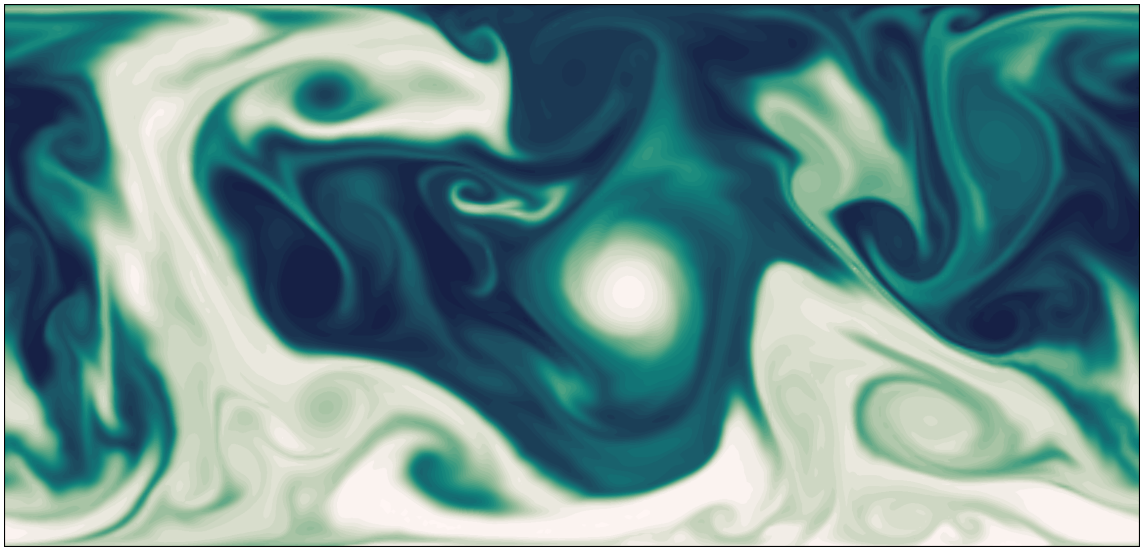
Towards weather and climate models in 16-bit arithmetic



Milan Klöwer
Atmospheric, Oceanic and Planetary Physics
University of Oxford

supervised by

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

A report submitted for the confirmation of status as DPhil candidate.

Oxford, April 2020

Word count

352	Abstract
933	Introduction
2484	Section 2
2923	Section 3
2618	Section 4
808	Section 5
686	Conclusions
10,804	Total

Abstract

Numerical models of weather and climate are a key development of the last decades for weather forecasting and climate predictions and of great importance to society and economy. They rely on computations of real numbers that are represented with 64-bit floating-point numbers on large supercomputers. Given the flattening of Moore's law, we are investigating the potential of 16-bit arithmetics, which are increasingly supported by specialised hardware due to the boom of deep learning applications. Alternatives to the dominating floating-point numbers are compared in this study with software emulators in low and medium complexity models. The results suggest that posit numbers are a generally preferable number format over floats, which is related to higher entropy encoding in our applications, which reduces the total rounding error at the same bit length. A self-organizing number format *Sonum16* is constructed with the lowest rounding errors of all available 16-bit formats due to its maximised entropy, but training and look-up tables are required. Given the comparison of decimal precision we identify the key requirements of a number format to reduce rounding errors. Motivated by the successful stochastic approaches in weather forecasting, a exact-in-expectation stochastic rounding mode was found largely superior to round-to-nearest in our applications by reducing rounding errors and improving regime dynamics in chaotic systems. In general, we show that the transition to 16-bit arithmetic is challenging. However, we provide several mitigation methods that make algorithms resilient against increased rounding errors while retaining the performance potential on 16-bit supporting hardware. Mixing 16-bit arithmetics for the majority of calculations with 32 bit for critically error-prone calculations was shown to be a promising approach that can be implemented on present-day hardware. Reduced precision communication between processors with 16 or even 8-bit numbers was found to introduce negligible errors, providing a performance potential for weather and climate models that are communication bandwidth limited. To systematically identify algorithmic changes to reduce rounding errors, we developed an analysis number format *sherlogs*, which creates bitpattern histograms, estimates the algorithmic information entropy and identifies problematic lines of code of complex algorithms to aid the transition towards 16-bit arithmetic in weather and climate models.

Contents

1 Introduction

Reliable weather forecasts and climate predictions are heavily dependent on the world's largest supercomputers and their performance increase of the last decades following Moore's law. Due to the flattening of processor clock rates, innovations are needed to provide weather and climate models with the required computational performance towards increased reliability of predictions. The Earth's climate system remains very difficult to predict even with the computational resources available, due to its complexity and non-linear dynamics that couple all features from the smallest time and length-scales to the largest. The forecast error of a weather forecast model has several origins [Palmer, 2015, 2012]: (i) Initial and boundary condition errors, which result from observations, data assimilation and external factors; (ii) model error, i.e. the difference between the mathematical model and the real world; (iii) discretisation error resulting from a finite spatial and temporal resolution of the discretised equations and (iv) rounding errors with finite precision arithmetic. The forecast error is largely dominated by (i-iii), depending on the forecast variable and the forecast lead time. In contrast, rounding errors are usually negligible when the IEEE-754 standard on 64-bit double precision floating-point numbers (Float64) is used, which is the default number format for the majority of operational weather forecasts and in climate models.

Reduced precision arithmetics allow for hardware-acceleration and lower energy consumptions due to smaller circuit sizes and an increased potential for vectorization [Jouppi *et al.*, 2017]. Limitations on runtime and energy consumption mean that an increased performance of simulations can be traded for more accurate predictions of weather and climate by increasing model resolution and complexity. The Integrated Forecast System at the European Centre for Medium-Range Weather Forecasts can be accelerated by 40% when 32-bit single precision (Float32) is used for the majority of calculations, with no impact on the forecast skill [Hatfield *et al.*, 2020; Váňa *et al.*, 2017]. MeteoSwiss achieved a similar performance increase with their weather forecast model COSMO [Rüdisühli *et al.*, 2013] by changing from Float64 to Float32. For the European ocean model NEMO mixing 32 and 64 bit calculations is a promising approach to keep accuracy-critical parts in high precision while increasing performance in others [Tintó Prims *et al.*, 2019].

Most of the 64 bits in Float64 do not contain information in simplistic chaotic models [Jeffress *et al.*, 2017]. Using less than 32-bit precision for algorithms in weather forecast models is investigated [Düben, 2018; Düben *et al.*, 2014; Thornes *et al.*, 2017], and

especially mixed-precision approaches with 16-bit half precision floats (Float16) are promising [Hatfield *et al.*, 2019]. Most research on reduced precision modelling for weather and climate applications makes use of software emulators Dawson & Düben [2017] that provide other arithmetics than the widely hardware-supported Float32 and Float64. Software emulators are often an order of magnitude slower than a hardware-implementation onto an arithmetic unit on a processor. However, software emulation allows a scientific evaluation of the use of reduced numerical precision and various alternative number formats can be tested without the need of special hardware (such as FPGAs, Russell *et al.* [2017]). Existing models of weather and climate can therefore be executed with arbitrary number formats and the impact of rounding errors can be investigated.

Deep learning applications receive a strongly increasing attention recently, which influences hardware development towards lower numerical precision in exchange for higher computational performance. More reduced precision number formats, especially at the length of 16 bit are therefore becoming supported on specialised hardware. Graphic processing units started to implement Float16 for increased performance [Markidis *et al.*, 2018]. Google's tensor processing units (TPU, Jouppi *et al.* [2017, 2018]) support the 16b-it BFloat16 format, a truncated version of Float32, as deep learning was found to be successful despite the degradation in precision with 16-bit arithmetic [Burgess *et al.*, 2019; Kalamkar *et al.*, 2019]. Stochastic rounding is proposed as an alternative rounding mode that was found to improve deep learning with low precision number formats substantially [Gupta *et al.*, 2015]. Despite the success of stochastic parametrizations in weather forecasting [Palmer, 2019], only stochastic bit flips were investigated previously [Düben *et al.*, 2014].

The design of floating-point numbers was recently criticised, which led to the development posit numbers, an alternative to floats [Gustafson, 2017] with more accurate results in some algorithms of linear algebra and machine learning [Gustafson & Yonemoto, 2017; Langroudi *et al.*, 2019]. Posits were previously only tested by Klöwer *et al.* [2019] with a software emulator in shallow water models, but further analysis is needed to thoroughly understand their impact on the simulated physics in weather and climate models. Some research on hardware implementations for posits have been carried out recently [Chaurasiya *et al.*, 2018; Chen *et al.*, 2018; Glaser *et al.*, 2017; van Dam *et al.*, 2019], but standardized posit hardware is not yet available. This study will therefore be based on software emulators to compare posits to other available number formats.

Reduced precision modelling is closely related to data compression which aims to archive data fast, without losing valuable information and, ideally, consuming the

smallest storage space possible. This usually poses a dilemma in which not all of the three requirements size, precision, and speed can be satisfied [Silver & Zender, 2017]. However, the encoding of real numbers in bits has to map easily onto the circuits of the arithmetic units in a processor to guarantee fast calculations to meet the performance requirements for reduced precision modelling. Information entropy is central to data compression [MacKay, 2003], and important to identify meaningless bits in uncompressed climate data [Jeffress *et al.*, 2017] which can be removed for successful compression [Zender, 2016]. Information theory is not yet a central perspective for reduced precision modelling, which will be partly addressed in this study.

The study is structured as follows: Section 2 describes various 16-bit number formats, rounding modes, decimal precision and implementations of user-defined number types in the Julia language. The impact of various 16-bit formats on the physics in simulations of the shallow water model are analysed in section 3. Section 4 introduces the newly developed self-organizing numbers sonums. Sherlogs, a new analysis number format is outlined in section 5 to estimate the algorithmic information entropy. We summarize the results in the conclusions in section 6 and provide a thesis outline in section 7. The appendix A.1 lists the open-source developments that have been made as key elements to reproduce the outcomes of this study.

2 16-bit number formats

2.1 Integers and fixed-point numbers

The simplest way to represent a real number in bits is the integer format. An n-bit signed integer starts with a sign bit followed by a sequence of integer bits, that are decoded as a sum of powers of two with exponents $0, 1, ..., n-2$. An positive integer x with signbit $b_0 = 0$ is therefore decoded in bits $b_1, ..., b_{n-2}$ as

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

To avoid multiple representations of zero and to simplify hardware implementations, negative integers, with a sign bit (red) being 1, are decoded with two's complement interpretation (denoted with an underscore) by flipping all other bits and adding 1 [Choo *et al.*, 2003]. For example in the 4-bit signed integer format (Int4), $1110_{\mathrm{Int4}} = 1010\_ = -2$. The largest representable integer for a format with n bits is therefore $2^{n-1} - 1$ and the spacing between representable integers is always 1.

Fixed-point numbers extend the integer format with n_f fraction bits to n_i signed integer bits to decode an additional sum of powers of two with negative exponents $-1, -2, ..., -n_f$. A positive fixed-point number is

$$x = \sum_{i=1}^{n_i-2} 2^{i-1} b_i + \sum_{i=1}^{n_f} 2^{-i} b_{n_i-2+i} \tag{2.2}$$

Every additional fraction bit reduces the number of integer bits, for example Q6.10 is the 16-bit fixed-point format with 6 signed integer bits and 10 fraction bits.

Flexibility regarding the dynamic range can therefore be achieved with integer arithmetic if fixed-point numbers are used [Russell *et al.*, 2017]. Unfortunately, we did not achieve convincing results with integer arithmetic for the applications in this study, as rescaling of the equations is desired to place many arithmetic calculations near the largest representable number [Klöwer *et al.*, 2019]. However, any result beyond will lead to disastrous results, as integer overflow usually returns a negative value following a wrap-around behaviour.

2.2 Floating-point numbers

The IEEE standard on floating-point arithmetic defines how floats encode a real number x in terms of a sign, and several exponent and significant bits

$$x = (-1)^{sign\ bit} \cdot 2^{e-bias} \cdot (1+f) \tag{2.3}$$

The exponent bits e are interpreted as unsigned integers, such that $e - bias$ converts them effectively to signed integers. The significant bits f_i define the significand as $f = \sum_{i=1}^{n_f} f_i 2^{-i}$ such that $(1+f)$ is in the bounds $[1, 2)$. An 8-bit float encodes a real number with a sign bit (red), $n_e = 3$ exponent bits (blue) and $n_f = 4$ fraction bits (black) as illustrated in the following example

$$3.14 \approx 01001001_{\text{Float8}} = (-1)^{0} \cdot 2^{4-bias} \cdot (1 + 2^{-1} + 2^{-4}) = 3.125 \tag{2.4}$$

with $bias = 2^{n_e-1} - 1 = 3$. Exceptions to Eq. 2.3 occur for subnormal numbers, infinity (Inf) and Not-a-Number (NaN) when all exponent bits are either zero (subnormals) or one (Inf when f=0, or NaN else). 16-bit half-precision floating point numbers (Float16) have 5 exponent bits and 10 significant bits. A truncated version of the Float32 format (8 exponent bits, 23 significant bits) is BFloat16 with 8 exponent bits and 7 significant bits. Characteristics of various formats are summarised in Table 2.1. A format with more exponent bits has a wider dynamic range of representable numbers but lower precision, as fewer bits are available for the significant. All floating-point formats have a fixed number of significant bits. Consequently, they have a constant number of significant digits throughout their range of representable numbers (subnormals excluded), which is in contrast to posit numbers, which are introduced in the next section.

