bounds these maximum errors well (Fig. 3.5c and d).

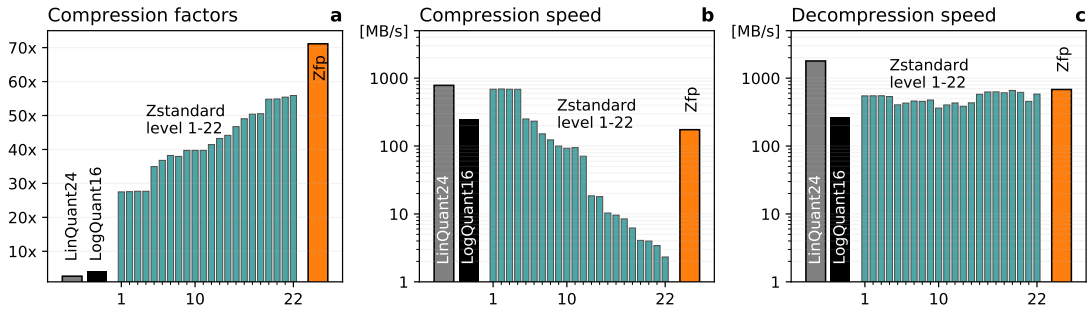

Figure 3.6 | Compressor performances. Compressing water vapour (specific humidity, variable code q) (3 mantissa bits retained, as in Fig. 3.11) with 24-bit linear quantization (LinQuant24), 16-bit logarithmic quantization (LogQuant16), round+lossless (Zstandard, compression level 1-22) and Zfp (precision-mode, including log-preprocessing): **a** Compression factors, **b** compression speed, **c** decompression speed. Timings are single-threaded on an Intel Core™ i7 (Kaby Lake) and do not include the writing to disk.

3.3.12 Compressor performances

Although different compressors and their performance are not within the central focus of this chapter, we analyse the compression and decompression speeds as a sanity check (Fig. 3.6). In order to find a data compression method that can be used operationally, a certain minimum data throughput should be achieved. The current 24-bit linear quantization method reaches compression speeds of almost 800 MB/s single-threaded on an Intel Core™ i7 (Kaby Lake) CPU in our application, excluding writing to disk. For the logarithmic quantization, this decreases to about 200 MB/s due to the ad-

ditional evaluation of a logarithm for every value. For Zstandard the user can choose between 22 compression levels, providing a trade-off between the compression speed (highest for level 1) and the compression factor (highest for level 22). Compression speed reduces from about 700 MB/s at compression level 1 to 2 MB/s at level 22 (Fig. 3.6b), such that for high compression factors about a thousand cores would be required in parallel to compress in real time the 2GB/s data production at ECMWF. For Zstandard at compression level 10 speeds of at least 100MB/s are achieved, but at the cost of about 50% larger file sizes. We use compression level 10 throughout this chapter as a compromise. The decompression speed is independent of the level (Fig. 3.6c). The additional performance cost of binary rounding is with 2 GB/s negligible. Zfp reaches compression speeds of about 200 MB/s (single-threaded, including the log-preprocessing) in our application, enough to compress ECMWF's data production in real time with a small number of processors in parallel.

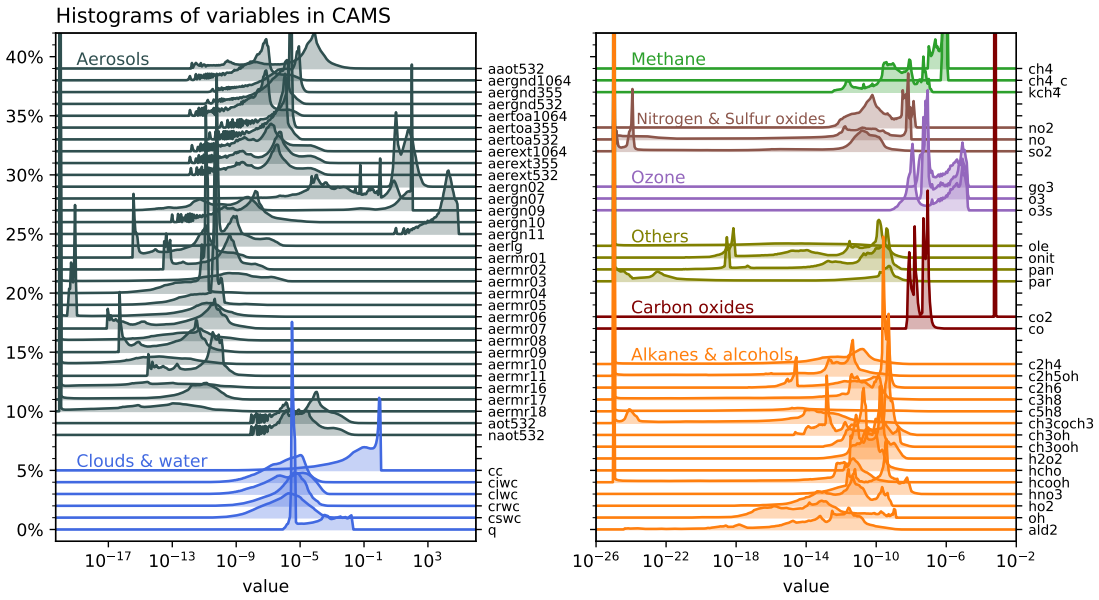

Figure 3.7 | Statistical distributions of all variables in CAMS. Histograms use a logarithmic binning and are staggered vertically for clarity. The variable abbreviations are explained in Table 3.1.

3.4 Results

3.4.1 Drawbacks of current compression methods

The Copernicus Atmospheric Monitoring Service (CAMS, Inness *et al.* [2019]) is performing operational predictions with an extended version of the Integrated Forecasting System IFS, the global atmospheric forecast model implemented by ECMWF. CAMS includes various atmospheric composition variables, like aerosols, trace and greenhouse gases that are important to monitor global air quality. The system monitors for example the spread of volcanic eruptions or emissions from wildfires. Most variables in CAMS have a multi-modal statistical distribution, spanning many orders of magnitude (Fig. 3.7).

The current compression technique for CAMS is the linear quantization, widely used in the weather and climate community through the data format GRIB2 [WMO, 2003]. CAMS uses the 24-bit version, which encodes values in a data array with integers from 0 to $2^{24} - 1$ (see section 3.3.9). These 24-bit unsigned integers represent values linearly distributed in the min-max range. Unused sign or exponent bits from the floating-point representation are therefore avoided and some of the trailing mantissa bits are discarded in quantization. Choosing the number of bits for quantization determines the file size, but the precision follows implicitly, leaving the required precision or amount of preserved information unassessed.

Although linear quantization bounds the absolute error, its linear distribution is unsuited for most variables in CAMS: Many of the available 24 bits are effectively unused as the distribution of the data and the quantized values match poorly (Fig. 3.8). Alternatively, placing the quantized values logarithmically in the min-max range better resolves the data distribution. As floating-point numbers are already approximately logarithmically distributed, this motivates compression directly within the floating-point format, which is also used for calculations in a weather or climate model and post-processing.

3.4.2 Bitwise real information content

Many of the trailing mantissa bits in floating-point numbers occur independently and at similar probability, i.e. with high information entropy [Jeffress *et al.*, 2017; Kleeman, 2011]. These seemingly random bits are incompressible [Huffman, 1952; MacKay, 2003; Ziv & Lempel, 1977], reducing the efficiency of compression algorithms. However, they probably also contain a vanishing amount of real information, which has to be analysed to identify bits with and without real information. The former should be conserved while

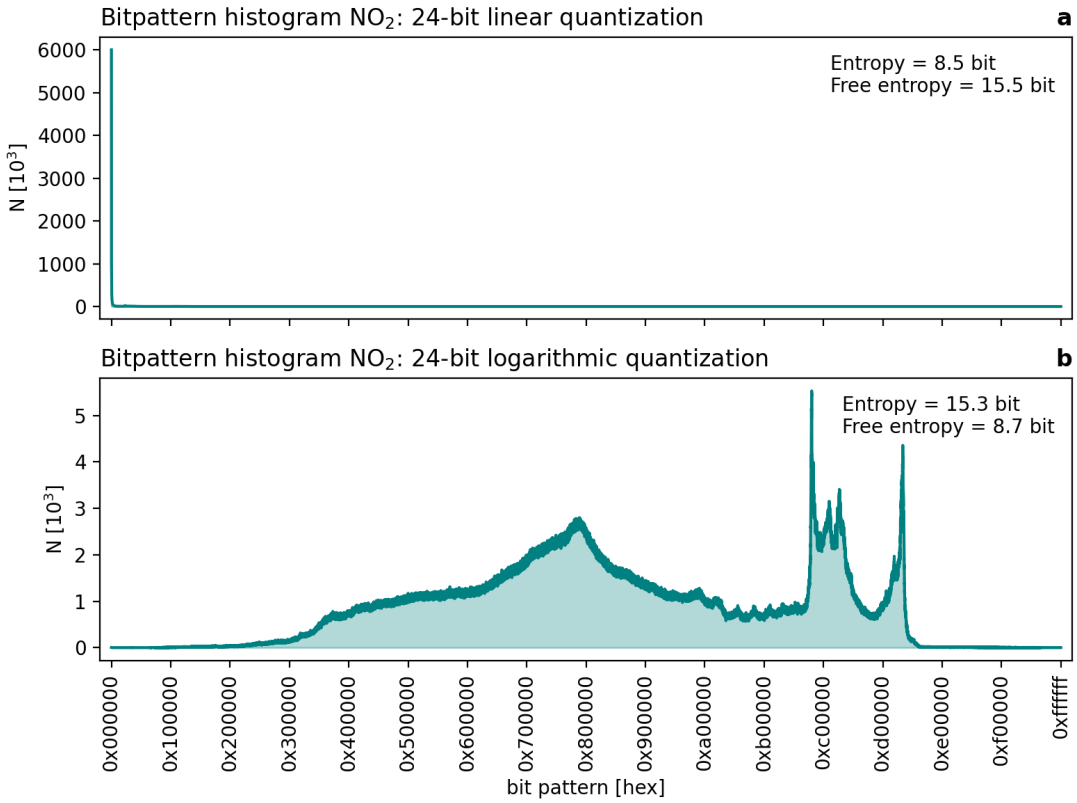

Figure 3.8 | Bitpattern histogram for linear and logarithmic quantization. a Linear 24-bit quantization and **b** 24-bit logarithmic quantization of nitrogen dioxide NO_2 mixing ratio [kg/kg]. All grid points and all vertical levels are used, consisting of $5.6 \cdot 10^7$ values with a range of $2 \cdot 10^{-14}$ to $2 \cdot 10^{-7}$ kg/kg. Bitpatterns are denoted in 24-bit hexadecimal. The free entropy is the difference between the available 24 bit and the bitpattern entropy and quantifies the number of effectively unused bits.

the latter should be discarded to increase compression efficiency.

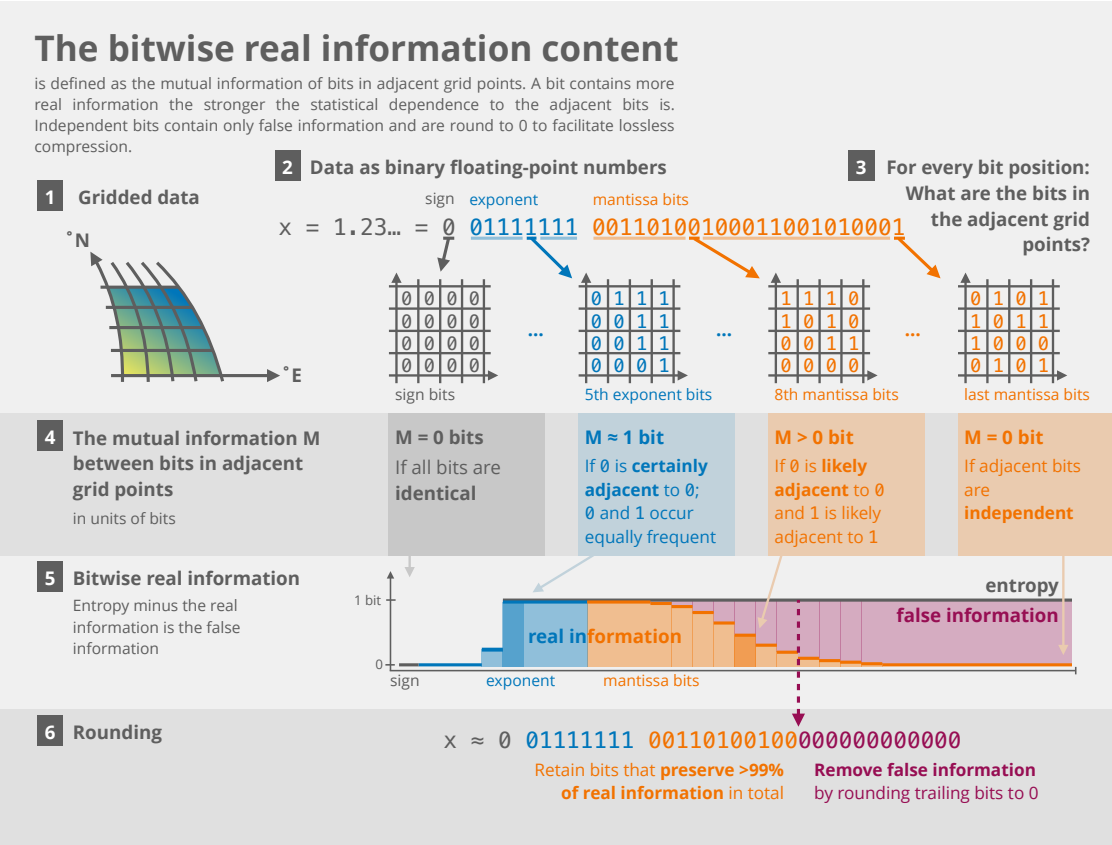

Figure 3.9 | The bitwise real information content explained schematically.

We define the bitwise real information content as the mutual information (DelSole [2004]; Kraskov *et al.* [2004]; MacKay [2003]; Pothapakula *et al.* [2019]; Schreiber [2000]; Shannon [1948] and sections 2.5.4, 3.3.1) of bits in adjacent grid points (Fig. 3.9). A bit contains more real information the stronger the statistical dependence to the adjacent bits is. Bits without real information are identified when this dependence is insignificantly different from zero and we regard the remaining entropy in these bits as false information. The adjacent bit can be found in any of the dimensions of the data, e.g. in longitude, time or in the ensemble dimension. However, always the same bit position is analysed, e.g. the dependence of the first mantissa bit with other first mantissa bits in adjacent grid points.

In general, this analysis can be applied to any n-dimensional gridded data array when its adjacent elements are also adjacent in physical space, including structured and unstructured grids. However, data without spatial or temporal correlation at the provided

resolution will be largely identified as false information due to the independence of adjacent grid points (Fig. 3.1 and 3.2). If valuable scientific information is present in such seemingly random data, then the bitwise real information content as defined here is unsuited.

Jeffress *et al.* [2017] formulate the bitwise information content for simple chaotic systems, assuming an inherent natural uncertainty which had to be defined. Their approach aims to enable reduced precision simulations on inexact hardware. Here, we reformulate the bitwise real information as the mutual information in adjacent grid points for the application in climate data compression. The quantization in the floating-point representation is used as an uncertainty, such that no additional assumption on the uncertainty of the underlying data has to be made. While most data compression techniques leave the choice of the retained precision to the user, the analysis here automatically determines a precision from the data itself based on the separation of real and false information bits.

Many exponent bits of the variables in CAMS have a high information content (Fig. 3.10), but information content decreases to zero within the first mantissa bits for most variables. Exceptions occur for variables like carbon dioxide (CO_2) with mixing ratios varying in a very limited range of 0.5-1.5 mg/kg (equivalent to about 330-990ppmv) globally. Due to the limited range, most exponent bits are unused and the majority of the real information is in mantissa bits 2 to 12.

The sum of real information across all bit positions is the total information per value, which is less than 7 bits for most variables. Importantly, the last few percent of total information is often distributed across many mantissa bits. This presents a trade-off where for a small tolerance in information loss many mantissa bits can be discarded, resulting in a large increase in compressibility (Fig. 3.4a). Aiming for 99% preserved information is found to be a reasonable compromise.

3.4.3 Compressing only the real information

Based on the bitwise real information content, we suggest a new strategy for data compression of climate variables: First, we diagnose the real information for each bit position. Afterwards, we round bits with no significant real information to zero, before applying lossless data compression. This allows us to minimise information loss but to maximise the efficiency of compression algorithms.

Bits with no or only little real information (but high entropy) are discarded via binary round-to-nearest as defined in the IEEE-754 standard (see section 2.2.2, IEEE [1985]).

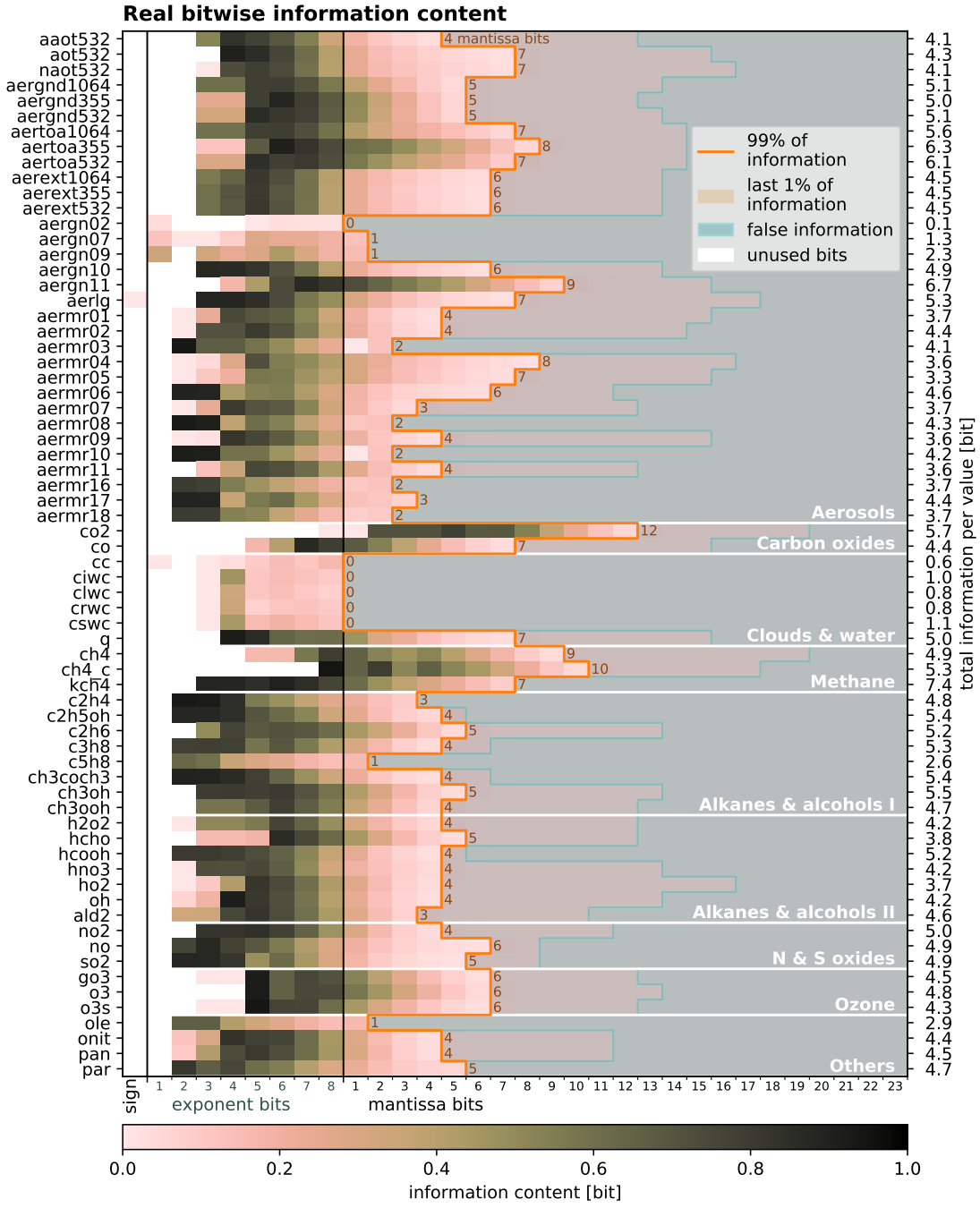

Figure 3.10 | Bitwise real information content for all variables in CAMS. The real information is calculated in all three spatial dimensions, revealing false information and unused bits, using the 32-bit encoding of single-precision floats. The bits that should be retained to preserve 99% of real information are enclosed in orange. Bits without any real information are shaded in grey-blue. The sum of the real information across bit positions per variable is the total information per value. Variable abbreviations are explained in Table 3.1.

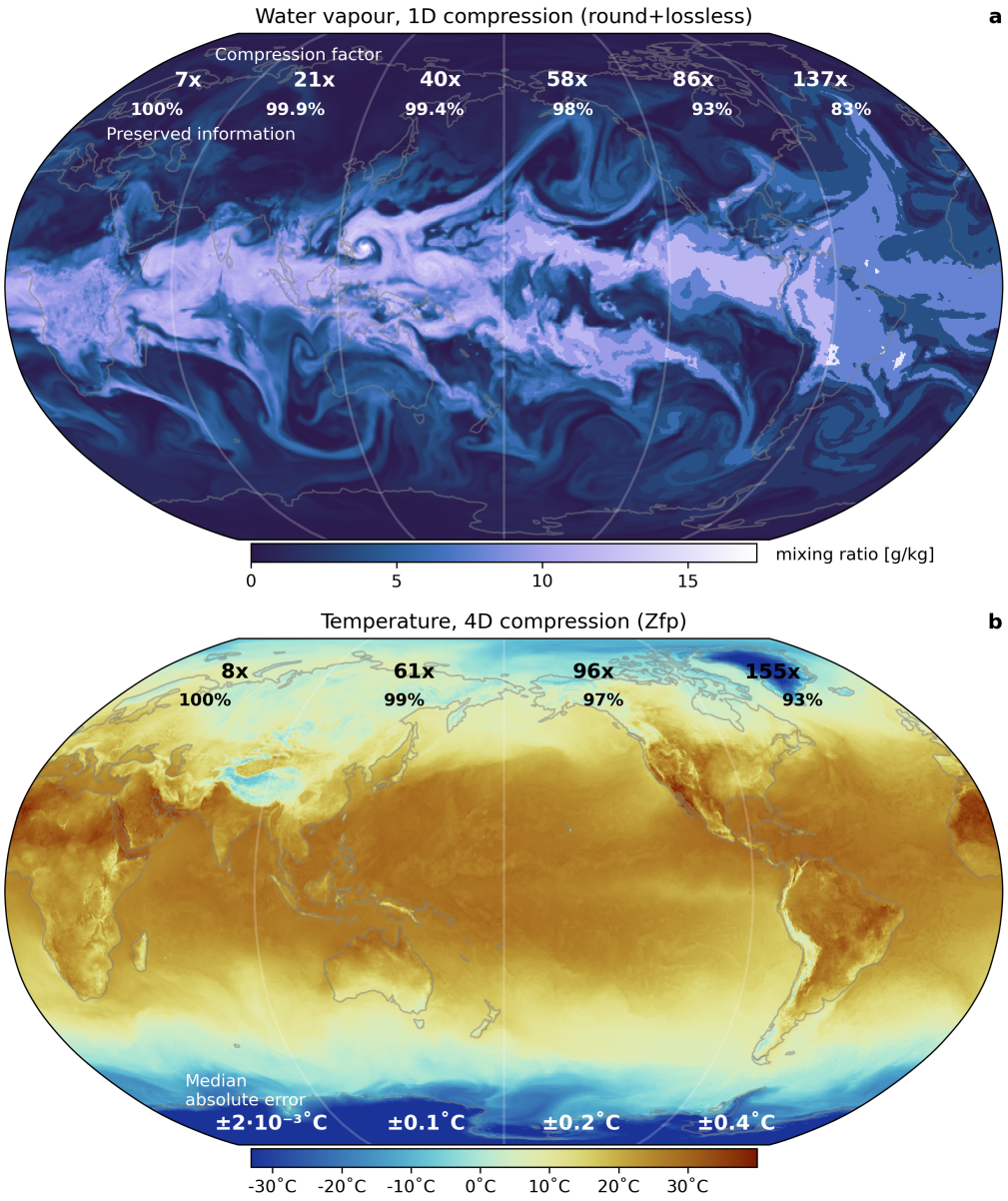

Figure 3.11 | Compression at various levels of preserved information. **a** Water vapour (specific humidity) compressed in the longitudinal dimension. The vertical level shown is at about 2 km geopotential altitude, but compression factors include all vertical levels. **b** Surface temperature compressed in the four space-time dimensions at various levels of preserved information with compression algorithm Zfp. Compression factors are relative to 64-bit floats.

This rounding mode is bias-free and therefore will ensure global conservation of quantities important in climate model data. Rounding removes the incompressible false information and therefore increases compressibility. While rounding is irreversible for the bits with false information, the bits with real information remain unchanged and are bitwise reproducible after decompression. Both the real information analysis and the rounding mode are deterministic, also satisfying reproducibility.

Lossless compression algorithms can be applied efficiently to rounded floating-point arrays (the *round+lossless* method). Many general-purpose lossless compression algorithms are available [Alted, 2010; Collet, 2020; Delaunay *et al.*, 2019; Deutsch, 1996; Huffman, 1952; Skibinski, 2020; Ziv & Lempel, 1977, 1978], which are based on dictionaries and other statistical techniques to remove redundancies. Most algorithms operate on bitstreams and exploit the correlation of data in a single dimension only, we therefore describe this method as 1-dimensional (1D) compression. Here, we use Zstandard for lossless compression [Collet, 2020], which has emerged as a widely available default in recent years (see section 3.3.10).

The compression of water vapour at 100% preserved information (16 mantissa bits are retained) yields a compression factor of 7x relative to 64-bit floats (Fig. 3.11a). At 99% of preserved information (7 mantissa bits are retained) the compression factor increases to 39x. As the last 1% of real information in water vapour is distributed across 9 mantissa bits, we recommend this compromise to increase compressibility. With this compression a 15-fold storage efficiency increase is achieved compared to the current method at 2.67x. Effectively only 1.6 bits are therefore stored per value.

Compressing all variables in CAMS and comparing error norms reveals the advantages of the 1D round+lossless method compared to the 24-bit linear quantization technique currently in use (Fig. 3.12). The maximum decimal errors (see section 2.3.3) are smaller for many variables due to the logarithmic distribution of floating-point numbers. Some variables are very compressible (>60x) due to many zeros in the data, which is automatically made use of in the lossless compression. Compression factors are between 3x and 60x for most variables, with a geometric mean of 6x when preserving 100% of information. Accepting a 1% information loss the geometric mean reaches 17x, which is the overall compression factor for the entire CAMS data set with this method when compared to data storage with 64 bits per value.

Furthermore, the 24-bit linear quantization could be replaced by a 16-bit logarithmic quantization, as the mean and absolute errors are comparable. The decimal errors are often even lower and naturally bound in a logarithmic quantization despite fewer available bits.

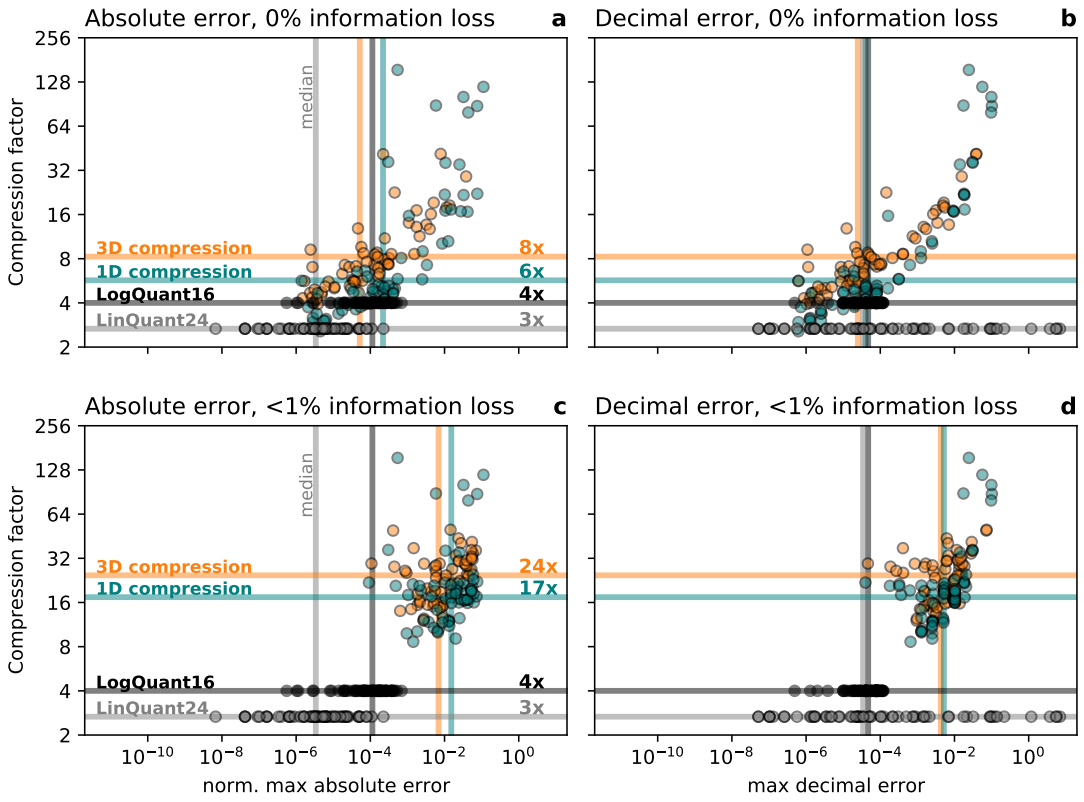

Figure 3.12 | Compression factors versus compression errors. The maximum absolute and decimal error for 24-bit linear and 16-bit logarithmic quantization (LinQuant24, LogQuant16) with 1-dimensional round+lossless and 3-dimensional Zfp compression. Every marker represents for one variable the global maximum of the **a, c** normalised absolute error, **b, d** decimal error for **a, b** 100% preserved information, and **c, d** 99% preserved information. The geometric mean of compression factors over all variables is given as horizontal lines. The median of the errors across all variables is given as vertical lines.

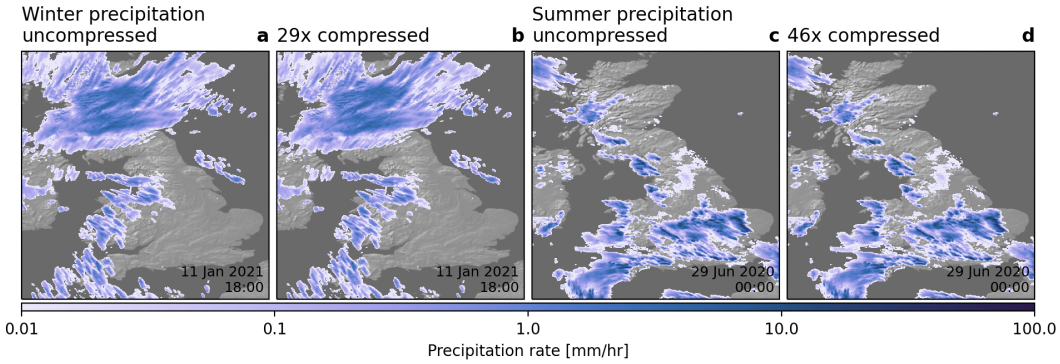

Figure 3.13 | Compression of radar-based observations of precipitation over Great Britain. a Precipitation for the hour preceding 18:00 UTC on 11 Jan 2021 from the UK MetOffice NIMROD data at about 1km horizontal resolution. **b** as a but the data was compressed preserving 99% of real information achieving compression factors of 29x relative to 64 bit. **c** and **d** as **a** and **b** but for 00:00 UTC on 29 Jun 2021 and achieving compression factors of 46x.

A broad applicability of the bitwise real information content analysis for compression is tested with further data sets: Radar-based observations of precipitation over Great Britain are similarly compressible using the same method (Fig. 3.13) and so are satellite measurements of brightness temperature with a very high resolution of about 300m horizontally (Fig. 3.14). Even for anthropogenic emissions of methane or nitrogen dioxide similar compression results are obtained, despite limited spatial correlation of point sources (Fig. 3.15). The bitwise real information content in this case is largely determined by the smooth background concentrations and therefore still sufficiently high to preserve the point sources.

In an operational setting we recommend the following workflow: First, for each variable the bitwise real information content is analysed from a representative subset of the data. Representative is, for example, a single time step for subsequent time steps if the statistics of the data distribution are not expected to change. From the bitwise real information the number of mantissa bits to preserve 99% of information is determined (the *keepbits*). Second, during the simulation the arrays that will be archived are rounded to the number of keepbits (which are held fixed) and compressed. The first step should be done offline, meaning once in advance of a data-producing simulation. Only the second step has to be performed online, meaning every time data is archived.

The presented round+lossless compression technique separates the lossy removal of false information and the actual lossless compression. This provides additional flexibilities as any lossless compressor can be used and application-specific choices can be made regarding availability, speed and resulting file sizes. However, most general-

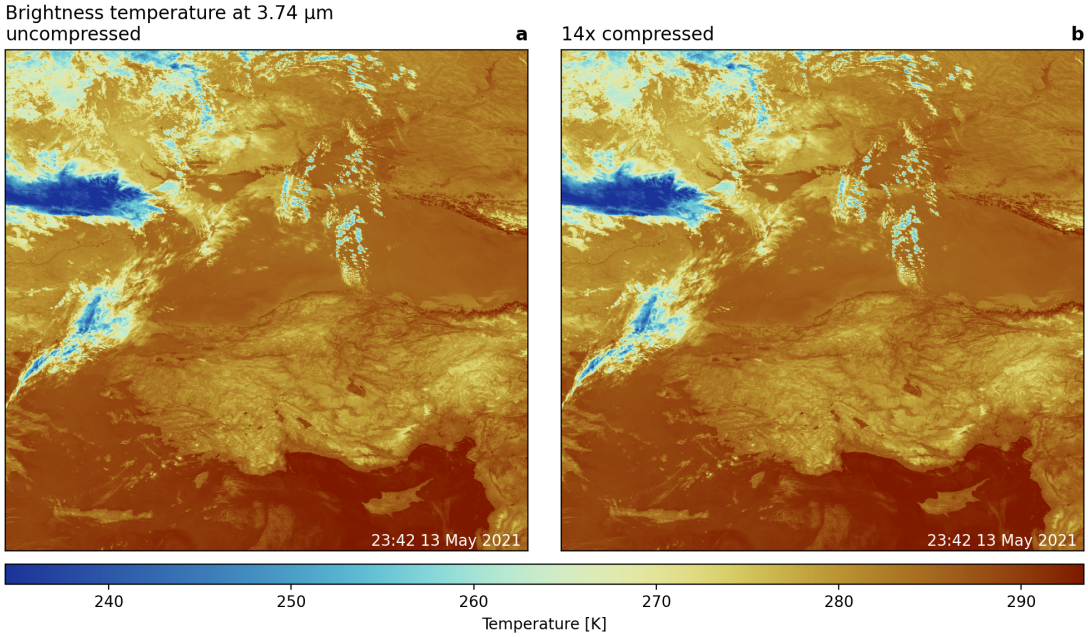

Figure 3.14 | Compression of satellite-based observations of brightness temperature over the Black Sea and Turkey. **a** Brightness temperature measured by the 3.74µm (I4) channel of the VIIRS sensor on board the Suomi-NPP satellite at about 300m horizontal resolution on the 13 May 2021. **b** as **a** but the data was compressed preserving 99% of real information with the round+lossless method achieving compression factors of 14x relative to 64 bit.

purpose lossless compression algorithms operate on bitstreams and require multidimensional data to be unravelled into a single dimension. Multidimensional correlation is therefore not fully exploited in this approach.