2.3 Posit numbers

Posit numbers arise from a projection of the real axis onto a circle (Fig. 2.1), with only one bitpattern for zero and one for Not-a-Real (NaR, or complex infinity), which serves as a replacement for Not-a-Number (NaN). The circle is split into *regimes*, determined by a constant $useed$, which always marks the north-west 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

Format	bits	exp bits	$minpos$	$maxpos$	ϵ	% NaR
Float64	64	11	$5.0 \cdot 10^{-324}$	$1.8 \cdot 10^{308}$	16.3	0.0
Float32	32	8	$1.0 \cdot 10^{-45}$	$3.4 \cdot 10^{38}$	7.6	0.4
Float16	16	5	$6.0 \cdot 10^{-8}$	65504	3.7	3.1
BFloat16	16	8	$9.2 \cdot 10^{-41}$	$3.4 \cdot 10^{38}$	2.8	0.4
Float8	8	3	$1.5 \cdot 10^{-2}$	15.5	1.9	12.5
Posit32	32	2	$7.5 \cdot 10^{-37}$	$7.5 \cdot 10^{37}$	8.8	0.0
Posit(16,1)	16	1	$3.7 \cdot 10^{-9}$	$3.7 \cdot 10^{9}$	4.3	0.0
Posit(16,2)	16	2	$1.4 \cdot 10^{-17}$	$1.4 \cdot 10^{17}$	4.0	0.0
Posit(8,0)	8	0	$1.5 \cdot 10^{-2}$	64	2.2	0.4
Int16	16	0	1	32767	0.8	0
Q6.10	16	0	$9.8 \cdot 10^{-4}$	32.0	3.7	0
LogFixPoint16	16	15	$5.4 \cdot 10^{-20}$	$1.8 \cdot 10^{19}$	3.2	0.0
Approx14	14	13	$5.4 \cdot 10^{-20}$	$9.1 \cdot 10^{18}$	2.6	0.8

Table 2.1: Some characteristics of various number formats. $minpos$ is the smallest representable positive number, $maxpos$ the largest. The machine precision ϵ, is the decimal precision at 1. % NaR denotes the percentage of bit patterns that represent not a number (NaN), infinity or not a real (NaR).

are eventually terminated by an opposite bit. The flexible length allows the significand (or mantissa) to occupy more bits when less regime bits are needed, which is the case for numbers around one. A resulting higher precision around one is traded against a gradually lower precision for large or small numbers. A positive posit number p is decoded as [Gustafson, 2017; Gustafson & Yonemoto, 2017; Klöwer *et al.*, 2019] (negative posit numbers are converted first to their two's complement, see Eq. 2.7)

$$p = (-1)^{sign\ bit} \cdot useed^k \cdot 2^e \cdot (1+f) \tag{2.5}$$

where k is the number of regime bits. e is the integer represented by the exponent bits and f is the fraction which is encoded in the fraction (or significant) bits. The base $useed = 2^{2^{e_s}}$ is determined by the number of exponent bits e_s. More exponent bits increase - by increasing $useed$ - the dynamic range of representable numbers for the cost of precision. The exponent bits themselves do not affect the dynamic range by changing the value of 2^e in Eq. 2.5. They fill gaps of powers of 2 spanned by $useed = 4, 16, 256, ...$ for $e_s = 1, 2, 3, ...$, and 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]. We will use a notation where Posit(n,e_s) defines the posit numbers with n bits including e_s exponent bits. A

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

$$57 \approx 01110111_{\text{Posit}(8,1)} = (-1)^0 \cdot 4^2 \cdot 2^1 \cdot (1 + 2^{-1} + 2^{-2}) = 56 \tag{2.6}$$

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 bits being 0, and positive, but subtract 1, for the bits being 1. The exponent bits are interpreted as unsigned integer and the fraction bits follow the IEEE floating-point standard for significant bits. For negative numbers, i.e. the sign bit being 1, all other bits are first converted to their two's complement (Choo *et al.* [2003], denoted with an underscore subscript) by flipping all bits and adding 1,

$$\begin{aligned} -0.28 \approx 11011110_{\text{Posit}(8,1)} &= 10100010\_ \\ &= (-1)^1 \cdot 4^{-1} \cdot 2^0 \cdot (1 + 2^{-3}) = -0.28125. \end{aligned} \tag{2.7}$$

After the conversion to the two's complement, the bits are interpreted in the same way as in Eq. 2.6.

Posits also come with a no overflow/no underflow-rounding mode: Where floats overflow and return infinity when the exact result of an arithmetic operation is larger than the largest representable number ($maxpos$), posit arithmetic returns $maxpos$ instead, and similarly for underflow where the smallest representable positive number ($minpos$) is returned. This is motivated as rounding to infinity returns a result that is infinitely less correct than $maxpos$, although often desired to indicate that an overflow occurred in the simulation. Instead, it is proposed to perform overflow-like checks on the software level to simplify exception handling on hardware [Gustafson, 2017]. Many functions are simplified for posits, as only two exceptions cases have to be handled, zero and NaR. Conversely, Float64 has more than 10^{15} bitpatterns reserved for NaN, but these 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 posit number framework also highly recommends *quires*, an additional register on hardware to store intermediate results. Dot-product operations 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 is available on some processors), but is technically difficult to implement on hardware for a general dot-product as the required

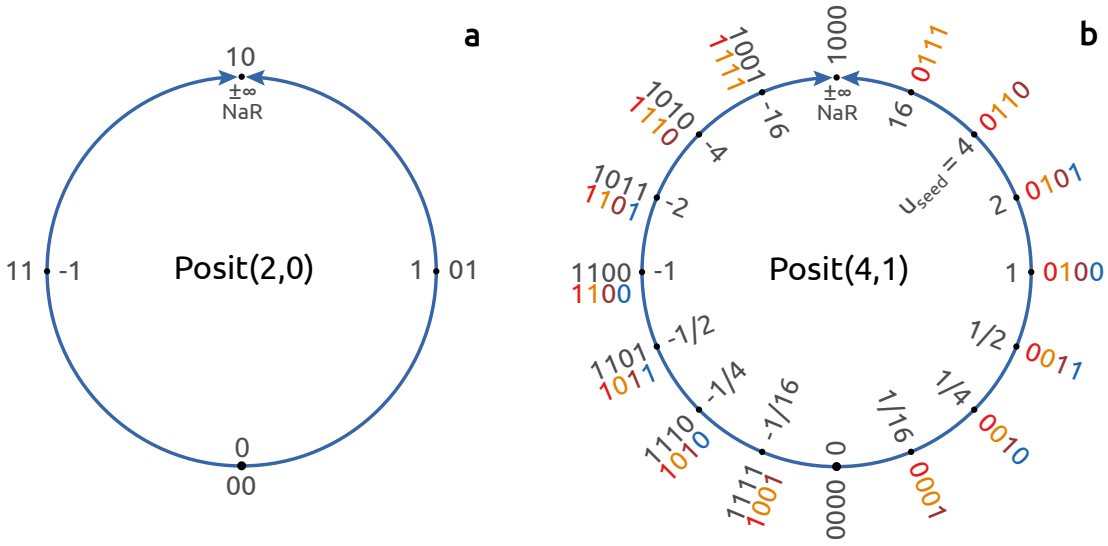


Figure 2.1: Two posit number formats obtained by projecting the real axis onto a circle. (a) 2bit Posit(2,0) and (b) 4bit Posit(4,1). The bit patterns are marked on the outside and the respective values on the inside of each circle. Bit patterns of negative numbers (black) have to be converted to their two's complement (colours) first (see text). At the top of every circle is complex infinity ($\pm\infty$) or NaR (Not-a-Real). After Gustafson [2017].

registers would need to be much larger in size. For fair comparison we do not take quires into account in this study. The posit number format is explained in more detail in Gustafson [2017]. In order to use posits on a conventional CPU we developed for the Julia programming language [Bezanson *et al.*, 2017] the posit emulator *SoftPosit.jl* [Klöwer & Giordano, 2019], which is a wrapper for the C-based library SoftPosit [Leong, 2020]. The type-flexible programming paradigm, facilitated by the Julia language, is outlined in 2.7.

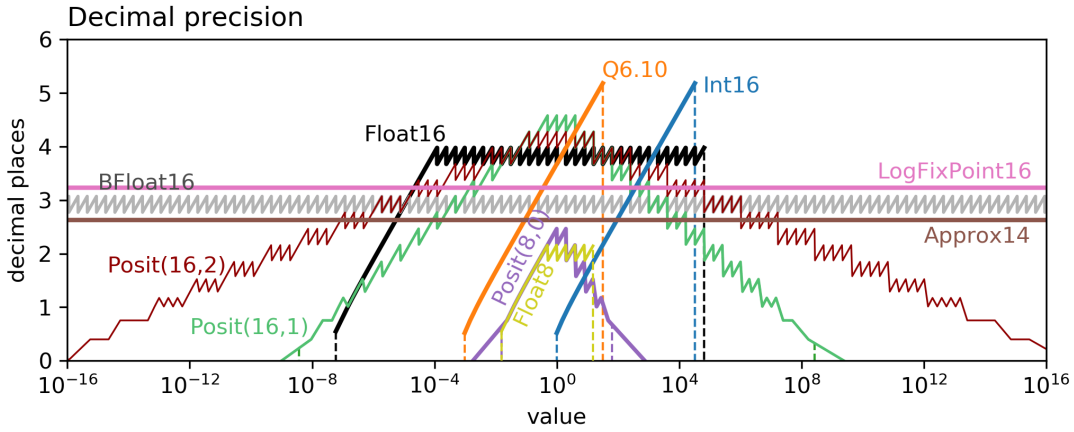


Figure 2.2: Decimal precision of various number formats. Dashed vertical lines indicate the range of representable numbers for each format. Float64, Float32 and Posit32 are beyond the axes limits.

2.4 Logarithmic fixed-point numbers

Fixed-point numbers have a limited range and for the applications in this study an unsuitable distribution of decimal precision. However, logarithmic fixed-point numbers are similar to floating-point numbers. A n-bit logarithmic fixed-point number is defined as

$$x = (-1)^{sign\ bit} \cdot 2^e \tag{2.8}$$

where e is encoded as an $(n-1)$-bit fixed-point number (Eq. 2.2). Consequently, logarithmic fixed-point numbers are equally spaced in log-space and have a perfectly flat decimal precision throughout the dynamic range of representable numbers (Fig. 2.2). We call the 16-bit logarithmic fixed-point numbers with 7 signed integer bits and

8 fraction bits LogFixPoint16. Approx14 is a proprietary number format developed by Singular Computing, which is essentially a 14-bit logarithmic fixed-point numbers with 7 signed integer bits and 6 fraction bits.

Logarithmic number formats have the advantage that no rounding error is applied for multiplication, as the addition of the exponents is exact with fixed-point numbers (as long as no under or overflow occurs). Hence, multiplication with LogFixPoint16 and Approx14 is not just exact but also fast, due to implementation as integer addition. Conversely, addition with logarithmic numbers is difficult. Adding two logarithmic numbers involves the computation of a logarithm, which, however, for low precision numbers can be implemented as a table look-up.

Both LogFixPoint16 and Approx14 come with a round-to-nearest rounding mode in log2-space. We consider $x_1 = 2^0 = 1$ and $x_2 = 2^1 = 2$ as two representable numbers as an example with x in between. With round-to-nearest (and tie to even) in linear-space all numbers x larger equal 1.5 are round up and others round down. With round-to-nearest in log2-space $2^{\frac{1}{2}} = \sqrt{2} = 1.414...$ is the log-midpoint as $\log_2(\sqrt{2}) = 0.5$. Consequently, the numbers between $\sqrt{2}$ (inclusive) and 2 will be round up and only numbers between 1 and less than $\sqrt{2}$ will round down. Hence, the linear range of numbers that will be round up is larger than those that will round down. This rounding is biased as the expectation of rounded uniformly distributed values between 1 and 2 is not equal to the expectation without rounding. Let $\mathrm{round}_{\log_2}(x)$ be the round-to-nearest function in log2-space and x be drawn N-times from a random uniform distribution $U(1, 2)$, then

$$\frac{1}{N}\sum_i^N x_i = 1.5 \neq \frac{1}{N}\sum_i^N \mathrm{round}_{\log_2}(x_i) = 3 - \sqrt{2} = 1.586... \tag{2.9}$$

We will investigate the effect of this round-to-nearest in log2-space in section 3.5.

2.5 Decimal precision

The decimal precision is defined as [Gustafson, 2017; Gustafson & Yonemoto, 2017]

$$\mathrm{decimal\,precision} = -\log_{10}|\log_{10}(\frac{x_{\mathrm{repr}}}{x_{\mathrm{exact}}})| \tag{2.10}$$

where x_{exact} is the exact result of an arithmetic operation and x_{repr} is the representable number that x_{exact} is rounded to, given a specified rounding mode. For the common round-to-nearest rounding mode, the decimal precision approaches infinity when the

exact result approaches the representable number and has a minimum in between two representable numbers. 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.

Fig. 2.2 compares the worst-case decimal precision for various 16 and 8-bit floats and posits, as well as 16-bit integers, the fixed-point format Q6.10 (6 integer bits, 10 fraction bits) and logarithmic fixed-point numbers LogFixPoint16 and Approx14. Float16 has a nearly constant decimal precision of almost 4 decimal places, which decreases for the subnormal numbers towards the smallest representable number $minpos$. 16-bit posits, on the other hand, show an increased decimal precision for numbers around 1 and a wider dynamic range, in exchange for less precision for numbers around 10^4 as well as 10^{-4}. The machine precision ϵ (in analogy to the machine error, also known as machine epsilon), defined as half the distance between 1 and the next representable number, is given in terms of decimal precision and is summarised in Table 2.1 for the various formats. Due to the no overflow/no underflow-rounding mode, the decimal precision is slightly above zero outside the dynamic range.

The decimal precision of 16-bit integers is negative infinity for any number below 0.5 (round to 0) and maximised for the largest representable integer $2^{15} - 1 = 32767$. Similar conclusions hold for the fixed-point format Q6.10, as the decimal precision is shifted towards smaller numbers by a factor of $\frac{1}{2}$ for each additional fraction bit.

2.6 Stochastic rounding

The default rounding mode for floats and posits is round-to-nearest tie-to-even. In this rounding mode an exact result x is rounded to the nearest representable number x_i. In case x is half-way between two representable numbers, the result will be tied to the even. A floating-point number x_i is considered to be even, if its significand ends in a zero bit. These special cases are therefore alternately round up or down, which removes a bias that otherwise persists (see Eq. 2.9 for an example of biased rounding). Let x_1 and x_2 be the closest two representable numbers to x and $x_1 \leq x < x_2$ then

$$\mathrm{round}_{\mathrm{nearest}}(x) = \begin{cases} x_1 & \text{if } x - x_1 < x_2 - x, \\ x_1 & \text{if } x - x_1 = x_2 - x \text{ and } x1 \text{ even}, \\ x_2 & \text{else}. \end{cases} \tag{2.11}$$

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. Let δ be the distance between x_1, x_2, then

$$\mathrm{round}_{\mathrm{stoch}}(x) = \begin{cases} x_1 & \text{with probability} \quad 1 - \delta^{-1}(x - x_1) \\ x_2 & \text{with probability} \quad \delta^{-1}(x - x_1). \end{cases} \tag{2.12}$$

This behaviour is illustrated in Fig. 2.3. 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. 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 big as for round-to-nearest, and when low-probability round away from nearest occurs, it can be up to $\pm\delta$, whereas for round-to-nearest the error is bound 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

$$\lim_{N\to\infty} \frac{1}{N} \sum_{i}^{N} \mathrm{round}_{\mathrm{stoch}}(x) = x \tag{2.13}$$

as follows by inserting Eq. 2.12. Stochastic rounding is therefore exact in expectation.

The stochastic rounding mode is implemented for Float16 and BFloat16. Software emulations of both number formats rely on conversion to Float32, such that the exact result (to the precision provided by Float32) is known before conversion back to 16 bit. Instead of calculating the probabilities given in Eq. 2.12, we add a stochastic perturbation $\xi \in [-\frac{\delta}{2}, \frac{\delta}{2}]$ to x before round-to-nearest. Let r be uniformly distributed in $[0, 1]$ then Eq. 2.12 can then be rewritten as

$$\mathrm{round}_{\mathrm{stoch}}(x) = \begin{cases} \mathrm{round}_{\mathrm{nearest}}(x + \frac{\delta}{2}(r - \frac{x - x_1}{\delta})) & \text{if } x_1 = 2^n \text{ and } x - x_1 < \frac{\delta}{4} \\ \mathrm{round}_{\mathrm{nearest}}(x + \delta(r - \frac{1}{2})) & \text{else}\,. \end{cases} \tag{2.14}$$

The special case only occurs for x being within $\frac{\delta}{4}$ larger than a floating-point number $x_1 = 2^n$, that means with zero significand. In this case the distance from x_1 to the previous float is only $\frac{\delta}{2}$, which has to be accounted for.

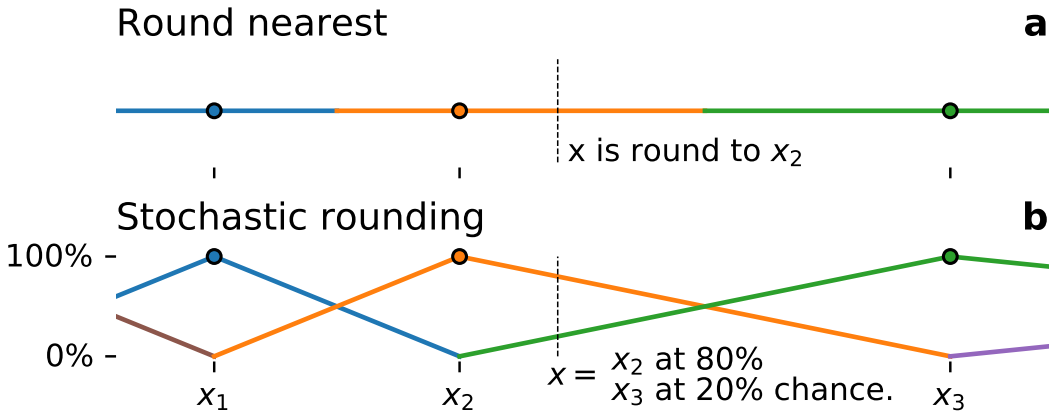


Figure 2.3: Stochastic rounding schematically. (a) For round to nearest, the number x is round to the nearest representable number, which is x_2 (orange) in the example. (b) For stochastic rounding, x is round down at chance p, which is proportional to the distance of the two closest representable numbers. In the example, x is round down at 80% chance, but round up at 20% chance.

2.7 A type-flexible programming paradigm

Julia's programming paradigms of *multiple-dispatch* and *type-stability* facilitate the use of arbitrary number formats without the need to rewrite an algorithm, while compiling functions for specific types [Bezanson *et al.*, 2017]. As this is an essential feature of Julia and extensively made use of here, we briefly outline the benefits of Julia by computing the harmonic sum $\sum_{i=1}^{\infty} \frac{1}{i}$ with various number types 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.

Executing the function `harmonic_sum` for the first time with a type `T` as the first argument, triggers Julia's *just-in-time* compiler (Fig. 2.4). The function is type-stable, as the types of all variables are declared and therefore known to the compiler. At the same time Julia allows for type-flexibility, as its *multiple-dispatch* means that calling `harmonic_sum` with another type `T2` will result in a separately compiled function for `T2`. We can therefore compute the harmonic sum with arbitrary number types, as long as the zero-element `zero(T)`; the one-element `one(T)`; addition; division; conversion from

```
function harmonic_sum(::Type{T},steps::Int=2000) where T

    s = zero(T)
    o = one(T)

    for i in 1:steps

        s_old = s
        s += o/T(i)

        if s == s_old    # check for convergence
            println(Float64(s),i)
            break
        end
    end
end
```

Figure 2.4: A type-flexible harmonic sum function in the Julia language.

integer and conversion to float are defined for `T`.

```
julia> using SoftPosit
julia> using BFloat16s
julia> harmonic_sum(Float16)
(7.0859375, 513)

julia> harmonic_sum(BFloat16)
(5.0625, 65)

julia> harmonic_sum(Posit16)
(7.77734375, 1024)
```

Figure 2.5: Harmonic sum example use of the posit emulator *SoftPosit.jl* in the Julia shell. `Posit16` is the Posit(16,1) standard.

The harmonic sum converges after 513 elements when using Float16 (Fig. 2.5). The precision of BFloat16 is so low that the sum already converges after 65 elements, as the addition of the next term $1/66$ is rounded back to 5.0625. We identify the addition of small terms to prognostic variables of size $\mathcal{O}(1)$ as one of the major challenges with low precision arithmetic, which is discussed in more detail in section 3.5. Using Posit(16,1), the sum only converges after 1024 terms, due to the higher decimal precision of posits between 1 and 10.

3 Impact on the physics

We investigate the impact that various 16-bit number formats have on the simulated physics of different low and medium-complexity models. The Lorenz 1963 and the shallow water model are introduced in Klöwer *et al.* [2019].

3.1 Error growth

Low precision number formats increase the rounding error, which grows over time in a chaotic system. As long as the rounding error is masked by other errors, its contribution to a degradation in forecast skill is negligible. We analyse the rounding error growth in the shallow water system, by running simulations with various 16-bit number formats and compare them to Float64.

The solution to the shallow water equations includes vigorous turbulence that dominates a meandering zonal current. Using either float or posit arithmetic in 16 bit the simulated fluid dynamics are very similar to a Float64 reference: As shown in a snapshot of tracer concentration (Fig. 3.1) turbulent stirring and mixing can be well simulated with posits. However, the Float16 simulation (Fig. 3.1d) deviates much faster than the posit simulations (Fig. 3.1b and c) from the Float64 reference (Fig. 3.1a), presumably due to 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 a first evidence that the accumulated rounding errors with posits are smaller than with floats. BFloat16 arithmetic is not able to simulate the shallow water dynamics, presumably as tendencies are too small to be added to the prognostic variables, an issue that also occurs in the Lorenz system (Fig. 3.8d) and even in the harmonic sum (Fig. 2.5).

To quantify differences between the different 16-bit arithmetics we perform short-term forecasts with the medium-resolution configuration. To quantify the error growth of 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 root mean square error (RMSE) of sea surface height with respect to Float64 simulations. Other variables yield similar results. Each forecast is performed several times from identical initial conditions but with the various number formats. To compare the magnitude of rounding error that are caused by a reduction in precision to

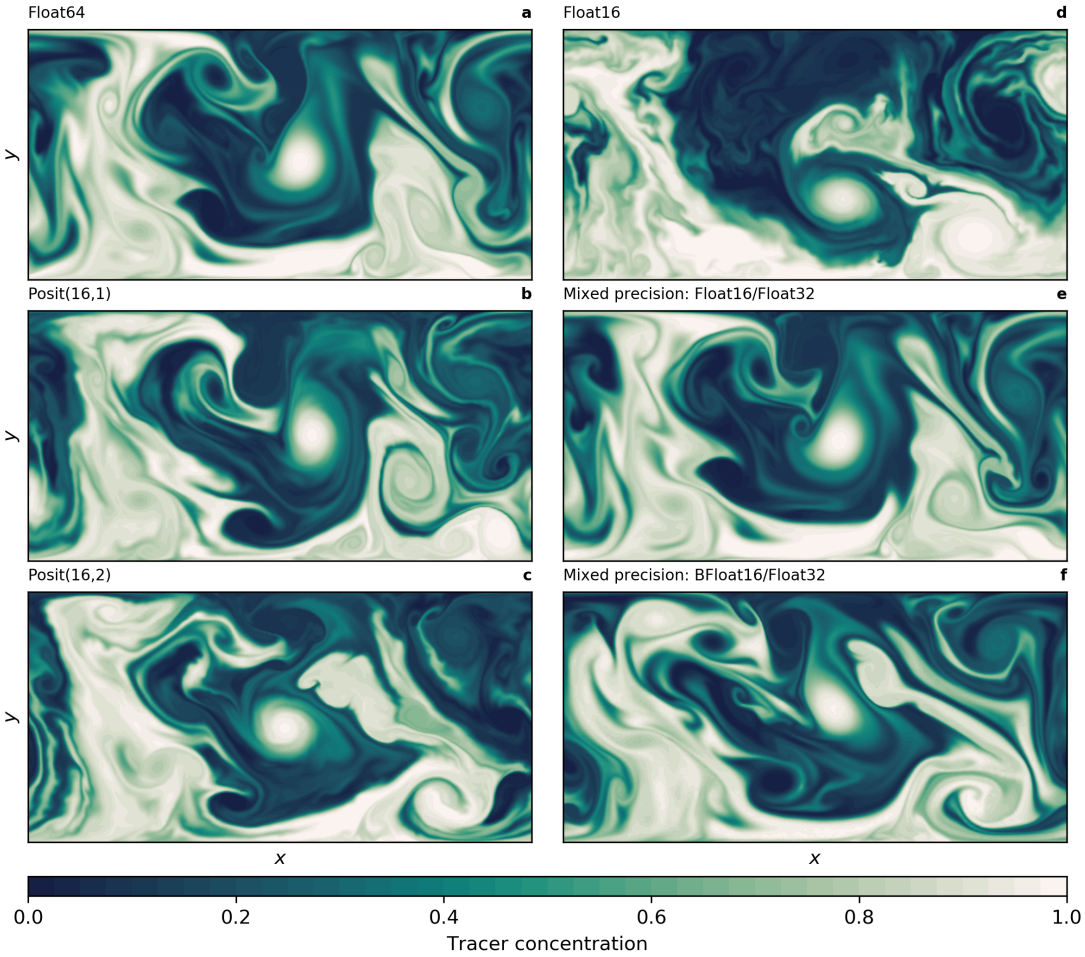


Figure 3.1: Snapshot of tracer concentration simulated by the shallow water model using different 16bit number formats. Mixed precision using Float32 for the prognostic variables only is used for (e) and (f). The tracer was injected uniformly in the lower half of the domain 50 days before. This simulation was run at an increased resolution of $\Delta = 5$km (400x200 grid points).