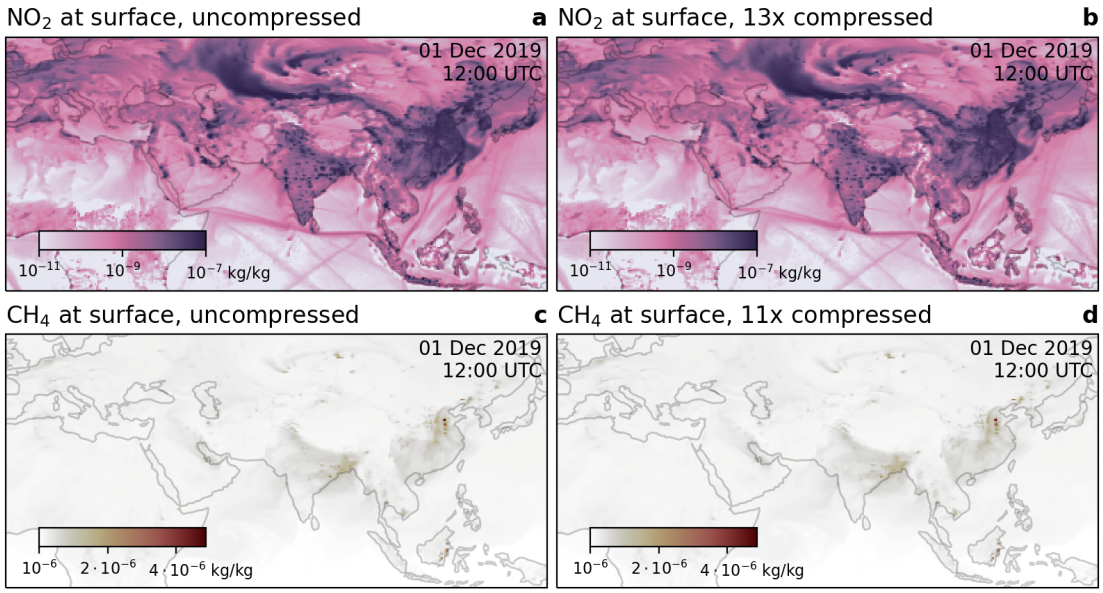

Figure 3.15 | Compression of nitrogen dioxide (NO_2) and methane (CH_4) at the surface. **a** Surface NO_2 concentrations preliminary result from fossil fuel combustion. **b** Surface CH_4 concentrations often include point sources, such as here in East China, East India and East Borneo. **b,d** as **a,c** but compressed preserving 99% of information achieving a compression factor of 13x, 11x, respectively.

We extend the ideas of information-preserving compression to modern multidimensional compressors. The analysis of the bitwise real information content leads naturally to the removal of false information via rounding in the round+lossless method. For other lossy compressors, however, the separation of real and false information has to be translated to the precision options of such compressors. While such a translation is challenging in general, we present results from combining the bitwise real information analysis with one modern multidimensional compressor in the next section.

3.4.4 Multidimensional climate data compression

Modern compressors have been developed for multidimensional floating-point arrays [Ballester-Ripoll *et al.*, 2020; Di & Cappello, 2016; Lindstrom & Isenburg, 2006; von Larcher & Klein, 2019; Zhao *et al.*, 2020], which compress in several dimensions simultaneously. We will compare the 1D round+lossless compression to Zfp, a modern compression al-

gorithm for two to four dimensions [Hammerling *et al.*, 2019; Lindstrom, 2014; Pinard *et al.*, 2020; Poppick *et al.*, 2020]. Zfp divides a d-dimensional array into blocks of 4^d values (i.e. the edge length is 4), which allows to exploit the correlation of climate data in up to 4 dimensions. To extend the concept of information-preserving compression to modern compressors like Zfp, the bitwise real information is translated to the precision options of Zfp (more details in section 3.3.11).

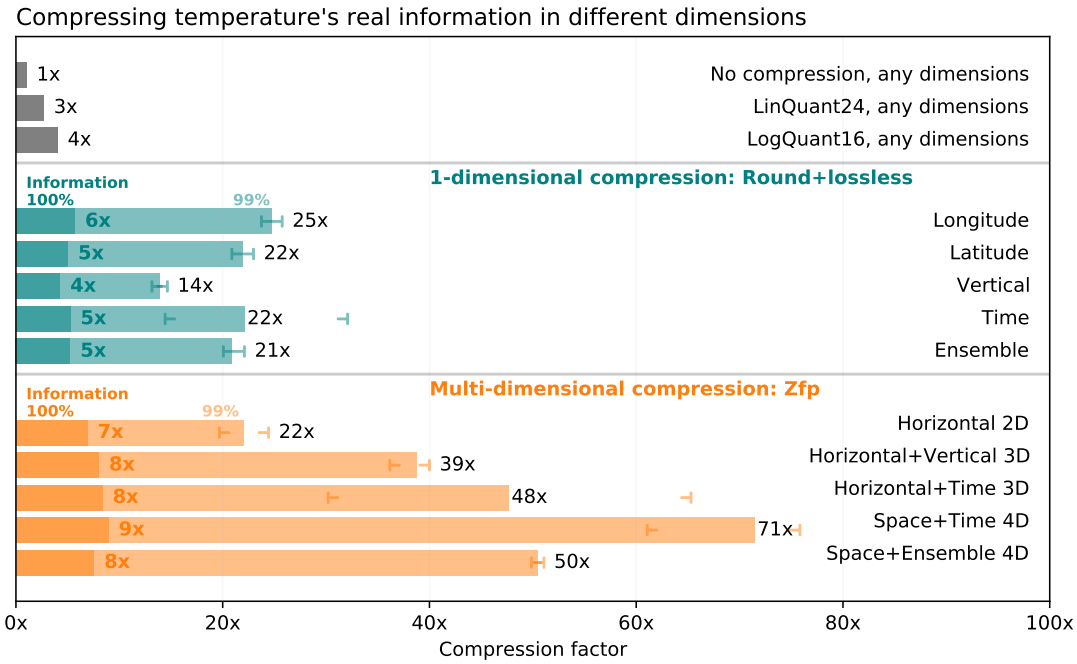

Figure 3.16 | Multidimensional compression allows for higher compression factors. 1-dimensional compression (round+lossless) of temperature reaches at most 25x when preserving 99% of real information with the round+lossless method, whereas 71x is reached with 4-dimensional (4D) space-time compression using Zfp compression. Preserving 100% of information considerably lowers the compression factors to 4-9x. Error brackets represent the min-max range of compression when applied to various data samples.

Multidimensional compression imposes additional inflexibilities for data retrieval: Data is compressed and decompressed in larger chunks, which can increase the load on the data archive. For example, if the data is compressed across the time dimension, data of several time steps have to be downloaded and decompressed although only data from a single time step might be requested. Downloads from an archive might therefore increase if the data chunking is not well suited to typical data requests from users.

For 1D compression the compressibility varies with the dimension: Longitude (i.e.

in the zonal direction) is more compressible, reaching 25x for temperature at 99% preserved information, than compressing in the vertical which yields only 14x (Fig. 3.16). This agrees with the predominantly zonal flow of the atmosphere as spatial correlation in the zonal direction is usually highest. For a constant number of retained mantissa bits, higher resolution in the respective dimensions increases the compressibility as also the correlation in adjacent grid points increases (Fig. 3.1 and 3.2).

For multidimensional compression it is generally advantageous to include as many highly correlated dimensions as possible. In that sense, including the hourly-resolved forecast lead time instead of the vertical dimension in 3D compression yields higher compression factors. 4D space-time compression is the most efficient, reaching 60-75x at 99% preserved information. For temperature this is equivalent to a median absolute error of 0.1°C (Fig. 3.11b).

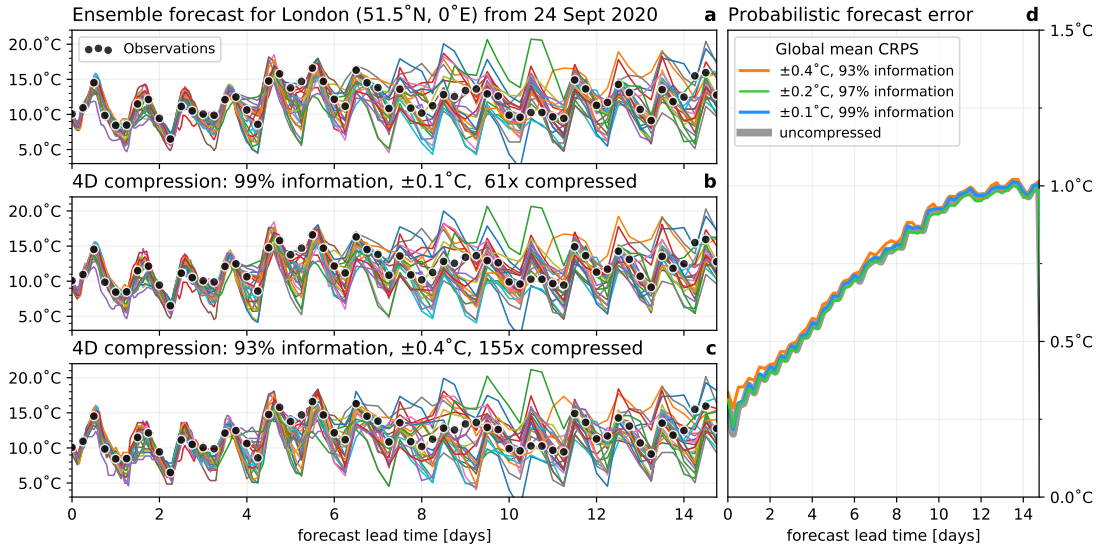

Figure 3.17 | Verification of an ensemble forecast with the probabilistic forecast error based on ensemble data with and without compression. a 25-member uncompressed ensemble forecast (lines) of surface temperature in London, UK from 24 Sept 2020 up to 15 days ahead. **b** as a but the data was compressed in 4-dimensional (4D) space-time with Zfp, preserving 99% of real information. **c** as **b** but only preserving 93% of real information. **d** Probabilistic forecast error (continuous ranked probability score, CRPS) for various levels of preserved information in the compression. The CRPS does not increases relative to the uncompressed reference for more than 93% of preserved information.

Compressing the entire CAMS dataset in the three spatial dimensions with Zfp while preserving 99% of the information yields an overall compression factor of 24x (Fig. 3.12). Maximum absolute error and decimal errors are for most variables very similar to 1D round+lossless (see Fig. 3.5 and section 3.3.11 for a discussion why they are not identical), providing evidence that a multidimensional compression is preferable for higher

compression factors.

Due to the limited meaning of error norms in the presence of uncertainties in the uncompressed reference data, the forecast error is assessed to quantify the quality of compressed atmospheric data. The continuous ranked probability score (CRPS, Hersbach [2000]; Matheson & Winkler [1976]; Zamo & Naveau [2018]), a generalisation of the root-mean-square error for probabilistic forecasts, is evaluated for global surface temperature using observations every 6 hours as truth (Fig. 3.17). Compared to the uncompressed data, no significant increase of the CRPS forecast error occurs for individual locations or globally at 99% and 97% preserved information. The usefulness for the end user of the global temperature forecast is therefore unaltered at these levels of preserved information in the compression. Contrarily, with an information loss larger than 5% the CRPS forecast error starts to increase, while large compression factors beyond 150x are achieved.

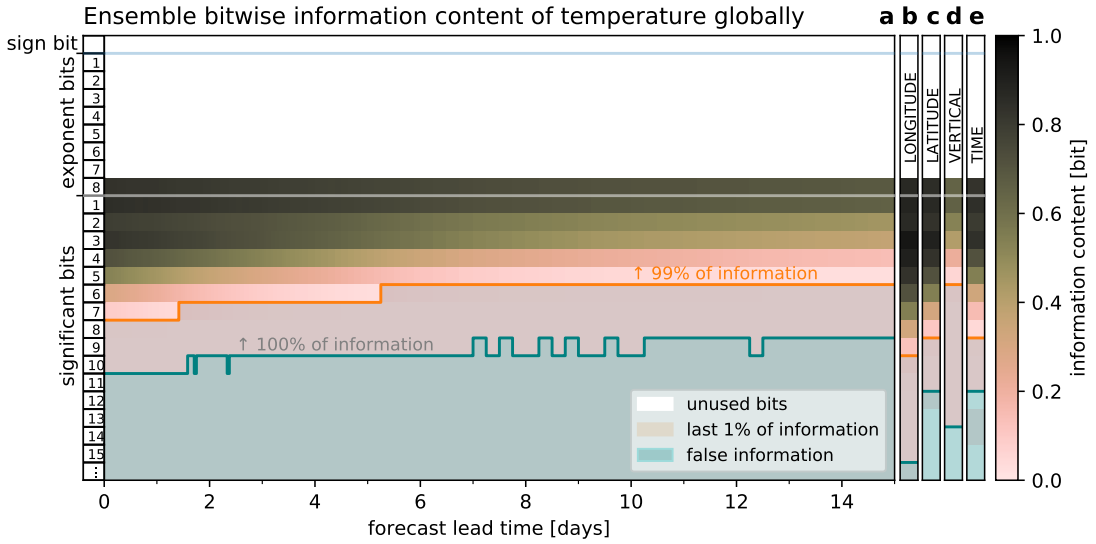

Figure 3.18 | Bitwise real information content for temperature in various dimensions. **a** ensemble, **b** longitude, **c** latitude, **d** vertical and **e** forecast lead time. The ensemble information effectively encodes the ensemble mean, which is less information than in most other dimensions. Longitude, latitude and forecast lead time have the highest total information which should be preserved in compression. The ensemble information decreases over time as the ensemble spread increases (Fig. 3.17).

3.4.5 A Turing test for data compression

In numerical weather predictions, progress in the development of global weather forecasts is often assessed using a set of error metrics, summarised in so-called *score cards*.

These scores cover important variables in various large scale regions, such as the temperature 2m above the surface over Europe or horizontal wind speed at different vertical levels in the Southern Hemisphere. With a similar motivation as in Baker *et al.* [2019], we suggest assessing the efficiency of climate data compression using similar scores, which have to be passed similar to a Turing test [Baker *et al.*, 2016; Turing, 1950]. The compressed forecast data should be indistinguishable from the uncompressed data in all of these score tests, or at least indistinguishable from the current compression method while allowing higher compression factors. Many score tests currently in use represent area-averages (such as Fig. 3.17d), which would also be passed with coarse-grained data – reducing the horizontal resolution from 10km to 20km, for example, yields a compression factor of 4x. It is therefore important to include resolution-sensitive score tests such as the maximum error in a region.

A Turing test for data compression could be used iteratively to optimise the level of preserved information in compression. Such an iteration could be formulated as follows

1. Create 4 seemingly identical data sets with

   `FULL` Full precision, i.e. no lossy compression was applied

   `CURR` Current compression method

   `INF0` Conservative information-preserving compression, e.g 99%

   `INF1` Progressive information-preserving compression, e.g. 98%

2. Disseminate `FULL`, `CURR`, `INF0`, and `INF1` to users, then

   i if `FULL`, `CURR`, `INF0`, and `INF1` are indistinguishable, define `INF1` as the new `INF0` and preserve less information in `INF1`, e.g. 97% instead of 98%

   ii if `CURR` is identified as least useful, conclude that information-preserving should be considered as the new default compression method and continue to i

   iii if `INF1` or `INF0` are identified as least useful, define `INF0` as the new `INF1` and preserve more information in `INF0`, e.g . 99.5% instead of 99%

3. Reiterate from 1.

While a compression method either passes or fails a conventional Turing test, there is additional value in conducting such a test. Evaluating the failures will highlight problems and evaluating the passes may identify further compression potential. An iterative Turing test as presented here can be used to translate an abstract property of data such as usefulness to users into a compression level (here % of preserved information).

3.5 Discussion

While weather and climate forecast centres produce very large amounts of data, especially for future cloud and storm-resolving models, only the real information content in this data should be stored. We have here presented a methodology to identify real and false information in atmospheric, and more generally, climate data. This novel information-preserving compression relies on the removal of false information via rounding and can then be used in combination with any lossless compression algorithm. Applied to CAMS data we show that a high compressibility can be achieved without increasing the forecast error. The entire data set is 17x smaller in the compressed form when compared to 64-bit values but preserves 99% of the real information. This is about 6-times more efficient when compared to the current compression method.

Alternatively, the analysis of the bitwise real information content can be used to inform multidimensional compressors. Ideally, climate data compression should exploit correlation in as many dimensions as possible for highest compression factors. The most important dimensions to compress along are longitude, latitude and time, which provide the highest compressibility. With Zfp we achieve factors of 60-75x for 4D space-time compression of temperature while preserving 99% of real information and without increasing forecast errors. Using the three spatial dimensions the entire set of variables in CAMS data can be compressed by 24x equivalently.

No additional uncertainty measure has to be assumed for the distinction of real and false information presented here. The uncertainty of a variable represented in a data array is directly obtained from the distribution of the data itself. Most lossy compression techniques leave the choice of precision to the user, which may lead to subjective choices or the same precision for a group of variables. Instead, our suggestion that 99% of information should be preserved may be altered by the user, which will implicitly determine the required precision for each variable individually.

To be attractive for large data sets, a compression method should enable compression as well as decompression at reasonable speeds. ECMWF produces data at about 2GB/s, including CAMS which creates about 15 MB/s. Data on ECMWF's archive is compressed once, but downloaded on average at 120 MB/s by different users, such that both high compression and decompression speeds are important. The (de)compression speeds obtained here are all at least 100MB/s single-threaded, but faster speeds are available in exchange for lower compression factors (Fig. 3.6). The real information is only analysed once and ultimately independent of the compressor choice.

Lossy compression inevitably introduces errors compared to the uncompressed data.

Weather and climate forecast data, however, already contains uncertainties which are in most cases larger than the compression error. For example, limiting the precision of surface temperature to 0.1°C (as shown in Fig. 3.11b) is well below the average forecast error (Fig. 3.17d) and also more precise than the typical precision of 1°C presented to end users of a weather forecast. Reducing the precision to the real information content does not just increase compressibility but also helps to directly communicate the uncertainty within the data set — an important, often neglected, information by itself.

Satisfying requirements on size, precision and speed simultaneously is an inevitable challenge of data compression. As the precision can be reduced without losing information, we revisit this trade-off and propose an information-preserving compression. While current archives likely use large capacities to store random bits, the analysis of the bitwise real information content is essential towards efficient climate data compression.

3.6 Appendix

Name	Code	Unit	Name	Code	Unit
Aerosols			**Carbon oxides**		
Aerosol optical thickness 532nm	aott532	1	Carbon dioxide	co2	kg/kg
Anthropogenic aot532	aaot532	1	Carbon monoxide	co	kg/kg
Natural aot532	naot532	1	**Clouds and water**		
Backscatter from ground at 1064nm	aergnd1064	$m^{-1}sr^{-1}$	Fraction of cloud cover	cc	1
Backscatter from ground at 355nm	aergnd355	$m^{-1}sr^{-1}$	Cloud ice water	ciwc	kg/kg
Backscatter from ground at 532nm	aergnd532	$m^{-1}sr^{-1}$	Cloud liquid water	clwc	kg/kg
Backscatter from top of atm at 1064nm	aertoa1064	$m^{-1}sr^{-1}$	Specific rain water	crwc	kg/kg
Backscatter from top of atm at 532nm	aertoa355	$m^{-1}sr^{-1}$	Specific snow water	cswc	kg/kg
Backscatter from top of atm at 532nm	aertoa532	$m^{-1}sr^{-1}$	Specific humidity	q	kg/kg
Aerosol extinction coefficient at 1064nm	aerext1064	m^{-1}	**Methane**		
Aerosol extinction coefficient at 355nm	aerext355	m^{-1}	Methane	ch4	kg/kg
Aerosol extinction coefficient at 532nm	aerext532	m^{-1}	Methane (chemistry)	ch4_c	kg/kg
Aerosol type 2 source/gain accum.	aergn02	kg/m^2	Methane loss rate	kch4	s^{-1}
Aerosol type 7 source/gain accum.	aergn07	kg/m^2	**Alkanes or alcohols**		
Aerosol type 9 source/gain accum.	aergn09	kg/m^2	Ethene	c2h4	kg/kg
Aerosol type 10 source/gain accum.	aergn10	kg/m^2	Ethanol	c2h5oh	kg/kg
Aerosol type 11 source/gain accum.	aergn11	kg/m^2	Ethane	c2h6	kg/kg
Aerosol large mode mixing ratio	aerlg	kg/kg	Propane	c3h8	kg/kg
Sea salt (0.03-0.5µm)	aermr01	kg/kg	Isoprene	c5h8	kg/kg
Sea salt (0.5-5µm)	aermr02	kg/kg	Acetone	ch3coch3	kg/kg
Sea salt (5-20µm)	aermr03	kg/kg	Methanol	ch3oh	kg/kg
Dust aerosol (0.03-0.55µm)	aermr04	kg/kg	Methyl peroxide	ch3ooh	kg/kg
Dust aerosol (0.55-0.9µm)	aermr05	kg/kg	Hydrogen peroxide	h2o2	kg/kg
Dust aerosol (0.9-20µm)	aermr06	kg/kg	Formaldehyde	hcho	kg/kg
Hydrophilic organic matter	aermr07	kg/kg	Formic acid	hcooh	kg/kg
Hydrophobic organic matter	aermr08	kg/kg	Nitric acid	hno3	kg/kg
Hydrophilic black carbon	aermr09	kg/kg	Hydroperoxy radical	ho2	kg/kg
Hydrophobic black carbon	aermr10	kg/kg	Hydroxyl radical	oh	kg/kg
Sulphate aerosol	aermr11	kg/kg	Aldehyde	ald2	kg/kg
Nitrate fine mode	aermr16	kg/kg	**Nitrogen and sulfur oxides**		
Nitrate coarse mode	aermr17	kg/kg	Nitrogen dioxide	no2	kg/kg
Ammonium aerosol	aermr18	kg/kg	Nitrogen monoxide	no	kg/kg
Others			Sulphur dioxide	so2	kg/kg
Olefins	ole	kg/kg	**Ozone**		
Organic nitrates	onit	kg/kg	Ozone mixing ratio 2	go3	kg/kg
Peroxyacetyl nitrate	pan	kg/kg	Ozone mixing ratio 1	o3	kg/kg
Paraffins	par	kg/kg	Stratospheric ozone	o3s	kg/kg

Table 3.1 | Variables, their codes and units in the Copernicus Atmospheric Monitoring Service.

4 Bitwise periodic orbits in chaotic systems

Contributions This chapter is largely based on the following publication[*]

M Klöwer, PV Coveney, EA Paxton, and TN Palmer, 2021. *Bitwise periodic orbits in chaotic systems simulated at low precision*, in preparation.

Abstract. Non-periodic solutions are an essential property of chaotic dynamical systems. Simulations with deterministic finite-precision numbers, however, always yield bitwise periodic orbits. At the high precision of 64-bit floating-point numbers such orbits are usually negligible due to very long periods. The emerging trend to accelerate simulations with low-precision numbers, such as 16-bit half precision floats, raises questions on the fidelity of such simulations of chaotic systems. Here, we revisit the 1-variable chaotic generalised Bernoulli map with floats, posits and logarithmic fixed-point numbers at various levels of precision using deterministic and stochastic rounding. The simulations represent the analytical Bernoulli map generally better the higher the precision of the number format. Stochastic rounding is especially beneficial as it prevents bitwise periodic orbits even at low precision. For the simulation of continuous systems the performance gain from low-precision arithmetic will often be reinvested in higher resolution, increasing the number of variables. Using the Lorenz 1996 system we provide evidence that the period length increases exponentially with the number of variables. Moreover, invariant measures are better approximated, indicating an overall improved simulation, with an increased number of variables than with increased precision. Extrapolating to complex simulations of natural systems, such as climate models with millions of variables, periodic orbit lengths are far beyond reach of present-day computers. Bitwise periodic orbits from low-precision simulations are therefore not expected to be problematic for any system larger than the simplest chaotic systems.

[*]with the following author contributions. Conceptualisation: MK, PVC, EAP. Data curation: MK. Formal Analysis: MK. Methodology: MK. Visualisation: MK. Writing – original draft: MK. Writing – review and editing: MK, PVC, EAP, TNP.

4.1 Introduction

Many natural systems exhibit chaotic dynamics. The chaos in weather prevents reliable forecasts beyond one or two weeks [Bauer *et al.*, 2015; Palmer, 2019b]. The turbulent flow of air or water around vehicles requires complex numerical simulations to optimize the drag [Cummings *et al.*, 2015; Moran, 2003]. Similarly, chaos is present in models of many-body problems from astrophysics [Cornish, 2001; Springel, 2005], chemical reaction networks [Coveney & Wan, 2016] or plasma in fusion reactors [D'Ippolito *et al.*, 2011; ITER Physics Expert Group on Confinement and Transport *et al.*, 1999; Ricci *et al.*, 2012]. As chaotic dynamics often prevent analytical solutions, numerical simulations with finite-precision floating-point numbers [IEEE, 1985] are used to approximate a system's state and predict its future.

However, simulating a deterministic, yet chaotic system with deterministic finite-precision numbers always results in bitwise periodic orbits due to a finite set of possible states [Boghosian *et al.*, 2019]. Such orbits are distinct from the unstable periodic orbits that describe the recurrence of a very similar, but not bitwise identical, state [Cvitanović, 2013]. Eternal periodicity in the simulation of chaotic systems violates a fundamental property of chaos, but periods are very long with the high precision of 64-bit floating-point numbers (Float64).

Computed dynamics have errors relative to analytical solutions. Model errors arise from the difference between the mathematical equations and the natural systems they represent, including unresolved processes and heuristic parameters. Errors in the initial or boundary conditions are a result of imperfect observations being assimilated into the numerical model [Ghil & Malanotte-Rizzoli, 1991]. Discretization errors occur when a continuous system is discretized into a number of variables that is often limited by available computational resources [Butcher, 2016]. In addition, there are rounding errors as a result of using finite-precision numbers to approximate real numbers [Higham, 2002]. In a chaotic system, all of these errors grow exponentially, but the largest sources of error mask smaller ones. Due to often negligible rounding errors, simulations are increasingly accelerating with low-precision calculations in exchange for computational performance [Fuhrer *et al.*, 2018; Nakano *et al.*, 2018; Váňa *et al.*, 2017]. The performance gain is then reinvested into a higher resolution with more independent variables, reducing the model and or discretization error.

16-bit low-precision computations are increasingly supported on modern processors, such as graphics processing units (GPU, Markidis *et al.* [2018]), tensor processing units (TPU, Jouppi *et al.* [2018b]) and also central processing units (CPU, Odajima *et al.*

[2020]; Sato *et al.* [2020]). While the standard and only widely available number format are floats, several alternatives have been proposed: Posits [Gustafson & Yonemoto, 2017]), logarithmic fixed-point numbers [Johnson, 2020], and floats with stochastic rounding [Hopkins *et al.*, 2020]. Currently lacking hardware support, these number formats are emulated in software for precision tests. In comparison to floats they provide a better understanding how the numerical precision affects the simulated dynamics.

Unstable periodic orbits are the skeleton of chaos [Cvitanović, 1991] and have been intensively studied to better understand the dynamical structure of chaotic systems [Lasagna, 2020; Leboeuf, 2004; Ruelle & Takens, 1971]. Chaotic trajectories follow a given periodic orbit in its vicinity but eventually diverge due to the orbit's instability and approach another orbit until diverging again [Maiocchi & Lucarini, 2021]. The spectrum of the periodic orbits is a decomposition of the attractor [Eckmann & Ruelle, 2004]. As numerical simulations are based on finite-precision arithmetic, the spectrum of periodic orbits is degraded, but to which extent is generally unclear. While very low arithmetic precision truncates chaotic attractors to simple loops or fixed points, the degradation can be between less obvious but substantial and irrelevant for sufficiently high precision.

Here, we compare the invariant measures as the statistical properties of two chaotic dynamical systems when simulated with different binary number formats and at various levels of precision. The methodology for finding bitwise periodic orbits is presented in section 4.2. We revisit the simulation of the generalised Bernoulli map with floats, posits, logfix and stochastic rounding in section 4.3 to better understand the effect of numerical precision in a system where the analytical invariant measure is known. In section 4.4 we turn to the N-variable Lorenz 1996 system to investigate the bitwise periodic orbit spectrum with an increasing number of variables. Section 4.5 summarizes and discusses the results.

4.2 Methods

Floating-point numbers are standardized following IEEE, 1985, 2008. Another number format used in this chapter are posits (Gustafson & Yonemoto [2017], and section 2.1.5), which have a slightly higher precision within the powers of 2 around ± 1, yet a wide dynamic range at the cost of a gradually lower precision away from ± 1. While posits have been proposed as a drop-in replacement for floats, they currently lack widely available hardware support. Additionally, there are Logarithmic fixed-point numbers, which

have received little attention apart from research implementations on custom hardware [Johnson, 2020] and are described in section 2.1.4 with our design choices in more detail. Stochastic rounding has recently emerged as an alternative rounding mode to the widely used deterministic round-to-nearest (section 2.2.2, IEEE [1985]), beneficial for scientific computing [Croci & Giles, 2020; Fasi & Mikaitis, 2021; Hopkins *et al.*, 2020; Paxton *et al.*, 2021] and is described in section 2.2.4. An improved pseudo random number generator for floats is presented in 4.2.1 which is used for Monte Carlo-based search for periodic orbits (section 4.2.2). Especially for systems of several variables, the search for periodic orbits becomes computationally very demanding and we outline our approach using distributed computing in section 4.2.3. The agreement between invariant measures is analysed with the Wasserstein distance, which is briefly described in section 4.2.4.

4.2.1 Improved random generation of uniformly distributed floats

Conventional random number generation for a float f from a uniform distribution $U(0,1)$ in $[0,1)$ uses the following technique: First, 10, 23, or 52 random bits (for Float16, Float32, or Float64, respectively) from an unsigned integer are used to set the mantissa bits of floating-point 1. This creates a floating-point number that is uniformly distributed in $[1,2)$ as all float formats are uniformly distributed in that range. Second, 1 is subtracted to obtain a float in $[0,1)$, i.e. $f \sim U(1,2) - 1$. While this approach is fast, it is statistically imperfect as the resulting distribution does not contain all floats in $[0,1)$. There are 2^{23} Float32s in $[1,2)$ but the subtraction maps those only to a subset of all $1065353216 \approx 2^{30}$ Float32s in $[0,1)$. This technique only samples from every second float in $[\frac{1}{2},1)$, every fourth in $[\frac{1}{4},\frac{1}{2})$ and so every $2n$-th float in $[2^{-n-1},2^{-n})$. Furthermore, the smallest positive number that can be obtained is about 10^{-7} for Float32 and 10^{-16} for Float64. This is many orders of magnitude larger than *minpos*, the smallest representable positive float, which is about 10^{-45}, 10^{-324} for Float32, Float64, respectively (Table 2.1).

We therefore developed a statistically improved conversion from a random unsigned integer to a uniformly distributed float in $[0,1)$. Counting the number of leading zeros l of a random unsigned integer yields $l = 0$ at probability $p = \frac{1}{2}$, $l = 1$ at probability $p = \frac{1}{4}$ and $l = k$ at probability $p = 2^{-k-1}$. These probabilities correspond exactly to the share of power-2 exponents in the unit range $[0,1)$ for floats. Consequently, we translate the number of leading zeros l to the respective exponent bits and use the remaining bits of the unsigned integers for the mantissa bits. The statistical flaws from the conventional conversion as presented above are avoided, but for practical reasons the smallest float that can be sampled is about 10^{-20} for both Float32 and Float64. It is therefore prac-

```julia
function randfloat(::Type{Float32})

    ui = rand(UInt64)         # 64 random bits with generator rng
    lz = leading_zeros(ui)    # count leading zeros of random UInt64

    # then convert leading zeros to exponent bits of Float32
    # e.g. 01111110, and bitshift to the right position via <<23
    e = ((126 - lz) % UInt32) << 23

    # reuse the last bits of ui for the 23 mantissa bits
    # unless the leading zeros reach into those for lz > 40
    ui = lz > 40 ? rand(UInt64) : ui

    # ui % UInt32 drops the first 32 bits
    # & 0x007f_ffff sets non-mantissa bits to 0
    # e | then combines exponent and mantissa
    # and reinterpret the UInt32 as Float32
    return reinterpret(Float32,e | ((ui % UInt32) & 0x007f_ffff))
end
```

Listing 4.1 | An improved random number generator for uniformly distributed floats. The Julia function `randfloat` takes a number format as argument (here only the version for Float32 is presented). `%` is the remainder after division, for unsigned integers effectively converting between unsigned integers by adding leading zeros or discarding leading bits. `?` indicates a one-line if-clause. `&` is the bitwise logical and-operation. `|` is the bitwise logical or-operation.

tically impossible to sample a zero with this technique, just as the chance of obtaining a zero in $[0, 1)$ is effectively 0 for floats with 32 or 64 bit. For Float32, for example, we implement this technique as shown in Listing 4.1, the implementations for Float16 and Float64 are similar. See our implementation in the Julia-package RandomNumbers.jl for further details. The random number generation for uniformly distributed floats as described here is used throughout this chapter.

4.2.2 Monte Carlo orbit search

The state of a deterministic dynamical system is entirely determined by $X = (X_1, X_2, ..., X_N)$, the vector of all its N prognostic variables at a given time step t. We define a periodic orbit when the state vector X^{t_0} at time step t_0 reoccurs at a later time step $t_1 > t_0$

$$X^{t_0} = X^{t_1} \tag{4.1}$$

Bitwise equality is hereby required with an exception for floats where $-0 = 0$, which arithmetically does not impact on the dynamical system. While $\frac{1}{-0} = -\infty \neq \infty = \frac{1}{0}$ in float arithmetic, we consider finite solutions only. The bitwise periodicity is in contrast to other studies investigating quasi-periodic orbits [Urminsky, 2010; Yalnız *et al.*, 2021], which require X^{t_0}, X^{t_1} to be close, but not bitwise identical. Bitwise periodic orbits are found in long simulations when Eq. 4.1 holds exactly and are very sensitive to the choice of the number format and numerical precision. This is distinct from unstable periodic orbits, numerically found via an iterative Newton method when Eq. 4.1 holds up to a numerical error [Viswanath, 2007]. Although there is an infinite number of unstable periodic orbits in the Lorenz 63 system [Capiński *et al.*, 2018; Lorenz, 1963], the number of bitwise periodic orbits is always finite.

The invariant measure of a chaotic system describes its attractor independent of the initial conditions. A deterministic chaotic system simulated with finite-precision arithmetic (also deterministic as opposed to stochastic rounding, for example) will converge to one of the periodic orbits for any initial condition. Based on all periodic orbits we can compute the invariant measure through a weighted average by the orbits' respective basins of attraction, i.e. the fraction of initial conditions that end up on a given periodic orbit. To find *all* periodic orbits in a deterministic dynamical system, all possible initial conditions have to be integrated and checked for periodicity. For a 1-variable system with $X \in [0, 1)$ (such as the Bernoulli map, see section 4.3) simulated with Float32 there are 1,065,353,216 initial conditions. For any larger system or higher precision number format it becomes computationally virtually impossible to consider all initial conditions.

We therefore use a Monte Carlo-based random sampling of the initial conditions to find a subset of all orbits. The orbits found are statistically expected to be those with the largest basins of attraction, and a robust estimate of their size is obtained for a sufficiently large sample of initial conditions. This procedure is explained in the following.