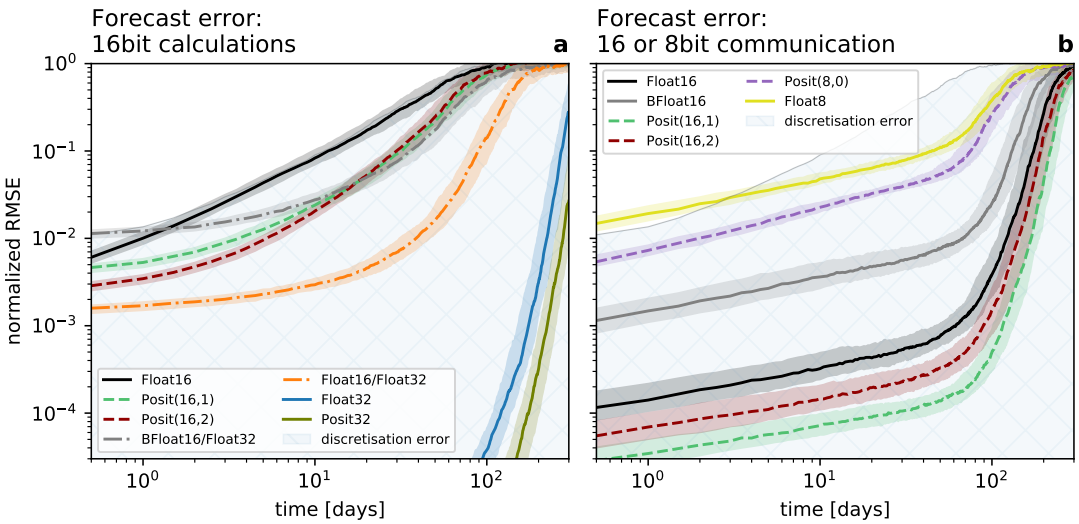


Figure 3.2: Forecast error measured as the root mean square error (RMSE) of sea surface height η taking Float64 as reference. (a) Forecast error for various 16-bit number formats and mixed 16-bit/32-bit simulations for which the prognostic variables are kept at Float32. (b) Forecast error for reduced 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 lines represent the median of 200 forecasts per number format. The shaded areas of each model configuration denote the interquartile range of the ensemble.

a realistic level of error that is caused by model discretisation, we also perform forecasts with Float64 and a low-resolution model configuration, which are used to estimate the discretization error. We normalise the RMSE by the climatological mean forecast error at very long lead times, which is the same for all model configurations. A normalised RMSE of 1 therefore means that all information of the initial conditions is removed by chaos.

The forecast error of Float16 is as large as the discretisation error and clearly outperformed by 16-bit posit arithmetic (Fig. 3.2a). Both Posit(16,1) and Posit(16,2) 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 very short lead times. Also at 32 bit, posits clearly outperform floats.

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. The mean state is an eastward flow of about 0.3 m/s, about 3 to 4 times weaker than individual velocities throughout the domain (Fig. 3.3a), 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. 3.3b). The variability is significantly increased by 10 – 30% with 16-bit arithmetic, especially with Posit(16,2). This is probably caused by rounding errors that are triggering local perturbation which increase variability.

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 (conventional notation)

$$\mathbf{u}_g = \frac{g}{f}\hat{\mathbf{z}} \times \nabla\eta \tag{3.1a}$$

$$\mathbf{u} = \mathbf{u}_g + \mathbf{u}_{ag} \tag{3.1b}$$

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

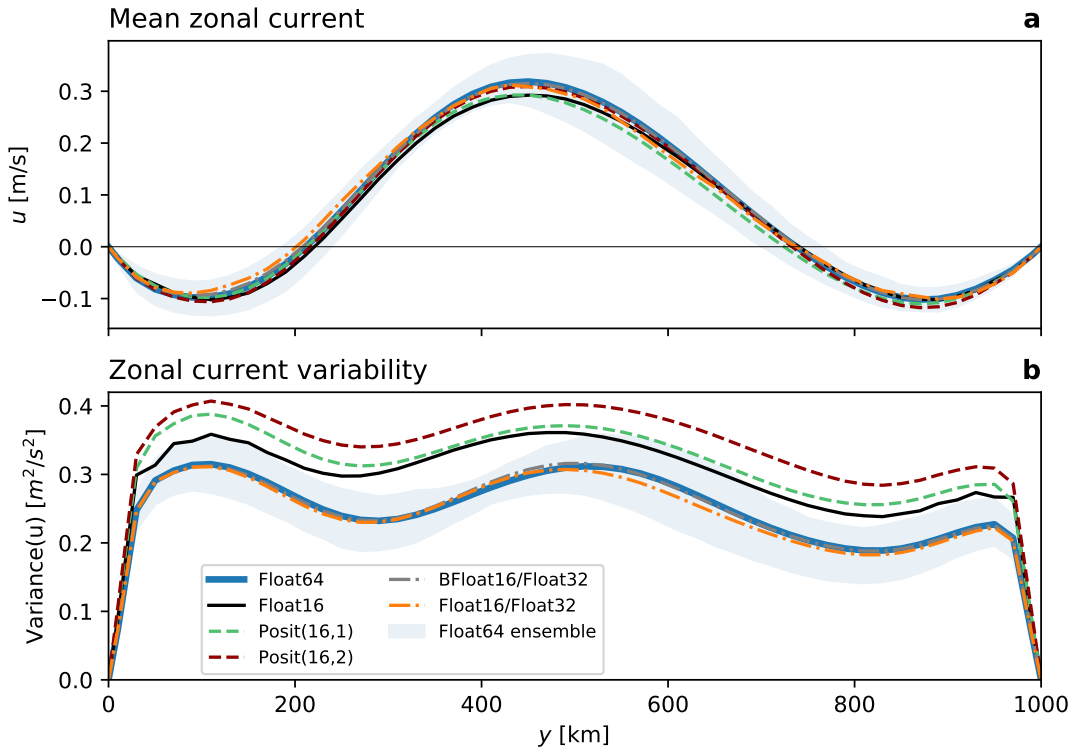


Figure 3.3: Climatology and variability of the zonal current. (a) Zonally-averaged zonal current u as a function of the meridional coordinate y. (b) Zonal variance of the zonal current as a function of y. The shaded area denotes the interquartile temporal variability around the (a) mean and (b) variance of reference simulation with Float64.

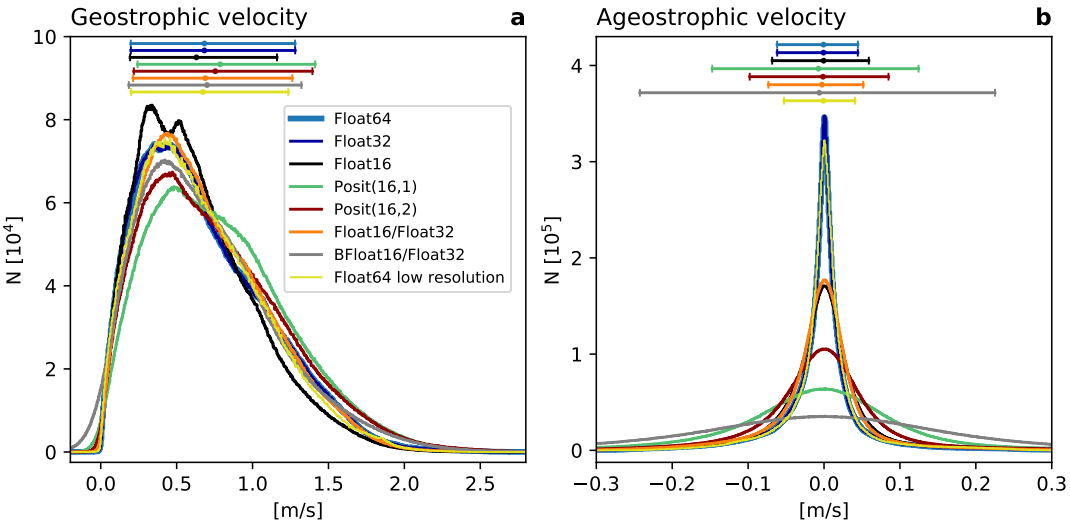


Figure 3.4: Geostrophic balance as simulated with different number formats. (a) Histograms of flow-parallel components of geostrophic velocity. (b) as (a) but for the ageostrophic velocities. Horizontal bars denote the mean, 10th and 90th-percentile in respective colours.

$\tilde{u}_g$ and $\tilde{u}_{ag}$ via

$$\tilde{u}_g = \frac{\mathbf{u}_g \cdot \mathbf{u}}{\|\mathbf{u}\|}, \quad \tilde{u}_{ag} = \frac{\mathbf{u}_{ag} \cdot \mathbf{u}}{\|\mathbf{u}\|}. \tag{3.2}$$

The geostrophic velocities in the shallow water simulations can reach up to 2 m/s, are hardly negative (i.e. against the flow) and have a mean of about 0.7 m/s (Fig. 3.4a). This behaviour is well simulated with 16-bit number formats, although posits increase the strength of geostrophic velocities slightly. Ageostrophic velocity components are found to be isotropic, and are oriented equally frequent with and against the prevailing flow, but rarely exceed $\pm$0.1m/s and are therefore 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 shows clearly fewer ageostrophic velocities around 0, pointing towards an increased number of simulated instabilities. Posits have an even further increased number of ageostrophic velocities, and especially Posit(16,1) increases the variance of those 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

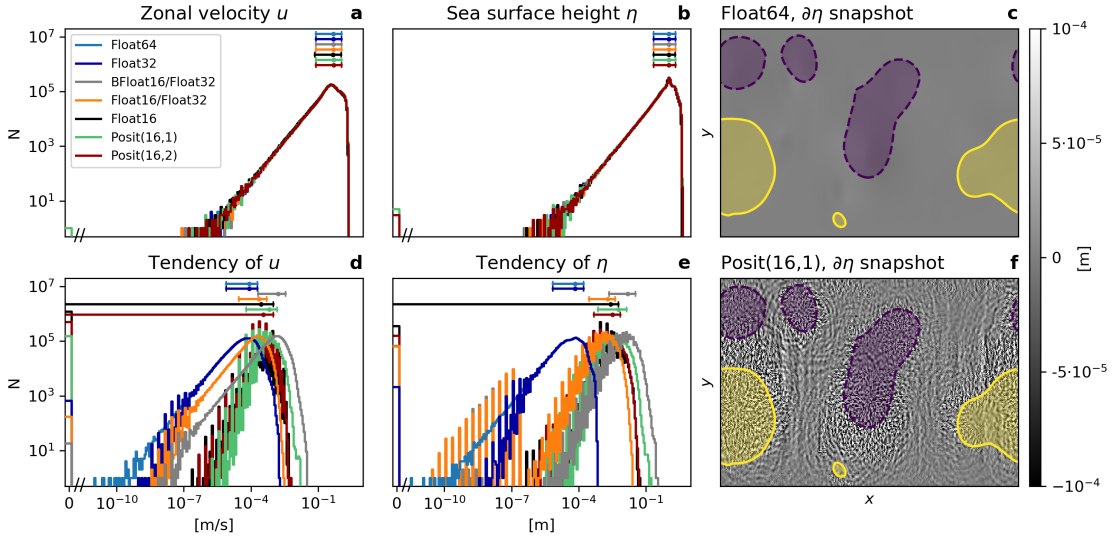


Figure 3.5: Histograms of the numeric values of the prognostic variables (a) zonal velocity u, (b) sea surface height η, and the respective tendencies of (d) u and (e) η, simulated with different 16, 32 and 64-bit number formats. Mean, 10th and 90th percentile are shown above the histograms in respective colors. Snapshots of the tendencies of η simulated with (c) Float64 and (f) Posit(16,1), other 16bit formats are similar. Areas of sea surface height anomalies exceeding ± 1.4 m are shown in purple (negativ) and yellow (positive). Note the break on the x-axis in (a,b,d) and (e).

in the simulations is largely maintained, rounding errors lead, likely due to an increase in ageostrophy, to a higher variability in the flow field.

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. 3.5a and b). However, the tendencies are increased by orders of magnitude with 16-bit arithmetics (Fig. 3.5d and e), as rounding errors cause gravity waves to radiate away from eddies (Fig. 3.5f). Gravity waves are identified from the tendency of sea surface height. Comparing their propagation to the location of anomalous sea surface height, which is used as a proxy for 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. 3.5c) and tend to have only a small impact on quasi-gestrophic dynamics, as they act on different time and length scales. It is unclear whether these

gravity waves cause the observed ageostrophic velocities.

The 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 error 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 error of 2.8, which explains why no change from initial conditions in the shallow water system can be simulated with BFloat16.

3.5 Mixed precision arithmetic in the shallow water model

In the previous simulations the entire shallow water equations were calculated 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 for numbers around 1. The prognostic variables are now reduced to Float16 or BFloat16 before calculations of the right-hand side and every term of the tendencies is converted back before addition to the prognostic variables. Using subscripts 16 and 32 to denote variables held at 16 and 32-bit precision, respectively, and let Float32() be the conversion function then the continuity equation in the shallow water system $\partial_t \eta = -\nabla \cdot (\mathbf{u}h)$ becomes

$$\frac{\partial \eta_{32}}{\partial t} = -\,\mathrm{Float32}(\partial_x(u_{16}h_{16}) + \partial_y(v_{16}h_{16})) \tag{3.3}$$

and similar for u and v in the momentum equations.

Snapshots of tracer concentration reveal well simulated geostrophic turbulence (Fig. 3.1e and f) with Float16/Float32 or BFloat16/Float32 and instabilities at fronts or in filaments are visibly reduced compared to pure 16-bit arithmetic. The forecast error is strongly reduced once the prognostic variables are kept as Float32 (Fig. 3.2a), supporting the hypothesis that the addition of tendencies to the prognostic variables is a key computation with low rounding error-tolerance. Despite BFloat16 not being suitable for shallow water simulations when applied to all computations, mixing BFloat16 with Float32 arithmetic yields a similar error growth to posits, which is well below the discretization error. Mean state or variability are virtually identical for both mixed precision cases (Fig. 3.3) compared to the Float64 reference. The geostrophic balance

is largely unaffected, but ageostrophic velocities increase in variance, especially for BFloat16 (Fig. 3.4). Gravity waves are similarly present for mixed precision although weaker for tendencies computed with Float16 (Fig. 3.5d) 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, 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 a possible solution to circumvent spurious behaviour due to 16-bit floating-point arithmetics when solving the shallow water equations. Performance benefits are still possible as most calculations are performed with 16 bit, with key computations in 32 bit to reduce the overall error. Depending on the application, the conversions between number formats are assumed to be of negligible cost. This 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, as it is the case for posits.

Inaccurate rounding: Logarithmic fixed-point numbers

Motivated by the successful use of BFloat16 to compute the tendencies, we want to test two logarithmic fixed-point number formats that have a similar distribution of decimal precision compared to BFloat16: LogFixPoint16 and Approx14. The former has a slightly higher precision (machine precision 3.2, see Table 2.1) than BFloat16 (machine epsilon 2.8) whereas Approx14 has a slightly lower (2.6). Logarithmic number formats have the advantage that multiplications (and divisions) do not involve rounding and are therefore exact (under or overflows excluded). Whether this is important in a simulation where both multiplications and additions (as well as subtractions) are required, is investigated. Similar to BFloat16, LogFixPoint16 and Approx14 are not suited for the entire model and we therefore restrict their use only to the computation of the tendencies in a mixed-precision approach with Float32 used for the prognostic variables.

The same benchmark simulation in the shallow water model as before is used to provide some evidence that logarithmic number formats violate conservation laws, at least with the current round-to-nearest rounding mode in log2-space (Fig. 3.6 and see Eq. 2.9). Both formats represent the general picture of tracer advection correctly,

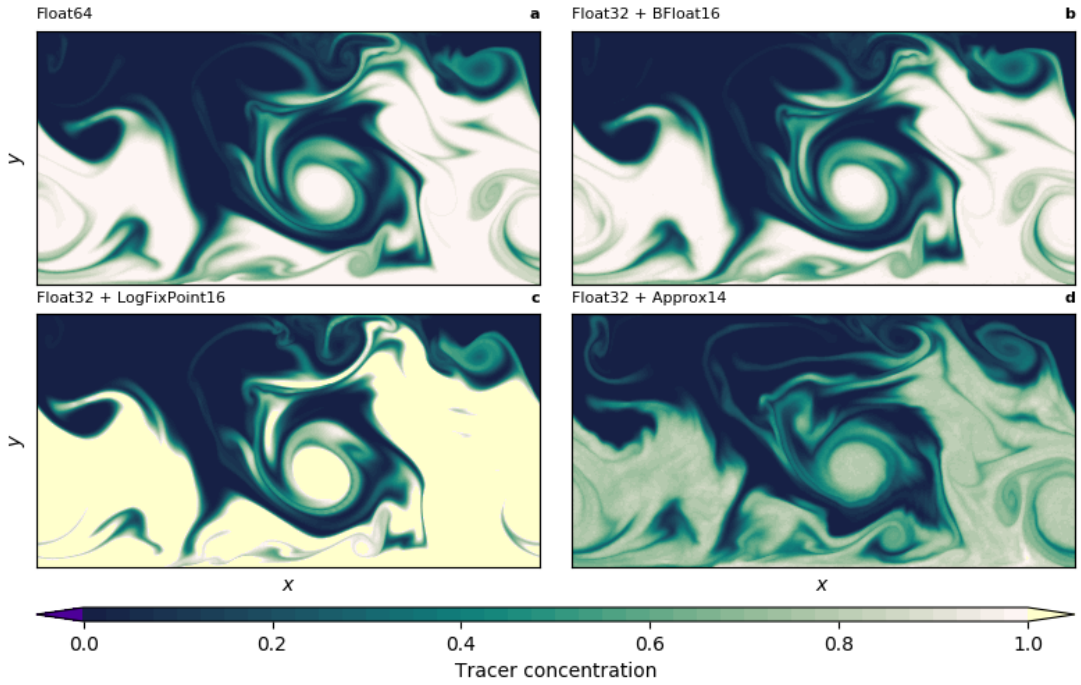


Figure 3.6: Snapshot of tracer concentration simulated by the shallow water model with logarithmic fixed-point numbers used to calculate the tendencies. Float32 is used for the prognostic variables. The tracer was injected uniformly in the lower half of the domain 25 days before. This simulation was run at a resolution of $\Delta = 5$km (400x200 grid points).

LogFixPoint16 clearly more accurate than Approx14, which is unsurprising given the higher precision. However, Approx14 introduces a tracer sink, whereas LogFixPoint16 increases the tracer concentrations beyond physically possible values. We argue that this is related to the rounding mode, such that a revision is necessary to ensure a conservative (or unbiased) rounding mode is in place that does not violate conservation laws. The differences between Approx14 (being a tracer sink) and LogFixPoint16 (being a tracer source) is presumably related to the exact implementation of rounding in additions. As the two formats violate conservation in either direction, this could mean that a conservative rounding is possible. This can presumably be achieved with a change of the rounding mode to represent exactly round-to-nearest (and tie to even) in linear-space although calculated in log2-space. A thorough analysis of other conserved quantities in the shallow water system, such as mass or energy, is necessary to assess whether logarithmic number formats are a promising alternative to floating-point numbers.

3.6 Reduced precision communication

A standard method to parallelise simulations is the distributed-memory parallelism via Message Passing Interface (MPI). We emulate MPI-like communication in the shallow water model with the copying of boundary values between 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 was used to communicate 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 show a negligible error for Float16 and posits (Fig. 3.7). 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. 3.5a 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. As even the worst 16-bit communication format, BFloat16, has a smaller error than the best mixed precision formats, Float16 with Float32, we 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

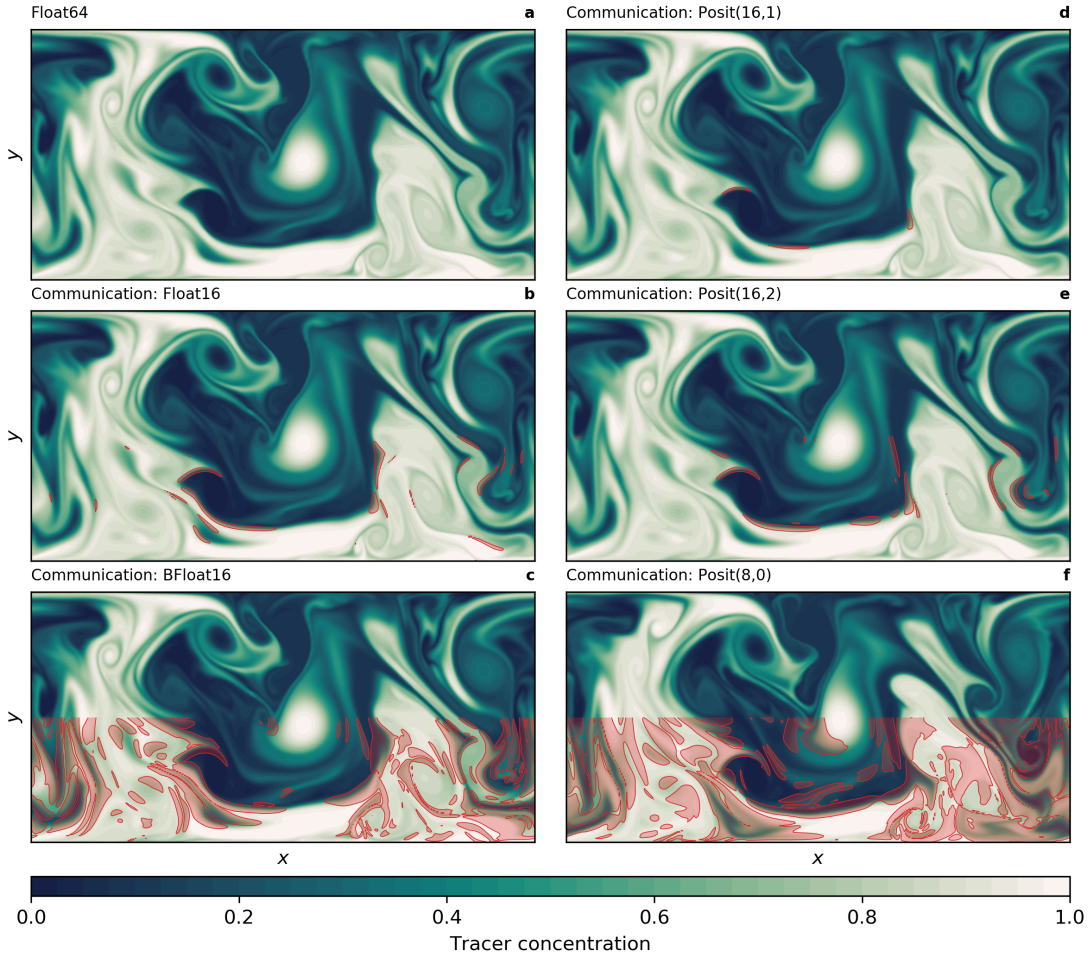


Figure 3.7: Snapshot of tracer concentration simulated by the shallow water model using reduced precision communication. The communication of boundary values occurs at every time step for the prognostic variables. Float64 was used for all calculations. Areas where the absolute error exceeds 0.05 are 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 at a resolution of $\Delta = 5$km (400x200 grid points).

the discretization error. Having said that, Float8 communication introduces an error that is comparably large initially but grows only linearly in the first 50 days of the simulation, which is in contrast to the exponential error growth observed for 16-bit arithmetic.

Reduced precision communication was not found to have a significant impact on either mean state, variability, geostrophy or tendencies. We acknowledge that 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 management is realized. However, in the case that communication is an identified performance bottleneck in a given application, the results here suggest that reliable model simulations can be achieved with 16 or even 8-bit communication. The range of values for the prognostic variables here is comparably small, facilitating 8-bit communication. Such reductions might be in general difficult to implement. Although we show that posits are a preferable number format to be used for 16-bit communication, it remains an open question how efficient an implementation can be, given the computational cost of the conversion between formats.

3.7 Stochastic rounding

Simple stochastic models like the Lorenz 1963 system are of such low dimensionality that the attractor drastically decreases in complexity for large rounding errors (Fig. 3.8b and d). Both Float16 and especially BFloat16 simulate an attractor that consists of an orbit with a relatively short period. Due to the deterministic nature of the rounding errors this orbit repeats and its period depends on the initial condition and the time step size Klöwer *et al.* [2019]. Although some 16-bit number formats like posits or sonums (see section 4) are able to simulate an attractor that resembles more the Float64 approximation to the analytic solution, we will investigate the effect of exact-in-expectation stochastic rounding (see section 2.6 for details) for Float16 and BFloat16, two number formats with the largest rounding errors for default round-to-nearest.