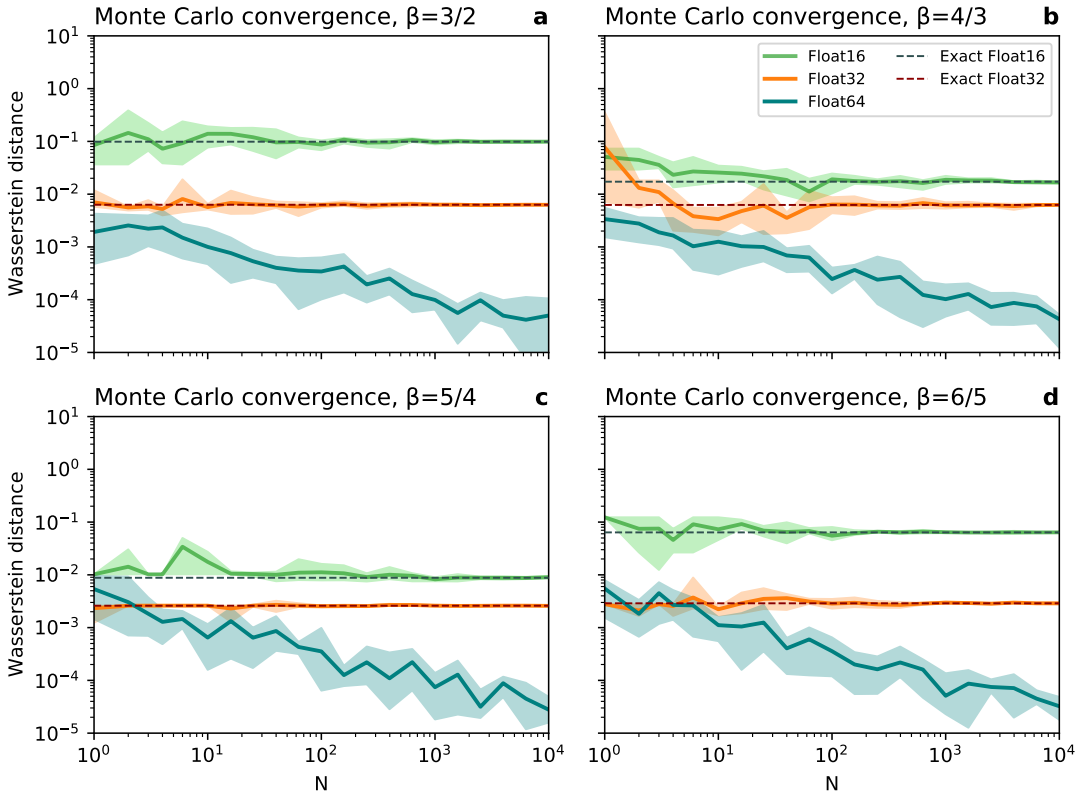

Figure 4.1 | Convergence of the Monte Carlo sampling to estimate invariant measures. The agreement between the analytical and simulated invariant measures (Figure 4.3) from N random initial conditions uniformly distributed in $[0, 1)$ are assessed with the Wasserstein distance (section 4.2.4). About $N = 1000$ random initial conditions allow for a robust estimate of the invariant measure with Float16 and Float32, virtually identical with exact invariant measure obtained from computing all 15,360 Float16 and 1,065,353,216 Float32 numbers in $[0, 1)$, respectively. The β parameter of the generalised Bernoulli map is **a** $\beta = \frac{3}{2}$, **b** $\beta = \frac{4}{3}$, **c** $\beta = \frac{5}{4}$, and **d** $\beta = \frac{6}{5}$. Solid lines represent the mean and shading the min-max range.

For the generalised Bernoulli map, the random number generator from section 4.2.1 is used to sample from all floats in $[0, 1)$ to obtain a representative subset of all initial conditions. While there is no guarantee that all orbits are found, those found have the largest basin of attraction. While it is easily possible to miss a periodic orbit, those missed have a very small basin of attraction and therefore a negligible contribution to the invariant measure. Estimating the invariant measure from these orbits is therefore

also expected to be an unbiased approximation that converges to the exact invariant measure. The exact invariant measure, on the other hand, is obtained by finding *all* simulated orbits and their exact basins of attraction rather than using a random set of initial conditions. We verify this methodology for Float16 and Float32, where the exact invariant measure can be calculated in Figure 4.1. While we cannot find all orbits with Float64, the Monte Carlo-based invariant measure converges to the analytical invariant measure and is for the same sample size a better approximation than using Float16 or Float32. Despite the high precision, a Float64 simulation of the generalised Bernoulli map still substantially degrades some properties of the analytical system: The topological entropy, measuring how trajectories diverge onto distinct orbits, is positive in the analytical system, representing chaotic solutions. However, even with the high precision of Float64 the topological entropy is negative, as trajectories eventually converge onto periodic orbits.

For the Lorenz 1996 system, the space of all possible initial conditions is much larger than the space the attractor occupies. It is therefore more efficient to only choose initial conditions randomly that are already part of, or at least close to the attractor. Basin of attraction is here therefore the relative share of the initial conditions from the attractor that end up on a given orbit, and not from all possible initial conditions. To obtain an initial condition for the Lorenz 1996 system, one first starts a high-precision simulation from a given initial condition including a small stochastic perturbation (see section 4.4 for more details). After disregarding a spin-up the information of the chosen initial condition is removed and the stochastic perturbation grows into a fully independent random initial condition. Converting a random time step from a high-precision simulation into the given number format then yields a point that is either on or close to the low-precision attractor.

4.2.3 Efficient orbit search with distributed computing

To find an orbit in a simulation, Eq. 4.1 is used after every time step to check for equality with the system's state at a previous time step. However, before an orbit is found, it is unknown whether a given initial condition X^{t_0} is already part of the orbit or still part of the trajectory that is yet to converge onto the orbit. It is possible to use the last time step of a very long spin-up simulation as initial condition. This strategy limits the chance that X^{t_0} is not yet part of the orbit, but does not provide a guarantee, nor is it efficient. Instead we successfully implemented a strategy whereby X^{t_0} is updated during simulation and slowly moves forward in time: Updates like $t_0 = \text{round}(\sqrt{t_1})$ or

$t_0 = \text{round}(\log(t_1))$, with t_0, t_1 integers, indicating the time steps, are used. In particular, we use several past time steps of the simulation as X^{t_0} to check for periodicity. Checking for periodicity with *all* past time steps is inefficient as they would have to be stored and $\frac{1}{2}(t_1 - t_0)^2$ checks, which therefore scale with t_1^2, have to be performed in total for all time steps from t_0 to t_1. In contrast, for a constant number of checks per time step, the total number of checks increases linearly with the simulation time.

Finding n orbits from n different initial conditions is a problem that is parallelizable into n independent processes calculated on $n_p \leq n$ processors. We follow ideas of the MapReduce framework: Each worker process starts with a different initial condition and simulates the dynamical system independently of other processes until an orbit is found. This orbit is passed to the main process, which reduces successively all n orbits found into a list of unique orbits, as several initial conditions can yield the same orbit. Instead of defining an orbit by all of the points on it, which would be computationally inefficient for very long orbits, we describe an orbit by its period length and minimum. The minimum of an orbit is the point for which the L^2 norm is minimised. While it is theoretically possible that an orbit has several minima with identical norms, this case was very rare in our applications. Two such falsely-identified unique orbits can still be merged in post-processing. A uniqueness check between two orbits (or one orbit and a list of orbits, in which case the uniqueness check is pairwise against every orbit in the list) is unsuccessful and yields a single orbit only if all of the three following criteria are fulfilled

1. **Length.** The two orbits must have the same period length.
2. **Minimum norm.** The norms of the orbits' minima have to be identical.
3. **Minimum.** The orbits' minima, including possible rotation of the variables for spatially periodic solutions, have to be bitwise identical.

While criterion 3 is sufficient to identify the uniqueness of two orbits (excluding the two minima exception as described above), it is computationally more efficient to check for criterion 3 only if criterion 2 is fulfilled, which is only checked if criterion 1 is fulfilled, hence the proposed order of the criteria.

4.2.4 Wasserstein distance

The invariant measure of a chaotic dynamical system is estimated with histogram binning. To assess the agreement of two histograms representing invariant measures (either simulated or analytical) we use the Wasserstein distance, a metric that derives from

the theory of optimal transport, with an L^1 cost. The Wasserstein distance $W_1(\mu, \nu)$ is defined as the least cost at which one can transport all probability mass from histogram μ to another histogram ν, where the cost to move mass m from a bin at location x to a bin at location y is $m|x-y|$ [Paxton *et al.*, 2021; Villani, 2003]. This gives a non-parametric method to compare probability distributions which accounts for both differences in the probabilities of events as well as their separations in the underlying space, so that closeness in Wasserstein distance truly corresponds to a natural notion of closeness between probability distributions [Villani, 2003, Thm 7.12].

4.3 Revisiting the generalised Bernoulli map

The generalised Bernoulli map [Parry, 1960] is a 1-variable chaotic system starting with $x_0 \in [0, 1)$ at time iteration $i = 0$ with the parameter $\beta > 1$ defined as

$$x_{i+1} = f_\beta(x_i) = \beta x_i \mod 1 \tag{4.2}$$

The modulo-operator mod satisfies that $x \in [0, 1)$ in all future iterations. Simulating this system was found to not represent well the periodic orbit spectrum [Boghosian *et al.*, 2019], which is closely related to the simulated invariant measure. For the generalised Bernoulli map the analytical invariant measure is known [Hofbauer, 1978]

$$h_\beta(x) = C \sum_{j=0}^{\infty} \beta^{-j} \theta(1_j - x) \tag{4.3}$$

The heaviside function is θ and 1_j is the j-th iteration of the Bernoulli map starting from $x_0 = 1$, i.e. $1_j = f_\beta^j(1)$. The normalization constant is chosen as $C = 1$, but for the calculation of Wasserstein distances renormalization is applied so that $C \int_0^1 h_\beta(x)\mathrm{d}x = 1$, which ensures that $Ch_\beta(x)$ is a probability density function. To better visualise the invariant measure for varying β we introduce a normalisation $h^*_\beta(x) = \frac{h_\beta(x)}{\max(h_\beta(x))}$, which is always in $[0, 1]$ and can be applied to the analytical invariant measure as well as simulated ones.

We are revisiting the generalised Bernoulli map with various number formats and rounding modes to understand better the previously suggested pathology [Boghosian *et al.*, 2019] as a function of arithmetic precision. While simulating the Bernoulli map numerically with a given number format, we perform both the multiplication and the subtraction in Eq. 4.2 with that format and avoid any conversion between number formats. This is in contrast to Boghosian *et al.* [2019], whose implementation converts x_i

to Float64 before multiplication with β (as Float64) and possible subtraction with 1 (as Float64) in the modulo. In summary, $x_{i+1} = \text{Float32}(\beta * \text{Float64}(x_i) \mod 1)$. While some hardware allows for fused multiply-add operations without intermediate rounding error, similar to the conversion to Float64 here, the fused conditional subtraction in the modulo is generally not supported on hardware.

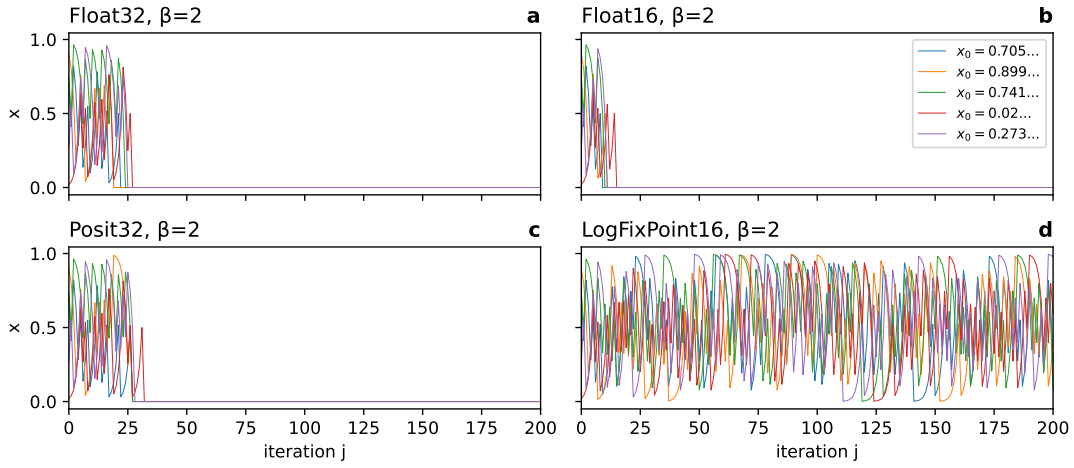

Figure 4.2 | The Bernoulli map simulated with different number formats. **a** Float32, **b** Float16, **c** Posit32, and **d** LogFixPoint16. The arithmetic operations in the Bernoulli map do not introduce rounding errors in **a**-**c**, only the initial conditions are subject to rounding, causing the attractor to collapse after a few iterations. However, the subtraction in the Bernoulli map causes rounding errors with logfix arithmetic in **d** that prevent the stalling at 0 from **a**-**c**.

4.3.1 The special $\beta = 2$ case

Boghosian *et al.* [2019] highlight that the Bernoulli map with $\beta = 2$, and similarly for every even integer, will collapse to $x = 0$ after n iterations with any float format at arbitrary high precision, where n is smaller than the number of bits in the format. The subtraction with 1 acts as a bitshift in base-2 format towards more significant bits, pushing zero bits in the mantissa until $x = 1$ and the modulo returns $x = 0$ (Figure 4.2a, b and c). This phenomenon occurs as the Bernoulli map with $beta = 2$ (and similarly for larger even integers) does not introduce any arithmetic rounding error: Both the multiplication with β and subtraction with 1 are exact with floats (and also with posits) in $[1, 2)$. The multiplication with $\beta = 2$ is exact as the base-2 exponent is simply increased by 1. The subtraction with 1 is exact as every finite positive float or posit can be written as $2^e(1 + f)$ (see Eq. 2.4) for some integer exponent e and a sum $f \in [0, 1)$ of powers of two with negative exponents. Constraining the range to $[1, 2)$, where the subtraction is applied, yields $e = 0$

and so subtracting 1 from the mantissa $1 + f$ returns f, again a sum of powers of two, which is exactly representable with floats or posits.

The only occurring rounding error is in the initial conditions. While a randomly chosen $x \in [0, 1)$ will have infinitely many non-zero mantissa bits at infinite precision, at finite precision those beyond the resolved mantissa bits will be rounded to 0. Therefore, the least significant mantissa bit remains 0 after each iteration of the Bernoulli map while the same 0 bit from the previous iteration is further shifted in. While this behaviour holds for floats and posits, it does not occur with logarithmic fixed-point numbers (logfixs). All multiplications are exact with logfixs (unless under or overflows occur, see section 2.1.4), but in contrast to floats and posits a rounding error occurs in the subtraction, with the possibility of setting the least significant mantissa bit to 1. On the next multiplication with $\beta = 2$ this 1-bit is shifted further in, hence, the rounding error is effective at preventing a collapse of the attractor (Figure 4.2d). The Bernoulli map with $\beta = 2$ and simulated with floats or posits is therefore special, as it is a chaotic system that does not involve any arithmetic rounding errors beyond the rounding of the initial conditions. However, the simulation of most other system, including the generalised Bernoulli map with $1 < \beta < 2$, involves rounding errors with any finite precision number format.

4.3.2 Bifurcation of the invariant measure

Given that the analytical invariant measure is known for the generalised Bernoulli map (Eq. 4.2), we can assess the representation of such a system with various number formats at different levels of precision. Boghosian *et al.* [2019] conclude that the invariant measure with Float32 is an inaccurate approximation of the analytical invariant measure. This difference is even more pronounced with Float16 arithmetic (Figure 4.3), however, with Float64 the invariant measure is comparably accurate. The question therefore arises whether the discrepancy of the invariant measures vanishes with higher precision, or whether a pathology persists at any precision level for some $\beta < 2$.

The analytical invariant measure of the generalised Bernoulli map consists of many step functions taking values on a discrete set of points (Fig. 4.3), which bifurcate with increasing (Fig. 4.4a). These *quantization levels* cannot be exactly represented with Float16 or Float32 arithmetic (Fig. 4.3), such that their bifurcation is visually blurred (Fig. 4.4c). However, the visually sharp representation of this bifurcation of the quantization levels with Float64 indicates a much more accurate approximation to the analytical invariant measure. Due to the higher precision of 32-bit posit arithmetic (Posit32) around ± 1, the

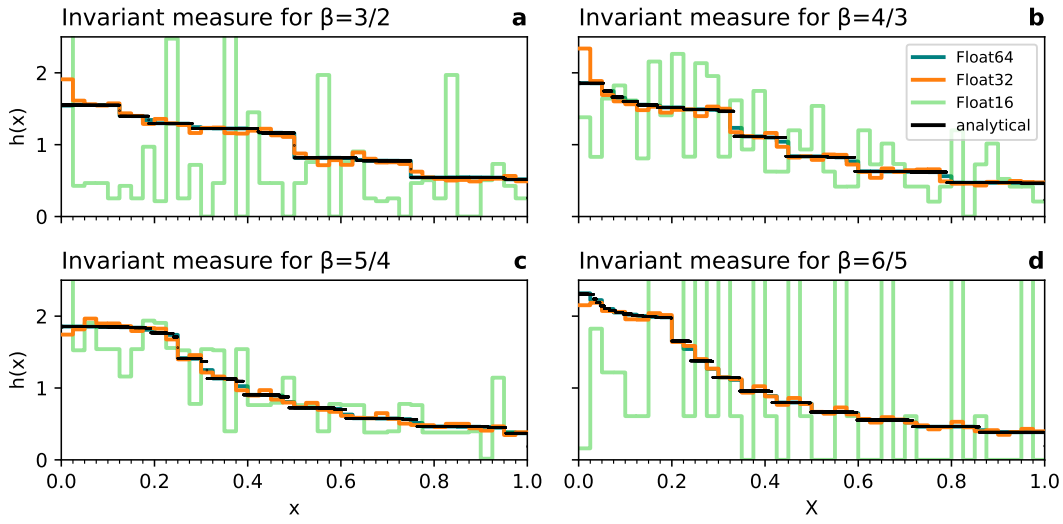

Figure 4.3 | The invariant measure of the generalised Bernoulli map. The generalised Bernoulli map is simulated with parameter **a** $\beta = \frac{3}{2}$, **b** $\beta = \frac{4}{3}$, **c** $\beta = \frac{5}{4}$, **d** $\beta = \frac{6}{5}$ and calculated with different number formats Float64, Float32 and Float16. The invariant measures of Float16 and Float32 are obtained from periodic orbits found by starting from 10,000 initial conditions $x_0 \in [0, 1)$ chosen from a random uniform distribution. For Float64, long integrations (10,000 iterations, disregarding a spin-up of 5,000 iterations) of the Bernoulli map are used instead. Histograms use the bin width 0.025. The analytical invariant measure is not binned, which accounts for the discrepancy to Float64.

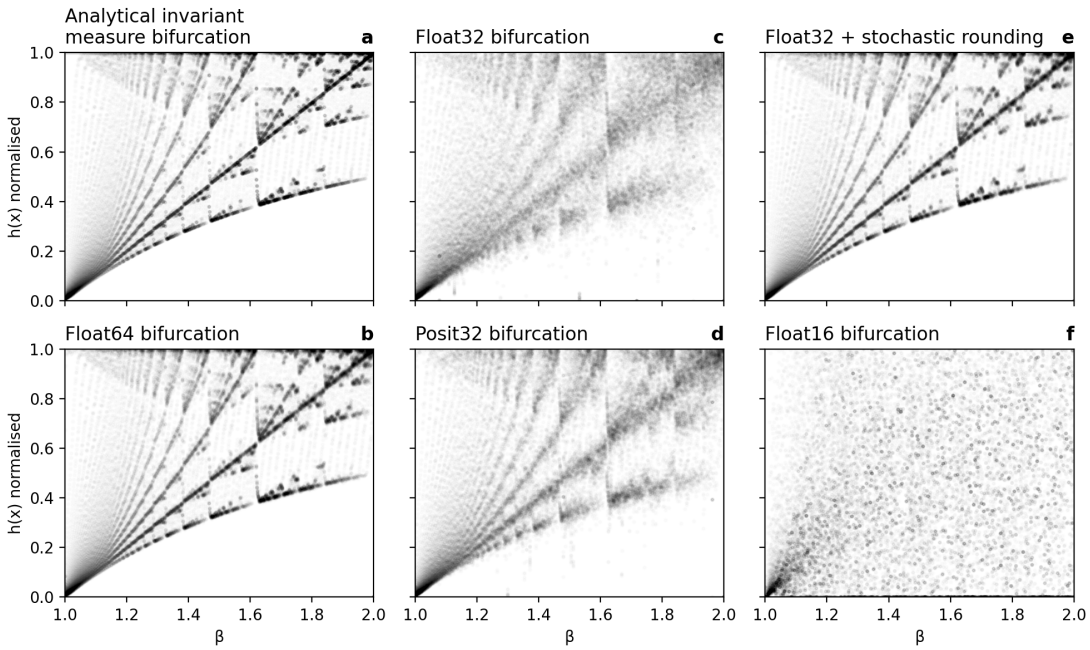

Figure 4.4 | Bifurcation of the quantization levels corresponding to the invariant measures in the generalised Bernoulli map as simulated with various number formats. a Analytical bifurcation $h_\beta(x)$ from the exact invariant measure, normalised by $\max(h_\beta(x))$, compared to the invariant measure by simulating the Bernoulli map with **b** Float64, **c** Float32, **d** Posit32, **e** Float32 and stochastic rounding, and **f** Float16.

bifurcation is slightly improved with Posit32 over Float32 (Fig. 4.4d). Given the inaccurate representation of the invariant measure with Float16 (Fig. 4.3), its bifurcation has little resemblance to the analytical bifurcation (Fig. 4.4f).

4.3.3 Effects of stochastic rounding

Augmenting Float32 with stochastic rounding considerably improves the bifurcation of the invariant measure (Fig. 4.4e) and makes it virtually indistinguishable from Float64 or the analytic bifurcation. However, stochastic rounding does not decrease the rounding error accumulated over many iterations in a forecast (Figure 4.5), such that the effective precision is not increased over deterministic rounding. But introducing stochasticity prevents the convergence onto short periodic orbits, which are otherwise present with deterministic rounding [Boghosian *et al.*, 2019]. Previously inaccessible regions of the attractor can be reached with stochastic rounding as the simulation is frequently pushed off any periodic orbit. While periodic orbits are not fully removed from solutions due to the use of pseudo random number generators (PRNG) that are themselves periodic, the periods of PRNGs are usually so long that effectively any periodicity is avoided. The period length of the conventional Mersenne Twister [Matsumoto & Nishimura, 1998] is $2^{19937} - 1$ and still sufficiently long with $2^{128} - 1$ for the faster Xoroshiro128+ [Blackman & Vigna, 2019] PRNG that is used in StochasticRounding.jl.

The agreement of the analytical and simulated invariant measures is quantified with the Wasserstein distance (section 4.2.4). For $\beta = 2$ the analytical invariant measure is the uniform distribution $U(0, 1)$, whereas all float and posit formats for both deterministic and stochastic rounding simulate a collapse of the attractor to zero such that the invariant measure is the Dirac delta distribution (Fig 4.2). The Wasserstein distance is in all these cases $W_1 = 0.5$ and does not decrease, i.e. improve, with precision. However, as previously mentioned, the rounding errors from logfixs prevent a collapse such that for LogFixPoint16 the invariant measure is much better approximated, with $W_1 = 0.05$.

For $1 < \beta < 2$ the Wasserstein distance is always reduced going to higher precision, supporting the inference that only the case $\beta = 2$ (and other even integers) presents a pathology where higher precision does not improve the simulated invariant measure arising from the generalised Bernoulli map (Fig. 4.6). The Wasserstein distances of Float32 with stochastic rounding are similarly low to Float64, but Float64 yields slightly lower distances in most cases.

However, using stochastic rounding with Float16 is worse than deterministic rounding as in this case the stochasticity makes it possible that the invariant measure of the

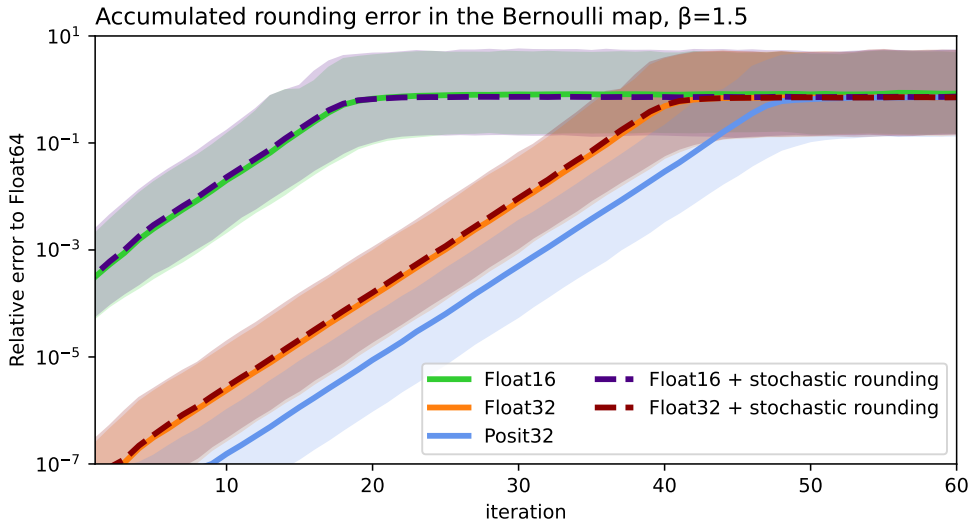

Figure 4.5 | Accumulated rounding error in the Bernoulli map. Starting from 10,000 random initial conditions, the accumulated rounding error is the relative error of the given number format relative to a Float64 integration. Solid lines represent median errors across all initial conditions, shading the interdecile range. Other choices for the parameter β of the generalised Bernoulli map yield a similar comparison between the number formats, but decreasing β towards 1 decreases the error growth for all formats similarly.

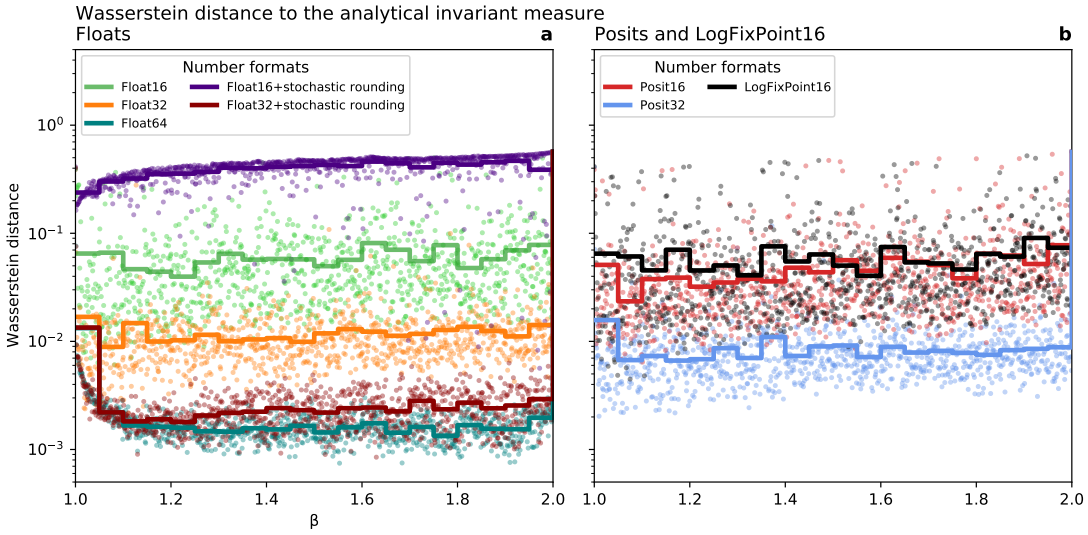

Figure 4.6 | Agreement between the simulated and analytical invariant measures in the generalised Bernoulli map. For all values except $\beta = 2$ a high precision number format yields a better agreement with the analytical Bernoulli map. Simulations using **a** Floats with and without stochastic rounding, **b** Posits and LogFixPoint16. The Wasserstein distances are calculated for the invariant measures obtained from 1000 simulations for every value of β. Scatter points denote individual Wasserstein distances, solid lines indicate averages across a range of as indicated by steps.

generalised Bernoulli map collapses to 0, which is a fixed point (Figure 4.7). While for Float32 the probability of such an occurrence is low and is not observed here, with Float16 most simulations collapse within a few thousand iterations transforming their invariant measures into Dirac distributions. Whether this problem generalises to other systems is questionable. We suspect that this is a feature of low dimensional dynamics, and would not arise in more physical, higher dimensional systems, where the chance of a stochastic perturbation moving one onto a fixed point becomes vanishingly small even at very low precision. Other natural systems do not have fixed points due to time-dependent forcing. The posit format is slightly better than floats at both 16 and 32 bit, as expected from the slightly higher precision.

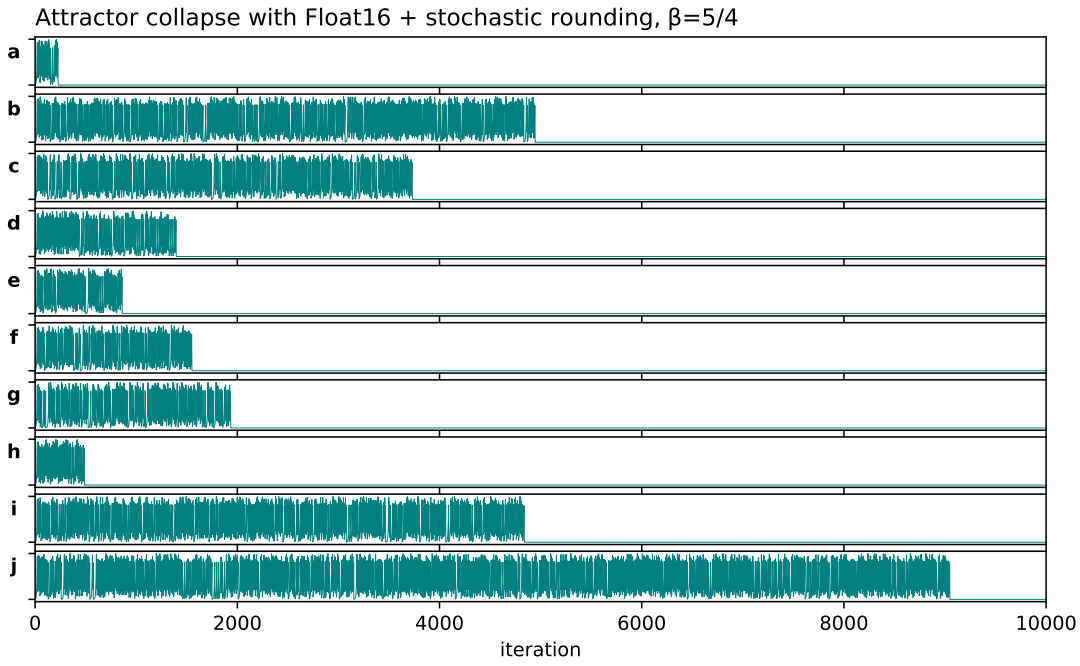

Figure 4.7 | Attractor collapse of the generalised Bernoulli map with Float16 and stochastic roudning. a-j Simulation of the Bernoulli starting from identical initial conditions and $\beta = 5/4$. Only the the state of the random number generator for stochastic rounding differs between **a**-**j**. After 10,000 iterations all simulations **a**-**j** stalled at the fixed-point 0 of the analytical attractor. The y-axes denote the value of x_i in [0,1].

4.4 Orbits in the Lorenz 1996 system

In contrast to the 1-variable generalised Bernoulli map, most continuous natural systems are simulated with as many variables as computationally affordable. Weather forecast and climate models often use more than 10 million independent variables that result from a discretisation of naturally continuous variables on the globe. While short periodic orbits in low precision are problematic in the simulation of few-variable systems as discussed in the previous section, this section tests the hypothesis that large systems are unaffected for all practical purposes.

4.4.1 The Lorenz 1996 system

To investigate the dependence of periodic orbits on the number of variables in the system we consider the chaotic Lorenz 1996 system [Hatfield *et al.*, 2017; Lorenz & Emanuel, 1998]. With N variables X_i, $i = 1, ..., N$ the Lorenz 1996 system is a system of coupled ordinary differential equations

$$\frac{\mathrm{d}X_i}{\mathrm{d}t} = (X_{i+1} - X_{i-2})X_{i-1} - X_i + F \tag{4.4}$$

in a 1-dimensional spatial domain with periodic boundary conditions, $X_{N+1} = X_1$ etc. The term $(X_{i+1} - X_{i-2})X_{i-1}$ implements nonlinear advection and drag is represented with the relaxation term $-X_i$. The forcing F is the single parameter in the Lorenz 1996 system fixed at the common default $F = 8$ which produces chaotic solutions. The forcing is steady in time and constant in space. The system exhibits dynamics of nonlinear wave-wave interactions (Fig. 4.8), which are reasonably independent of the number of variables (Figure 4.8b). The system can be integrated with as little as $N = 4$ variables without an obvious degradation of the simulated dynamics.

The initial conditions are in equilibrium $X_i = F, \forall i i \neq j$ with only a single variable which is slightly perturbed, $X_j = F + 0.005 + 0.01\epsilon$ with $\epsilon \sim U(0, 1)$, drawn from a random uniform distribution in $[0, 1)$. Due to the periodic boundary conditions, the system is spatially invariant once the information from the initial conditions is removed through the chaotic dynamical evolution, after some several hundred time steps (Figure 4.8a). The invariant measure μ of one variable X_i is therefore identical to that of any other, $\mu(X_i) = \mu(X_j), \forall i, j$, as will be further discussed below.

The Lorenz 1996 system is discretized in time using the 4-th order Runge-Kutta scheme

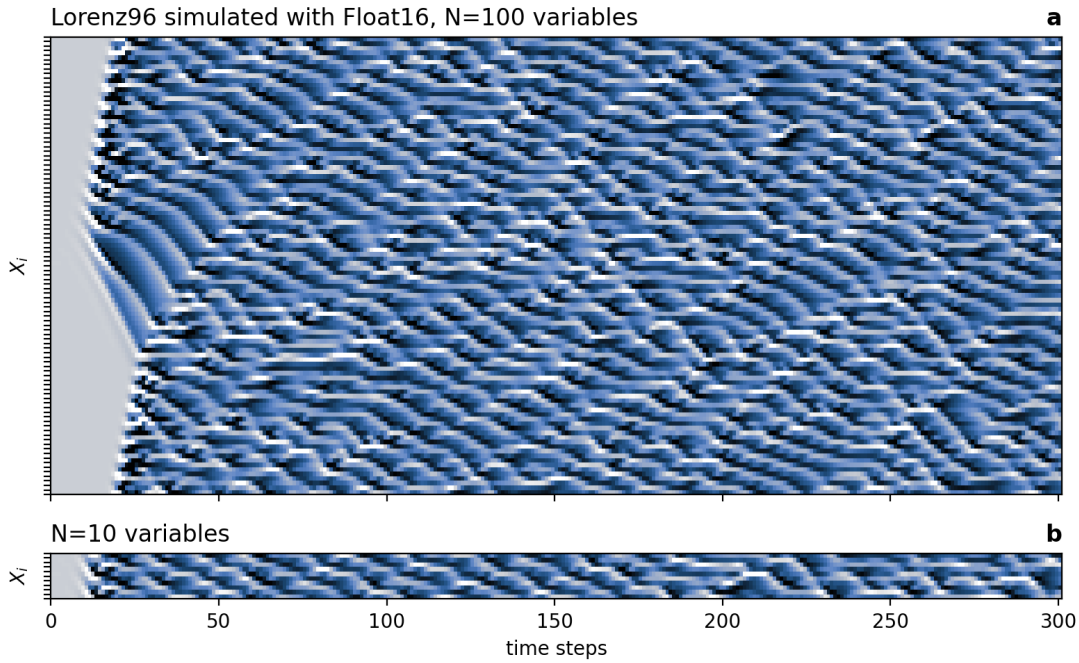

Figure 4.8 | The Lorenz 1996 system simulated with Float16 arithmetic. a 100 variables are used, starting from equilibrium with a small perturbation in X_{50}, and **b** 10 variables starting with a small perturbation in X_5. Rectangles visible in shading represent individual variables and time step.

[Butcher, 2008] with a time step of $\Delta t = 0.01$. At this temporal resolution the system can also be integrated using a low-precision number format such as Float16 (Figure 4.8). For more details and a software implementation see Lorenz96.jl (Klöwer, 2019/2021).

4.4.2 Longer orbits with more variables

Using $N = 4$ variables in the Lorenz 1996 system simulated with Float16, the longest periodic orbit we find is 6756 time steps long (Figure 4.9 and Table 4.1). The basin of attraction is about 0.82, meaning that more than 80% of the randomly chosen initial conditions converge onto this orbit. Increasing the number of variables to $N = 5$, the longest periodic orbit found increased to a length of 294,995 time steps at a similarly large basin of attraction. For $N > 9$ the orbit search becomes computationally very demanding and requires more than several days on sizable compute clusters of 100 cores. For $N = 9$ though, the longest periodic orbit we were able to find has a period of 32,930,252,532 time steps. For a list of all bitwise periodic orbits found see Table 4.1.

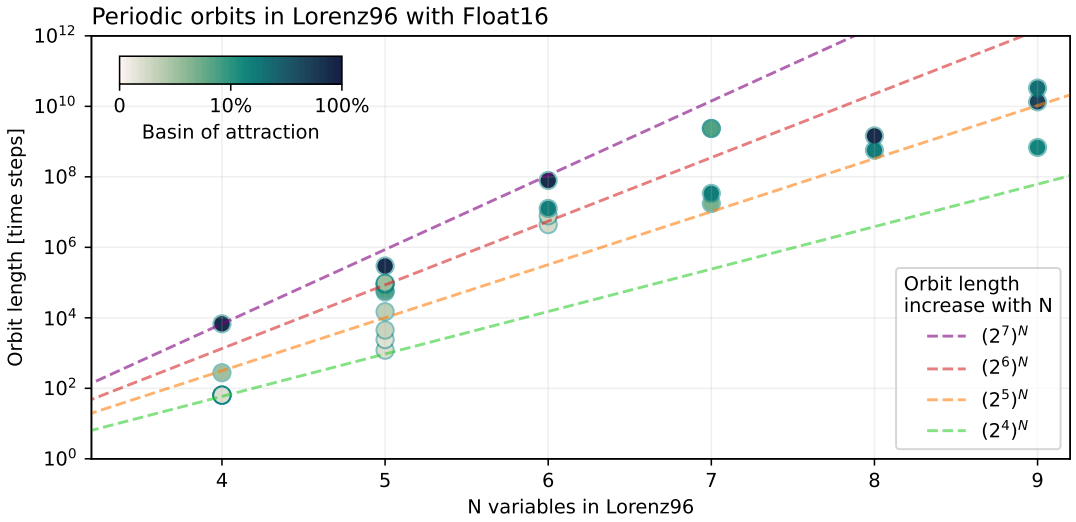

Figure 4.9 | Periodic orbits in Lorenz96 simulated with Float16 and an increasing number of variables. Initial conditions are randomly taken from a spin-up simulation. Size of the basins of attraction (shading) correspond to the share of initial conditions that converge to the respective periodic orbit. Dashed lines provide an orientation for the exponential increase in orbit lengths with the number of variables.

In most cases between 4 and 9 variables in the Lorenz 1996 system, the longest orbit is also the one with the largest basin of attraction. The longer the orbit the larger the occupied state space of possible values the variables X_i can take at a given precision. Consequently, the assumption is that it is most likely that a given trajectory ends up on

the longest orbit. However, we also found a counter examples as, the longest orbit with $N = 8$ variables is shorter than the longest with $N = 7$ variables (Figure 4.9).

4.4.3 More variables instead of higher precision

The orbit length increases approximately exponentially following a scaling of about 16^N to 128^N from $N = 4$ to $N = 9$. Such an exponential increase translates to about 4 to 7 effective bits of freedom (as $2^4 = 16, 2^7 = 128$) for every additional variable in Lorenz 1996 represented with Float16. However, the computational resources limit the orbit search for larger N, making it hard to constrain this exponential scaling further. Assuming a similar exponential orbit increase holds for larger N, extrapolation of these findings suggests orbit lengths on the order of about 10^{100000} for million-variable systems simulated with Float16. This is far beyond the reach of any computational resources currently available. In that sense, while a simulation of such large systems would eventually be periodic, a periodic solution will never be reached.

Longer orbits are promising to avoid periodic solutions in low precision, but short periodic orbits do not necessarily misrepresent a reference invariant measure. We assess the agreement of invariant measures using the Wasserstein distance as before. As a reference invariant measure $\mu(X_{\text{ref}})$ we integrate the Lorenz 1996 system for 1,000,000 time steps with $N = 500$ variables using Float64 arithmetic. The Wasserstein distance is then $W_1(\mu(X_{\text{ref}}), \mu(X))$ with X representing the N variables from a Lorenz 1996 simulation using either Float16 or Float64 arithmetic.

Using only $N = 4$ variables in the simulation of Lorenz 1996 yields an invariant measure with little resemblance to the reference (Fig. 4.10a and e), regardless of the number format. While more variables yield an invariant measure that converges to the reference, there is virtually no difference whether Float16 or Float64 arithmetic is used (Figure 4.10b-e). The Wasserstein distance significantly reduces with an increasing number of variables, but not with higher precision. Given a certain availability of computational resources a better invariant measure is therefore obtained by reducing the precision and reinvesting the performance gain into more variables.

4.5 Discussion

We analysed bitwise periodic orbits in the generalised Bernoulli map with different number formats and levels of arithmetic precision. While there are very special cases (such as $\beta = 2$) in which the system's simulation is greatly degenerated at any precision, in

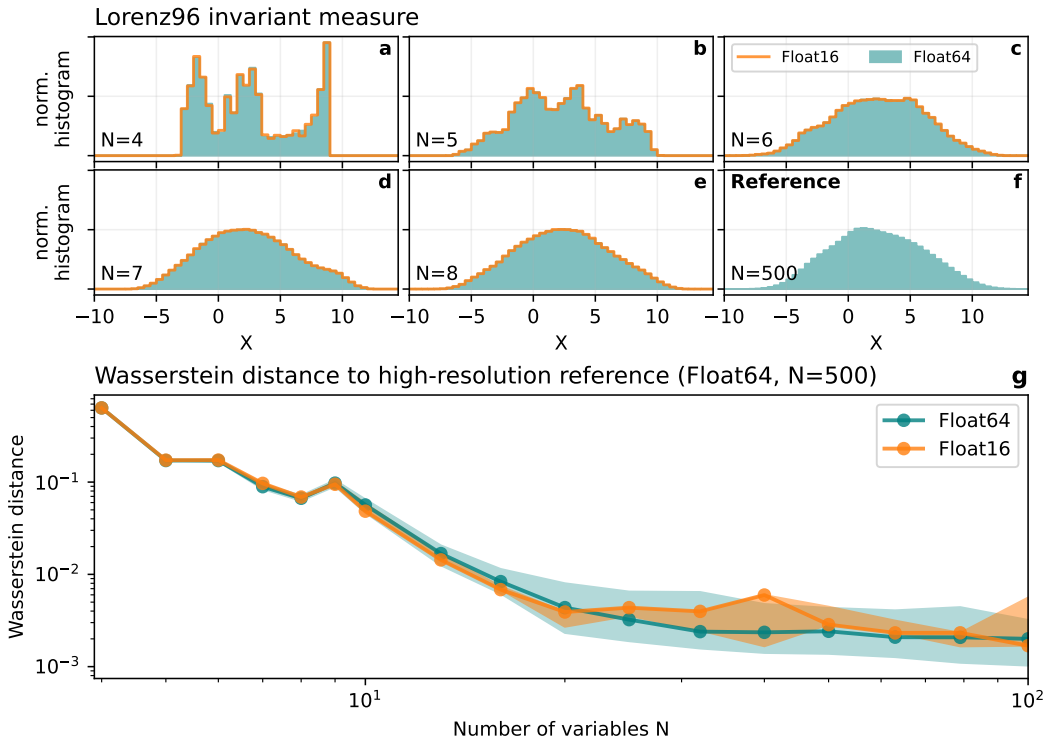

Figure 4.10 | Improvement of the simulated Lorenz 1996 invariant measure with increasing number of variables. **a** The invariant measure of Lorenz 1996 simulated with 4 variables. Using Float16 or Float64 arithmetic yields a virtually identical invariant measure. **b-e** as **a** but with an increasing number of variables. **f** The reference invariant measure obtained from 500 variables using Float64 arithmetic. **g** The Wasserstein distance of the simulated Lorenz 1996 system with respect to the reference. Invariant measures are taken from all available variables, which are invariant due to periodic boundary conditions and spatially-independent forcing (see Eq. 4.4). Shadings in **g** represent the 5-95% confidence interval and solid lines the median obtained from an ensemble simulation with 51 members starting from slightly perturbed random initial conditions.

all other cases simulations were found to improve with higher precision. 16 and 32-bit arithmetic result in short bitwise periodic orbits in the Bernoulli map, but a sufficiently high precision reduces the error in the invariant measure to a minimum.

Stochastic rounding is found to be especially beneficial for 32-bit floats, as it prevents bitwise periodic orbits. A chaotic trajectory is regularly pushed off any periodic orbit due to the stochastic perturbation in rounding. Trajectories travel thereby through otherwise unreachable state space, improving the simulated invariant measures. However, in the Bernoulli map stochastic rounding also causes a non-zero chance that the system collapses on the fixed point. But for any precision higher than Float16 and in systems with more variables this chance quickly vanishes. In many complex natural systems the fixed points are not close to the attractor, further limiting the relevance of this issue in practice.

Increasing the number of variables in the Lorenz 1996 system, we find that more variables improve the simulated invariant measure much more than increased precision with fewer variables does: Doubling the amount of variables yields a more accurate invariant measure compared to a high-resolution and high-precision reference than doubling the precision. This provides evidence that computational resources should be invested in higher resolution rather than higher precision for the simulation of continuous chaotic systems.

From a rigorous mathematical perspective, we would like to know how the period of the largest bitwise periodic orbit increases with the number of variables and precision. Experimentally, we find orbits that exponentially increase in length with the number of independent variables in the Lorenz 1996 system. Every variable is found to add between 4 and 7 maximum entropy bits that extend orbits by a factor of 2^4 to 2^7. Assuming similarly a maximum entropy for additional mantissa bits beyond a sufficiently high precision, the expected length of bitwise periodic orbits doubles with every additional mantissa bit in precision.

For more complex simulations of natural systems such as million-variable climate models, the expected bitwise periodic orbits extrapolate to lengths beyond $10^{100,000}$ time steps even in 16-bit precision. Bitwise periodic orbits are therefore not in reach on future generations of supercomputers, especially when the gained performance from low-precision simulations is reinvested into higher resolution. In the context of climate models, this supports a vision of low-precision but high-resolution simulations with added stochasticity to accelerate and improve climate predictions in decades to come.

4.6 Appendix

Period length	basin	minimum
Float16, N=4		
64	0.148301	Float16[8.4, -1.172, 1.208, 1.527]
64	0.025185	Float16[1.217, 1.563, 8.38, -1.199]
277	0.011029	Float16[1.031, 1.361, 8.43, -1.286]
6756	0.815485	Float16[8.47, -1.344, 0.783, 1.106]
Float16, N=5		
1205	0.0005	Float16[-4.465, 2.676, 1.983, 3.566, 4.45]
2415	0.0003	Float16[0.794, 7.3, -0.1984, -1.412, 0.09503]
4485	0.0019	Float16[2.105, 3.656, 4.258, -4.375, 2.752]
14925	0.0032	Float16[-0.2502, 0.8438, 7.652, -0.1371, -1.393]
53995	0.0047	Float16[-0.14, -1.364, -0.1969, 0.7847, 7.605]
59945	0.0612	Float16[7.586, -0.1426, -1.432, -0.198, 0.8076]
88110	0.0275	Float16[1.212, 7.62, -0.5586, -0.641, 0.1917]
91980	0.0792	Float16[1.065, 7.64, -0.4578, -0.8286, 0.09595]
97215	0.0112	Float16[1.34, 7.562, -0.6274, -0.4536, 0.3428]
294995	0.8103	Float16[-1.157, -0.03723, 0.952, 7.598, -0.3352]
Float16, N=6		
4405392	0.001	Float16[-3.361, 0.6084, 1.09, 3.375, 4.805, -0.4067]
7820184	0.002	Float16[1.955, 3.916, 3.021, -4.24, 0.5537, 0.701]
12688470	0.181	Float16[1.222, 1.088, 1.614, 3.6, 3.574, -4.023]
78874782	0.816	Float16[3.219, 4.9, -0.2283, -3.436, 0.3677, 0.713]
Float16, N=7		
17531430	0.03	Float16[-0.4985, 2.64, 5.97, -2.188, 0.1743, 0.2306, 0.103]
33926067	0.18	Float16[-0.4238, -0.704, 2.56, 5.92, -1.828, -0.795, 0.8286]
2355085796	0.79	Float16[6.113, -1.858, -0.6133, 0.7876, -0.1544, -0.707, 1.994]
Float16, N=8		
569018386	0.1875	Float16[-1.604, 1.616, 2.139, 2.68, -2.373, -0.6445, 0.5273, 3.215]
1449659326	0.8125	Float16[1.264, 1.165, 3.008, -2.305, -1.467, 0.797, 2.666, -2.35]
Float16, N=9		
681602535	0.15625	Float16[-3.072, -2.104, -0.4575, -0.08777, 1.392, 0.8486, 1.859, 4.867, 5.086]
13428881973	0.59375	Float16[1.604, 4.234, 4.36, -4.59, -0.2253, 0.362, 1.207, 1.533, 0.793]
32930252532	0.25	Float16[0.769, 1.876, -1.593, -0.4277, -0.3481, 3.348, 6.816, -1.179, -1.096]

Table 4.1 | Bitwise periodic orbits in the Lorenz 1996 system simulated with Float16 and N variables. The period length is given as the number of time steps. The basin is the fraction of initial conditions ending up on a given orbit. The minimum is one point on the orbit for which the norm is minimized. Simulations performed with Lorenz96.jl.

5 A 16-bit shallow water model

Contributions This chapter is largely based on the following publications [*]

M Klöwer, PD Düben, and TN Palmer, 2019. *Posits as an alternative to floats for weather and climate models*, **CoNGA'19: Proceedings of the Conference for Next Generation Arithmetic**, Singapore, 10.1145/3316279.3316281.

M Klöwer, PD Düben, and TN Palmer, 2020. *Number Formats, Error Mitigation, and Scope for 16-Bit Arithmetics in Weather and Climate Modeling Analyzed With a Shallow Water Model*, **Journal of Advances in Modeling Earth Systems**, 10.1029/2020MS002246.

Abstract. The need for high-precision calculations with 64-bit or 32-bit floating-point arithmetic for weather and climate models is questioned. Lower precision numbers can accelerate simulations and are increasingly supported by modern computing hardware. This paper investigates the potential of 16-bit arithmetic when applied within a shallow water model that serves as a medium complexity weather or climate application. There are several 16-bit number formats that can potentially be used (IEEE half-precision, BFloat16, posits, integer and fixed-point). It is evident that a simple change to 16-bit arithmetic will not be possible for complex weather and climate applications as it will degrade model results by intolerable rounding errors that cause a stalling of model dynamics or model instabilities. However, if the posit number format is used as an alternative to the standard floating-point numbers the model degradation can be significantly reduced. Furthermore, a number of mitigation methods, such as rescaling, reordering and mixed precision, are available to make model simulations resilient against a precision reduction. If mitigation methods are applied, 16-bit floating-point arithmetic can be used successfully within the shallow water model. The results show the potential of 16-bit formats for at least parts of complex weather and climate models where rounding errors would be entirely masked by initial condition, model or discretisation error.

[*] with the following author contributions. Conceptualisation: MK, PDD. Data curation: MK. Formal Analysis: MK. Methodology: MK. Visualisation: MK. Writing – original draft: MK. Writing – review and editing: MK, PDD, TNP.

5.1 Introduction

Predictions of weather and climate remain very difficult despite the use of the world's fastest supercomputers. Although the available computational resources have vastly increased over the last decades, forecast errors remain and have several origins [Palmer, 2015, 2012]. They can be categorised as initial and boundary condition errors, model errors and discretisation errors. For instance, uncertainties in the observational data and their assimilation contribute to the errors in the initial conditions [Ghil & Malanotte-Rizzoli, 1991]; discrepancies between the mathematical model and the real world cause model errors; and the finite spatial and temporal resolution result in discretisation errors. The forecast error is in general a combination and respective contributions can be different for different variables and forecast lead times [Jung *et al.*, 2010; Palmer, 2019a]. 64-bit double-precision floating-point numbers (Float64, IEEE [1985] and section 2.1.3) are used by default for weather and climate models since the 1980s with the rise of 64-bit computing. The Float64 format introduces rounding errors [Higham, 2002] that are largely negligible compared to the other mentioned sources of error.

Faster calculations and communication on computing architectures can be achieved with reduced-precision floating-point numbers, with a trade-off between speed and precision, provided a hardware-accelerated implementation. Deep learning algorithms require only low numerical precision [Sun *et al.*, 2020; Wang *et al.*, 2018] but high computational performance. The recent boom of machine learning applications increased the demand on hardware-accelerated reduced-precision calculations, such that hardware developments increasingly offer more flexibility on low-precision number formats. While 16-bit arithmetic was not available for use on commodity supercomputing hardware in the past, today most hardware vendors offer the use of 16-bit formats, such as 16-bit half-precision floating-point numbers (Float16, IEEE [2008] and section 2.1.3), on the next generation of hardware [Burgess *et al.*, 2019; Sato *et al.*, 2020].

Graphic processing units (GPU) started to support Float16 for increased performance [Markidis *et al.*, 2018]. Google's tensor processing units (TPU, Jouppi *et al.* [2017, 2018b]) support the 16-bit BFloat16 format, a truncated version of 32-bit single-precision floats (Float32), as this format is sufficiently precise for many deep learning applications [Burgess *et al.*, 2019; Gupta *et al.*, 2015; Kalamkar *et al.*, 2019]. The world's fastest supercomputers have reached peak performances of 500 Petaflop/s (10^{17} floating-point operations per second) according to TOP500.org [Dongarra & Luszczek, 2011] with Float64 as of June 2021, but peak performances with Float16 are already beyond the exascale milestone (10^{18} flop/s, Kudo *et al.* [2020]; Kurth *et al.* [2018]).

A gained speed from low-precision calculations can free resources to increase the complexity and therefore the forecast skill in weather and climate models [Bauer *et al.*, 2020]. The European Centre for Medium-Range Weather Forecasts reduces the runtime by 40% but not the forecast skill in their forecast model when using almost entirely Float32 instead of Float64 [Váňa *et al.*, 2017]. MeteoSwiss profited similarly with Float32 in their forecast model [Fuhrer *et al.*, 2018; Rüdisühli *et al.*, 2013]. For the European ocean model NEMO [Madec *et al.*, 2017] a mix of 32-bit and 64-bit arithmetic is a promising approach to keep precision-critical parts in high precision while increasing performance in others [Tintó Prims *et al.*, 2019].

Software emulators for other number formats than Float32 and Float64 are often used to investigate rounding errors caused by lower precision formats [Dawson & Düben, 2017]. Although emulators are considerably slower than hardware-accelerated formats, they allow a scientific evaluation of the introduced errors without requiring specialised hardware [Johnson, 2020], such as FPGAs [Russell *et al.*, 2017]. Unfortunately, the computational performance cannot be assessed with software emulators.

Reducing the precision raises questions of the bitwise real information content (see chapter 3). In simplistic chaotic models only a minority of bits contain real information [Jeffress *et al.*, 2017], providing an information theoretic argument for reduced-precision calculations. Recent research covers reduced precision in floating-point arithmetic in parts of weather forecast models, such as the dynamical core [Chantry *et al.*, 2019; Düben *et al.*, 2014; Hatfield *et al.*, 2020; Thornes *et al.*, 2017]; physical parameterisations [Saffin *et al.*, 2020]; the ocean model [Tintó Prims *et al.*, 2019]; the land-surface model [Dawson *et al.*, 2018]; data assimilation [Hatfield *et al.*, 2017, 2018]. In contrast to these studies, in this chapter we will evaluate various 16-bit arithmetics, as other formats than floats have gained little attention for weather and climate models. Options for reduced-precision approaches are discussed and we present ways to mitigate rounding errors.

Although floating-point numbers are the dominating number format in scientific computing, alternatives have been proposed [Gustafson & Yonemoto, 2017]. Posit numbers claim to provide more effective precision in algorithms of machine learning and linear algebra [Chen *et al.*, 2018; Gustafson & Yonemoto, 2017; Langroudi *et al.*, 2019], compared to floats with the same number of bits. Posits were initially tested in simplistic weather and climate simulations [Klöwer *et al.*, 2019] – research that is extended here, providing a more thorough investigation of various 16-bit number formats.

Is 16-bit arithmetic useful within weather and climate models? Which 16-bit formats are most promising, and can model simulations be made resilient against a reduction in precision to 16 bits? These questions are covered in this chapter. We apply several

types of 16-bit arithmetic and test their impact on the simulated dynamics in a shallow water model that serves as a medium-complexity weather and climate application.

The chapter is structured as follows: Section 5.2 introduces the methods specific to this chapter, including an introduction to the shallow water equations and their discretisation. Section 5.3 presents the results of using various 16-bit arithmetic in the shallow water model and section 5.4 summarizes and discusses the results in this chapter.

5.2 Methods

The methods are structured as follows: Section 5.2.1 introduces the shallow water equations and our discretisation thereof. For 16-bit arithmetic it is important to scale and reorder the equations which is presented in section 5.2.2. Section 5.2.3 describes a semi-Lagrangian advection scheme resilient to 16-bit arithmetic. The concept of mixed precision and reduced-precision communication are explained in section 5.2.4 and 5.2.5.

An extensive discussion of the number formats used in this chapter is found in section 2.1.

5.2.1 The shallow water model

The shallow water model used here is an updated version of the one used in Klöwer *et al.* [2019], and most details described therein are also valid here. A vertical integration of the Navier-Stokes equations yields the shallow water equations, that can be used to understand many features of the general circulation of atmosphere and ocean as well as some two-dimensional non-linear interactions on shorter length and time scales [Gill, 1982; Vallis, 2006]. The shallow water equations for the prognostic variables velocity $\mathbf{u} = (u, v)$ and sea surface elevation η are

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} + f\hat{\mathbf{z}} \times \mathbf{u} = -g\nabla\eta + \mathbf{D} + \mathbf{F} \tag{5.1a}$$

$$\frac{\partial \eta}{\partial t} + \nabla \cdot (\mathbf{u}h) = 0. \tag{5.1b}$$

η can be interpreted as pressure for the atmosphere [Gill, 1982]. The shallow water system is forced with a zonal wind stress $\mathbf{F}$. The dissipation term $\mathbf{D}$ removes energy on large scales (bottom friction, Arbic & Scott [2008]) and on small scales (diffusion, [Griffies & Hallberg, 2000]). The non-linear term $(\mathbf{u}\cdot\nabla)\mathbf{u}$ represents advection of momentum. The term $f\hat{\mathbf{z}} \times \mathbf{u}$ is the Coriolis force and $-g\nabla\eta$ is the pressure gradient force, with g being

the gravitational acceleration. Eq. 5.1b is the shallow water-variant of the continuity equation, ensuring conservation of mass.

The shallow water equations are solved in the (x, y)-plane over the zonally periodic rectangular domain $L_x \times L_y$, of size 2000 km $\times$1000 km. We associate x with the zonal and y with the meridional direction. The domain is centred at 45N with the beta-plane approximation [Vallis, 2006]. The boundary conditions are periodic in zonal direction and no-slip at the northern and southern boundary. The layer thickness is $h = \eta + H(x)$, with

$$H(x) = H_0 - H_1 \exp\left(-H_\sigma^{-2}(x - \tfrac{L_x}{2})^2\right) \tag{5.2}$$

being the undisturbed depth, representing a meridional mountain ridge at $x = \frac{L_x}{2}$ spanning from the southern to the northern boundary. The standard depth is $H_0 = 500$ m. The ridge has a maximum height of $H_1 = 50$ m and a characteristic width of $H_\sigma = 300$ km, which makes the zonal current barotropically unstable. The flow regime is therefore governed both by eddy-mean flow as well as eddy-eddy interactions [Ferrari & Wunsch, 2010]. The wind stress forcing $\mathbf{F} = (F_x, 0)$ is constant in time, acts only on the zonal momentum budget

$$Fx = \frac{F_0}{\rho h} \cos\left(\pi \left(y {L_y}^{-1} - 1\right)\right)^2 \tag{5.3}$$

and vanishes at the boundaries. The water density is $\rho = 1000\,\mathrm{kg\,m^{-3}}$ and $F_0 = 0.12$ Pa. The wind forcing acts as a continuous input of large-scale kinetic energy that is balanced by the dissipation term.

The dissipation term $\mathbf{D}$ is the sum

$$\mathbf{D} = -\frac{c_D}{h}\|\mathbf{u}\|\mathbf{u} - \nu\nabla^4\mathbf{u} \tag{5.4}$$

of a quadratic bottom drag with dimensionless coefficient $c_D = 10^{-5}$ [Arbic & Scott, 2008] and a biharmonic diffusion with viscosity coefficient $\nu \approx 1.33\times10^{11}\ \mathrm{m^4\,s^{-1}}$ [Griffies & Hallberg, 2000].

The shallow water equations are discretised using 2nd order centred finite differences on an Arakawa C-grid [Arakawa & Lamb, 1977] and the fourth-order Runge-Kutta method [Butcher, 2016] is used for time integration of the pressure, Coriolis and advective terms, whereas a semi-implicit method is used for the dissipative terms $\mathbf{D}$. We present results of simulations with three different levels of resolution: high resolution

simulations with a grid spacing of $\Delta = 5\,\text{km}$ (400x200 grid points), medium resolution simulations with a grid spacing of $\Delta = 20\,\text{km}$ (100x50 grid points) and low resolution with a grid-spacing of $\Delta = 40\,\text{km}$ (50x25 grid points). The grid spacing in x and y-direction is always equal, $\Delta x = \Delta y = \Delta$. The time step $\Delta t = 282\,\text{s}$ at $\Delta = 20\text{km}$ is chosen to resolve surface gravity waves, traveling at an estimated phase speed of $\sqrt{gH_0}$ with a CFL number close to 1 and gravitational acceleration $g = 10\,\text{ms}^{-1}$. The advection terms are discretised using an energy and enstrophy conserving scheme [Arakawa & Hsu, 1990; Salmon, 2004, 2007].

The shallow water equations are extended with an advection equation for tracers. Temperature and humidity or salinity are examples of tracers in the atmosphere and the ocean. For simplicity, we regard them as passive here, such that they do not influence the flow. The advection of a passive tracer q given a velocity $\mathbf{u}$ is governed by

$$\frac{\partial q}{\partial t} + \mathbf{u} \cdot \nabla q = 0. \tag{5.5}$$

A semi-Lagrangian advection scheme [Smolarkiewicz & Pudykiewicz, 1992] is used to discretise Eq. 5.5. In this discretisation, the tracer concentration for a given grid cell, which is considered to be the arrival point, is calculated from the concentration at a departure point at the previous time step. Starting from each arrival point, velocities are traced back to find the respective departure points. While the arrival points correspond to grid cells, the departure points in general do not coincide with the grid cells, such that an interpolation is required. The concentrations at the surrounding grid points are bilinearly interpolated onto the departure point, which is then used as the concentration at the arrival point one advective time step later.

5.2.2 Scaling and reordering the shallow water equations

Equations can be rescaled via multiplication with a constant rescaling factor s to shift the dynamic range occurring in an algorithm towards larger numbers (for $s > 1$) or towards smaller numbers ($s < 1$). Rescaling can be used to adjust the number range to the decimal precision of the number formats (see section 2.3.3). If nonlinear terms are considered, this multiplicative rescaling is ineffective, as the rescaling factor s appears inside the non-linear terms. The nonlinear terms are therefore effectively invariant under multiplicative rescaling and only the linear terms are scaled by s. Fig. 5.1 includes histograms of numbers occurring in the rescaled Lorenz system [Jeffress *et al.*, 2017; Kwasniok, 2014; Lorenz, 1963; Tantet *et al.*, 2018] as an example from Klöwer *et al.*

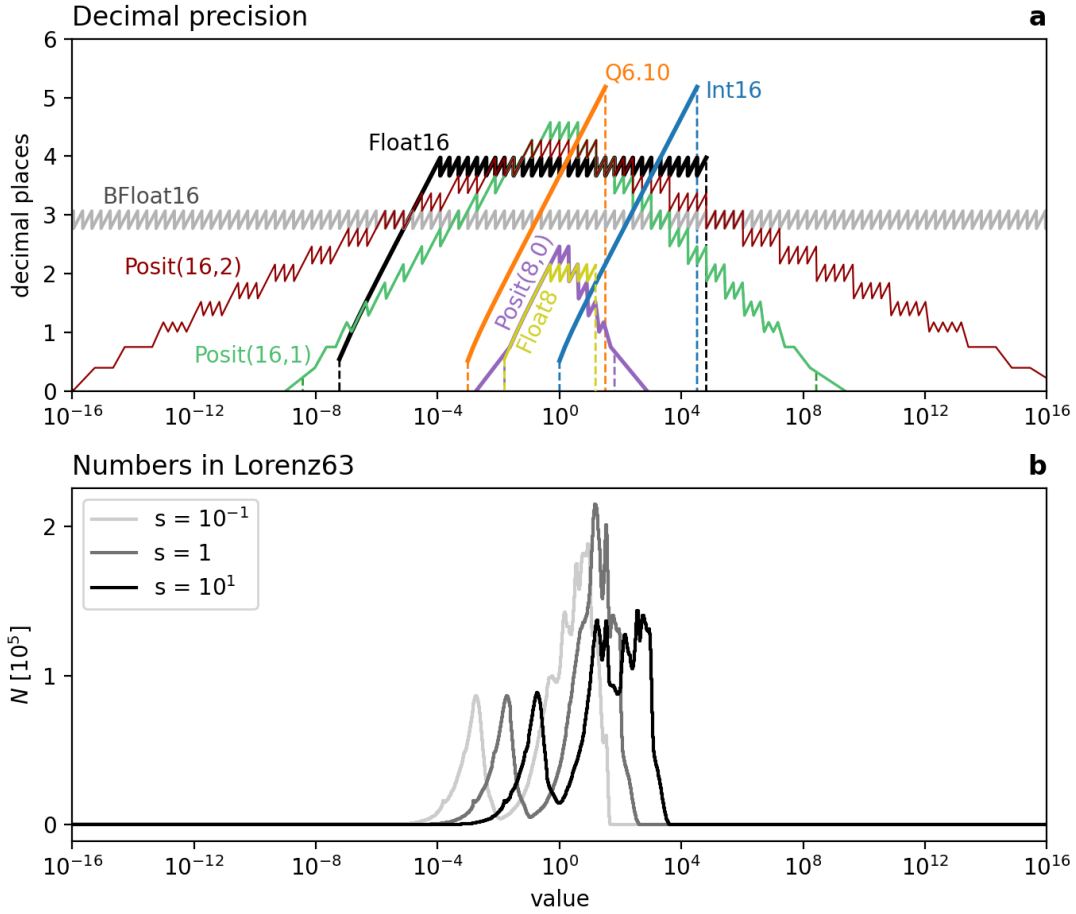

Figure 5.1 | Matching arithmetic results to the decimal precision of number formats. **a** Decimal precision for various number formats. Dashed vertical lines indicate for each format the range from *minpos* to *maxpos* of representable numbers. Float64, Float32 and Posit32 are beyond the axes limits. **b** Histogram of results of the absolute values of all arithmetic operations in Lorenz 1963 system rescaled by *s* from [Klöwer *et al.*, 2019].

[2019]. For posit arithmetic it is preferable to use $s = \frac{1}{10}$ in the Lorenz system to scale the prognostic variables to match the range of highest decimal precision around ± 1, which increases the complexity of the Lorenz attractor and decreases the average rounding error.

Furthermore, it is sometimes possible to avoid intermediate arithmetic results, which may be outside the dynamic range of a number format, by changing the order in which multiplications and divisions are executed. In general, it is preferable to combine such operations to a single multiplication with a constant which can be precomputed. Although this will have a negligible effect on the rounding error for floating-point arithmetic due to the approximately constant decimal precision throughout the range of numbers (subnormals excluded), it reduces the risk of over or underflow (see also Fig. 6.2 and chapter 6 in general).

The dynamic range of representable numbers with 16-bit arithmetic is for most formats discussed here considerably smaller than with Float32 or Float64 (Table 2.1). It is therefore important to rescale the calculations to limit the range of arithmetic results to stay within the bounds of a given 16-bit format.

The prognostic variables in the shallow water equations are typically $\mathcal{O}(1\,\mathrm{ms}^{-1})$ for $\mathbf{u}$, $\mathcal{O}(1\,\mathrm{m})$ for η and $\mathcal{O}(1)$ for q in the barotropic setup is used (section 5.2.1). Their physical units are therefore retained in the discretised numerical model and we do not apply a rescaling of the shallow water equations as a whole in this chapter. Chapter 6 rescales the shallow water equations fully to avoid subnormals, which is discussed in more detail in section 6.2.1.

In this chapter, however, we introduce dimensionless gradients as the grid spacing Δ in units of meter is large for geophysical flows. We therefore use dimensionless Nabla operators $\tilde{\nabla} = \Delta\nabla$. The continuity equation Eq. 5.1b, for example, reads then as

$$\eta^{n+1} = \eta^n + \tilde{\Delta t}\left(-\tilde{\nabla}\cdot(\mathbf{u}h)^n\right) \tag{5.6}$$

for an explicit time stepping scheme. $\tilde{\Delta t}$ is the rescaled time step, a Runge-Kutta coefficient times $\frac{\Delta t}{\Delta}$ to combine a division by a large value for Δ and a subsequent multiplication with a large value for Δt into a single multiplication with $\tilde{\Delta t}$. The other terms are rescaled accordingly ($\tilde{f} = f\Delta$; $\tilde{\mathbf{F}} = \mathbf{F}\Delta$). As these terms remain constant, they can be precomputed at higher precision during model initialisation. The momentum equations are rescaled similarly.

Diffusion is an example of a discretization scheme that requires rescaling for the arithmetic results to fit into the limited dynamic range. Biharmonic diffusion [Griffies

& Hallberg, 2000] calculates a fourth-derivative in space, which is with $\mathcal{O}(10^{-20})$ often very small in geophysical applications, due to the large physical dimensions, when using meters as a unit for length. Contrarily, biharmonic viscosity coefficients are typically very large $\mathcal{O}(10^{11})$. We therefore rescale $\mathbf{D}$ accordingly

$$\tilde{\mathbf{D}} = -\frac{\tilde{c_D}}{h}\|\mathbf{u}\|\mathbf{u} - \tilde{\nu}\tilde{\nabla}^4\mathbf{u} \tag{5.7}$$

with $\tilde{c_D} = c_D\Delta = 0.2\,\mathrm{m}$, and $\tilde{\nu} = \nu\Delta^{-3} \approx 0.16\,\mathrm{ms}^{-1}$, which are precomputed. The term $\tilde{\mathbf{D}}$ is computed instead of $\mathbf{D}$ and the scaling eventually undone when multiplying with the rescaled time step $\tilde{\Delta}t$.

5.2.3 A 16-bit semi-Lagrangian advection scheme

The semi-Lagrangian advection scheme is reformulated for 16-bit arithmetics. In the absence of sources and sinks, the Lagrangian point-of-view states that the tracer concentration q does not change following a flow trajectory. The concentration q at departure points $\mathbf{x}_d$ at time t is therefore the same as the concentration at time $t + \Delta t_{\mathrm{adv}}$ at arrival points $\mathbf{x}_a$, which are chosen to coincide with the grid points. The advective time step Δt_{adv} does not have to be equal to the time step of other terms, hence we distinguish it here from Δt. Based on the flow velocity at the arrival point, the departure point is derived. To avoid large numbers for the coordinates (in our case $L_x = 2 \cdot 10^6$m), non-dimensional departure points $\tilde{\mathbf{x}}_{d,rel}$ relative to the arrival point are computed as

$$\tilde{\mathbf{x}}_{d,rel} = -\mathbf{u}(\mathbf{x}_a, t + \Delta t_{\mathrm{adv}})\left(\frac{\Delta t_{\mathrm{adv}}}{\Delta}\right). \tag{5.8}$$

A scaling with the grid-spacing inverse Δ^{-1} is applied such that all terms are $\mathcal{O}(1)$ and therefore representable with 16-bit arithmetics. In practice, when converting the relative departure point $\tilde{\mathbf{x}}_{d,rel}$ to an array index for the interpolation, the floor function is used in combination with integer arithmetics. This essentially separates a computation with reals into two parts. One that can be computed with integers without rounding errors, and a calculation with reals, with a removed offset to reduce rounding errors.