Simulating the Lorenz system with BFloat16 plus stochastic rounding drastically improves the complexity of the attractor (Fig. 3.8c). Therefore, the main problem of simulating a complex attractor even for BFloat16 is not a small amount of representable numbers which undersamples the state space but the rounding errors that occur when calculating the tendencies, and presumably especially the addition of the tendencies to the prognostic variables (see section 3.5). Stochastic rounding introduces a relative error that scales inversely with the precision of a given number format (see section 2.6). As

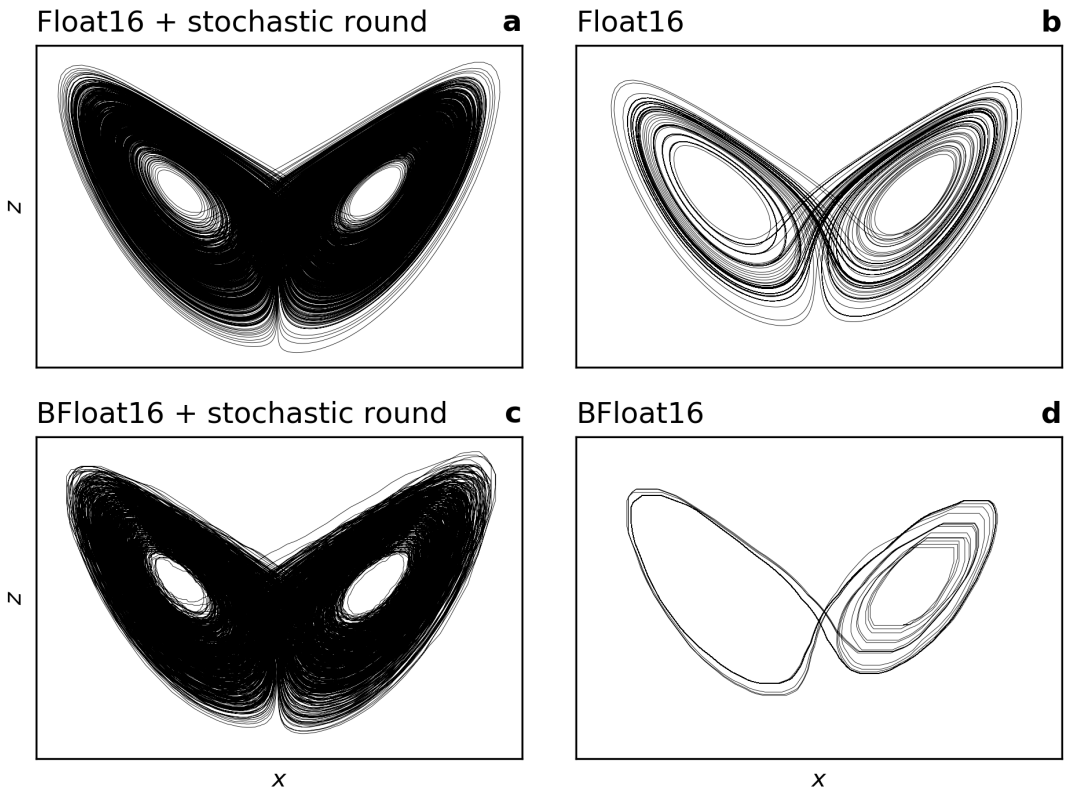


Figure 3.8: Lorenz attractor simulated with stochastic rounding. Deterministic round-to-nearest reduces the complexity of the Lorenz attractor for (b) Float16 and (d) BFloat16. With stochastic rounding for (a) Float16 and (c) BFloat16, however, the complexity of the attractor is considerably increased.

the precision of BFloat16 is considerably lower than for Float16, the introduced errors are also larger, which can be seen as the trajectories do not represent smooth lines, but are clearly perturbed, especially for higher z-values (where the absolute rounding errors are larger). Hence, the stochastic rounding error for Float16 is much lower, such that trajectories are smooth and do not show a visible sign of stochastic perturbations (Fig. 3.8a), and represent well the attractor simulated at high precision [Klöwer *et al.*, 2019]. Technically, this attractor is also subject to repetition as the underlying random number generator (Xoroshiro128+, Blackman & Vigna [2019]) has a finite period of 2^{128}, which is practically of negligible importance.

Motivated by the very promising effect that stochastic rounding has on simulations of the Lorenz system with 16-bit arithmetics, we implement stochastic rounding in the

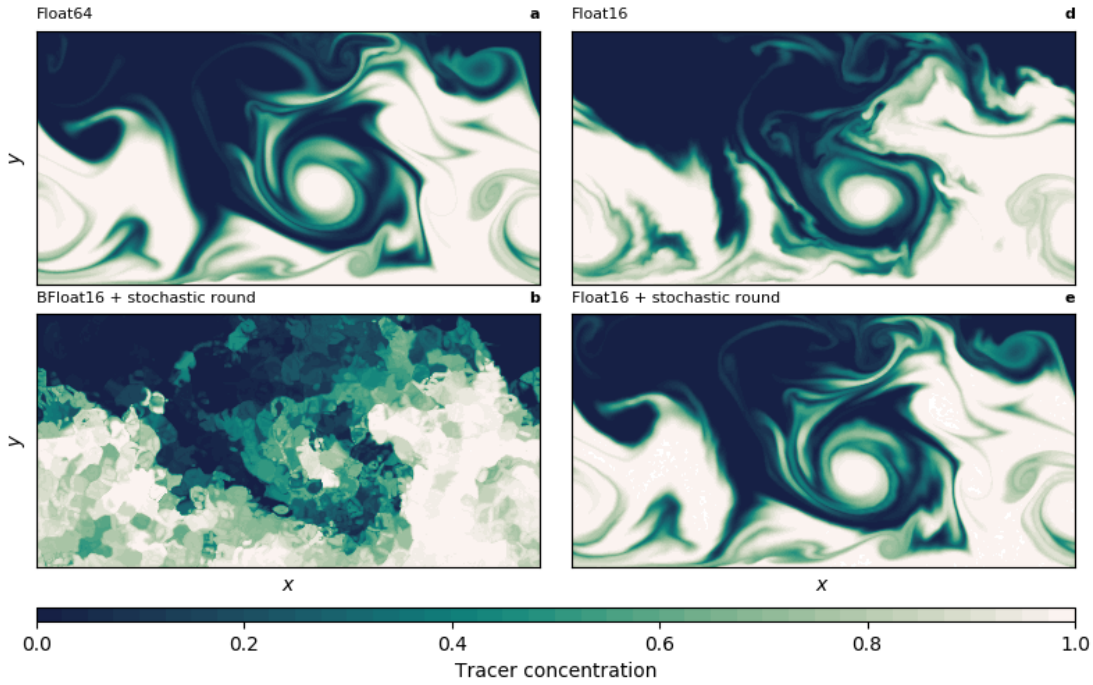


Figure 3.9: Snapshot of tracer concentration simulated by the shallow water model with stochastic rounding. The tracer was injected uniformly in the lower half of the domain 25 days before. This simulation was run at a resolution of $\Delta = 5$km (400x200 grid points).

shallow water model used previously to investigate stochastic rounding in systems that are subject to spatial discretization and stability constraints. Repeating the simulations presented in Fig. 3.1 and 3.7 with stochastic rounding for BFloat16, which was previously found to not allow time development, shows now, despite unrealistic dynamics, a vague resemblance with the reference simulation (Fig. 3.9b). Stochastic rounding for Float16 has a promising effect: Small scale instabilities, present for round to nearest, are largely absent and the tracer advection well resembles the reference simulation with Float64. We therefore conclude that stochastic rounding has the potential to be superior to round-to-nearest in the applications presented here. Further analysis is needed to assess its presumably very positive effect on low-precision simulations.

4 Sonums

A new number format, called Self-Organizing NUMbers (sonums), is developed. Motivation, design and implementations are discussed in the following.

4.1 Numbers that learn from data

The design of floating-point numbers in the 1970s and 1980s was motivated to meet several criteria: (i) hardware-friendly, i.e. the number format was designed to map easily from existing arithmetic circuits into bits; (ii) multi-purpose, i.e. initially declared as format for *scientific computations* it was supposed to be able to represent very large numbers $O(10^{-100})$ to $O(10^{-300})$ as well as very tiny numbers $O(10^{-100})$ to $O(10^{-300})$ with the same precision, to allow the use in many different fields of science (iii) error analysis-friendly, i.e. especially Float64 was designed to allow for very precise calculations, such that most scientists would not need to perform a numerical error analysis.

In the following we will relax these criteria and seek to find a number format that has the lowest rounding errors for a given application. We thereby ignore any hardware limitations, and create this number format purely on the basis of a software emulator. In fact, we end up designing it based on look-up tables. Arithmetic operations are with look-up tables not functions that calculate the result based on the inputs, but return the result from an array where the inputs determine the indices. This is in general only feasible for a small set of different inputs, as the underlying arrays have to be stored in memory. Look-up tables for 16-bit arithmetic, as considered here, are of a size of several GB which is unfeasible for current computing hardware. Look-up tables therefore are stored in RAM whereas a storage in much smaller low-level caches is necessary for speed. Every additional bit quadruples the size, such that look-up tables are only attractive for very low precision number formats or can be used for software emulators for up to 16 bit. We also do not aim to create a multi-purpose number format, but conversely a number format that is flexible enough to accommodate the precision requirements of a given algorithm as good as possible.

Motivated by the decimal precision analysed in the previous sections for floats and posits, the new number format is supposed to represent numbers in a given application with most precision for the numbers that occur most frequently and with no bitpatterns for numbers that never occur. Consequently, this number format is supposed have bitpatterns that occur equally frequent. This concept aligns with maximising entropy,

which will be discussed in the next section. In analogy to the unsupervised learning of self-organizing maps, which are mostly used in two or more dimensions, we call the new number format self-organizing numbers, or *sonums* in short. Note that the self-organization is here carried out on the real number axis, i.e. in one dimension.

We will make use of ideas introduced by the posit framework as introduced in section 2.3 as most redundant bitpatterns that occur in floats (± 0,$\pm\infty$, and a very large share of bitpatterns for NaNs, see Table 2.1) are removed for posits and only 2 bitpatterns for zero and NaR are retained as exceptions. The sonum circle is therefore designed in analogy to the posit circle (Fig. 2.1), but will be populated differently except for zero and NaR, which are mapped to identical bitpatterns for both formats. As illustrated in Fig. 4.1 sonums retain the symmetry with respect to zero, such that there is a reversible map (the two's complement, Choo *et al.* [2003]) between a number and its negation. However, sonums do not have a symmetry with respect to the multiplicative inverse as posits or floats have (note that for posits or floats this symmetry is only perfect when the significand is zero, otherwise rounding is applied, and excluding subnormal numbers). In the illustrated sonum circle it is therefore the idea to keep the real number value for s_1 to s_7 flexible and subject to training. In fact, sonums can be trained to replicate exactly the behaviour of posits with the same number of bits, but for any number of exponent bits. An n-bit sonum format has $m = 2^{n-1} - 1$ real number values that have to be defined. For 4-bit sonums $m = 7$, for 8-bit $m = 127$ and for 16-bit $m = 32767$. The size of the look-up tables scales with m^2 and is therefore quartic with the number of bits. Making use of commutativity for addition and multiplication as well as anti-commutativity for subtraction, the size reduces by a factor of two for those operations. The required size is therefore about 8KB per table for 8 bit and about 1GB per table for 16 bit. Division tables are twice that size. Sonums bear some similarity with type II unums, the predecessor of posits [Gustafson & Yonemoto, 2017].

In the following we will describe the maximum entropy training (also called quantile quantization) for sonums and present ways to train sonums to minimize the decimal rounding error.

4.2 Maximum entropy training

Given a data set D, regarded as j-element array of real numbers or a high precision approximation, we wish to find the sonum values s_i for $i \in 1, ..., 2^{n-1} - 1$ to maximise the entropy for an n-bit sonum format when representing the numbers in D. The

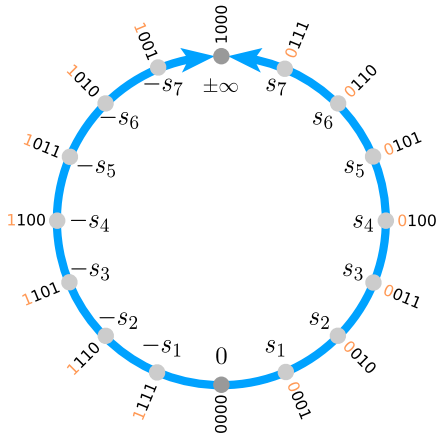


Figure 4.1: The 4bit sonum circle. Two bitpatterns are predefined: zero and NaR (Not-a-Real, or complex infinity), the remaining bitpatterns s_1 to s_7 can be user-defined and are usually subject to training based on provided data. Sonums are always symmetric with respect to zero.

information entropy H (or Shannon entropy, MacKay [2003]) is defined as

$$H = -\sum_i p_i \log_2(p_i) \tag{4.1}$$

where i is one possible state (here: bitpattern) with probability p_i, such that $\sum_i p_i = 1$ (Note that we define $p_i \log_2(p_i) = 0$ for $p_i = 0$). As we use the logarithm with base 2, the information entropy has units of bits. For a uniformly distributed probability, i.e. $p_i = \frac{1}{m}$ with m possible states the entropy is maximised to $n = \log_2(m)$ bits. In other words, the entropy is maximised when all states are equally likely and is zero for a discrete Dirac delta distribution.

We apply the concept of information entropy to the encoding of the standard uniform distribution $U(0, 1)$ between 0 and 1 with Float16, as an example (Fig. 5.1). Analysing the bitpattern histogram, we observe no bitpatterns in Float16 occur that are associated with negative numbers or numbers larger than 1. Converting the frequency of occurrence of every bitpattern into a probability p_i, we calculate the entropy as 12 bit of theoretically 16 bit that are available in Float16. This can be roughly interpreted as follows: The sign bit is unused as only positive numbers occur. One bit is redundant as only values in $(0, 1)$ occur and none in $(1, \infty)$. Another two bits are unused due to the uneven bitpattern distribution between $(0, 1)$.

Maximising the entropy for the standard uniform distribution $U(0, 1)$ with sonums

means that the values $s_i, i \in \{1, ..., m\}$ will be associated with numbers that are equidistantly distributed between 0 and 1. In theory therefore, $s_i = \frac{i}{m}$, which corresponds to the fixed-point numbers with a range from 0 to 1. In practice, one bitpattern is reserved for 0 and one for NaR, such that the entropy is not perfectly maximised. Furthermore, due to the symmetry with respect to zero, sonums have only 15 bit entropy as half the bitpatterns are reserved for all $-s_i$, which are not actually used. This poses only an issue in this artificial example, as many applications produce numbers that are symmetric with zero.