5.2.4 Mixed precision

In many models, it may be challenging to use 16-bit arithmetic throughout the entire model. Some model components will be more sensitive to a reduction in precision when

compared to others and it often makes sense to reduce precision only in those components where results are not deteriorated, while keeping precision high in precision-sensitive components. This approach is called *mixed precision* and is already used for the reduction to single precision in ocean and atmosphere models [Tintó Prims *et al.*, 2019; Váňa *et al.*, 2017]. Mixed precision requires conversion between high and low-precision formats, which introduces an additional computational step. The conversion either has to be done efficiently on hardware, such as NVidia's tensor cores [Haidar *et al.*, 2018; Hatfield *et al.*, 2019]. Otherwise, the number of conversions should be kept low to reduce overhead [Higham *et al.*, 2019]. Mixed precision is therefore especially attractive when using low precision for a comparably independent algorithm, such that conversion back to high precision is only required once for the result.

5.2.5 Reduced-precision communication for distributed computing

Complex weather and climate models rely on parallelisation to distribute the computational cost of simulations efficiently among the processing units in a large cluster or supercomputer [Fuhrer *et al.*, 2018]. Parallel execution typically requires domain decomposition, where the spatial domain is split into many subdomains to be calculated separately on individual processing units [Chan & Mathew, 1994; Toselli & Widlund, 2004]. Domain decomposition requires communication of the boundary values of a subdomain with the neighbouring subdomains. If 16-bit arithmetic cannot be used within the entire model, it may still be possible to reduce precision in the communication between processors [Fan *et al.*, 2019].

Not all weather and climate models would benefit from a reduced-precision communication as the acceleration potential depends on many factors specific to a model and the used hardware, e.g. number of nodes in a cluster and how shared and distributed memory is managed. It will also be important whether communication is latency or volume bound. Latency bound communication is bound by the time a package of information requires to travel between processors. In contrast, volume bound communication is limited by the bandwidth that is available for communication. Only the latter will benefit from a reduction in data volume, which can be achieved with reducing precision.

However, if communication volume is an identified bottleneck in a given application, which is often the case in weather and climate models, a reduction in computing time can be achieved with reduced-precision communication. Reliable model simulations might be possible with 16 or even 8-bit communication, which would result in a significant reduction in communication volume. In general, various lossy and lossless data

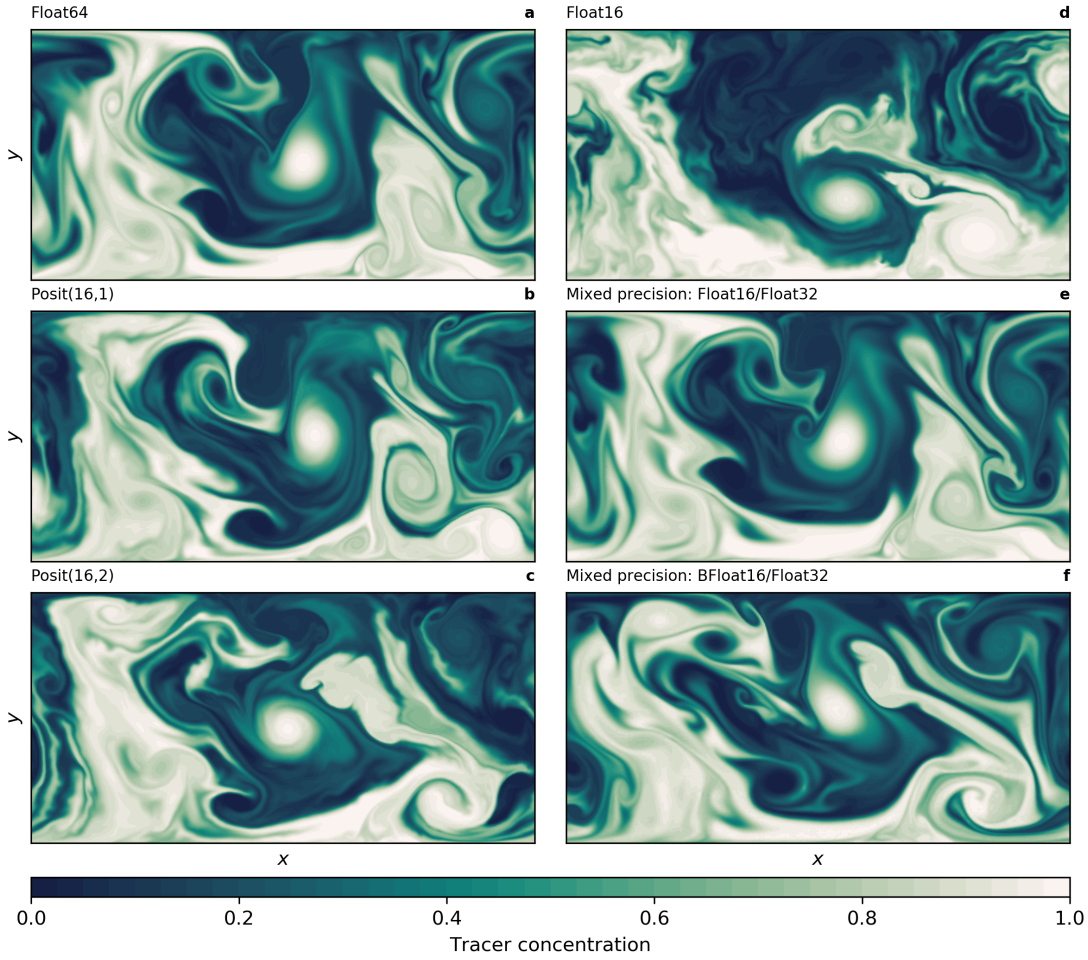

Figure 5.2 | Snapshot of tracer concentration simulated by the shallow water model using different 16-bit number formats. The high-resolution configuration ($\Delta = 5$km) is used with 400x200 grid points. The mixed-precision simulations presented in **e** and **f** use Float32 for the prognostic variables only. The tracer was injected uniformly in the lower half of the domain 50 days before the time step shown.

compression techniques can be used to reduce communication volume. Lossless communication allows for bit-reproducible compression, but reductions in communication volumes are small due to many high entropy mantissa bits (see also section **??** and chapter 3 in general). We therefore restrict ourselves to lossy type conversions to Float16, Float8 or Posit8, which introduce rounding errors to the data that is sent around while the overhead due to encoding before and decoding after communication remains small.

5.3 Impact of low precision on the physics

The shallow water model simulates vigorous turbulence interacting with a zonal current (Fig. 5.2). Both float and posit arithmetic in 16 bit present very similar fluid dynamics in comparison to the Float64 reference. A snapshot of tracer concentration many simulated days after initialisation reveals turbulent mixing of the tracer that is also well simulated with posits. However, with Float16 the simulation deviates faster from the reference than with Posit(16,1) (also called Posit16, as 1 exponent bit is the default configuration for 16-bit posits, see Table 2.1) and to a lesser degree with Posit(16,2). This is presumably due to larger rounding errors triggering the small scale instabilities visible in the snapshot as wavy filaments and fronts. These instabilities are clearly triggered by Float16 arithmetics, but to a lower degree also visible for posits. This provides some visual evidence that accumulated rounding errors are reduced with posits, especially Posit(16,1). BFloat16 arithmetic is not able to simulate the shallow water dynamics, as tendencies are too small to be added to the prognostic variables. Hence, a stalling of the simulated flow is observed. The results with mixed precision, i.e. using Float16/Float32 and BFloat16/Float32, will be discussed in section 5.3.5.

5.3.1 Error growth

Short-term forecasts at medium-resolution ($\Delta = 10$km) are performed to analyse the differences between different 16-bit arithmetics. To quantify the error growth caused by rounding errors with different arithmetics in a statistically robust way, we create a number of forecasts with each member starting from one of 200 randomly picked start dates from a 50-year long control simulation. The forecast error in the shallow water model is computed as the root mean square error (RMSE) of sea surface height η with respect to Float64 simulations. Other variables yield similar results. Each forecast is performed several times with the various number formats but from identical initial conditions. The error growth caused by rounding errors is additionally compared to the er-

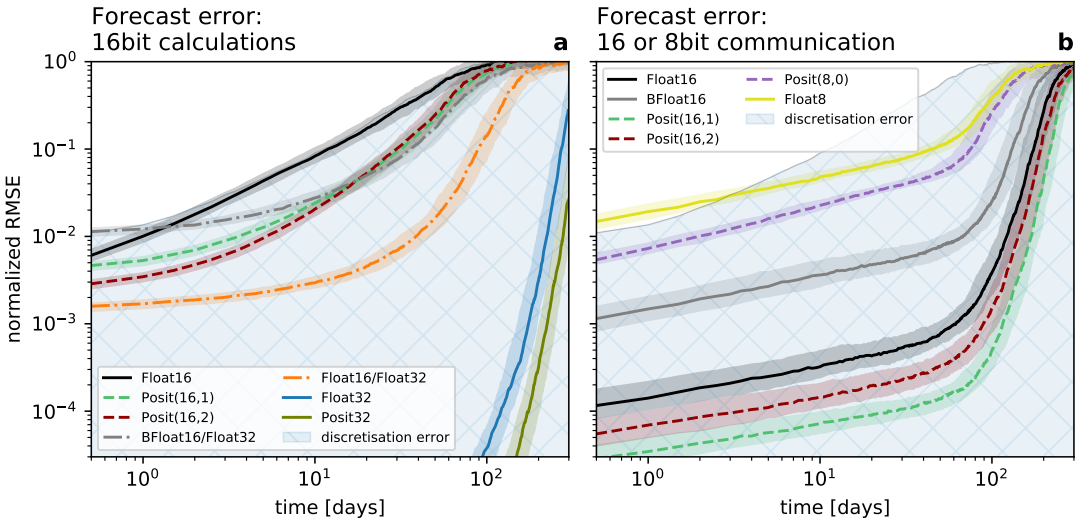

Figure 5.3 | Forecast error of sea surface height η measured as root mean square error (RMSE) taking Float64 as reference. a Forecast error for various 16-bit number formats and mixed 16/32-bit simulations for which the prognostic variables are kept in Float32. **b** Forecast error for reduced-precision communication in 8 or 16 bit with various number formats used for encoding, with Float64 used for all calculations. The communication of boundary values occurs at every time step for the prognostic variables. The RMSE is normalised by a mean forecast error at very long lead times. Solid or dashed lines represent the median of 200 forecasts per number format. The shaded areas of each model configuration denote the interquartile range of the forecast experiments.

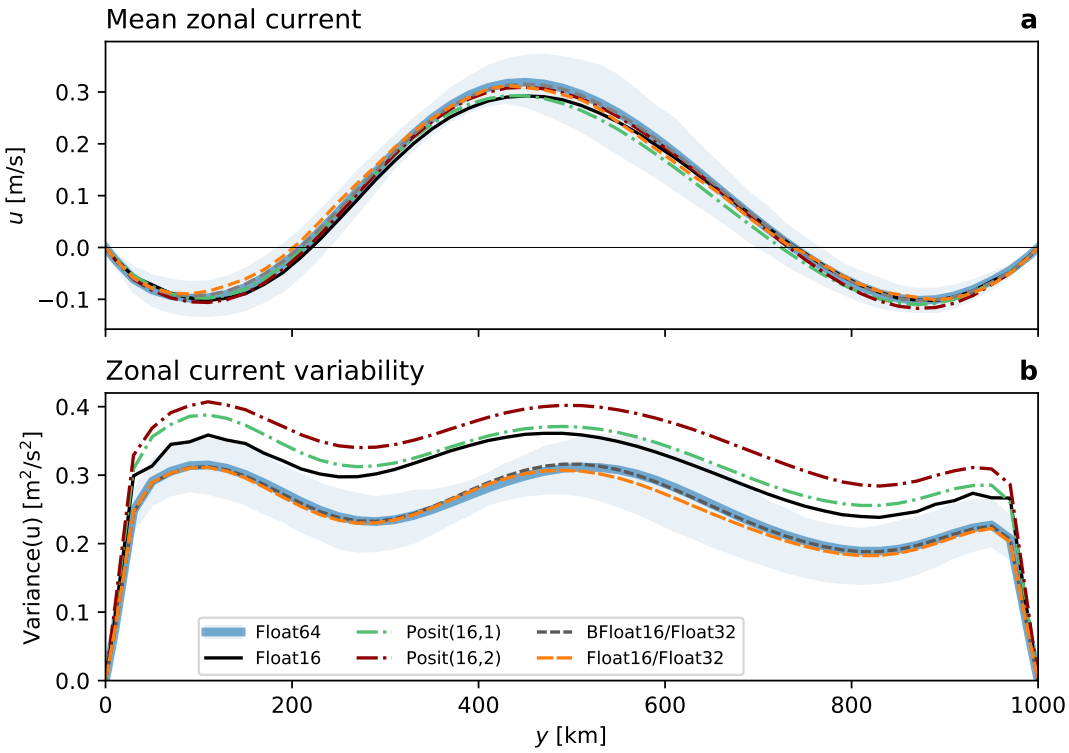

Figure 5.4 | Climatology and variability of the zonal current in the medium-resolution simulations. **a** Zonally-averaged zonal current u as a function of the meridional coordinate y. **b** Zonal variance of the zonal current as a function of y. The dashed lines for BFloat16/Float32 and Float16/Float32 are almost identical. The shaded area denotes the interquartile temporal variability around the **a** mean and **b** variance of reference simulations with Float64.

ror introduced by discretisation. A low-resolution model configuration with $\Delta = 20$km is used to quantify a realistic level of discretisation error. The RMSE is normalised by the climatological mean forecast error at very long lead times, which is the same for all model configurations. When the normalised RMSE reaches 1 all information on the initial conditions is removed by the chaotic evolution of the shallow water system.

The forecast error of Float16 is as large as the discretisation error and clearly larger than with 16-bit posit arithmetic (Fig. 5.3a). Both 16-bit posit formats, i.e. Posit(16,1) with 1 exponent bit and Posit(16,2) with two, yield a forecast error that is several times smaller than Float16. The forecast error of 32-bit arithmetic is several orders of magnitude smaller and is only after 200 days as large as the error for 16-bit arithmetic at short lead times of about 10 days. Also at 32 bit, posit arithmetic clearly causes smaller rounding errors and therefore outperform floats.

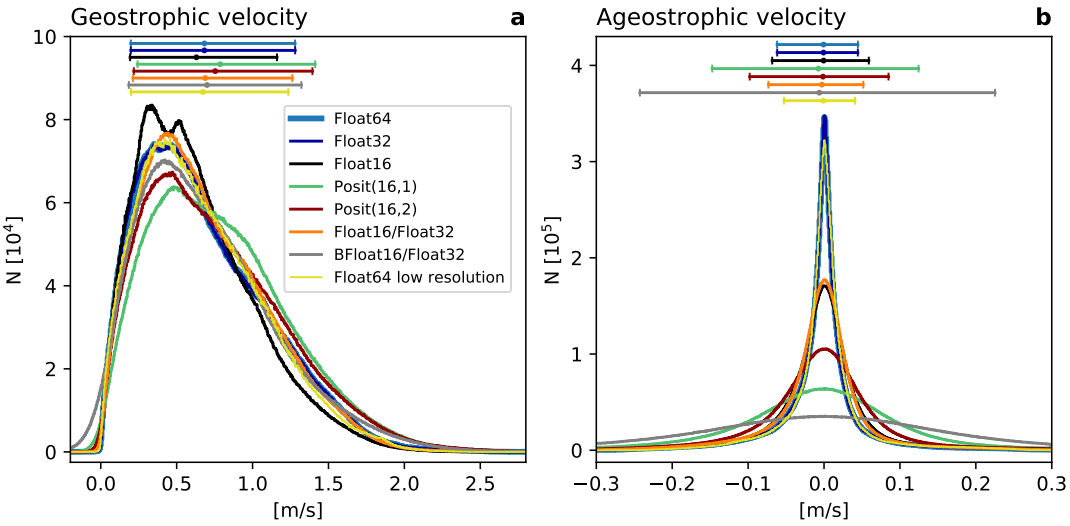

Figure 5.5 | Geostrophic balance as simulated with different number formats. a Histograms of flow-parallel components of geostrophic velocity. **b** as (a) but for the ageostrophic velocities, as defined by Eq. 5.10. Horizontal bars denote the mean, 10th and 90th-percentile in respective colours.

5.3.2 Mean and variability

To investigate the effect of rounding errors on the climatological mean state of the shallow water system, we zonally average the zonal velocity u. This average is based on 300-day long simulations starting from 200 different initial conditions, which cover the various states in the long-term variability of the shallow water system. The climatology from a single very long simulation has not been assessed.

The mean state is an eastward flow of about 0.3 m/s, about 3 to 4 times weaker than instantaneous velocities throughout the domain (Fig. 5.4a), which is typical for turbulent flows. A weak westward mean flow is found at the northern and southern boundary. No 16-bit format was found to have a significant impact on the mean state. The variability of the flow around its mean state is high throughout the domain (Fig. 5.4b). The variability is significantly increased by 10 – 30% with 16-bit arithmetic, especially with Posit(16,2). This is presumably caused by rounding errors that are triggering local perturbations which increase variability.

5.3.3 Geostrophy

The turbulence in shallow water simulations is largely geostrophic, such that the pressure gradient force opposes the Coriolis force. The resulting geostrophic velocities $\mathbf{u}_g$

can be derived from the sea surface height η as

$$\mathbf{u}_g = \frac{g}{f}\hat{\mathbf{z}} \times \nabla\eta \tag{5.9a}$$

$$\mathbf{u} = \mathbf{u}_g + \mathbf{u}_{ag} \tag{5.9b}$$

and deviations from the actual flow $\mathbf{u}$ are the ageostrophic velocity components $\mathbf{u}_{ag}$. We project both components on the actual velocities to obtain the flow-parallel components $\tilde{u}_g$ and $\tilde{u}_{ag}$ via

$$\tilde{u}_g = \frac{\mathbf{u}_g \cdot \mathbf{u}}{\|\mathbf{u}\|}, \quad \tilde{u}_{ag} = \frac{\mathbf{u}_{ag} \cdot \mathbf{u}}{\|\mathbf{u}\|}. \tag{5.10}$$

The geostrophic velocities in the shallow water simulations can reach up to 2 m/s, are rarely negative (i.e. against the flow $\mathbf{u}$) and have a mean of about 0.7 m/s (Fig. 5.5a). This behaviour is well simulated with 16-bit number formats, although both 16-bit posit formats increase the mean of geostrophic velocities slightly. Ageostrophic velocity components are found to be largely isotropic, and are therefore oriented equally frequent with and against the prevailing flow. They rarely exceed $\pm$0.1m/s and are comparably small, as expected in geostrophically balanced turbulence. Ageostrophic velocities can be seen as a measure of the physical instabilities in the flow field and their variance is indeed increased when simulated with 16-bit number formats. Float16 and posits show clearly fewer ageostrophic velocities around 0, pointing towards an increased number of simulated instabilities. Especially Posit(16,1) increases the variance of ageostrophic velocities by more than a factor of two. It is unclear where in the model integration rounding errors of 16-bit arithmetic trigger instabilities that lead to the observed increase in ageostrophy. We conclude that although the geostrophic balance in the simulations is maintained, rounding errors lead, likely due to an increase in ageostrophy, to a higher variability in the flow field.

5.3.4 Gravity waves

As 16-bit arithmetics have no significant impact on the climatological mean state, histograms of prognostic variables are also not changed (Fig. 5.6a and b). However, the tendencies are increased by orders of magnitude with 16-bit arithmetics (Fig. 5.6c and d) as rounding errors cause gravity waves to radiate away from eddies (Fig. 5.6f). Gravity waves are identified from the tendency of sea surface height η. Comparing their propagation to the location of anomalous sea surface height, which is used as a proxy to

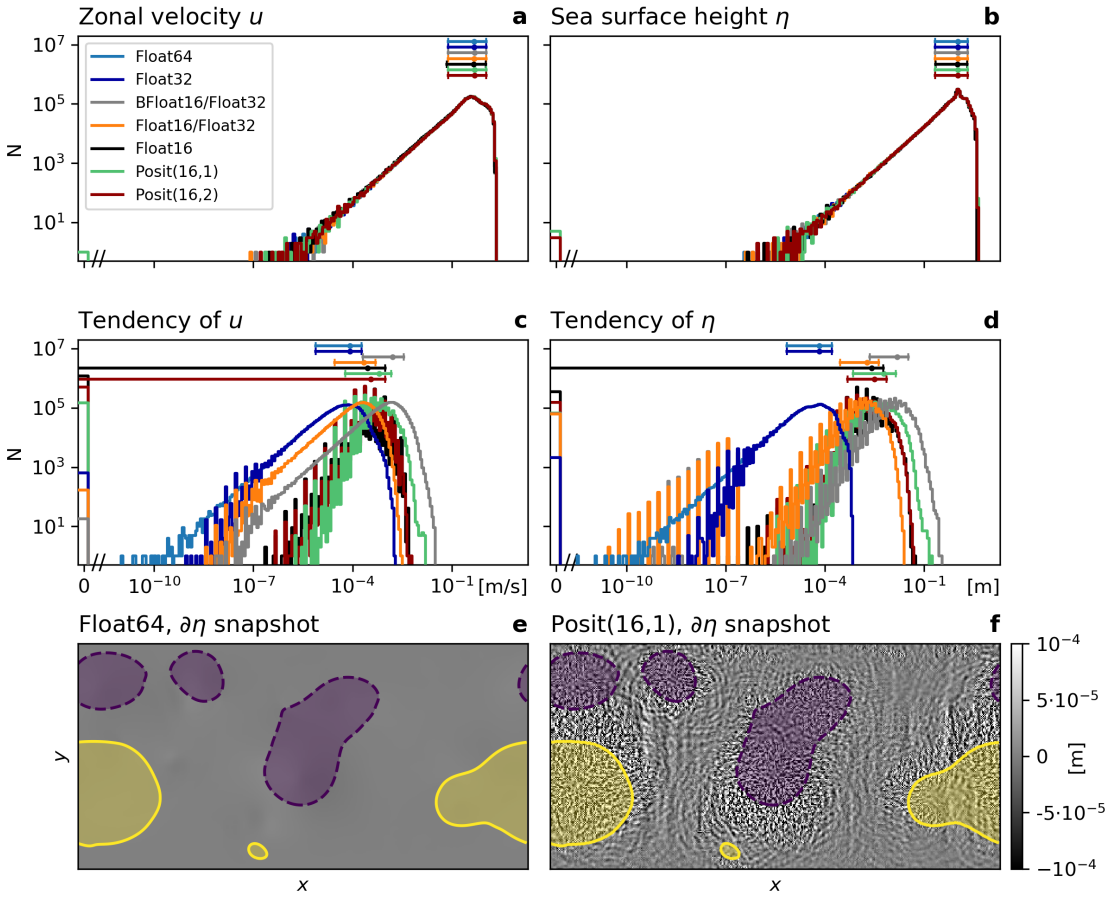

Figure 5.6 | Detecting gravity waves due to rounding errors from low precision. Histograms of the numeric values of the prognostic variables **a** zonal velocity u, **b** sea surface height η, and the respective tendencies of **c** u and **d** η, simulated with different 16, 32 and 64-bit number formats. Mean, 10th and 90th percentile are shown above the histograms in respective colours. Snapshots of the tendencies of η simulated with **e** Float64 and **f** Posit(16,1). Snapshots are similar for other 16-bit formats (not shown here). Areas of sea surface height anomalies exceeding ± 1.4 m are shown in purple (negative) and yellow (positive). Note the break on the x-axis close to zero in **a-d**.

locate eddies, we assume that rounding errors in regions of high eddy activity lead to instabilities that propagate away in the form of gravity waves. These gravity waves are not present in Float64 simulations (Fig. 5.6e) and tend to have only a small impact on quasi-geostrophic dynamics, as they act on different time and length scales. It is unclear but possible that gravity waves cause the observed increased ageostrophic velocities for 16-bit arithmetic.

Tendencies are about 4 orders of magnitude smaller than the prognostic variables. This poses a problem for number formats with a machine epsilon, measured as decimal precision, significantly lower than 4 decimal places (Table 2.1). Float16 has a machine epsilon of 3.7, which is presumably close to the lower limit beyond which the addition of tendencies will be round back. The BFloat16 number format has a machine epsilon of 2.8, which explains why flow is stalling when simulated with BFloat16.

5.3.5 Mixed-precision results

In the previous simulations the entire shallow water simulation was performed with the specified number format. As the addition of tendencies to the prognostic variables was identified as a key calculation that is error-prone, we investigate now the benefits of mixed-precision arithmetic, where Float32 is used for the prognostic variables but the tendencies are computed with either Float16 or BFloat16, two number formats that have the lowest decimal precision around 1. The prognostic variables are now reduced to Float16 or BFloat16 before the tendencies are calculated and every term of the tendencies is converted back before addition to the prognostic variables. The continuity equation (Eq. 5.1b) then becomes

$$\frac{\partial \eta_{32}}{\partial t} = -\,\text{Float32}(\partial_x(u_{16}h_{16}) + \partial_y(v_{16}h_{16})) \tag{5.11}$$

and similar for u and v in Eq. 5.1a. Subscripts 16 and 32 denote variables held at 16 and 32-bit precision, respectively, and Float32() is the conversion function.

Snapshots of tracer concentration reveal well simulated geostrophic turbulence (Fig. 5.2e and f) with Float16/Float32 and BFloat16/Float32 and instabilities at fronts or in filaments are visibly reduced compared to 16-bit arithmetic for all calculations. The forecast error growth is strongly reduced once the prognostic variables are kept as Float32 (Fig. 5.3a), supporting the hypothesis that the addition of tendencies to the prognostic variables is a key computation with low rounding error-tolerance. Although BFloat16 is not suitable for shallow water simulations when applied to all computations, mixing

BFloat16 with Float32 yields a similar error growth to posits, which is well below the discretisation error. Mean state or variability are virtually identical for both mixed-precision cases (Fig. 5.4) compared to the Float64 reference. The geostrophic balance is largely unaffected, but ageostrophic velocities increase in variance, especially for BFloat16 (Fig. 5.5). Gravity waves are similarly present for mixed precision although weaker for tendencies computed with Float16 (Fig. 5.6d) and, as discussed, they tend to not interact with the geostrophic time and length scales. Although the results show that Float16 is generally a preferable number format over BFloat16 for the applications presented here, we acknowledge that the conversion between Float32 and Float16 will come with some computational cost. In contrast, the conversion between BFloat16 and Float32 is computationally very cheap as both formats have the same number of exponent bits. Removing significant bits, applying rounding, and padding trailing zeros, are the only operations for this conversion. Following the results here, mixing 16 and 32-bit precision is found to be an attractive solution to circumvent spurious behaviour due to 16-bit floating-point arithmetics. Performance benefits are still possible as most calculations are performed with 16 bit, with error-critical computations in 32 bit to reduce the overall error.

Using mixed-precision in our shallow water model, 77% of the arithmetic operations are performed in 16 bit and the remaining 23% in 32 bit. Assuming Float16/BFloat16 to be two times faster than Float32 and conversion costs to be negligible this would yield another 40% reduction in computing time on top of a reduction from Float64 to Float32. However, this depends on the soft and hardware implementation considered. Some of the 16-bit accelerators (GPU/TPU) can increase the flop rate by more than a factor of 2 when compared to Float32. In addition, the shallow water model regarded here has a comparably simple right-hand side, such that more complex models will spend more time to compute tendencies which will come with a larger performance increase.

Mixed-precision is an attractive solution as hardware-accelerated 16-bit floating-point arithmetic is already available on graphic or tensor processing units and implementations therefore do not rely on the development of future computing hardware, which is the case for posits.

5.3.6 Reduced-precision communication results

A standard method to parallelise simulations is the distributed-memory parallelism via Message Passing Interface (MPI, The MPI Forum [1993]). We emulate MPI-like communication in the shallow water model with the copying of boundary values between

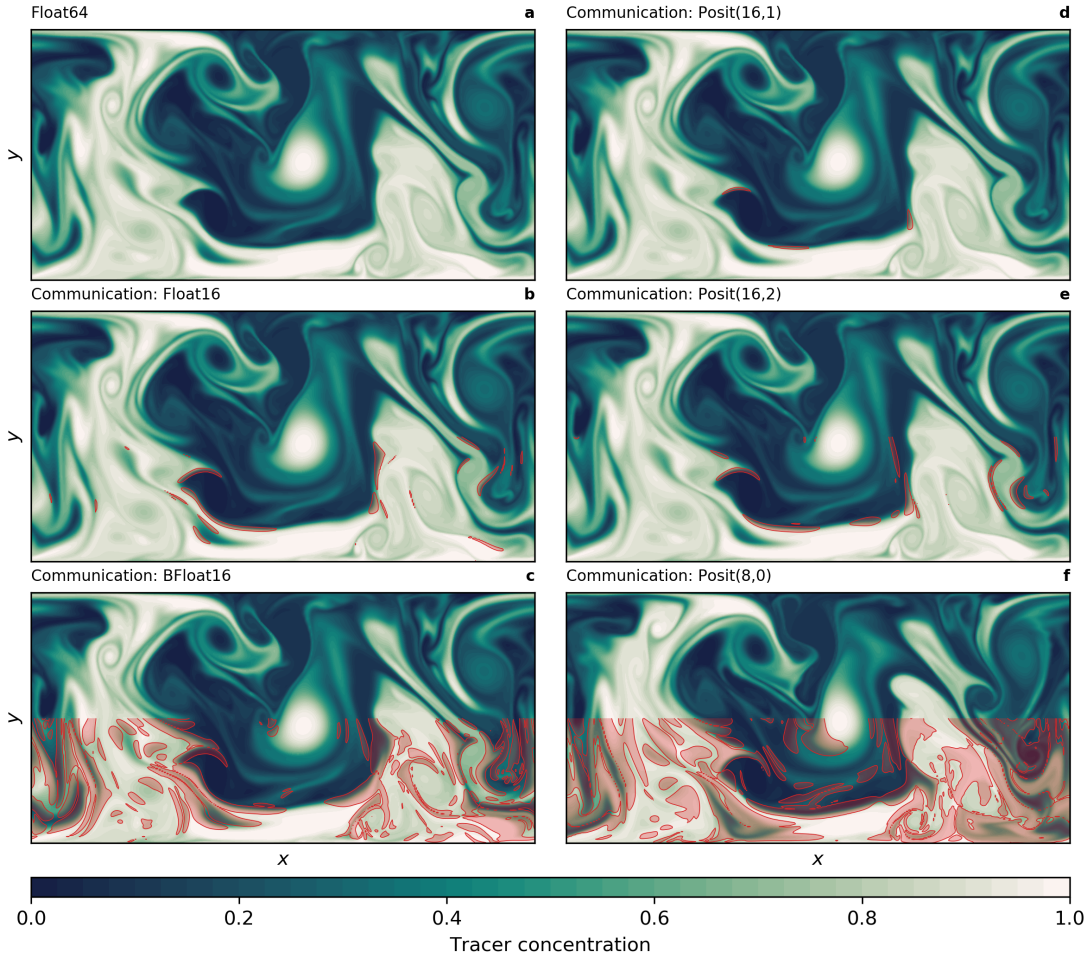

Figure 5.7 | Snapshot of tracer concentration simulated by the shallow water model using reduced-precision communication. The communication of boundary values occurs at every time step for the prognostic variables. Float64 was used for all calculations. Areas where the absolute error exceeds 0.05 are shaded in red only in the lower half of the domain. The tracer was injected uniformly in the lower half of the domain 50 days before. This simulation was run in the high-resolution configuration ($\Delta = 5$km).

the right and left boundary (periodic boundary conditions). Although the shallow water model does not run in parallel, reducing the precision in the copying of boundary values introduces an equivalent error as if reduced-precision MPI communication was used between subdomains. Reduced precision is applied for the communication of the prognostic variables at every Runge-Kutta substep.

Regarding snapshots of tracer concentration simulated with reduced-precision communication shows a negligible error for Float16 and 16-bit posits (Fig. 5.7). The error is largest at fronts and not concentrated around the boundaries. Encoding the communication with BFloat16 introduces a larger error than for the other 16-bit formats as the decimal precision is with 2.8 clearly lower (Table 2.1) for the range of values occurring within the prognostic variables (Fig. 5.6a and b). The errors are quantified by the RMSE of surface height η as before and are up to about two orders of magnitude smaller than the errors that result from 16-bit arithmetic. Even the worst format for 16-bit communication, BFloat16, has a smaller error than the best mixed-precision formats, Float16 with Float32. We therefore extend the short-term forecast experiments to include two 8-bit formats, Posit(8,0) and Float8 (see Table 2.1 for a description). Both formats are found to be suitable for reduced-precision communication here and do not introduce an error that is larger than the discretisation error. Having said that, Float8 communication introduces an error that is comparably large in the first days but growths only linearly in the first 50 days of the simulation, which is in contrast to the exponential error growth observed for 16-bit arithmetic.

5.4 Discussion

Future high-performance computing architecture will increasingly support 16-bit arithmetics. The wall-clock time for weather and climate simulations could be greatly reduced if computationally demanding algorithms were run at such reduced precision. We tested a number of options for 16-bit arithmetic for weather and climate applications in a shallow water model. The best results were achieved with 16-bit posits which appear very promising for application in high-performance computing for Earth System modelling. Float16 can be used to perform forecasts with the shallow water model while the application of BFloat16 was not successful.

In general, 16-bit arithmetics were not found to alter the climatological mean state or the large-scale dynamics. However, variability and ageostrophic velocities were increased, such that second and higher-order statistics should undergo testing to assess

the models reliability. Depending on the application, an increased variability does not necessarily deteriorate the model, especially for more realistic model set-ups than considered here. However, our findings suggest that reduced-precision changes need to be done carefully as specific simulation features can change without obvious impact on mean diagnostics.

Shallow water simulations with 16-bit arithmetic required rescaling of some terms but no major revisions of the model code or algorithms. Given that only floats are currently hardware-supported, we investigated mixed-precision approaches. Keeping the prognostic variables at 32 bit while computing the tendencies in 16 bit reduced the rounding errors significantly. We also showed that numerical precision for communication between compute nodes can be greatly reduced down to 16 or even 8-bit without introducing a large error. Reduced-precision communication was not found to have a significant impact on either mean state, variability, geostrophy or tendencies.

A *perfect model* is used in this study, such that any form of model or initial condition error is ignored and only the number format is changed between simulations. Solely discretisation errors are estimated by lowering the spatial resolution by a factor of 2. Although this is essential here to analyse the impact of rounding errors isolated from other errors, it is in general not a realistic configuration for weather or climate models. More complex models include many other sources of forecast error, such that the contribution of rounding errors from 16-bit arithmetic would likely be dwarfed by model, discretisation or initial condition errors.

Only the most common discretisation method for fluid dynamics was used in this study: Finite differences with an explicit time stepping scheme. But various other discretisation methods exist, such as finite element or volume, spectral methods and implicit time stepping. These methods come with different algorithms and associated precision requirements. Consequently, some might be less tolerant to rounding errors than the method used in this study.

There is currently no hardware available for posit arithmetic that we could have used for performance testing and it is seems impossible to make credible estimates whether such hardware would be faster or slower when compared to hardware optimised for Float16 arithmetic (as this does not only depend on theoretical considerations but also on investments into chip design). We therefore cannot draw any conclusion about the performance of posit arithmetic operations in comparison to Float16 or the other formats.

Until progress is made on hardware implementations for posits, the results here suggest that also 16-bit float arithmetic can successfully be used for parts of complex

weather and climate models with the potential for acceleration on graphic and tensor processing units. It is therefore recommended to adapt a type-flexible programming paradigm, ideally in a language that supports portability, with algorithms written to reduce the dynamic range of arithmetic results. Hardware progress on central, graphic or tensor processing units, with various numbers formats supported, can subsequently be utilised to accelerate weather and climate simulations.

6 Running on 16-bit hardware

Contributions This chapter is largely based on the following publication*

M Klöwer, S Hatfield, M Croci, PD Düben and TN Palmer, 2021. *Fluid simulations accelerated with 16 bit: Approaching 4x speedup on A64FX by squeezing ShallowWaters.jl into Float16*, **Journal of Advances in Modeling Earth Systems**, in review. Preprint 10.1002/essoar.10507472.2

Abstract. Most Earth-system simulations run on conventional CPUs in 64-bit double-precision floating-point numbers Float64, although the need for high-precision calculations in the presence of large uncertainties has been questioned. Fugaku, currently the world's fastest supercomputer, is based on A64FX microprocessors, which also support the 16-bit low-precision format Float16. We investigate the Float16 performance on A64FX with ShallowWaters.jl, the first fluid circulation model that runs entirely with 16-bit arithmetic. The model implements techniques that address precision and dynamic range issues in 16 bit. The precision-critical time integration is augmented to include compensated summation to minimize rounding errors. Such a compensated time integration is as precise but faster than mixed precision with 16 and 32-bit floats. As subnormals are inefficiently supported on A64FX the very limited range available in Float16 is $6{\cdot}10^{-5}$ to 65504. We develop the analysis-number format Sherlogs.jl to log the arithmetic results during the simulation. The equations in ShallowWaters.jl are then systematically rescaled to fit into Float16, using 97% of the available representable numbers. Consequently, we benchmark speedups of 3.8x on A64FX with Float16. Adding a compensated time integration the speedup is 3.6x. Although ShallowWaters.jl is simplified compared to large Earth-system models, it shares essential algorithms and therefore shows that 16-bit calculations are indeed a competitive way to accelerate Earth-system simulations on available hardware.

6.1 Introduction

The first numerical weather prediction models have recently moved away from 64-bit double-precision floating-point numbers for higher computational efficiency in lower

*with the following author contributions. Conceptualisation: MK, SH, MC. Data curation: MK. Formal Analysis: MK. Methodology: MK. Visualisation: MK. Writing – original draft: MK. Writing – review & editing: MK, SH, MC, PDD, TNP.

precision [Govett *et al.*, 2017; Nakano *et al.*, 2018; Rüdisühli *et al.*, 2013; Váňa *et al.*, 2017]. While both Float32 and Float64 formats are widely available for high-performance computing, support for 16-bit arithmetic is only available on mainstream hardware for a few years, due to the demand for low precision by the deep learning community. The transition towards 16 bit is challenging for an existing application: Rounding errors from low precision have to be controlled and a limited range of representable numbers cannot be exceeded without causing often catastrophic under and overflows. But the potential performance gains are promising, with 4x speedups compared to 64-bit calculations, not to mention the reduced energy consumption.

The current boom in machine learning applications is supported by advances in microprocessors. Instead of conventional central processing units (CPU), graphic and tensor processing units GPU, TPU [Jouppi *et al.*, 2018a, 2017, 2018b; Steinkraus *et al.*, 2005] are used, which are better suited for the workloads of machine learning. While most supercomputers from TOP500.org are based on Intel CPUs with the x86-64 architecture [Dongarra & Luszczek, 2011], many new installations transition towards GPUs or alternative microprocessor architectures [Zheng, 2020]. The trend is towards heterogeneous computing with specialised hardware, which is both a challenge and an opportunity for weather and climate models [Bauer *et al.*, 2021a,b]. Fugaku, the world's fastest supercomputer as of 2020, is based on Fujitsu's A64FX processors with ARM architecture [Odajima *et al.*, 2020; Sato *et al.*, 2020]. The A64FX also implements the Float16 format (1 sign, 5 exponent and 10 mantissa bits) and Fujitsu promises a 4x increase in the number of floating-point operations per second.

Float16 is the 16-bit variant of Float32 and Float64 and is defined in the 2008 revision of the IEEE-754 standard on floating-point arithmetic [IEEE, 1985, 2008]. Alternatives such as BFloat16 [Burgess *et al.*, 2019; Kalamkar *et al.*, 2019], minifloats [Fox *et al.*, 2020], logarithmic fixed-point numbers [Johnson, 2018, 2020; Sun *et al.*, 2020], posits [Chaurasiya *et al.*, 2018; Gustafson & Yonemoto, 2017; Klöwer *et al.*, 2019, 2020; Langroudi *et al.*, 2019; Zhang & Ko, 2020] and stochastic rounding [Croci & Giles, 2020; Hopkins *et al.*, 2020; Mikaitis, 2020; Paxton *et al.*, 2021] have been investigated, but most of these are not available on standard supercomputing hardware. Currently only floats (and integers) enjoy a widely available support in terms of hardware, libraries and compilers that ultimately make it possible to execute complex computational applications.

The use of low-precision number formats is motivated as in the presence of large uncertainties in the climate system rounding errors are masked by other sources of error [Palmer, 2015]. Typical rounding errors from high-precision calculations are many orders of magnitude smaller than errors in the observations, from coarse resolution

or underrepresented physical processes. Low-precision calculations are therefore, at least in theory, sufficient without a loss in accuracy for a weather forecast or a climate prediction. Emulated in parts of weather and climate models, 16-bit half precision has been shown to be a potential route to accelerated simulations [Chantry *et al.*, 2019; Dawson *et al.*, 2018; Hatfield *et al.*, 2019; Klöwer *et al.*, 2020].

Although weather and climate model data often comes with large uncertainties, many intermediate calculations inside a model simulation require a higher precision. Time integration is often a precision-critical part of numerical simulations of dynamical systems. Stability constraints require small time steps such that tendencies are often several times smaller than the prognostic variables [Courant *et al.*, 1967]. Adding the two yields a loss of precision from the tendency as small increments can only be poorly resolved in low precision [Gill, 1951; Kahan, 1965; Møller, 1965]. In extreme cases this can lead to a model stagnation [Croci & Giles, 2020], and is often dealt with using mixed-precision approaches [Dawson *et al.*, 2018; Klöwer *et al.*, 2020; Tintó Prims *et al.*, 2019], where the tendencies are computed in low precision, but converted to a high-precision format before addition. This is beneficial as a large share of computing time is accelerated with low precision, while precision-critical operations are kept in high precision.

Precision loss in calculations can be analysed with a variety of available tools, like FPBench [Damouche *et al.*, 2017], CADNA [Jézéquel & Chesneaux, 2008], Verrou [Fevotte & Lathuilière, 2019], and Verificarlo [Denis *et al.*, 2016]. Such tools are often either based on interval arithmetic, providing rigid rounding error bounds, or on stochastic arithmetic to assess the rounding error growth. While these can be useful to identify the minimal decimal precision for simulating chaotic systems, analysing the limited dynamic range of low-precision number formats is largely unaddressed in these tools.

In this chapter, we present, to our knowledge, the first fluid circulation model that runs entirely in hardware-accelerated 16-bit floats on the ARM architecture-based microprocessor A64FX. Strategies are presented to solve precision and range issues with 16-bit arithmetic: In section 6.2 we scale the shallow water equations, and an appropriate scale is found with the newly-developed analysis-number format Sherlogs.jl. Additionally, a compensated time integration is presented to minimise precision issues. Section 6.3 analyses the rounding errors of Float16 in ShallowWaters.jl and benchmarks the performance compared to Float64. Section 6.4 discusses the results.

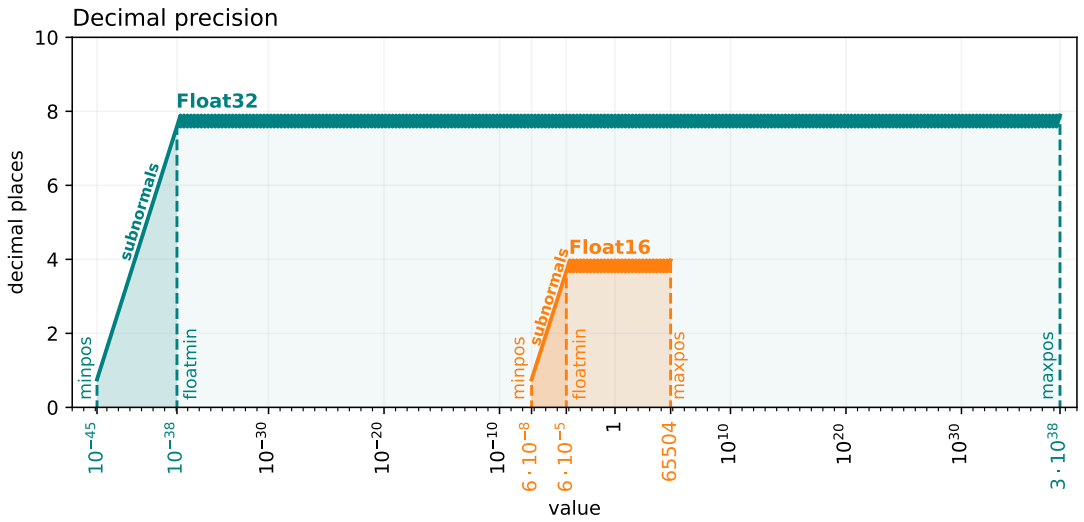

Figure 6.1 | Decimal precision of Float16 and Float32 over the range of representable numbers. The decimal precision is worst-case, i.e. given in terms of decimal places that are at least correct after rounding (see section 2.3.3). The smallest representable number (*minpos*, see section 2.1.3), the smallest normal number (*floatmin*) and the largest representable number (*maxpos*) are denoted with vertical dashed lines. The subnormal range is between *minpos* and *floatmin*.

6.2 Methods

The following methods describe in section 6.2.1 the scaling of the shallow water equations, which is extended from the dimensionless gradients introduced in section 5.2.2. The shallow water model presented here is an updated version from the one presented in chapter 5. Section 6.2.2 describes how to choose a scale guided by an analysis number format, a concept introduced in section 2.4.2. To reduce rounding errors in the time integration we describe the compensated time integration in section 6.2.3.

6.2.1 Scaling the shallow water equations

The shallow water equations describe atmospheric or oceanic flow idealised to two horizontal dimensions. They result from a vertical integration of the Navier-Stokes equations [Gill, 1982; Vallis, 2006] and are simplified but representative of many weather and climate models, which are usually solved with many vertically-coupled horizontal layers. They describe the time evolution of the prognostic variables velocity $\mathbf{u} = (u, v)$,

and interface height η in the following form

$$\begin{aligned}
&\partial_t \mathbf{u} + \mathbf{u} \cdot \nabla \mathbf{u} + f\mathbf{z} \times \mathbf{u} = -g\nabla\eta + \nu_B \nabla^4 \mathbf{u} - r\mathbf{u} + \mathbf{F} \\
&\partial_t \eta + \nabla \cdot (\mathbf{u}h) = 0 \\
&\partial_t q + \mathbf{u} \cdot \nabla q = -\tau(q - q_0)
\end{aligned} \tag{6.1}$$

defined over a rectangular domain with zonal and meridional coordinates x, y of size $L_x = 8000\,\text{km}$, $L_y = 4000\,\text{km}$, respectively. The domain is a zonal channel with boundary conditions being periodic in x. The channel setup is motivated by zonal flows like the Antarctic Circumpolar Current but highly idealised [Jansen *et al.*, 2015a,b].

The non-linear momentum advection is $\mathbf{u} \cdot \nabla \mathbf{u}$. The Coriolis force is $f\hat{\mathbf{z}} \times \mathbf{u} = (-fv, fu)$ with the Coriolis parameter f using a β-plane approximation at 45°N. The pressure gradient $-g\nabla\eta$ scales with a reduced gravitational acceleration $g = 0.01\,\text{ms}^{-2}$ to represent baroclinic ocean/atmosphere dynamics [Gill, 1982], which is in contrast to the barotropic model setup in chapter 5. The zonal wind forcing $\mathbf{F} = (F_x, 0)$ is a meridional shear $F_x = F_0 \sin(\omega t) \tanh(2\pi(L_y^{-1} - \frac{1}{2}))$ which reverses seasonally ($\omega^{-1} = 365\,\text{days}$). Lateral diffusion of momentum is described by $\nu_B \nabla^4 \mathbf{u}$, with biharmonic viscosity coefficient ν_B. Linear bottom friction is represented by $-r\mathbf{u}$ which decelerates the flow at a time scale of $r^{-1} = 300\,\text{days}$. The equation for interface height η is the shallow water-variant of the continuity equation, ensuring conservation of volume (and mass as the density ρ is constant). The layer thickness is $h = \eta + H$ of a fluid with depth H at rest. Several meridional ridges on the seafloor trigger instabilities in the zonal flow, but they are small compared to the fluid depth. The shallow water equations are complemented with an advection for the passive tracer q , which is stirred by the flow through $\mathbf{u} \cdot \nabla q$ and slowly ($\tau^{-1} = 100\,\text{days}$) relaxed back to a reference q_0.

In order to control the range of numbers occurring in the simulation, the shallow water equations are scaled with a multiplicative constant. The evaluation of linear terms is not affected, but the non-linear terms involve an unscaling. The same constant s is chosen for zonal velocity u and meridional velocity v, such that $\hat{u} = su$ and $\hat{v} = sv$. Additionally, we use dimensionless spatial gradients $\hat{\partial}_x = \Delta x \partial_x, \hat{\nabla} = \Delta x \nabla$, etc. by scaling the equations with the grid spacing Δx. For simplicity, we use the same Δx in x and y-direction but generalisation to less regular grids is possible. The grid spacing Δx is then combined with the time step $\widehat{\Delta t} = \frac{\Delta t}{\Delta x}$ and $\hat{\partial}_t = \Delta x \partial_t$. Due to the 4th-order gradient in the viscosity, we scale its coefficient as $\hat{\nu_B} = \Delta x^{-3} \nu_B$. Using the potential vorticity $h^{-1}(f + \zeta)$, with the relative vorticity $\zeta = \partial_x v - \partial_y u$, and the Bernoulli potential $\frac{1}{2}(u^2 + v^2) + g\eta$, the shallow water equations can be written into a scaled form as

$$\begin{aligned}
\hat{\partial}_t \hat{u} &= \frac{[s\Delta x f] + \hat{\zeta}}{\hat{h}} \frac{\hat{v}\hat{h}}{s} - \hat{\partial}_x \left([\frac{1}{2s}](\hat{u}^2 + \hat{v}^2) + [\frac{sg}{s_\eta}]\hat{\eta} \right) + \hat{\nu_B} \hat{\nabla}^4 \hat{u} - [r\Delta x]\hat{u} + [s\Delta x F_x] \\
\hat{\partial}_t \hat{v} &= -\frac{[s\Delta x f] + \hat{\zeta}}{\hat{h}} \frac{\hat{u}\hat{h}}{s} - \hat{\partial}_y \left([\frac{1}{2s}](\hat{u}^2 + \hat{v}^2) + [\frac{sg}{s_\eta}]\hat{\eta} \right) + \hat{\nu_B} \hat{\nabla}^4 \hat{v} - [r\Delta x]\hat{v} + [s\Delta x F_y]
\end{aligned} \tag{6.2}$$

Square brackets denote pre-computed constants and only the volume fluxes uh, vh have to be unscaled on every time step. As the volume fluxes are quadratic terms, the evaluation of $\hat{u}\hat{h}$ scales as s^2, which therefore has to be partly unscaled with s^{-1}. The continuity equation is rescaled with s_η, i.e. $\hat{\eta} = s_\eta \eta$ as well as $\hat{h} = \hat{\eta} + s_\eta H$, and the tracer advection equation is rescaled with s_q, so that $\hat{q} = s_q q$

$$\begin{aligned}
\hat{\partial}_t \hat{\eta} &= -\hat{\partial}_x (\frac{\hat{u}\hat{h}}{s}) - \hat{\partial}_y (\frac{\hat{v}\hat{h}}{s}) \\
[s\hat{\partial}_t]\hat{q} &= \left(-\hat{u}\hat{\partial}_x \hat{q} - \hat{v}\hat{\partial}_y \hat{q} \right) - [\tau \Delta x](\hat{q} - \hat{q_0})
\end{aligned} \tag{6.3}$$

ShallowWaters.jl solves these scaled shallow water equations with 2nd order finite differencing on a regular, but staggered Arakawa C-grid [Arakawa & Lamb, 1977]. The advection of potential vorticity uses the energy and enstrophy-conserving scheme of Arakawa & Hsu [1990]. The tracer advection equation for q is solved with a semi-Lagrangian advection scheme [Diamantakis, 2013; Smolarkiewicz & Pudykiewicz, 1992]. This scheme calculates a departure point for every arrival grid point one time step ago. The tracer field is then interpolated onto the departure point, which is used as the tracer concentration at the arrival point for the next time step. More details on the implementation of the semi-Lagrangian advection scheme is described in section 5.2.3. The time integration of ShallowWaters.jl is discussed in section 6.2.3.

6.2.2 Choosing a scale with Sherlogs

The scaling of equations has to be implemented carefully when using number formats with a limited dynamic range, such as Float16 (Fig. 6.1). Subnormals for Float16 are in the range of $6 \cdot 10^{-8}$ to $6 \cdot 10^{-5}$ (see section 2.1.3) and are inefficiently supported on some hardware, such that their occurrence causes large performance penalties. This reduces the available range of Float16 even further and a simulation has to fit as best as possible in the remaining 9 orders of magnitude between $6.104 \cdot 10^{-5}$ and 65504. A single overflow, i.e. a result above 65504, will abort the simulation. Understanding the

range of numbers that occur in all operations and ideally in which lines of the code is therefore very important. For most algorithms this is very difficult to achieve unless the numbers are directly measured within the simulation.

```
julia> using ShallowWaters, Sherlogs       # load packages
julia> # run ShallowWaters with Sherlog16 which logs all arithmetic results
julia> run_model(Sherlog16)                # use Sherlog16 as number format

julia> get_logbook()                       # retrieve the bitpattern histogram
65536-element LogBook(1112720887, 1484631, 1378491, 1024411, ... , 0, 0, 0)

julia> # run ShallowWaters with DrWatson16 recording a stack trace when f=true
julia> f(x) = 0 < abs(x) < floatmin(Float16)  # true for subnormals
julia> run_model(DrWatson16{f})            # use DrWatson16 as number format

julia> get_stacktrace(1)                   # retrieve the first stack trace
3-element Vector{Base.StackTraces.StackFrame}:
* at DrWatson16.jl:52 [inlined]            # subnormal occurred in *
caxb!(...) at time_integration.jl:320      # inside this function
time_integration(...) at time_integration.jl:82 # called from here
```

Listing 6.1 | Example usage and output of Sherlogs.jl, a package for Sherlogs and DrWatson, two analysis-number formats that can be combined with type-flexible functions in Julia. Using Sherlog16 as the first argument of `run_model` runs ShallowWaters.jl with Float16 but also logs the bitpattern of every arithmetic result into a *logbook* of length $2^{16} = 65536$ to create a bitpattern histogram. `DrWatson16{f}` uses Float16 but also records a stack trace (a list of calling functions and respective lines of code) every time the function `f(x)` evaluates to `true` with the arithmetic result `x`. Here, a subnormal arises in a multiplication ($*$ in line 14 here) in line 320 of the code in script `time_integration.jl`.

We therefore developed the analysis number format Sherlogs. See section 2.4.2 for a general description of code composability and analysis number formats. Sherlog16, for example, uses Float16 to compute, but after every arithmetic operation the result is also logged into a bitpattern histogram. Running a simulation with Sherlogs will take considerably longer due to the overhead from logging the arithmetic results, which can be obtained in the form of a bitpattern histogram upon completion. The bitpattern histogram will reveal information such as the smallest and largest occurring numbers or how well an algorithm fits into a smaller dynamic range. An example usage of Sherlogs is given in Listing 6.1.

Sherlogs are implemented in the package Sherlogs.jl, which makes use of the type-flexible programming paradigm in Julia [Bezanson *et al.*, 2017]. A function is written in an abstract form, which is then dynamically dispatched to the number format provided and compiled just-in-time. Such a number format can therefore be, for example, Float64 or Float16, but also any user-defined number format such as Sherlogs.

An appropriate scaling s, s_η, s_q has to be chosen for a given set of parameters. The bitpattern histogram of the entirely unscaled shallow water equations simulated with Float32 reveals range issues that would arise with Float16 (Fig. 6.2a). A large share (10%) of the arithmetic results would be below the representable range of Float16. Consequently, running the model without any scaling modifications in Float16 would round many numbers to 0, causing so-called underflows that deteriorate the simulated dynamics [Klöwer *et al.*, 2020]. Most of these underflows occur in the calculation of gradients, which consequently have to be non-dimensionalised as previously suggested [Klöwer *et al.*, 2019]. This also largely removes a resolution-dependence of the bitpattern histograms, such that Float16 simulations are possible across a wide range of resolutions. Dimensionless gradients are a major improvement to fit ShallowWaters.jl into the available range with Float16, yet 3% of the arithmetic results are subnormals (Fig. 6.2b). On A64FX a flag can be set to avoid the performance penalty from subnormals by flushing every occurring subnormal to zero. The smallest representable number is therefore $6.104 \cdot 10^{-5}$.

Using the DrWatson number format from Sherlogs.jl identifies the addition of the tendencies to the prognostic variables u, v, η as prone to produce subnormals (Listing 6.1). We therefore increase the scales to scale up the prognostic variables and consequently their tendencies. Choosing $s = 2^6, s_\eta = 1$ reduces the amount of subnormals to 0.04%, while leaving about a factor two headspace between the largest occurring numbers (about 30000) to avoid overflows beyond 65504 (Fig. 6.2b). The compensated time integration (see section 6.2.3) increases this share to about 0.2%.

The idealised tracer in ShallowWaters.jl takes values in (-1,1), so we scale this variable by $s_q = 2^{15}$ in order to use most of the Float16 range. This is to allow as many bitpatterns as possible for the interpolation in the semi-Lagrangian advection scheme, which uses non-dimensional departure points on a locally relative grid for 16-bit arithmetic, as described in section 5.2.3.

Consequently, the fully scaled shallow water equations are squeezed well into Float16, making near-optimal use of the available bitpatterns, of which only 3% are unused (NaNs excluded). In contrast, a simulation with Float32 does not make use of at least 81% of available bitpatterns (Fig. 6.2), assuming that for a simulation run long enough all bitpatterns within the used range occur eventually. Extrapolating this to Float64 with a representable range of $5 \cdot 10^{-324}$ to $2 \cdot 10^{308}$ the share of unused bitpatterns is at least 97.5%. This computational inefficiency can be overcome with 16-bit number formats and systematic scaling as presented in this chapter. However, scaling leaves the precision issues with low-precision formats unaddressed, for which we present the com-

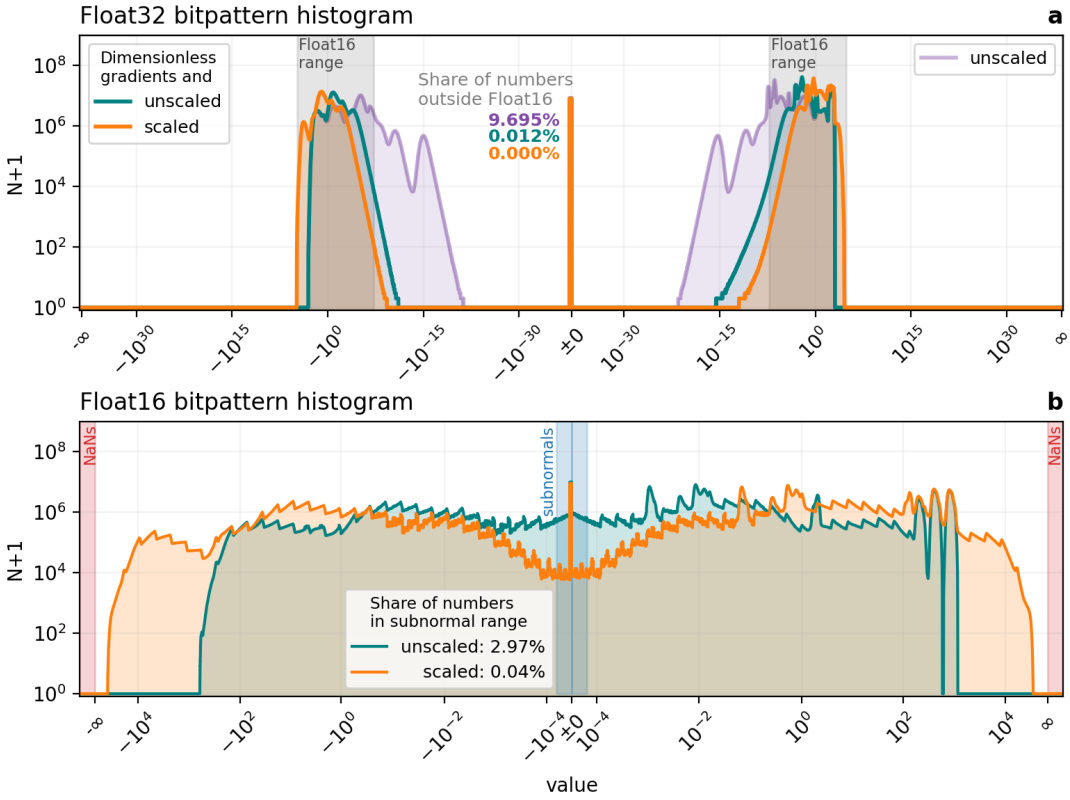

Figure 6.2 | Bitpattern histogram of all arithmetic results in ShallowWaters.jl. a 200-day simulation at $\Delta x = 20\,\mathrm{km}$ based on Float32 arithmetic. The share of numbers outside the Float16 range (grey shading) are colour-coded to the respective histograms. **b** as **a** but based on Float16. Bitpattern histograms are created with Sherlogs.jl. The logarithmic y-axis denotes the number of occurrences N of the respective bitpattern during the simulation. The histograms span all available bitpatterns (`0x0000` to `0xffff` in hexadecimal) in the respective formats evenly but are sorted and relabelled with the corresponding values for readability. The range of bitpatterns that are subnormals or interpreted as Not-A-Number (NaN) are marked. Bitpatterns histograms are without compensated time integration.

pensated time integration in the next section.

6.2.3 A compensated time integration

To minimise the precision loss in the time-integration, we adopt compensated summation as an alternative approach to mixing precision. Compensated summation is a simple, yet powerful technique that prevents the accumulation of rounding errors in the computation of large sums. Since the addition of multiple terms is ubiquitous in scientific computing, compensated summation can be used to improve the accuracy of many algorithms such as numerical linear algebra operations, integration or optimisation. Here we use compensated summation to augment the resilience to rounding errors of our half-precision time-stepping method.

The first version of compensated summation was used by Gill [1951] in a Runge-Kutta integrator scheme in fixed-point arithmetic, and the idea was subsequently extended to floating-point arithmetic by Kahan [1965], Møller [1965] and others [Higham, 1993; Linnainmaa, 1974; Vitasek, 1969]. That we are aware of, our paper is the first work in which compensated summation is used in a fluid circulation model with 16-bit arithmetic. To understand compensated summation, consider the following naïve algorithm for the summation of all the entries of a length-n vector a with elements a_i, $i = 1, ..., n$

```
sum = 0              # variable to store the sum
for ai in a          # loop over all elements of a
    sum += ai        # accumulate each element into sum
end
return sum
```

Listing 6.2 | A naïve summation algorithm.

This algorithm is prone to rounding errors, which accumulate at a rate proportional to n [Higham, 1993]. Furthermore, the algorithm might cause stagnation, a phenomenon for which the partial sum becomes too large, causing each subsequent addition to be neglected due to rounding. Compensated summation offers a much better alternative at the cost of introducing an additional compensation variable `c` (Listing 6.3)

At infinite precision, the compensation `c` will remain 0. At finite precision, however, calculating `c = (temp-sum) - aic` will estimate the rounding error in the addition `sum + aic` and subsequently attempt to compensate for it in the next iteration through `aic = ai - c`. For base-2 floating point arithmetic we have exactly `sum + ai = temp + c`,

```
c = 0                       # compensation, initially 0
sum = 0                     # variable to store the sum
for ai in a                 # loop over all elements of a
    aic = ai - c            # compensate rounding error from previous iteration
    temp = sum + aic        # add next element of a, but store in temp
    c = (temp-sum) - aic    # rounding error from sum+aic
    sum = temp              # copy addition back to sum
end
return sum
```

Listing 6.3 | An algorithm for compensated summation.

i.e. the compensation variable `c` correctly captures the rounding errors in the addition. Compensated summation prevents the rounding errors from accumulating, and the overall summation error will stay a mere multiple of machine precision [Higham, 1993]. Overall, the compensation `c` can be interpreted as a storage variable for rounding errors and effectively prevents rounding errors in the summation from growing beyond machine precision.

Compensated summation is especially useful in settings in which the order of summation cannot be manipulated to prevent rounding error growth. Time integration schemes, for which the state variables are updated sequentially, are especially amenable to augmentation by compensated summation. Over a time period T the number of terms to be added scales as $T\Delta t^{-1}$, proportional to one for each time step. The naïve algorithm would cause rounding errors to grow like $\mathcal{O}(T\Delta t^{-1}\varepsilon)$, causing errors to counter-intuitively grow as the time-step is refined. With compensated summation the rounding errors will stay $\mathcal{O}(\varepsilon)$.

ShallowWaters.jl uses the 4th-order Runge-Kutta scheme [Butcher, 2008] to integrate the non-dissipative terms in time: The momentum advection $\mathbf{u} \cdot \nabla\mathbf{u}$; the Coriolis force $f\hat{\mathbf{z}} \times \mathbf{u}$; the pressure gradient $-g\nabla\eta$; the wind forcing $\mathbf{F}$; and the conservation of volume $-\nabla \cdot (\mathbf{u}h)$ are summarised as the right-hand side function `rhs`. The time integration is now augmented with compensated summation. The rounding error c_u that occurs in the addition of the total tendency du to the previous time step u_n is calculated and stored. On the next time step, this rounding error is subtracted from the total tendency du in an attempt to compensate for the rounding error from the previous time step. This is illustrated here for the zonal velocity u in isolation, although in practice the time integration has to update the prognostic variables u, v, η simultaneously. A

compensated time integration for u with RK4 can be written as

$$\begin{aligned} k_1 &= \mathrm{rhs}(u + \tfrac{\Delta t}{2} k_1) \\ k_2 &= \mathrm{rhs}(u + \tfrac{\Delta t}{2} k_2) \\ k_3 &= \mathrm{rhs}(u + \tfrac{\Delta t}{2} k_3) \\ k_4 &= \mathrm{rhs}(u + \Delta t k_3) \\ du &= \tfrac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4) \\ u^*_{n+1} &= u_n + du \\ c_u &= (u^*_{n+1} - u_n) - du \end{aligned} \tag{6.4}$$

with $c_u = 0$ as initial condition. The addition $u_n + du$ usually suffers from rounding errors as described above. The loss of precision in du is calculated in c_u (which is only 0 in exact arithmetic). The compensation is analogously implemented with c_v, c_η for the other prognostic variables.

The dissipative terms, i.e. biharmonic diffusion of momentum $\nu_B \nabla^4 \mathbf{u}$ and bottom friction $-r\mathbf{u}$, are integrated with a single forward step after the Runge-Kutta integration in ShallowWaters.jl and summarized as $\mathrm{rhs}_{\mathrm{diss}}$. To compensate for rounding errors for both the dissipative and non-dissipative terms simultaneously, c_u from Eq. 6.4 is subtracted from the total dissipative tendency du_{diss}. In that sense, the rounding error from Eq. 6.4 is attempted to be compensated subsequently in Eq. 6.5, and vice versa.

$$\begin{aligned} du_{\mathrm{diss}} &= \Delta t\, \mathrm{rhs}_{\mathrm{diss}}(u^*_{n+1}) - c_u \\ u_{n+1} &= u^*_{n+1} + du_{\mathrm{diss}} \\ c_u &= (u_{n+1} - u^*_{n+1}) - du_{\mathrm{diss}} \end{aligned} \tag{6.5}$$

Only the addition of the total tendency is compensated here to minimise the amount of additional calculations, which increases when also compensating the 3 sub steps in RK4.

The compensated time integration is an alternative to mixed-precision approaches. While those aim to keep the precision high in the precision-critical calculations, the compensated time integration introduces a new variable to compensate for the rounding errors in one precision-critical calculation. With compensated time integration all variables can be kept in 16 bit, and no conversions between number formats are necessary.

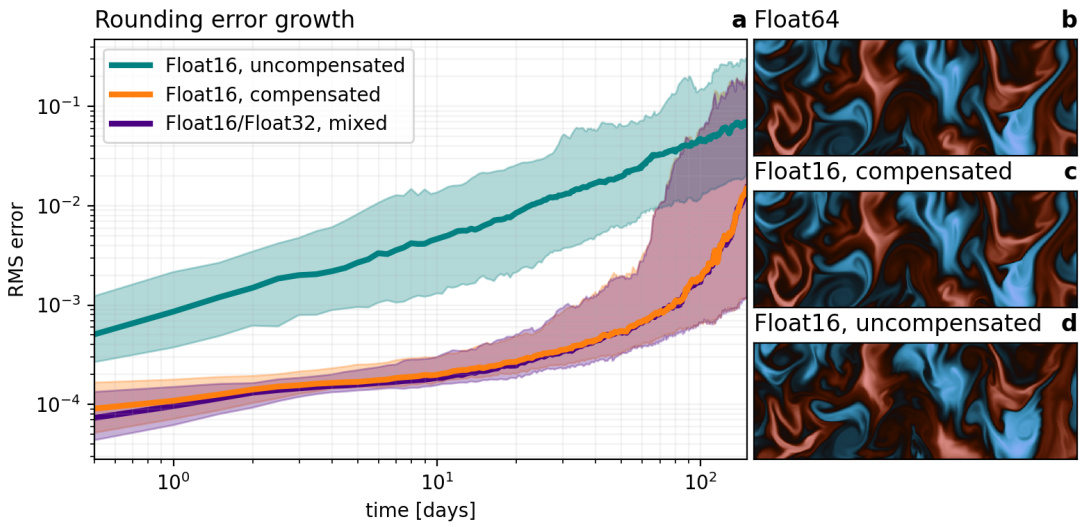

Figure 6.3 | Rounding error growth with Float16 in ShallowWaters.jl using compensated time integration or mixed precision. **a** Errors are root-mean square (RMS) errors of zonal velocity u relative to Float64. Solid lines denote the median and shadings the interdecile confidence interval. **b,c** Snapshots of tracer q from a zoom into Fig. 6.4 after 100 days of simulation and **d** as **c** but without compensated time integration.

6.3 Results

6.3.1 Minimizing precision issues in 16 bit

The accumulated rounding error from mixing precision and compensated time integration is now assessed. ShallowWaters.jl is started from identical, in Float16 perfectly representable, initial conditions in a domain of 8000 km by 4000 km. The model is spun-up to reach a turbulent flow domain-wide, while the tracer starts from an idealised checkerboard pattern to better highlight the turbulence everywhere in the domain. The grid consists of 3000x1500 points at about 2.7 km grid-spacing. With Float16 and without compensated time integration, the accumulated rounding error for zonal velocity u compared to Float64 exponentially increases 100-fold in the first 150 days (Fig. 6.3a). With mixed precision, using Float16 for the tendencies and Float32 for the prognostic variables (see section 5.3.5), this rounding error growth is strongly reduced. Errors after a few time steps without mixed precision are reached after about 100 days (about 25,000 time steps) of integration. After that the error growth accelerates and chaos removes the information of the initial conditions.

Using a compensated time integration, the rounding error from Float16 is strongly

reduced and matches well with the error growth of mixed precision. From the perspective of rounding errors the two methods are therefore equivalently suited to reduce rounding errors with 16-bit arithmetic. The rounding error growth of the other prognostic variables is similar. The positive effect of compensated time integration is well illustrated in snapshots of tracer mixing where even after 100 days of simulation only a very slight deviation from the Float64 reference is observable (Fig. 6.3b, c and d).

Even after 100 days of simulation a large simulation (3000x1500 grid points) with Float16 shows minimal errors in the tracer mixing compared to Float64 (Fig. 6.4). Only at regions near the boundaries, where the mixing is enhanced, a difference is visible. The remaining rounding error is small and will be masked in a more realistic setup by model or discretization errors. Reducing the precision in calculations raises concerns about the numerical conservation of physically conserved quantities like mass. The compensated time integration conserves the mass in the shallow water equations with Float16 (<0.002% change within 500 days compared to Float64), similar to mixed precision (Fig. 6.5). Without compensated time integration for Float16 the conservation is with 0.05% change over 500 days less accurate. Similar results were obtained for the conservation of the tracer. We will now assess the speedups with Float16 compared to Float64 on A64FX.