The generalization to arbitrary distributions, i.e. for any data set D, is therefore proposed as follows. In short, the array D is first sorted then split into m chunks of equal size. For each chunk the midpoint is found which is identified as the corresponding value for s_i. This can be written as an algorithm as shown in Fig. 4.2. We use the arithmetic average between the minimum and maximum value (i.e. midpoint) in each chunk to satisfy the round-to-nearest rounding mode.

```
function maxentropy_classification(m::Int,data::AbstractArray)

    N = length(data)
    n = N ÷ m       # integer division: number of data points per chunk

    # throw away random data for equally sized chunks of data
    data = shuffle(data[:])[1:n*m]
    sort!(data)

    # reshape data into a matrix, each chunk one column
    datam = reshape(view(data,:),(n,m))

    # array of sonum values
    s = Array{Float64,1}(undef,m)

    for i in 1:m
        # midpoint: arithmetic mean of min and max within chunk
        s[i] = (datam[1,i] + datam[end,i])/2
    end

    return s
end
```

Figure 4.2: A maximum entropy classification algorithm to train sonums.

Once sonums are trained (i.e. the values s_i are set) the decimal precision can be calculated. An example is given in Fig. 4.3, which shows how decimal precision of sonums follow the distribution of data from the Lorenz 1996 model, which will be introduced and discussed in section 4.4. After training the look-up tables have to be filled, which means that every arithmetic operation between all possible unique pairs of sonums

is precomputed. This is for 8-bit sonums (Sonum8) fast, and even for 16-bit sonums (Sonum16) completed within a few minutes. Subsequently, sonums can be used as a number format like floats and posits, however, sonums will presumably only yield reliable results for the application they were trained on. We will investigate in section 4.4 how sonums compare to floats and posits in the Lorenz 1996 system.

4.3 Minimising the decimal error

In the previous section we discussed a maximum entropy approach for training sonums, however, there are other training approaches possible that we want to investigate. Given a data set $D_j, j \in \{1, ..., N\}$ of length N, and a maximum entropy-trained set of sonum values $s_i, i \in \{1, ..., m\}$ we may want to know whether the s_i actually minimize the average rounding error ARE

$$ARE = \frac{1}{N} \sum_{j}^{N} |D_j - \mathrm{round}_s(D_j)| \tag{4.2}$$

with

$$\mathrm{round}_s(x) = \arg\min_{s_i \in s} |x - s_i| \tag{4.3}$$

being the round-to-nearest rounding function for a given set of sonums s. Alternatively, one could require the average decimal error ADE to be minimized

$$ADE = \frac{1}{N} \sum_{j}^{N} |\log_{10}(\frac{D_j}{\mathrm{round}_s(D_j)})| \tag{4.4}$$

which is equivalent to the (linear) average rounding error ARE when the logarithm with base 10 is applied to D_j beforehand and the rounding function is changed accordingly. Based on the framework around decimal precision presented in the previous section one may argue that it is more important to minimize ADE than ARE, but further analysis is needed to assess this with respect to a statistic like forecast error.

How to find s given D to minimize either ARE or ADE? We are therefore seeking a one-dimensional classification method that sorts all values in D into classes s_i. A classification is therefore a clustering and the two terms can be used interchangeably. Using the Jenks Natural Breaks Classification [Jenks & Caspall, 1971] is proposed and presented in the following in a modified version, that was found to be better suited in first tests for our applications. The Jenks classification is usually applied on multi-

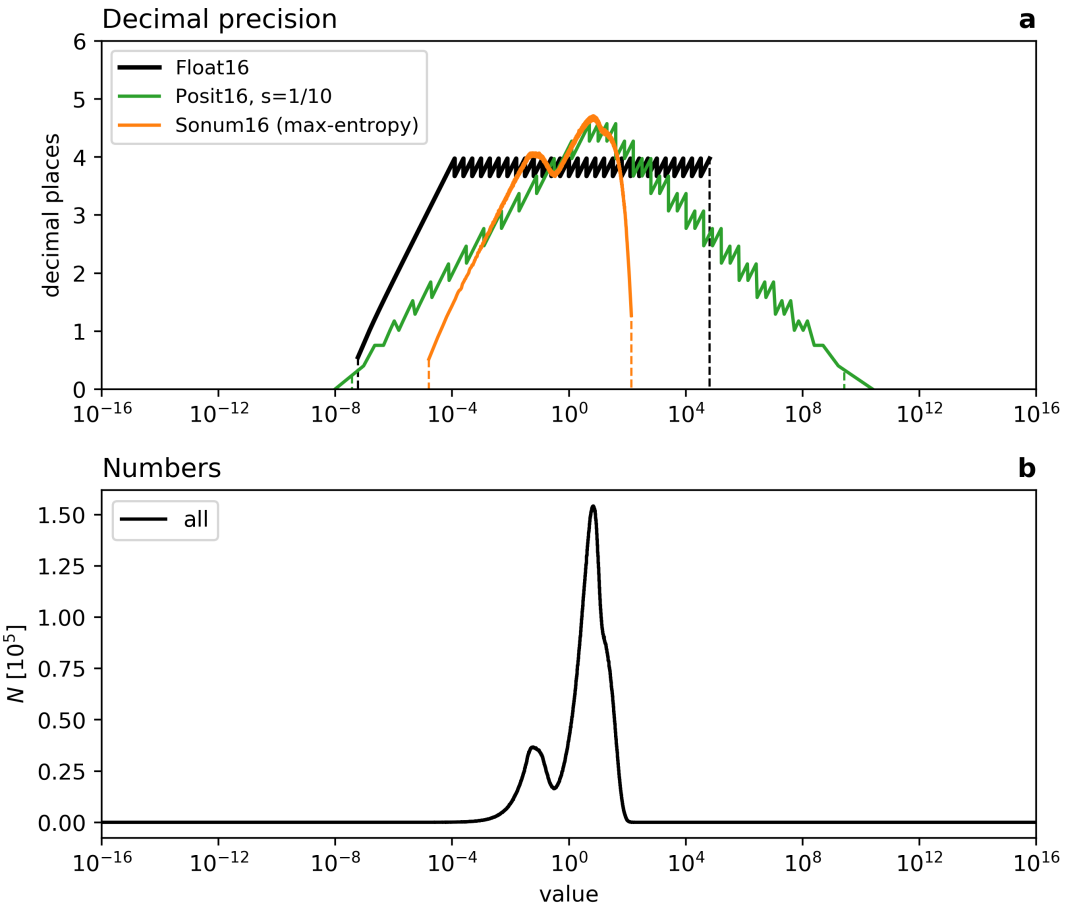


Figure 4.3: Decimal precision of sonums trained on data from the Lorenz 1996 model via maximum entropy. (a) Decimal precision of Sonum16 in comparison to Float16 and Posit(16,1). Note that the decimal precision distribution is shifted by one decade to the right to account for the scaling $s = 10^{-1}$ used. (b) Histogram of the numbers that occur in Lorenz 1996 that were used for training.

modal distributions with a few classes. Here, we are attempting to find up to 32767 (for 16-bit sonums) classes from millions of data points or more, which complicates the convergence of this iterative algorithm. The original Jenks algorithm is a method to minimize in-class variance while maximizing the variance between classes.

The modified Jenks classification algorithm is presented in a simplified version.

(0) Convert all D_j to their absolute value $|D_j|$.

(1) Define m (arbitrary) initial classes, each with an upper break point b_i, such that all D_j within the previous break point b_{i-1} and the i-th break point b_i belong to class i. The m-th class break point is the maximum in D_j. We choose the maximum entropy method of the previous section as a initial classification.

Then loop over

(2) For each class i, calculate an (unnormalized) error norm E_i of values in that class with respect to the class midpoint. The error can be the total rounding error or the total decimal error for example.

(3) Calculate the sum of the error norms of all classes $\sum_i E_i$, which is important to assess convergence. Dividing by N yields the average rounding or decimal error, depending on which error norm was used.

(4) For each class i except the last one, compare the error E_i to the next class error E_{i+1}.

(4.1) If $E_i < E_{i+1}$: Increase b_i by r, which will be defined shortly. That means, shift the break point to the right on the real axis to make the i-th class bigger and the $(i+1)$-th class smaller. This will increase E_i and decrease E_{i+1}.

(4.2) Else: Decrease b_i by r.

Choosing an appropriate value for r, which is a flux of data point from one class to a neighbouring class, is difficult. We found that r should scale with the size of the donating class, such that a certain share of points should be passed on. Additionally, we decrease the flux r if the previous flux direction was opposite (4.2 was evaluated instead of 4.1 or vice versa), which is helpful to aid convergence. However, we increase the flux r if the previous flux direction was the same, which accelerates convergence.

In the next section we will test sonums against floats and posits. At the moment we will restrict this analysis to sonums which were trained with the maximum entropy classification. Using sonums with minimised rounding error or decimal error is subject to further analysis to satisfy convergence.

4.4 Sonums in Lorenz 1996

Sonum16 will be tested against floats and posits in the one-level Lorenz 1996 model Hatfield *et al.* [2018]; Lorenz & Emanuel [1998], which is a simple chaotic weather model, described by the following equations

$$\frac{dX_k}{dt} = X_{k-1}(X_{k+1} - X_{k-2}) - X_k + F \tag{4.5}$$

with $k \in \{1, 2, ..., 36\}$. Periodic boundary conditions are applied, such that $X_0 = X_{36}$ and $X_{37} = X_1$. The first term on the right-hand side represents advection and the second is a relaxation term. The forcing is set to $F = 8$ and independent of space and time. Although the Lorenz 1996 model can be extended to a two or three-level version, such that levels can be interpreted as large, medium and small-scale variables, the model used here is the simple one-level version. The model is spun-up from rest $X_k = 0$ with a small perturbation in one variable. Scaling can be applied by multiplication with a constant s, such that $\hat{X}_k = sX_k$

$$\frac{d\hat{X}_k}{dt} = s^{-1}\hat{X}_{k-1}(\hat{X}_{k+1} - \hat{X}_{k-2}) - \hat{X}_k + F \tag{4.6}$$

which controls the range of numbers, occuring in the simulation. As similarly suggested for the Lorenz 63 model [Klöwer *et al.*, 2019], we use $s = 10^{-1}$ (which is precomputed) for the simulation of Eq. 4.6 with posits to center the arithmetic operations around 1. This is beneficial for posit arithmetic as otherwise the prognostic variables X_k are $O(10)$.

A Hovmoeller diagram illustrates the chaotic dynamics simulated by the Lorenz 1996 model (Fig. 4.4). The initial perturbation in X_1 is advected throughout the domain within the first time steps. After this first wake, the model's state becomes chaotic. Posit(16,1) represents well the dynamic, as shown in the comparison with Float64 for reference.

To quantify the forecast error, we run a set of 1000 forecasts per number format, starting from a random time step of a long control simulation. The simulation with Float64 is taken as reference truth. Float16 has an exponential error growth that starts much earlier than the error growth for Posit(16,1) (Fig. 4.5). Both formats have on average an identical error growth rate. Posits clearly cause a reduced rounding error compared to floats at all lead times, making posits a better suited number format for the simulation of the Lorenz 1996 model.

We now use the long control simulation with Float64 to produce a dataset that contains all prognostic variables as well as the arithmetic results of all intermediate

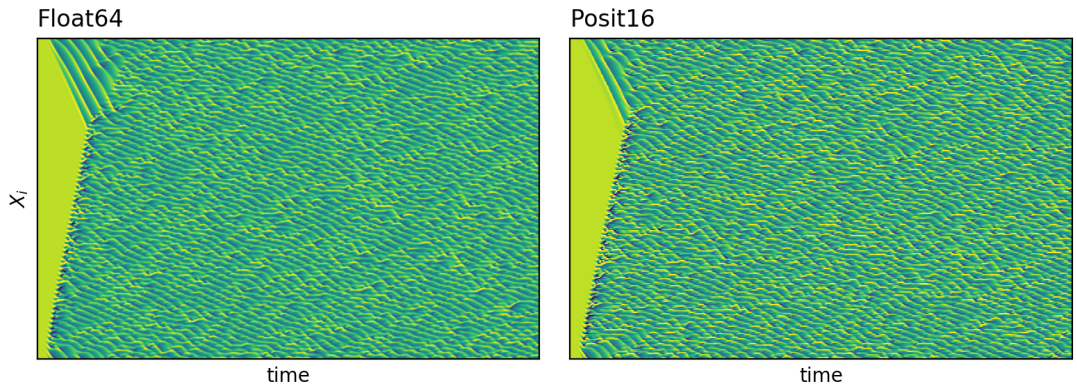


Figure 4.4: Solution of the Lorenz 1996 model presented as Hovmoeller diagram. (a) Float64 arithmetic, (b) Posit(16,1) arithmetic.

terms calculated for the tendencies. 16-bit sonums are trained with this dataset, such that they can self-organize around the numbers that occur most frequently within the Lorenz 1996 model (Fig. 4.3b). Consequently, Sonum16 has a slightly higher decimal precision compared to posits for the mode of the data distribution. A second mode is created for the tendencies, for which numbers of the order of 10^{-1} frequently occur. The decimal precision of Sonum16 drastically drops beyond the largest numbers $O(100)$ in the Lorenz 1996 model, as no bitpatterns have to be used to encode these real numbers.

After training, the sonum circle (Fig. 4.1) is defined. All arithmetic operations are precomputed creating look-up tables for multiplication, addition and subtraction. No look-up table is created for division as the Lorenz 1996 equations (Eq. 4.6) are written division-free (s^{-1} is precomputed). We can now quantify the forecast error as for floats and posits, running a set of 1000 forecasts from the same initial conditions as used for floats and posits.