6.3.2 Approaching 4x speedup on A64FX

The A64FX is a microprocessor developed by Fujitsu based on the ARM-architecture. It powers not just the fastest supercomputer in the world as of June 2021 (measured by TOP500, Dongarra & Luszczek [2011]), Fugaku, but also a number of smaller systems around the world, including Isambard 2 which we use here. The A64FX has a number of features intended to accelerate machine learning applications. Notably, it allows not just Float32 and Float64 arithmetic but also Float16. Official benchmarks of the A64FX demonstrate a cost increase which is linear with the number of bits. In that sense, Float32 can be twice as fast in applications than Float64, while Float16 can be four times as fast, when optimized well. In practice, speedups in complex applications are due to a mix of factors: In compute-bound applications, the wall-clock time is largely given by the clock rate of the processor and the vectorization of arithmetic operations (such that small sets of them are performed in parallel on a single processor core). Using Float16 instead of Float64 allows to put four times as many numbers through the vectorization, theoretically allowing for 4x speedups. The performance of memory-bound applications, on the other hand, is largely determined by the data transfer rate between

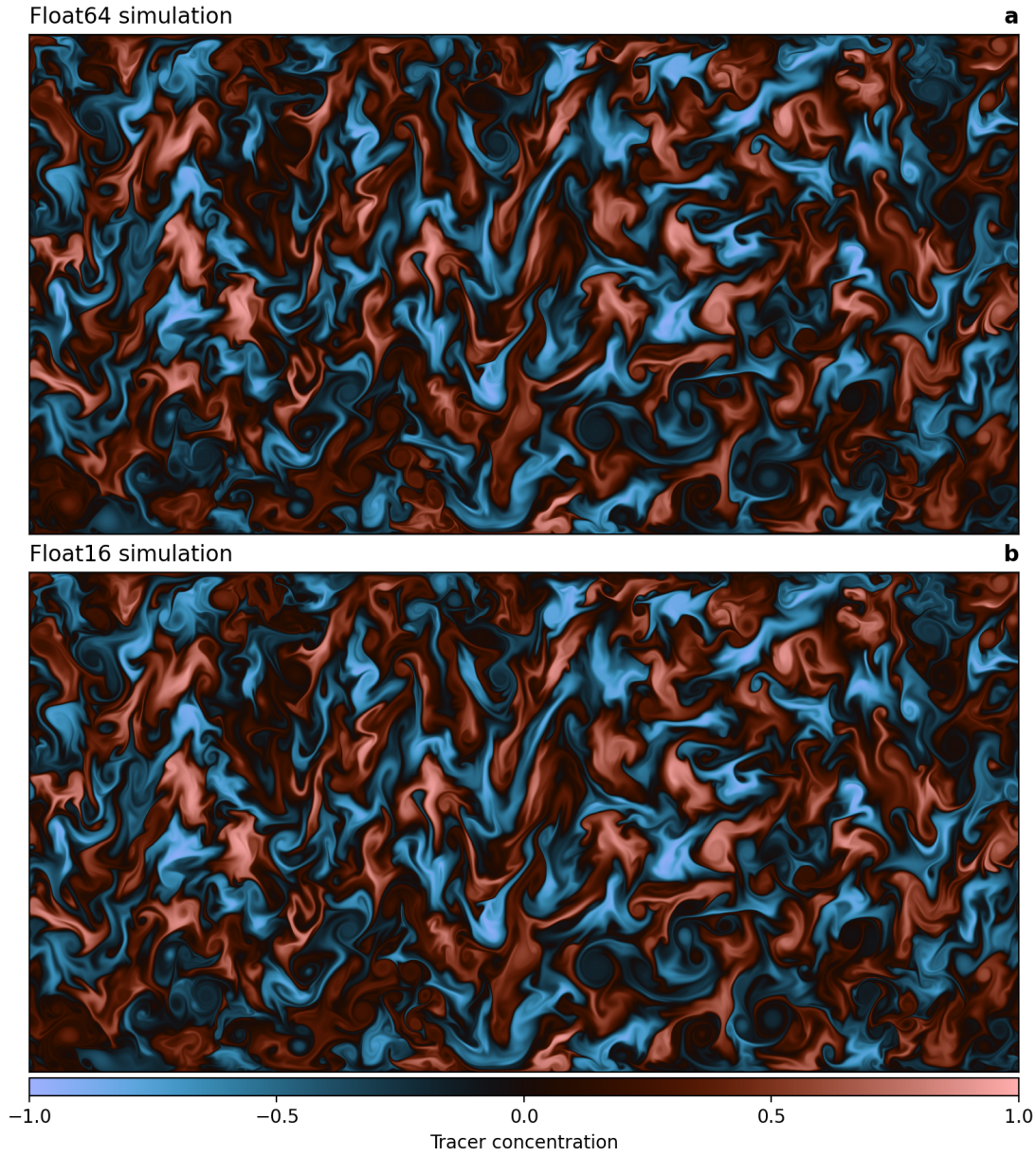

Figure 6.4 | Turbulent tracer mixing as simulated by ShallowWaters.jl. a Simulation based on Float64 arithmetic and **b** Float16 with compensated time integration. Snapshot is taken after 100 days of simulation (about 25,000 time steps) with 3000x1500 grid points starting from identical initial conditions. Remaining errors between **a** and **b** from low-precision Float16 are tolerable and will be masked by other sources of error in a less idealised model setup.

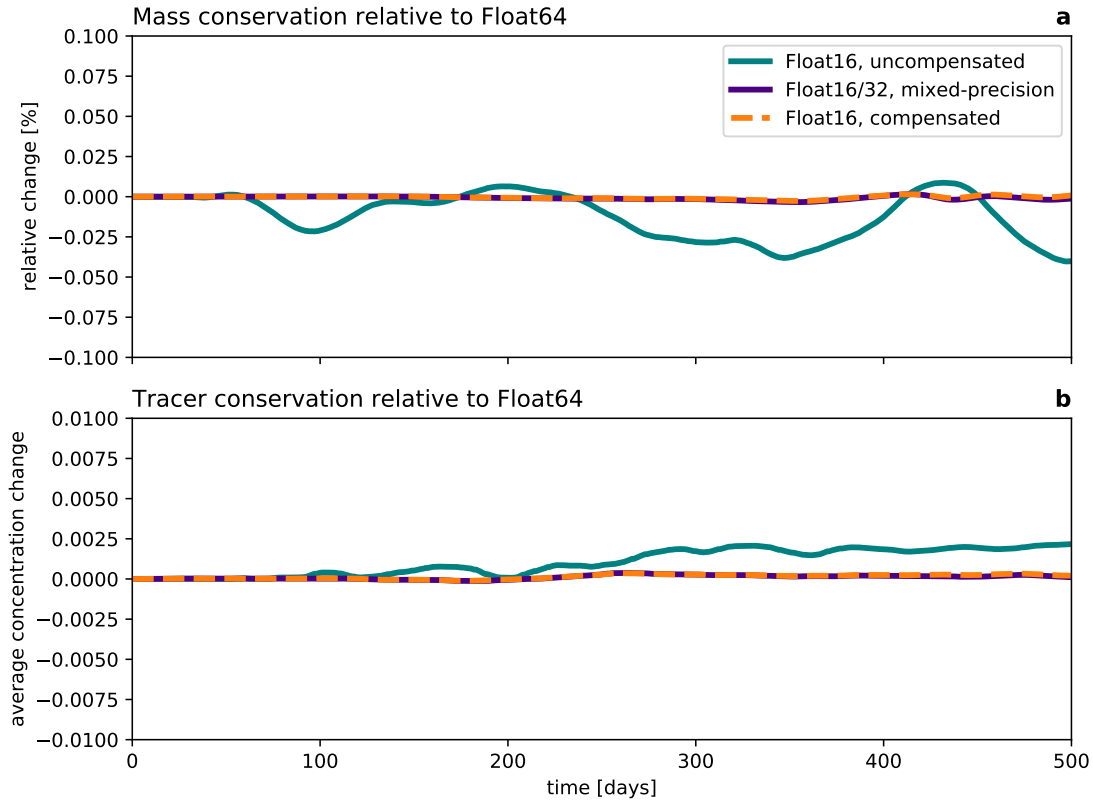

Figure 6.5 | Mass and tracer conservation with Float16 arithmetic. **a** Mass conservation relative to Float64. **b** Tracer conservation relative to Float64 in units of tracer concentration with initial conditions in (-1,1), see Fig. 6.4. Both mass and tracer are well conserved with Float16 arithmetic. Best conservations are obtained with compensated time integration or mixed precision.

the processor and its various levels of caches that increase in size but decrease in bandwidth. Using Float16 instead of Float64 allows to load four times as many numbers from memory, which theoretically translates to 4x speedup as well.

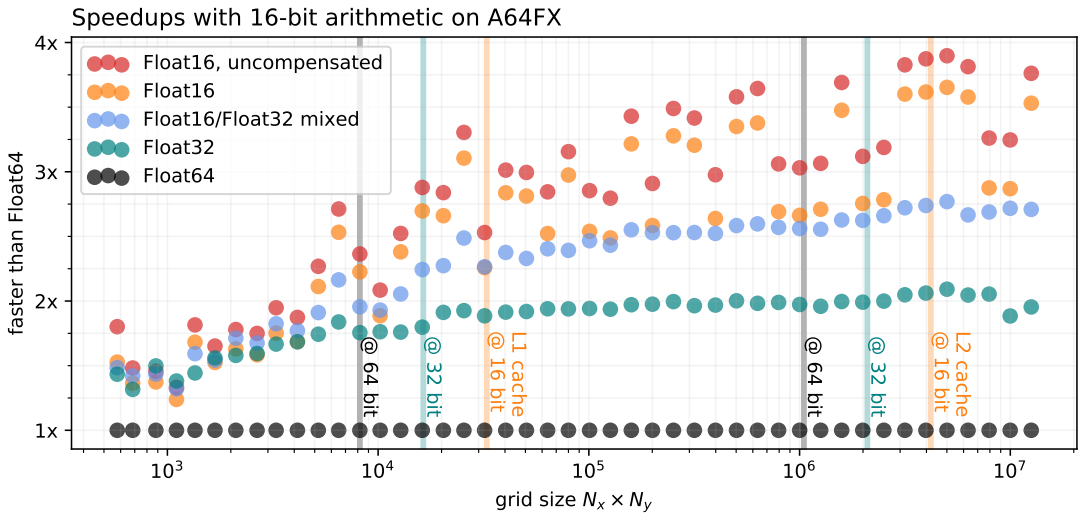

Figure 6.6 | Performance increase from Float16 when running ShallowWaters.jl at varying grid sizes on A64FX. The grid size is the total number of grid points $N_x \times N_y$. All timings are single-threaded median wall clock times relative to Float64, excluding compilation, model initialisation and memory pre-allocation. The corresponding size of the L1 and L2 cache (64KiB, 8 MiB) of A64FX is given as vertical lines for arrays of 16, 32 and 64-bit floats.

ShallowWaters.jl is a memory-bound application for which the biggest benefit from Float16 will be the reduction of the size of the arrays by a factor of four when compared to Float64. The arrays can therefore be read faster from memory with a potential speedup of 4x. We benchmark ShallowWaters.jl at varying grid sizes, excluding compilation, model initialisation and memory pre-allocation. With grid sizes of 10^5 (about 450x225 grid points) and larger, there is a clear improvement from using Float32 instead of Float64 which approaches 2x speedups (Fig. 6.6). Using Float16 these speedups reach up to 3.8x for grid sizes beyond $3 \cdot 10^6$ (about 2450x1225 grid points). The dependency of the speedup on the grid size is complicated: While larger grids usually experience more acceleration on A64FX in Float16, there are ranges where the speedup drops to 3-3.25x. This is likely due to peculiarities in the memory and cache hierarchy of the A64FX, such that the performance benefit of Float16 cannot always be fully realised. A detailed assessment of these peculiarities is beyond the scope of this chapter, but it is nevertheless reassuring that, even in the worst case, Float16 is still at least three times faster than Float64 for these large grids.

As discussed in previous sections, using a compensated time integration can be used to minimise the rounding errors, which comes with a small additional computational cost: Using the compensated time integration the speedups drop to about 3.6x for large grids. Nevertheless, a compensated time integration yields higher performances than mixing the precision of Float16 and Float32, which approaches only 2.75x here. Consequently, a compensated time integration for Float16 is, although as precise, faster than mixed precision.

6.4 Discussion

Low-precision calculations for weather and climate simulations are a potential that is not yet fully exploited. While the first weather forecast models are moving towards Float32, 16-bit arithmetics will likely find increasing support on future supercomputers. We present, to our knowledge, the first fluid simulation that runs entirely in hardware-accelerated 16 bit with minimal rounding errors but at almost 4x the speed. The simulations were performed on A64FX, the microprocessor that is used in Fugaku, the fastest supercomputer as of November 2020 (TOP500, Dongarra & Luszczek [2011]).

The complex partial differential equations underlying weather and climate simulations are difficult to fit into the limited range of Float16, but here we have presented a method to do this more systematically. We present Sherlogs.jl to analyse number formats. Sherlogs.jl allows to assess any changes to the scaling of the equations to minimise underflows while making the most of the available representable numbers. In our case, subnormal floating-point numbers had to be avoided and scaling of the equations dropped the amount of subnormals occurring from 10 to below 0.2%.

Using 16-bit floats will likely cause precision issues in fluid simulations. While mixed precision has been used to minimise rounding errors in precision-critical calculations, we have presented here an approach that compensates for rounding errors to allow for simulations entirely within 16-bit arithmetic. The compensated time integration minimises rounding errors from this precision-critical part of a simulation at a slightly higher cost. Benchmarking in comparison to mixed precision shows that the compensated time integration is faster in ShallowWaters.jl while being as precise as mixed precision.

Alternatives to floats have been discussed for weather and climate simulations previously [Klöwer *et al.*, 2019, 2020]. Although posit numbers [Gustafson & Yonemoto, 2017] are more precise in these applications, the improvement from floats to posits is smaller than using mixed precision and therefore also smaller than the compensated

time integration. In that sense, algorithms that are low precision-resilient are far more important than the actual choice of the number format, especially given that only floats are widely hardware-supported.

The work in this chapter shows that a naive translation of the mathematical equations into code will likely fail with 16-bit arithmetic. However, this does not mean that 16-bit arithmetic is unsuited for the numerical solution of complex partial differential equations such as the shallow water equations. But it means that both precision and range issues have to be addressed in the design of the algorithms used. A compensated time integration is a low precision-resilient algorithm, and scaling is essential to fit the very limited range of Float16.

While 16-bit hardware is largely designed for machine learning, its potential to increase computational efficiency extends to weather and climate applications too. 16-bit calculations are indeed a competitive way to also accelerate Earth-system simulations on available hardware.

7 Concluding discussion

Numerical models of weather and climate are high-performance applications for the world's largest supercomputers, producing very large amounts of data. As 32 and 64-bit floating-point numbers are the only widely available number formats for scientific computing, this thesis investigates the potential for lower-precision computations and data compression for weather and climate modelling. An information-preserving compression for climate data is derived, presented and discussed that distinguishes between the real and false information in data without a prior knowledge of the data's uncertainty. Discarding the false information bits was shown to be efficient to achieve compression factors between 10 and 50x for many different variables. The analysis of the bitwise real information content shows that different variables require different levels of precision. Hence, using a default level of precision for all of them, will be inefficient: Either large amounts of false information is retained when using unnecessarily high precision, or real information discarded when the precision is not high enough for some variables. Given that default precisions are generally high, a large share of a climate data archive stores essentially random bits without any real information. Removing such bits before storage will reduce the load on the data archive, data access servers and bandwidth constraints of users. If information-preserving compression is combined with low-precision computing, the freed resources from the supercomputer and the data archive can also be used to archive variables at higher resolution.

But understanding the impact of low-precision arithmetic on chaotic systems is difficult. Using simple chaotic dynamical systems as well as the more complex shallow water equations we investigate several ways how low precision affects the simulated dynamics. With low precision the number of discrete states in a system is exponentially reduced, causing bitwise periodicity with shorter periods compared to high precision. While low precision truncates the periodic orbit spectrum and therefore the simulated attractors, we find no evidence for a significant degradation of the orbit spectrum in a system with more than just a few variables. A system with many variables is estimated to have bitwise periodic orbits that are far beyond reach for any computer, even at low precision. Given that low-precision computing is applied to accelerate simulations such that freed resources can be reinvested into improving resolution through more variables, there is little practical concern. In our applications, the reduction in periodic orbit lengths due to low precision is compensated with higher resolution.

Furthermore, stochastic rounding can be applied to avoid periodicity and to reduce

rounding errors. This rounding mode was found beneficial in simulations of simple chaotic systems to reduce average rounding errors without the need for higher precision. Although stochastic rounding is currently unsupported by available hardware, it is nevertheless possible to include a software-implementation into a few operations without sacrificing performance. Mixed-precision with stochastically rounded conversions between low and high precision might be a competitive approach as it circumvents the need for stochastic arithmetics on hardware and can be used for low-precision communication too.

However, in the shallow water equations we found evidence that a simulation's resilience to low precision can highly depend on details of the numerical discretisation schemes implemented: With a conventional time integration Float16 simulations include greatly amplified gravity waves, impact the geostrophic balance and rounding errors grow similarly fast as discretisation errors. However, with a compensated time integration to reduce the rounding errors in one addition per time step and variable, Float16 simulations are a competitive alternative to simulations with Float64. On the other hand, alternative number formats such as posits can provide more precision at the same number of bits compared to floats. But given the lack of widely available hardware support, their future in high-performance computing is questionable. In general, we find low-precision resilient algorithms far more important than the choice of the number format to reduce the impact of rounding errors from low precision. While floats are not the best number format for weather and climate models, they may be good enough even at 16 bits when combined with carefully designed algorithms.

To test the performance of the, to our knowledge, first 16-bit fluid circulation model ShallowWaters.jl, we execute it on Fujitsu's A64FX, the first available central processing unit that supports Float16 arithmetic. We approach 4x speedups with Float16 compared to Float64 at minimal rounding errors due to the compensated time integration and avoid issues from the small representable range of Float16 by systematic rescaling. Both techniques show the importance of algorithms that are designed for 16-bit computing and that a naive transition to lower precision will likely fail. However, this also provides an opportunity as it will be easier to port algorithms that are more resilient to low precision to future hardware. Many weather and climate forecasting centres have a new supercomputer regularly and the time to next upgrade can be less than the time to rewrite and test core algorithms. In that sense, a weather or climate model with core algorithms that are developed for 16-bit arithmetics will be much easier to transition to new computing hardware. On the other hand, a model that relies on 64-bit arithmetic will likely miss out on future advances in computing hardware and an avoidable

dependency is created.

While it remains an open question whether an entire weather or climate model can be executed with 16-bit arithmetics, this thesis finds no reason that it is not possible. The bitwise real information content of no variable was found to exceed 16 bits; scaling techniques are presented to squeeze complex algorithms into a limited dynamic range; and precision issues can be addressed by modifications of the numerical schemes, as presented for the compensated time integration. Converting complex algorithms for 16-bit arithmetic can be a challenging endeavour but modern programming languages can accelerate efforts. In this thesis we exploited the programming paradigms of code composability and type flexibility to develop a shallow water model that can run with many different number formats and on different computing architectures, without any code changes. With a combination of these techniques and an increased effort there is little reason to assume that 16-bit weather and climate models are not possible. While the development can be challenging, such a new generation of computationally efficient models will provide a competitive performance on modern computing architecture towards more reliable weather and climate forecasts in the future.

Appendix

A.1 Open-source software developments

Several open-source software packages that were developed as part of this thesis are briefly described in the following. For more information see the respective repositories.

A.1.1 ShallowWaters.jl

- Author: M Klöwer
- github.com/milankl/ShallowWaters.jl
- Version: 0.5.0
- Language: Julia

A shallow water model with a focus on type-flexibility and 16-bit number formats. ShallowWaters allows for Float64/32/16, Posit32/16/8, BFloat16, LogFixPoint16, Sonum16, Float32/16 and BFloat16 with stochastic rounding and in general every number format with arithmetics and conversions implemented. ShallowWaters also allows for mixed-precision and reduced precision communication.

ShallowWaters uses an energy and enstrophy conserving advection scheme and a Smagorinsky-like biharmonic diffusion operator. Tracer advection is implemented with a semi-Lagrangian advection scheme. Strong stability-preserving Runge-Kutta schemes of various orders and stages are used with a semi-implicit treatment of the continuity equation. Boundary conditions are either periodic (only in x direction) or non-periodic super-slip, free-slip, partial-slip, or no-slip. Output via NetCDF.

A.1.2 SoftPosit.jl

- Authors: M Klöwer, M Giordano, C Leong
- github.com/milankl/SoftPosit.jl
- Version: 0.3.0
- Languages: Julia, C

SoftPosit.jl is a software emulator for posit arithmetic. The package exports the Posit8, Posit16, Posit32 number types among other non-standard types, as well as arithmetic operations, conversions and additional functionality. The package is a wrapper for the SoftPosit C-library written by C Leong.

A.1.3 StochasticRounding.jl

- Author: M Klöwer
- github.com/milankl/StochasticRounding.jl
- Version: 0.6.0
- Language: Julia

StochasticRounding.jl is a software emulator for stochastic rounding in the Float32, Float16 and BFloat16 number formats. Both 16bit implementations rely on conversion to and from Float32 and stochastic rounding is only applied for arithmetic operations in the conversion back to 16bit. Float32 with stochastic rounding uses Float64 internally. Xoroshiro128+ is used as a high-performance random number generator.

A.1.4 Sherlogs.jl

- Authors: M Klöwer, Ben Arthur
- github.com/milankl/Sherlogs.jl
- Version: 0.2.0
- Language: Julia

Sherlogs.jl provides a number format Sherlog64 that behaves like Float64, but inspects code by logging all arithmetic results into a 16-bit bitpattern histogram during calculation. Sherlogs can be used to identify the largest or smallest number occurring in functions, and where algorithmic bottlenecks are that limit the ability for your functions to run in low precision. A 32-bit version is provided as Sherlog32, which behaves like Float32. A 16-bit version is provided as Sherlog16{T}, which uses T for computations as well as for logging.

Sherlogs.jl also exports the analysis number format DrWatson. DrWatson64f behaves like Float64 but evaluates the function f after every arithmetic operation. If the result of f is true a stacktrace is recorded, which can be used to identify where in the code certain arithmetic results occur.

A.1.5 BitInformation.jl

- Author: M Klöwer
- github.com/milankl/BitInformation.jl
- Version: 0.2.0
- Language: Julia

BitInformation.jl is a package for the analysis of bitwise information in Julia Arrays. Based on counting the occurrences of bits in floats (or generally any bittype) across various dimensions of an array, this package provides functions to calculate quantities like the bitwise real information content, the mutual information, the redundancy or preserved information between arrays.

BitInformation.jl also implements various rounding modes (round,bitshave,bitset,halfshave etc.) efficiently with bitwise operations. Furthermore, transformations like XOR-delta, bittranspose, or signed_exponent are implemented.

A.1.6 LogFixPoint16s.jl

- Author: M Klöwer
- github.com/milankl/LogFixPoint16s.jl
- Version: 0.3.0
- Language: Julia

LogFixPoint16s.jl is a software emulator for 16-bit logarithmic fixed-point numbers with an adjustable number of integer and fraction bits.The package exports the LogFixPoint16 number type, as well as arithmetic operations, conversions and additional functionality. The software emulator is based on integer addition for multiplication, division and square root, and based on lookup tables for addition and subtraction.

A.1.7 Float8s.jl

- Authors: M Klöwer, J Sarnoff
- github.com/milankl/Float8s.jl
- Version: 0.1.0
- Language: Julia

Float8s.jl is a software emulator for the 8-bit floating-point format Float8 with 3 exponent and 4 significant bits. The package provides the `Float8` number type, as well as arithmetic operations, conversions and additional functionality. The software emulator is based on conversion to and from Float32, which is used for arithmetic operations.

A.1.8 Lorenz96.jl

- Author: M Klöwer
- github.com/milankl/Lorenz96.j

- Version: 0.3.0
- Language: Julia

Lorenz96.jl is a type-flexible one-level Lorenz 1996 model, which supports many different number formats, as long as conversions to and from Float64 and arithmetics are defined. Lorenz96.jl supports mixed-precision: Different number types can be defined for prognostic variables and calculations on the right-hand side, with automatic conversion on every time step. The equations are scaled such that the dynamic range of numbers can be changed. The scaled equations are written division-free.

Lorenz96.jl also exports functions that allow for the efficient search using distributed computing to find bitwise periodic orbits. A MapReduce approach is used with Julia's Distributed package.

A.1.9 Lorenz63.jl

- Author: M Klöwer
- github.com/milankl/Lorenz63.jl
- Version: 0.2.0
- Language: Julia

Lorenz63.jl is a type-flexible Lorenz 1963 model, which supports many different number formats, as long as conversions to and from Float64 and arithmetics are defined. The Lorenz equations are scaled such that the dynamic range of numbers can be changed. The scaled equations are written division-free.

A.1.10 ZfpCompression.jl

- Authors: M Klöwer, P Lindstrom
- github.com/milankl/ZfpCompression.jl
- Version: 0.2.0
- Languages: Julia, C

Julia bindings for the data compression library zfp v0.5.5, written by P Lindstrom. Zfp is a library to compress numerical arrays in 2-4 dimensions. It uses lossy but error-bounded local approximations for data with spatio-temporal correlation in 2 to 4 dimensions. Higher compression factors will be achieved the higher the correlation in as many dimensions as possible.

A.1.11 LinLogQuantization.jl

- Author: M Klöwer
- github.com/milankl/LinLogQuantization.jl
- Version: 0.2.0
- Language: Julia

Linear and logarithmic quantisation for Julia arrays into 8, 16, 24 or 32-bit. Quantisation is a lossy compression method that divides the range of values in an array in equidistant quantums and encodes those as unsigned integers from 0 to 2^{n-1} where n is the number of bits available. The quantums are either equidistant in linear space or in logarithmic space, which has a denser encoding for values close to the minimum in trade-off with a less dense encoding close to the maximum.

Acknowledgements

I am very grateful for all the contributions of co-authors and discussions with all of my colleagues over the last years. I have learned a lot from you. While a thesis is traditionally the work of a single author, you are always included whenever I used the term *we*. I especially value the support and the very fruitful discussions with my supervisors Tim Palmer and Peter Düben, and appreciate the freedom to develop my own ideas. For a self-propelled student it is easier to fly without headwind.

Particularly, I would like to thank Adam Paxton for countless efforts to correct my sloppy approach to mathematics; Mosè Giordano for teaching me many tricks about Julia; Matteo Croci for many discussions how to make stochastic rounding faster and faster; Debbie Hopkins for having improved my general style of writing when we wrote our comment together; Valentin Churavy and Jeffrey Sarnoff for Julia-related bug fixes; Simon MacIntosh-Smith for access to the A64FX processors on Isambard 2; Sam Hatfield and Josh Dorrington for RaspberryPi and Google challenges in my favourite office; Simon Proud and Lucy Harrington for providing further data sets to test our compression method; Miha Razinger and Juanjo Dominguez for being great mentors during two ESoWCs; Peter Coveney for writing countless emails to discuss periodic orbits; Chiara Maiocchi for explaining unstable periodic orbits to me; Joe Bates for having introduced me to logarithmic fixed-point numbers; Myles Allen and David Lee for supervising me on decarbonising aviation; Sören Thomsen for many initiatives to decarbonise science together; Mike Morrison for great ideas on how to make science user-friendly; and Najar's for having fed me countless falafels over the last years whenever my work rhythm was very offbeat.

Special thanks to the editors and many anonymous reviewers that have read and improved our manuscripts over the last years without getting appropriate credit for these efforts — particularly also the reviewers that took the time to formulate appreciation among all the constructive criticism. Thanks

for your contribution to make science more welcoming to students and early career scientists that do not always submit perfect manuscripts. Many thanks also to Geoff, Lukas, Chloë and Žiga for proofreading my thesis.

I would like to thank the whole Julia community for an uncountable effort to develop a programming language that turned out to be the secret behind much of my progress over the last years. I have benefited a lot from learning Julia and I hope I was able to return the favour through the development and contribution to several Julia packages. Many thanks also to everyone who developed the matplotlib plotting library, which was used for every figure in this thesis, and taught me a lot about data visualisation.

Over the last four years I have received funding from various sources which I am very grateful for. The European Research Council and therefore indirectly the European Union for my stipend; the Natural Environmental Research Council for my tuition fees, the Copernicus Programme and the ECMWF Summer of Weather Code 2020 and 2021 and therefore indirectly the European Commission for funding my research on climate data compression; Jesus College for several travel grants that supported my attendance at conferences; and a big greedy publisher for the first prize at a student conference — thanks for the money, but I still will not publish with you.

Then I am very grateful to all the more than 30 people I had the opportunity to live with in the Vic over the last four years. You've always been like a family to me; I've tried to invest a lot of time and energy into this amazing community and you have always more than returned the favour with laughter and calories, with support and activity, with challenge and knowledge, with lessons and help, with Gwen and Winston, and definitely turned me into a better human.

Finally, I would like to thank my mum for always having worked hard to support me towards the opportunities to do what I want do.

Funding

I gratefully acknowledge funding from

- The European Research Council ERC under grant number 741112 *An Information Theoretic Approach to Improving the Reliability of Weather and Climate Simulations*
- The UK Natural Environmental Research Council NERC under grant number NE/L002612/1.
- The Copernicus Programme (EU Commission) through the ECMWF Summer of Weather Code 2020 and 2021
- The Graduate Research Allowance provided by Jesus College Oxford in 2018 and 2019

References

J Ackmann, PD Düben, TN Palmer & PK Smolarkiewicz (2021). Mixed-precision for Linear Solvers in Global Geophysical Flows. *arXiv:2103.16120 [physics]*. 42

F Alted (2010). Why Modern CPUs Are Starving and What Can Be Done about It. *Computing in Science Engineering*, **12**, 68–71, 10.1109/MCSE.2010.51. 64

A Arakawa & YJG Hsu (1990). Energy Conserving and Potential-Enstrophy Dissipating Schemes for the Shallow Water Equations. *Monthly Weather Review*, **118**, 1960–1969, 10.1175/1520-0493(1990)118<1960:ECAPED>2.0.CO;2. 105, 128

A Arakawa & VR Lamb (1977). Computational Design of the Basic Dynamical Processes of the UCLA General Circulation Model. In J. Chang, ed., *Methods in Computational Physics: Advances in Research and Applications*, vol. 17 of *General Circulation Models of the Atmosphere*, 173–265, Elsevier, 10.1016/B978-0-12-460817-7.50009-4. 104, 128

BK Arbic & RB Scott (2008). On Quadratic Bottom Drag, Geostrophic Turbulence, and Oceanic Mesoscale Eddies. *Journal of Physical Oceanography*, **38**, 84–103, 10.1175/2007JPO3653.1. 103, 104

AH Baker, DM Hammerling, SA Mickelson, H Xu, MB Stolpe, P Naveau, B Sanderson, I Ebert-Uphoff, S Samarasinghe, F De Simone, F Carbone, CN Gencarelli, JM Dennis, JE Kay & P Lindstrom (2016). Evaluating lossy data compression on climate simulation data within a large ensemble. *Geoscientific Model Development*, **9**, 4381–4403, 10.5194/gmd-9-4381-2016. 43, 72

AH Baker, DM Hammerling & TL Turton (2019). Evaluating image quality measures to assess the impact of lossy data compression applied to climate simulation data. *Computer Graphics Forum*, **38**, 517–528, 10.1111/cgf.13707. 3, 43, 52, 53, 72

R Ballester-Ripoll, P Lindstrom & R Pajarola (2020). TTHRESH: Tensor Compression for Multidimensional Visual Data. *IEEE Transactions on Visualization and Computer Graphics*, **26**, 2891–2903, 10.1109/TVCG.2019.2904063. 42, 68

P Bauer, A Thorpe & G Brunet (2015). The quiet revolution of numerical weather prediction. *Nature*, **525**, 47–55, 10.1038/nature14956. 41, 77

P Bauer, T Quintino, N Wedi, A Bonanni, M Chrust, W Deconinck, M Diamantakis, P Düben, S English, J Flemming, P Gillies, I Hadade, J Hawkes, M Hawkins, O Iffrig, C

Kühnlein, M Lange, P Lean, O Marsden, A Müller, S Saarinen, D Sarmany, M Sleigh, S Smart, P Smolarkiewicz, D Thiemert, G Tumolo, C Weihrauch & C Zanna (2020). The ECMWF Scalability Programme: Progress and Plans. 10.21957/gdit22ulm. 42, 102

P Bauer, PD Dueben, T Hoefler, T Quintino, TC Schulthess & NP Wedi (2021a). The digital revolution of Earth-system science. *Nature Computational Science*, **1**, 104–113, 10.1038/s43588-021-00023-0. 2, 42, 124

P Bauer, B Stevens & W Hazeleger (2021b). A digital twin of Earth for the green transition. *Nature Climate Change*, **11**, 80–83, 10.1038/s41558-021-00986-y. 42, 124

J Bezanson, A Edelman, S Karpinski & VB Shah (2017). Julia: A Fresh Approach to Numerical Computing. *SIAM Review*, **59**, 65–98, 10.1137/141000671. 16, 31, 129

D Blackman & S Vigna (2019). Scrambled Linear Pseudorandom Number Generators. *arXiv:1805.01407 [cs]*. 90

BM Boghosian, PV Coveney & H Wang (2019). A New Pathology in the Simulation of Chaotic Dynamical Systems on Digital Computers. *Advanced Theory and Simulations*, **2**, 1900125, 10.1002/adts.201900125. 3, 77, 85, 86, 87, 90

N Burgess, J Milanovic, N Stephens, K Monachopoulos & D Mansell (2019). Bfloat16 Processing for Neural Networks. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, 88–91, 10.1109/ARITH.2019.00022. 101, 124

JC Butcher (2008). Runge–Kutta Methods. In *Numerical Methods for Ordinary Differential Equations*, chap. 3, 137–316, John Wiley & Sons, Ltd, 10.1002/9780470753767.ch3. 95, 133

JC Butcher (2016). *Numerical Methods for Ordinary Differential Equations*. Wiley, Chichester, West Sussex, United Kingdom, 3rd edn. 77, 104

MJ Capiński, D Turaev & P Zgliczyński (2018). Computer assisted proof of the existence of the Lorenz attractor in the Shimizu–Morioka system. *Nonlinearity*, **31**, 5410–5440, 10.1088/1361-6544/aae032. 81

TF Chan & TP Mathew (1994). Domain decomposition algorithms. *Acta Numerica*, **3**, 61–143, 10.1017/S0962492900002427. 109

M Chantry, T Thornes, T Palmer & P Düben (2019). Scale-Selective Precision for Weather and Climate Forecasting. *Monthly Weather Review*, **147**, 645–655, 10.1175/MWR-D-18-0308.1. 102, 125

R Chaurasiya, J Gustafson, R Shrestha, J Neudorfer, S Nambiar, K Niyogi, F Merchant & R Leupers (2018). Parameterized Posit Arithmetic Hardware Generator. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, 334–341, IEEE, Orlando, FL, USA, 10.1109/ICCD.2018.00057. 124

J Chen, Z Al-Ars & HP Hofstee (2018). A matrix-multiply unit for posits in reconfigurable logic leveraging (open)CAPI. In *Proceedings of the Conference for Next Generation Arithmetic on - CoNGA '18*, 1–5, ACM Press, Singapore, Singapore, 10.1145/3190339.3190340. 14, 102

H Choo, K Muhammad & K Roy (2003). Two's complement computation sharing multiplier and its applications to high performance DFE. *IEEE Transactions on Signal Processing*, **51**, 458–469. 7, 15

D Cohen (1981). On Holy Wars and a Plea for Peace. *Computer*, **14**, 48–54, 10.1109/C-M.1981.220208. 5

Y Collet (2020). Facebook/zstd. Facebook. 54, 64

NJ Cornish (2001). Chaos and gravitational waves. *Physical Review D*, **64**, 084011, 10.1103/PhysRevD.64.084011. 77

R Courant, K Friedrichs & H Lewy (1967). On the Partial Difference Equations of Mathematical Physics. *IBM Journal of Research and Development*, **11**, 215–234, 10.1147/rd.112.0215. 125

PV Coveney & S Wan (2016). On the calculation of equilibrium thermodynamic properties from molecular dynamics. *Physical Chemistry Chemical Physics*, **18**, 30236–30240, 10.1039/C6CP02349E. 77

M Croci & MB Giles (2020). Effects of round-to-nearest and stochastic rounding in the numerical solution of the heat equation in low precision. *arXiv:2010.16225 [cs, math]*. 24, 79, 124, 125

RM Cummings, WH Mason, SA Morton & DR McDaniel (2015). *Applied Computational Aerodynamics: A Modern Engineering Approach*. Cambridge University Press. 77

P Cvitanović (1991). Periodic orbits as the skeleton of classical and quantum chaos. *Physica D: Nonlinear Phenomena*, **51**, 138–151, 10.1016/0167-2789(91)90227-Z. 78

P Cvitanović (2013). Recurrent flows: The clockwork behind turbulence. *Journal of Fluid Mechanics*, **726**, 1–4, 10.1017/jfm.2013.198. 77

N Damouche, M Martel, P Panchekha, C Qiu, A Sanchez-Stern & Z Tatlock (2017). Toward a Standard Benchmark Format and Suite for Floating-Point Analysis. In S. Bogomolov, M. Martel & P. Prabhakar, eds., *Numerical Software Verification*, Lecture Notes in Computer Science, 63–77, Springer International Publishing, Cham, 10.1007/978-3-319-54292-8_6. 125

A Dawson & PD Düben (2017). Rpe v5: An emulator for reduced floating-point precision in large numerical simulations. *Geoscientific Model Development*, **10**, 2221–2230, 10.5194/gmd-10-2221-2017. 102

A Dawson, PD Düben, DA MacLeod & TN Palmer (2018). Reliable low precision simulations in land surface models. *Climate Dynamics*, **51**, 2657–2666, 10.1007/s00382-017-4034-x. 42, 102, 125

X Delaunay, A Courtois & F Gouillon (2019). Evaluation of lossless and lossy algorithms for the compression of scientific datasets in netCDF-4 or HDF5 files. *Geoscientific Model Development*, **12**, 4099–4113, 10.5194/gmd-12-4099-2019. 55, 64

T DelSole (2004). Predictability and Information Theory. Part I: Measures of Predictability. *Journal of the Atmospheric Sciences*, **61**, 2425–2440, 10.1175/1520-0469(2004)061<2425:PAITPI>2.0.CO;2. 60

C Denis, P De Oliveira Castro & E Petit (2016). Verificarlo: Checking Floating Point Accuracy through Monte Carlo Arithmetic. In *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*, 55–62, 10.1109/ARITH.2016.31. 125

P Deutsch (1996). DEFLATE Compressed Data Format Specification version 1.3. 64

S Di & F Cappello (2016). Fast Error-Bounded Lossy HPC Data Compression with SZ. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 730–739, 10.1109/IPDPS.2016.11. 42, 68

M Diamantakis (2013). The semi-Lagrangian technique in atmospheric modelling: Current status and future challenges. 18. 128

DA D'Ippolito, JR Myra & SJ Zweben (2011). Convective transport by intermittent blob-filaments: Comparison of theory and experiment. *Physics of Plasmas*, **18**, 060501, 10.1063/1.3594609. 77

J Dongarra & P Luszczek (2011). TOP500. In D. Padua, ed., *Encyclopedia of Parallel Computing*, 2055–2057, Springer US, Boston, MA, 10.1007/978-0-387-09766-4_157. 2, 101, 124, 136, 140

PD Düben, H McNamara & T Palmer (2014). The use of imprecise processing to improve accuracy in weather & climate prediction. *Journal of Computational Physics*, **271**, 2–18, 10.1016/j.jcp.2013.10.042. 3, 102

JP Eckmann & D Ruelle (2004). Ergodic theory of chaos and strange attractors. In B.R. Hunt, T.Y. Li, J.A. Kennedy & H.E. Nusse, eds., *The Theory of Chaotic Attractors*, 273–312, Springer, New York, NY, 10.1007/978-0-387-21830-4_17. 78

Q Fan, DJ Lilja & SS Sapatnekar (2019). Using DCT-based Approximate Communication to Improve MPI Performance in Parallel Clusters. In *2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC)*, 1–10, IEEE, London, United Kingdom, 10.1109/IPCCC47392.2019.8958720. 42, 109

M Fasi & M Mikaitis (2021). Algorithms for Stochastically Rounded Elementary Arithmetic Operations in IEEE 754 Floating-Point Arithmetic. *IEEE Transactions on Emerging Topics in Computing*, 1–1, 10.1109/TETC.2021.3069165. 79

R Ferrari & C Wunsch (2010). The distribution of eddy kinetic and potential energies in the global ocean. *Tellus A*, 10.3402/tellusa.v62i2.15680. 104

F Fevotte & B Lathuilière (2019). Debugging and Optimization of HPC Programs with the Verrou Tool. In *2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness)*, 1–10, 10.1109/Correctness49594.2019.00006. 3, 125

S Fox, S Rasoulinezhad, J Faraone, D Boland & P Leong (2020). A Block Minifloat Representation for Training Deep Neural Networks. In *International Conference on Learning Representations*. 124

O Fuhrer, T Chadha, T Hoefler, G Kwasniewski, X Lapillonne, D Leutwyler, D Lüthi, C Osuna, C Schär, TC Schulthess & H Vogt (2018). Near-global climate simulation at 1 km resolution: Establishing a performance baseline on 4888 GPUs with COSMO 5.0. *Geoscientific Model Development*, **11**, 1665–1681, 10.5194/gmd-11-1665-2018. 77, 102, 109

M Ghil & P Malanotte-Rizzoli (1991). Data Assimilation in Meteorology and Oceanography. In R. Dmowska & B. Saltzman, eds., *Advances in Geophysics*, vol. 33, 141–266, Elsevier, 10.1016/S0065-2687(08)60442-2. 77, 101

AE Gill (1982). *Atmosphere-Ocean Dynamics*. No. 30 in International Geophysics Series, Acad. Press, San Diego. 103, 126, 127

S Gill (1951). A process for the step-by-step integration of differential equations in an automatic digital computing machine. *Mathematical Proceedings of the Cambridge Philosophical Society*, **47**, 96–108, 10.1017/S0305004100026414. 125, 132

M Govett, J Rosinski, J Middlecoff, T Henderson, J Lee, A MacDonald, N Wang, P Madden, J Schramm & A Duarte (2017). Parallelization and Performance of the NIM Weather Model on CPU, GPU, and MIC Processors. *Bulletin of the American Meteorological Society*, **98**, 2201–2213, 10.1175/BAMS-D-15-00278.1. 124

SM Griffies & RW Hallberg (2000). Biharmonic Friction with a Smagorinsky-Like Viscosity for Use in Large-Scale Eddy-Permitting Ocean Models. *MONTHLY WEATHER REVIEW*, **128**, 12. 103, 104, 107

S Gupta, A Agrawal, K Gopalakrishnan & P Narayanan (2015). Deep Learning with Limited Numerical Precision. *arXiv:1502.02551 [cs, stat]*. 24, 101

JL Gustafson (2015). *The End of Error: Unum Computing*. Chapman and Hall/CRC, 1st edn. 3, 13

JL Gustafson & I Yonemoto (2017). Beating Floating Point at its Own Game: Posit Arithmetic. *Supercomputing Frontiers and Innovations*, **4**, 16. 13, 14, 22, 28, 29, 78, 102, 124, 140

A Haidar, S Tomov, J Dongarra & NJ Higham (2018). Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, 1–11, IEEE Press, Dallas, Texas, 10.1109/SC.2018.00050. 109

DM Hammerling, AH Baker, A Pinard & P Lindstrom (2019). A Collaborative Effort to Improve Lossy Compression Methods for Climate Data. In *2019 IEEE/ACM 5th International Workshop on Data Analysis and Reduction for Big Scientific Data (DRBSD-5)*, 16–22, 10.1109/DRBSD-549595.2019.00008. 69

S Hatfield, A Subramanian, T Palmer & P Düben (2017). Improving Weather Forecast Skill through Reduced-Precision Data Assimilation. *Monthly Weather Review*, **146**, 49–62, 10.1175/MWR-D-17-0132.1. 93, 102

S Hatfield, P Düben, M Chantry, K Kondo, T Miyoshi & T Palmer (2018). Choosing the Optimal Numerical Precision for Data Assimilation in the Presence of Model Error. *Journal of Advances in Modeling Earth Systems*, **10**, 2177–2191, 10.1029/2018MS001341. 102

S Hatfield, M Chantry, P Düben & T Palmer (2019). Accelerating High-Resolution Weather Models with Deep-Learning Hardware. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '19, 1–11, Association for Computing Machinery, Zurich, Switzerland, 10.1145/3324989.3325711. 42, 109, 125

S Hatfield, A McRae, T Palmer & P Düben (2020). Single-Precision in the Tangent-Linear and Adjoint Models of Incremental 4D-Var. *Monthly Weather Review*, **148**, 1541–1552, 10.1175/MWR-D-19-0291.1. 102

K Haustein, MR Allen, PM Forster, FEL Otto, DM Mitchell, HD Matthews & DJ Frame (2017). A real-time Global Warming Index. *Scientific Reports*, **7**, 15417, 10.1038/s41598-017-14828-5. 1

H Hersbach (2000). Decomposition of the Continuous Ranked Probability Score for Ensemble Prediction Systems. *Weather and Forecasting*, **15**, 559–570, 10.1175/1520-0434(2000)015<0559:DOTCRP>2.0.CO;2. 71

NJ Higham (1993). The Accuracy of Floating Point Summation. *SIAM Journal on Scientific Computing*, **14**, 783–799, 10.1137/0914050. 132, 133

NJ Higham (2002). *Accuracy and Stability of Numerical Algorithms*. SIAM. 3, 77, 101

NJ Higham, S Pranesh & M Zounon (2019). Squeezing a Matrix into Half Precision, with an Application to Solving Linear Systems. *SIAM Journal on Scientific Computing*, **41**, A2536–A2551, 10.1137/18M1229511. 109

F Hofbauer (1978). β-Shifts have unique maximal measure. *Monatshefte für Mathematik*, **85**, 189–198, 10.1007/BF01534862. 85

M Höhfeld & SE Fahlman (1992). Probabilistic rounding in neural network learning with limited precision. *Neurocomputing*, **4**, 291–299, 10.1016/0925-2312(92)90014-G. 24

M Hopkins, M Mikaitis, DR Lester & S Furber (2020). Stochastic rounding and reduced-precision fixed-point arithmetic for solving neural ordinary differential equations. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, **378**, 20190052, 10.1098/rsta.2019.0052. 78, 79, 124

N Hübbe, A Wegener, JM Kunkel, Y Ling & T Ludwig (2013). Evaluating Lossy Compression on Climate Data. In J.M. Kunkel, T. Ludwig & H.W. Meuer, eds., *Supercomputing*, Lecture Notes in Computer Science, 343–356, Springer, Berlin, Heidelberg, 10.1007/978-3-642-38750-0_26. 43

DA Huffman (1952). A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, **40**, 1098–1101, 10.1109/JRPROC.1952.273898. 58, 64

IEEE (1985). IEEE Standard for Binary Floating-Point Arithmetic. *ANSI/IEEE Std 754-1985*, 1–20, 10.1109/IEEESTD.1985.82928. 1, 8, 9, 14, 20, 42, 61, 77, 79, 101, 124

IEEE (2008). IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, 1–70, 10.1109/IEEESTD.2008.4610935. 1, 8, 101, 124

A Inness, M Ades, A Agustí-Panareda, J Barré, A Benedictow, AM Blechschmidt, JJ Dominguez, R Engelen, H Eskes, J Flemming, V Huijnen, L Jones, Z Kipling, S Massart, M Parrington, VH Peuch, M Razinger, S Remy, M Schulz & M Suttie (2019). The CAMS reanalysis of atmospheric composition. *Atmospheric Chemistry and Physics*, **19**, 3515–3556, 10.5194/acp-19-3515-2019. 58

ITER Physics Expert Group on Confinement and Transport, ITER Physics Expert Group on Confinement Modelling and Database & IPB Editors (1999). Chapter 2: Plasma confinement and transport. *Nuclear Fusion*, **39**, 2175–2249, 10.1088/0029-5515/39/12/302. 77

D James (1990). Multiplexed buses: The endian wars continue. *IEEE Micro*, **10**, 9–21, 10.1109/40.56322. 5

MF Jansen, AJ Adcroft, R Hallberg & IM Held (2015a). Parameterization of eddy fluxes based on a mesoscale energy budget. *Ocean Modelling*, **92**, 28–41, 10.1016/j.ocemod.2015.05.007. 127

MF Jansen, IM Held, A Adcroft & R Hallberg (2015b). Energy budget-based backscatter in an eddy permitting primitive equation model. *Ocean Modelling*, **94**, 15–26, 10.1016/j.ocemod.2015.07.015. 127

S Jeffress, P Düben & T Palmer (2017). Bitwise efficiency in chaotic models. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, **473**, 20170144, 10.1098/rspa.2017.0144. 2, 42, 43, 45, 58, 61, 102, 105

F Jézéquel & JM Chesneaux (2008). CADNA: A library for estimating round-off error propagation. *Computer Physics Communications*, **178**, 933–955, 10.1016/j.cpc.2008.02.003. 3, 125

J Johnson (2018). Rethinking floating point for deep learning. *arXiv:1811.01721 [cs]*. 124

J Johnson (2020). Efficient, arbitrarily high precision hardware logarithmic arithmetic for linear algebra. *arXiv:2004.09313 [cs, math]*. 78, 79, 102, 124

N Jouppi, C Young, N Patil & D Patterson (2018a). Motivation for and Evaluation of the First Tensor Processing Unit. *IEEE Micro*, **38**, 10–19, 10.1109/MM.2018.032271057. 124

NP Jouppi, C Young, N Patil, D Patterson, G Agrawal, R Bajwa, S Bates, S Bhatia, N Boden, A Borchers, R Boyle, Pl Cantin, C Chao, C Clark, J Coriell, M Daley, M Dau, J Dean, B Gelb, TV Ghaemmaghami, R Gottipati, W Gulland, R Hagmann, CR Ho, D Hogberg, J Hu, R Hundt, D Hurt, J Ibarz, A Jaffey, A Jaworski, A Kaplan, H Khaitan, D Killebrew, A Koch, N Kumar, S Lacy, J Laudon, J Law, D Le, C Leary, Z Liu, K Lucke, A Lundin, G MacKean, A Maggiore, M Mahony, K Miller, R Nagarajan, R Narayanaswami, R Ni, K Nix, T Norrie, M Omernick, N Penukonda, A Phelps, J Ross, M Ross, A Salek, E Samadiani, C Severn, G Sizikov, M Snelham, J Souter, D Steinberg, A Swing, M Tan, G Thorson, B Tian, H Toma, E Tuttle, V Vasudevan, R Walter, W Wang, E Wilcox & DH Yoon (2017). In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, 1–12, Association for Computing Machinery, Toronto, ON, Canada, 10.1145/3079856.3080246. 101, 124

NP Jouppi, C Young, N Patil & D Patterson (2018b). A domain-specific architecture for deep neural networks. *Communications of the ACM*, **61**, 50–59, 10.1145/3154484. 2, 77, 101, 124

T Jung, MJ Miller & TN Palmer (2010). Diagnosing the Origin of Extended-Range Forecast Errors. *Monthly Weather Review*, **138**, 2434–2446, 10.1175/2010MWR3255.1. 101

W Kahan (1965). Pracniques: Further remarks on reducing truncation errors. *Communications of the ACM*, **8**, 40, 10.1145/363707.363723. 125, 132

D Kalamkar, D Mudigere, N Mellempudi, D Das, K Banerjee, S Avancha, DT Vooturi, N Jammalamadaka, J Huang, H Yuen, J Yang, J Park, A Heinecke, E Georganas, S Srinivasan, A Kundu, M Smelyanskiy, B Kaul & P Dubey (2019). A Study of BFLOAT16 for Deep Learning Training. *arXiv:1905.12322 [cs, stat]*. 101, 124

R Kleeman (2011). Information Theory and Dynamical System Predictability. *Entropy*, **13**, 612–649, 10.3390/e13030612. 42, 43, 58

M Klöwer & M Giordano (2019). SoftPosit.jl - A posit arithmetic emulator. Zenodo, 10.5281/zenodo.3590291. 16

M Klöwer, PD Düben & TN Palmer (2019). Posits as an alternative to floats for weather and climate models. In *Proceedings of the Conference for Next Generation Arithmetic 2019 on - CoNGA'19*, 1–8, ACM Press, Singapore, Singapore, 10.1145/3316279.3316281. 14, 28, 102, 103, 105, 106, 124, 130, 140

M Klöwer, PD Düben & TN Palmer (2020). Number Formats, Error Mitigation, and Scope for 16-Bit Arithmetics in Weather and Climate Modeling Analyzed With a Shallow Water Model. *Journal of Advances in Modeling Earth Systems*, **12**, e2020MS002246, 10.1029/2020MS002246. 3, 42, 124, 125, 130, 140

R Kouznetsov (2020). A note on precision-preserving compression of scientific data. *Geoscientific Model Development Discussions*, 1–9, 10.5194/gmd-2020-239. 17, 19, 43, 47

A Kraskov, H Stögbauer & P Grassberger (2004). Estimating mutual information. *Physical Review E*, **69**, 066138, 10.1103/PhysRevE.69.066138. 44, 60

S Kudo, K Nitadori, T Ina & T Imamura (2020). Implementation and Numerical Techniques for One EFlop/s HPL-AI Benchmark on Fugaku. In *2020 IEEE/ACM 11th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*, 69–76, 10.1109/ScalA51936.2020.00014. 101

M Kuhn, J Kunkel & T Ludwig (2016). Data Compression for Climate Data. *Supercomputing Frontiers and Innovations*, **3**, 75–94, 10.14529/jsfi160105. 43

T Kurth, S Treichler, J Romero, M Mudigonda, N Luehr, E Phillips, A Mahesh, M Matheson, J Deslippe, M Fatica, Prabhat & M Houston (2018). Exascale deep learning for climate analytics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, 1–12, IEEE Press, Dallas, Texas. 101

F Kwasniok (2014). Enhanced regime predictability in atmospheric low-order models due to stochastic forcing. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, **372**, 20130286, 10.1098/rsta.2013.0286. 105

HF Langroudi, Z Carmichael & D Kudithipudi (2019). Deep Learning Training on the Edge with Low-Precision Posits. *arXiv:1907.13216 [cs, stat]*. 102, 124

D Lasagna (2020). Sensitivity of long periodic orbits of chaotic systems. *Physical Review E*, **102**, 052220, 10.1103/PhysRevE.102.052220. 78

P Leboeuf (2004). Periodic orbit spectrum in terms of Ruelle-Pollicott resonances. *Physical Review. E, Statistical, Nonlinear, and Soft Matter Physics*, **69**, 026204, 10.1103/PhysRevE.69.026204. 78

SH Leong (2020). SoftPosit. Zenodo, 10.5281/zenodo.3709035. 16

P Lindstrom (2014). Fixed-Rate Compressed Floating-Point Arrays. *IEEE Transactions on Visualization and Computer Graphics*, **20**, 2674–2683, 10.1109/TVCG.2014.2346458. 42, 43, 69

P Lindstrom & M Isenburg (2006). Fast and Efficient Compression of Floating-Point Data. *IEEE Transactions on Visualization and Computer Graphics*, **12**, 1245–1250, 10.1109/TVCG.2006.143. 43, 68

S Linnainmaa (1974). Analysis of some known methods of improving the accuracy of floating-point sums. *BIT Numerical Mathematics*, **14**, 167–202, 10.1007/BF01932946. 132

EN Lorenz (1963). Deterministic Nonperiodic Flow. *Journal of the Atmospheric Sciences*, **20**, 130–141, 10.1175/1520-0469(1963)020<0130:DNF>2.0.CO;2. 3, 81, 105

EN Lorenz & KA Emanuel (1998). Optimal Sites for Supplementary Weather Observations: Simulation with a Small Model. *Journal of the Atmospheric Sciences*, **55**, 399–414, 10.1175/1520-0469(1998)055<0399:OSFSWO>2.0.CO;2. 93

DJC MacKay (2003). *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, Cambridge, UK, 1st edn. 35, 36, 42, 58, 60

G Madec, R Bourdallé-Badie, PA Bouttier, C Bricaud, D Bruciaferri, D Calvert, J Chanut, E Clementi, A Coward, D Delrosso, C Ethé, S Flavoni, T Graham, J Harle, D Iovino, D Lea, C Lévy, T Lovato, N Martin, S Masson, S Mocavero, J Paul, C Rousset, D Storkey, A Storto & M Vancoppenolle (2017). NEMO ocean engine. Report, 10.5281/zenodo.3248739. 102

CC Maiocchi & V Lucarini (2021). Decomposing the Dynamics of the Lorenz 1963 model using Unstable Periodic Orbits: Averages, Transitions, and Quasi-Invariant Sets. *arXiv:2108.04181 [cond-mat, physics:nlin, physics:physics]*. 78

S Malardel, N Wedi, N Deconinck, M Diamantakis, C Kuehnlein, G Mozdzynski, M Hamrud & P Smolarkiewicz (2016). A new grid for the IFS. *ECMWF Newsletter*. 43

S Markidis, SWD Chien, E Laure, IB Peng & JS Vetter (2018). NVIDIA Tensor Core Programmability, Performance Precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 522–531, 10.1109/IPDPSW.2018.00091. 77, 101

JE Matheson & RL Winkler (1976). Scoring Rules for Continuous Probability Distributions. *Management Science*, **22**, 1087–1096, 10.1287/mnsc.22.10.1087. 71

M Matsumoto & T Nishimura (1998). Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, **8**, 3–30, 10.1145/272991.272995. 90

M Mikaitis (2020). Stochastic Rounding: Algorithms and Hardware Accelerator. *arXiv:2001.01501 [cs, math]*. 24, 124

O Møller (1965). Quasi double-precision in floating point addition. *BIT Numerical Mathematics*, **5**, 37–50, 10.1007/BF01975722. 125, 132

F Molteni, R Buizza, TN Palmer & T Petroliagis (1996). The ECMWF Ensemble Prediction System: Methodology and validation. *Quarterly Journal of the Royal Meteorological Society*, **122**, 73–119, 10.1002/qj.49712252905. 42

J Moran (2003). *An Introduction to Theoretical and Computational Aerodynamics*. Courier Corporation. 77

A Müller, W Deconinck, C Kühnlein, G Mengaldo, M Lange, N Wedi, P Bauer, PK Smolarkiewicz, M Diamantakis, SJ Lock, M Hamrud, S Saarinen, G Mozdzynski, D Thiemert, M Glinton, P Bénard, F Voitus, C Colavolpe, P Marguinaud, Y Zheng, J Van Bever, D Degrauwe, G Smet, P Termonia, KP Nielsen, BH Sass, JW Poulsen, P Berg, C Osuna, O Fuhrer, V Clement, M Baldauf, M Gillard, J Szmelter, E O'Brien, A McKinstry, O Robinson, P Shukla, M Lysaght, M Kulczewski, M Ciznicki, W Piątek, S Ciesielski, M Błażewicz, K Kurowski, M Procyk, P Spychala, B Bosak, ZP Piotrowski, A Wyszogrodzki, E Raffin, C Mazauric, D Guibert, L Douriez, X Vigouroux, A Gray, P Messmer, AJ Macfaden & N New (2019). The ESCAPE project: Energy-efficient Scalable Algorithms for Weather Prediction at Exascale. *Geoscientific Model Development*, **12**, 4425–4441, 10.5194/gmd-12-4425-2019. 2

LK Muller & G Indiveri (2015). Rounding Methods for Neural Networks with Low Resolution Synaptic Weights. *arXiv:1504.05767 [cs]*. 24

M Nakano, H Yashiro, C Kodama & H Tomita (2018). Single Precision in the Dynamical Core of a Nonhydrostatic Global Atmospheric Model: Evaluation Using a Baroclinic Wave Test Case. *Monthly Weather Review*, **146**, 409–416, 10.1175/MWR-D-17-0257.1. 77, 124

T Odajima, Y Kodama, M Tsuji, M Matsuda, Y Maruyama & M Sato (2020). Preliminary Performance Evaluation of the Fujitsu A64FX Using HPC Applications. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, 523–530, 10.1109/CLUSTER49012.2020.00075. 2, 77, 124

T Palmer (2014). Climate forecasting: Build high-resolution global climate models. *Nature News*, **515**, 338, 10.1038/515338a. 42

T Palmer (2015). Modelling: Build imprecise supercomputers. *Nature News*, **526**, 32, 10.1038/526032a. 2, 3, 42, 101, 124

T Palmer (2019a). The ECMWF ensemble prediction system: Looking back (more than) 25 years and projecting forward 25 years. *Quarterly Journal of the Royal Meteorological Society*, **145**, 12–24, 10.1002/qj.3383. 101

TN Palmer (2012). Towards the probabilistic Earth-system simulator: A vision for the future of climate and weather prediction. *Quarterly Journal of the Royal Meteorological Society*, **138**, 841–861, 10.1002/qj.1923. 101

TN Palmer (2019b). Stochastic weather and climate models. *Nature Reviews Physics*, **1**, 463–471, 10.1038/s42254-019-0062-2. 42, 77

TN Palmer, A Döring & G Seregin (2014). The real butterfly effect. *Nonlinearity*, **27**, R123–R141, 10.1088/0951-7715/27/9/R123. 3

W Parry (1960). On the β-expansions of real numbers. *Acta Mathematica Academiae Scientiarum Hungarica*, **11**, 401–416, 10.1007/BF02020954. 85

EA Paxton, M Chantry, M Klöwer, L Saffin & T Palmer (2021). Climate Modelling in Low-Precision: Effects of both Deterministic & Stochastic Rounding. *arXiv*. 79, 85, 124

T Pelkonen, S Franklin, J Teller, P Cavallaro, Q Huang, J Meza & K Veeraraghavan (2015). Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment*, **8**, 1816–1827, 10.14778/2824032.2824078. 55

A Pinard, DM Hammerling & AH Baker (2020). Assessing Differences in Large Spatio-temporal Climate Datasets with a New Python package. In *2020 IEEE International Conference on Big Data (Big Data)*, 2699–2707, IEEE, Atlanta, GA, USA, 10.1109/BigData50022.2020.9378100. 69

A Poppick, J Nardi, N Feldman, AH Baker, A Pinard & DM Hammerling (2020). A statistical analysis of lossily compressed climate model data. *Computers & Geosciences*, **145**, 104599, 10.1016/j.cageo.2020.104599. 69

PK Pothapakula, C Primo & B Ahrens (2019). Quantification of Information Exchange in Idealized and Climate System Applications. *Entropy*, **21**, 1094, 10.3390/e21111094. 44, 60

P Ricci, FD Halpern, S Jolliet, J Loizu, A Mosetto, A Fasoli, I Furno & C Theiler (2012). Simulation of plasma turbulence in scrape-off layer conditions: The GBS code, simulation results and code validation. *Plasma Physics and Controlled Fusion*, **54**, 124047, 10.1088/0741-3335/54/12/124047. 77

S Rüdisühli, A Walser & O Fuhrer (2013). COSMO in single precision. *Cosmo Newsletter*, 5–1. 102, 124

D Ruelle & F Takens (1971). On the nature of turbulence. *Les rencontres physiciens-mathématiciens de Strasbourg-RCP25*, **12**, 1–44. 78

FP Russell, PD Düben, X Niu, W Luk & T Palmer (2017). Exploiting the chaotic behaviour of atmospheric models with reconfigurable architectures. *Computer Physics Communications*, **221**, 160–173, 10.1016/j.cpc.2017.08.011. 7, 102

L Saffin, S Hatfield, P Düben & T Palmer (2020). Reduced-precision parametrization: Lessons from an intermediate-complexity atmospheric model. *Quarterly Journal of the Royal Meteorological Society*, **146**, 1590–1607, 10.1002/qj.3754. 102

R Salmon (2004). Poisson-Bracket Approach to the Construction of Energy- and Potential-Enstrophy- Conserving Algorithms for the Shallow-Water Equations. *JOURNAL OF THE ATMOSPHERIC SCIENCES*, **61**, 21. 105

R Salmon (2007). A General Method for Conserving Energy and Potential Enstrophy in Shallow-Water Models. *Journal of the Atmospheric Sciences*, **64**, 515–531, 10.1175/JAS3837.1. 105

M Sato, Y Ishikawa, H Tomita, Y Kodama, T Odajima, M Tsuji, H Yashiro, M Aoki, N Shida, I Miyoshi, K Hirai, A Furuya, A Asato, K Morita & T Shimizu (2020). Co-Design for A64FX Manycore Processor and "Fugaku". In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–15, 10.1109/SC41405.2020.00051. 2, 78, 101, 124

C Schär, O Fuhrer, A Arteaga, N Ban, C Charpilloz, SD Girolamo, L Hentgen, T Hoefler, X Lapillonne, D Leutwyler, K Osterried, D Panosetti, S Rüdisühli, L Schlemmer, TC Schulthess, M Sprenger, S Ubbiali & H Wernli (2020). Kilometer-Scale Climate Models: Prospects and Challenges. *Bulletin of the American Meteorological Society*, **101**, E567–E587, 10.1175/BAMS-D-18-0167.1. 42

T Schreiber (2000). Measuring Information Transfer. *Physical Review Letters*, **85**, 461–464, 10.1103/PhysRevLett.85.461. 44, 60

CE Shannon (1948). A mathematical theory of communication. *The Bell System Technical Journal*, **27**, 623–656, 10.1002/j.1538-7305.1948.tb00917.x. 35, 36, 42, 43, 44, 60

JD Silver & CS Zender (2017). The compression–error trade-off for large gridded data sets. *Geoscientific Model Development*, **10**, 413–423, 10.5194/gmd-10-413-2017. 43

P Skibinski (2020). Inikep/lzbench. 54, 55, 64

PK Smolarkiewicz & JA Pudykiewicz (1992). A Class of Semi-Lagrangian Approximations for Fluids. *Journal of the Atmospheric Sciences*, **49**, 2082–2096, 10.1175/1520-0469(1992)049<2082:ACOSLA>2.0.CO;2. 105, 128

V Springel (2005). The cosmological simulation code gadget-2. *Monthly Notices of the Royal Astronomical Society*, **364**, 1105–1134, 10.1111/j.1365-2966.2005.09655.x. 77

D Steinkraus, I Buck & P Simard (2005). Using GPUs for machine learning algorithms. In *Eighth International Conference on Document Analysis and Recognition (ICDAR'05)*, 1115–1120 Vol. 2, 10.1109/ICDAR.2005.251. 124

B Stevens, M Satoh, L Auger, J Biercamp, CS Bretherton, X Chen, P Düben, F Judt, M Khairoutdinov, D Klocke, C Kodama, L Kornblueh, SJ Lin, P Neumann, WM Putman, N Röber, R Shibuya, B Vanniere, PL Vidale, N Wedi & L Zhou (2019). DYAMOND: The DYnamics of the Atmospheric general circulation Modeled On Non-hydrostatic Domains. *Progress in Earth and Planetary Science*, **6**, 61, 10.1186/s40645-019-0304-z. 42

X Sun, N Wang, Cy Chen & Jm Ni (2020). Ultra-Low Precision 4-bit Training of Deep Neural Networks. In *NeurIPS*, 12. 101, 124

A Tantet, V Lucarini & HA Dijkstra (2018). Resonances in a Chaotic Attractor Crisis of the Lorenz Flow. *Journal of Statistical Physics*, **170**, 584–616, 10.1007/s10955-017-1938-0. 105

C The MPI Forum (1993). MPI: A message passing interface. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, Supercomputing '93, 878–883, Association for Computing Machinery, New York, NY, USA, 10.1145/169627.169855. 118

T Thornes, P Düben & T Palmer (2017). On the use of scale-dependent precision in Earth System modelling: Scale-Dependent Precision in Earth System Modelling. *Quarterly Journal of the Royal Meteorological Society*, **143**, 897–908, 10.1002/qj.2974. 102

O Tintó Prims, MC Acosta, AM Moore, M Castrillo, K Serradell, A Cortés & FJ Doblas-Reyes (2019). How to use mixed precision in ocean models: Exploring a potential reduction of numerical precision in NEMO 4.0 and ROMS 3.6. *Geoscientific Model Development*, **12**, 3135–3148, 10.5194/gmd-12-3135-2019. 42, 102, 109, 125

A Toselli & O Widlund (2004). *Domain Decomposition Methods - Algorithms and Theory*. Springer Science & Business Media. 109

AM Turing (1950). I.—COMPUTING MACHINERY AND INTELLIGENCE. *Mind*, **LIX**, 433–460, 10.1093/mind/LIX.236.433. 3, 72

DJ Urminsky (2010). Shadowing unstable orbits of the Sitnikov elliptic three-body problem. *Monthly Notices of the Royal Astronomical Society*, **407**, 804–811, 10.1111/j.1365-2966.2010.16974.x. 81

GK Vallis (2006). *Atmospheric and Oceanic Fluid Dynamics*. Cambridge University Press. 103, 104, 126

F Váňa, P Düben, S Lang, T Palmer, M Leutbecher, D Salmond & G Carver (2017). Single Precision in Weather Forecasting Models: An Evaluation with the IFS. *Monthly Weather Review*, **145**, 495–502, 10.1175/MWR-D-16-0228.1. 42, 77, 102, 109, 124

C Villani (2003). *Topics in Optimal Transportation*. American Mathematical Soc. 85

D Viswanath (2007). Recurrent motions within plane Couette turbulence. *Journal of Fluid Mechanics*, **580**, 339–358, 10.1017/S0022112007005459. 81

E Vitasek (1969). The numerical stability in solution of differential equations. In J.L. Morris, ed., *Conference on the Numerical Solution of Differential Equations*, Lecture Notes in Mathematics, 87–111, Springer, Berlin, Heidelberg, 10.1007/BFb0060017. 132

T von Larcher & R Klein (2019). On identification of self-similar characteristics using the Tensor Train decomposition method with application to channel turbulence flow. *Theoretical and Computational Fluid Dynamics*, **33**, 141–159, 10.1007/s00162-019-00485-z. 42, 68

P Voosen (2020). Europe is building a 'digital twin' of Earth to revolutionize climate forecasts. *Science | AAAS*, 10.1126/science.abf0687. 42

N Wang, J Choi, D Brand, CY Chen & K Gopalakrishnan (2018). Training Deep Neural Networks with 8-bit Floating Point Numbers. *arXiv:1812.08011 [cs, stat]*. 101

Z Wang, A Bovik, H Sheikh & E Simoncelli (2004). Image quality assessment: From error visibility to structural similarity. *IEEE Transactions on Image Processing*, **13**, 600–612, 10.1109/TIP.2003.819861. 52, 53

WMO (2003). Guide to the WMO Table Driven Code Form Used for the Representation and Exchange of Regularly Spaced Data In Binary Form: FM 92 GRIB Edition 2. 2, 58

J Woodring, S Mniszewski, C Brislawn, D DeMarle & J Ahrens (2011). Revisiting wavelet compression for large-scale climate data using JPEG 2000 and ensuring data precision. In *2011 IEEE Symposium on Large Data Analysis and Visualization*, 31–38, 10.1109/LDAV.2011.6092314. 43

G Yalnız, B Hof & NB Budanur (2021). Coarse graining the state space of a turbulent flow using periodic orbits. *Physical Review Letters*, **126**, 244502, 10.1103/PhysRevLett.126.244502. 81

M Zamo & P Naveau (2018). Estimation of the Continuous Ranked Probability Score with Limited Information and Applications to Ensemble Weather Forecasts. *Mathematical Geosciences*, **50**, 209–234, 10.1007/s11004-017-9709-7. 71

CS Zender (2016). Bit Grooming: Statistically accurate precision-preserving quantization with compression, evaluated in the netCDF Operators (NCO, v4.4.8+). *Geoscientific Model Development*, **9**, 3199–3211, 10.5194/gmd-9-3199-2016. 17, 19, 43, 47

H Zhang & SB Ko (2020). Design of Power Efficient Posit Multiplier. *IEEE Transactions on Circuits and Systems II: Express Briefs*, **67**, 861–865, 10.1109/TCSII.2020.2980531. 124

K Zhao, S Di, X Liang, S Li, D Tao, Z Chen & F Cappello (2020). Significantly Improving Lossy Compression for HPC Datasets with Second-Order Prediction and Parameter Optimization. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '20, 89–100, Association for Computing Machinery, New York, NY, USA, 10.1145/3369583.3392688. 42, 68

W Zheng (2020). Research trend of large-scale supercomputers and applications from the TOP500 and Gordon Bell Prize. *Science China Information Sciences*, **63**, 171001, 10.1007/s11432-020-2861-0. 124

J Ziv & A Lempel (1977). A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, **23**, 337–343, 10.1109/TIT.1977.1055714. 58, 64

J Ziv & A Lempel (1978). Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, **24**, 530–536, 10.1109/TIT.1978.1055934. 64

183 references in total.