Sonum16 has a smaller forecast error compared to posits for the important lead times where the normalised RMSE exceeds 1% (Fig. 4.5). Interestingly, although the error growth is much faster for the first time steps, it levels off afterwards and approaches the same error growth rate as for floats and posits once the normalised RMSE exceeds about 1%. This points towards a higher potential of Sonum16, when the cause of the initial rapid error growth is understood and circumvented with adjustments in the training method. As discussed in the previous section, sonums can be trained to minimize the average decimal rounding error, an aspect that requires further analysis to understand the optimal distribution of decimal precision for a given application. Nevertheless,

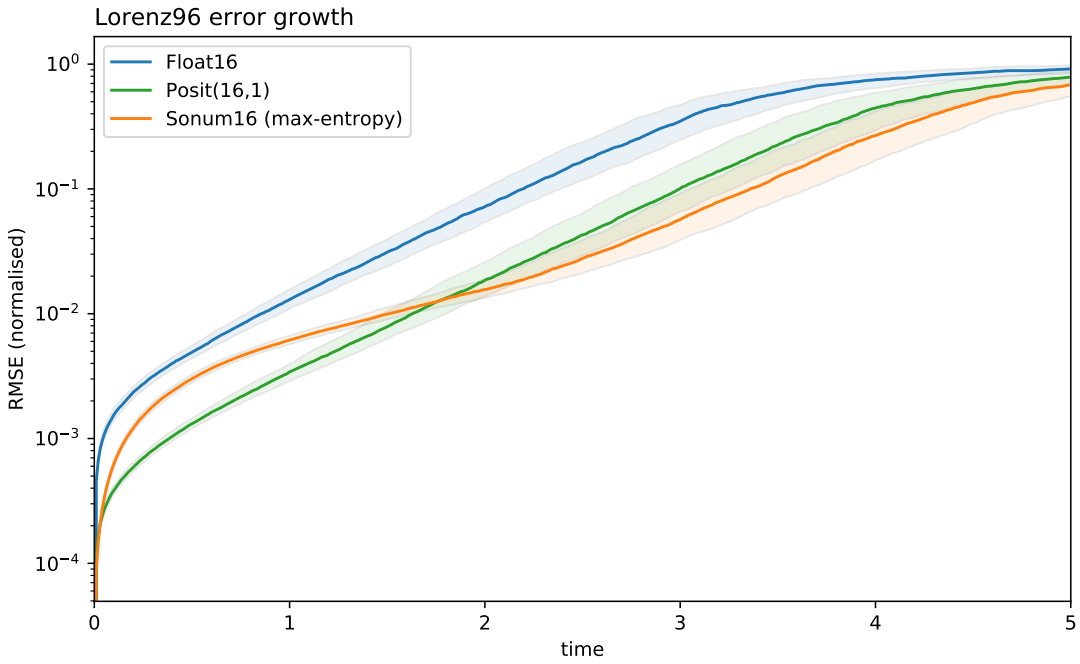


Figure 4.5: Error growth in the Lorenz 1996 system as simulated with Float16, Posit(16,1) or Sonum16. The error has been normalised by the climatological forecast error. Shaded areas denote the interquartile range of 1000 forecasts with the respective format.

sonums already provide perspectives towards an optimal number format for a given application. The main characteristics presumably are: (i) high precision for the most frequently occurring numbers, (ii) a tapered precision towards the smallest numbers and (iii) no redundant bitpatterns for very large and very small numbers that do not occur. Posits fulfill these criteria better than floats, which is likely the reason why they outperform floats in the applications presented here.

5 Sherlogs

This chapter will introduce the analysis number format Sherlogs, which can be used to estimate the algorithmic information entropy and to identify bottlenecks in algorithms that prevent them to be executed successfully with 16-bit arithmetic.

5.1 Logging arithmetic results

A central challenge in the development of the shallow water model used in section 3 was to identify subroutines that are error-prone and cause a degradation of the simulation when executed with 16-bit numbers. For simple models, such as the Lorenz systems, this is a comparably easy task, due to the low algorithmic complexity. For the shallow water model, knowledge about the simulated physics, the constants involved and physical scaling arguments were required to identify problematic parts of the model. Therefore, a try-and-error approach was successful, that does not scale well to models of higher complexity and the required time to understand an algorithm from the perspective of rounding errors increases rapidly.

To overcome this issue, we developed an analysis number format that helps to identify error-prone subroutines in models of high complexity by logging the arithmetic results that occur in every arithmetic operation performed. This number format is called Sherlogs and is considered to be an analysis format, as it behaves numerically like default floating-point numbers (either Float32 or Float64), but executes an additional logging function on every call to one of the arithmetic functions which is used to log the arithmetic result in a histogram. Within the type-flexibility of the Julia language framework, we can execute any type-stable function with Sherlogs as number format without the need to rewrite or adjust this algorithm in any way. Sherlogs come in various versions: Sherlog64 behaves like Float64, Sherlog32 behaves like Float32 and Sherlog16 behaves like any 16-bit number format that can be passed on as a parameter. Every Sherlog format takes a 16-bit number format as a parameter that defines the bins of the bitpattern histogram.

The resulting bitpattern histograms that are produced by Sherlogs will show whether certain bitpatterns occur very frequently in an algorithm or whether some bitpatterns do not occur at all. Highly frequent bitpatterns are likely associated with rounding errors, as a range of values was probably rounded to one representable number, but can also result from the repeating computation of constant values. An algorithm that

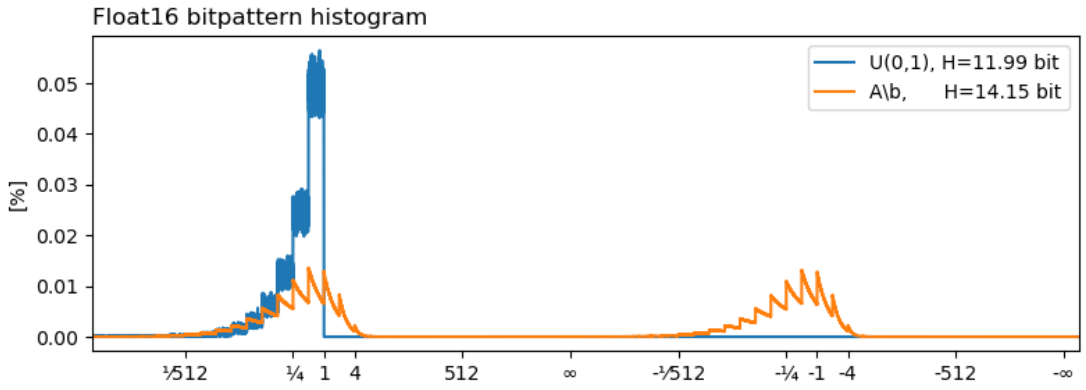


Figure 5.1: Algorithmic information entropy of the LU-decomposition. Float16 is used as number format for binning. The entries in the input arrays A and b are uniformly distributed in $(0, 1)$ (blue), whereas the numbers in the LU-decomposition are shown in orange.

produces a comparably smooth bitpattern histogram can likely be executed successfully with low precision arithmetics, as its algorithmic information entropy is higher than for multi-model bitpattern distributions, as discussed in the next section.

Similar to Sherlogs, we developed another number format *DrWatson*, which logs the stacktrace when a provided condition is fulfilled. This condition can be passed on to DrWatson as parameter and is usually a comparison like *x greater than c*, to identify parts of a function where very large numbers occur. Using DrWatson as a number format, therefore can identify where an arithmetic result that is of a certain size occurs. Consequently, DrWatson provides the information in which line of the code a certain arithmetic result occurs, which facilitates the debugging with respect to rounding errors.

5.2 Algorithmic information entropy

Bitpattern histograms and algorithmic information entropy is introduced with the example of the solution of a linear equation system, which is performed with a standard LU-decomposition for matrices. We create a matrix A and a vector b, both with random uniformly distributed entries in $(0, 1)$ and solve $Ax = b$ for x. Without detailed knowledge of the LU-decomposition algorithm, we can convert the entries in A and b to Sherlogs and execute Julia's LU-decomposition.

The retrieved bitpattern (Fig. 5.1) based on bins corresponding to representable

numbers with Float16 reveals that although the inputs A and b have only positive entries, the algorithm requires both positive as well as negative numbers similarly. Furthermore, the largest numbers (in absolute terms) produced are around 4, with very rarely numbers that are larger. As we do not observe any spikes, or number for the bitpatterns associated with $\pm\infty$, we conclude that this algorithm can likely be executed with Float16 without major degradation in the accuracy of the result, which can be verified.

Given the bitpattern histogram (which depends on the inputs A and b as well as the number format used for logging) we can calculate the algorithmic information entropy for the LU-decomposition which is estimated to be 14.15 bit, relative to the maximum entropy of 16 bit. Training sonums with the maximum entropy method on this bitpattern histogram will taper the precision for numbers of $O(10)$ and larger and will allocate more precision for numbers around 1, which matches the general idea of the decimal precision of posits. We would therefore argue that posits are also a more suitable number format for the LU-decomposition presented here.

Following this methodology, we can estimate the algorithmic information entropy of various algorithms, including chaotic models of atmospheric and oceanic circulation. Future analysis will investigate this for the shallow water model presented in section 3, which will provide a more systematic approach to the rounding error mitigation methods already discussed in Klöwer *et al.* [2019].

6 Conclusions

Computational speed is a limitation of current weather and climate models towards more reliable weather forecasts and climate predictions. Recent progress on specialised computing hardware was shown to yield significant performance increase on graphic and tensor processing units, especially for low precision number formats that use 16 bit to encode real numbers. This study offers perspectives for weather and climate models computed in 16-bit arithmetic to benefit from hardware acceleration that will be available on future computing architecture.

Using a software emulator we have tested various different 16-bit number formats for the computation of weather and climate models of low and medium complexity. The investigate data types are floating-point numbers (Float16 and BFloat16), posit numbers (Posit(16,1) and Posit(16,2)), logarithmic fixed-point numbers (LogFixPoint16 and Approx14), as well as a newly developed self-organizing number format called Sonum16. Among those number formats we identified posits as the best general-purpose format for a minimal rounding error in our applications. Once trained on a specific application, sonums were shown to outperform all existing number formats. Due to the self-organization of sonums (which can be done to maximise entropy or to minimize rounding errors) we identified the key requirements for number formats to reduce rounding errors: A high decimal precision for the most frequent numbers (often the prognostic variables); a tapered precision towards zero and towards the largest occurring numbers; and as few redundant bitpatterns as possible.

Every number format comes with a rounding mode for (at least some of) the arithmetic operations. Default rounding mode for floats and posits is round-to-nearest, but we have tested two additional modes in this study: Round-to-nearest in log2-space and stochastic exact-in-expectation rounding. While the former was shown to be biased, resulting in violation of conservation laws, the latter was found to be a better suited rounding mode for our applications. Stochastic rounding significantly reduced the rounding error and overcame issues that led to a stalling of the model dynamics with round-to-nearest. Although stochastic rounding was emulated in software, the results promote stochastic rounding in hardware as a way to make existing 16-bit number formats more suitable for weather and climate models. Including stochasticity in hardware could be an efficient way to represent uncertainty in ensemble forecasts in addition to or instead of the existing stochastic parameterizations of unresolved processes.

This study suggests that replacing Float32 or Float64 with 16-bit arithmetic for all

calculations in weather and climate models will likely fail, as mitigation methods need to be applied, possibly on various levels to make algorithms resilient against increased rounding errors with low precision numbers. One mixed-precision approach is proposed, which uses Float32 for the prognostic variables and various 16-bit formats to calculate the tendencies. This circumvents the issue that arises when the increments of the prognostic variables with small tendencies get round back. Mixed-precision is a promising approach as it can be implemented on present-day hardware and does not require future processors. In case the communication between processors is a performance limitation we identified 16 and even 8-bit encodings for the communicated variables to be largely sufficient and the introduced error negligible compared to other sources of error. For a complex weather or climate model distributed on large supercomputers this could drastically reduce the bandwidth of communication.

Additionally to the algorithmic changes proposed by Klöwer *et al.* [2019] for rescaling, re-ordering and precomputing these mitigations methods make it more realistic to run the first complex weather and climate models in 16-bit arithmetics in the near future. To systematically identify algorithmic changes that are needed to reduce rounding errors with low precision numbers, we developed the analysis number format sherlogs, that can be used to create bitpattern histograms of arbitrarily complex algorithms, estimate the algorithmic information entropy and identify lines of code that may cause serious rounding errors. With this tool, we hope to aid the transition of existing complex models towards 16-bit arithmetic.

Although this study provided evidence that the transition towards 16-bit arithmetic for weather and climate models is a difficult challenge, we presented several approaches that have the potential to make this transition a success towards more reliable weather forecasts and climate predictions.

7 Thesis outline

7.1 Outline

A thesis outline is presented. Only missing contributions to complete the prosed contents of the thesis are discussed.

1 Information entropy and error norms

1.1 Information entropy

This section will be extended to a general introduction to information theory.

1.2 Norms for rounding errors

This section will extend the brief discussion about rounding errors presented in this report to a more complete section on rounding error norms in general.

1.3 Binary and decimal precision

2 Number formats and rounding modes

2.1 Integer and fixed-point numbers

2.2 Floating-point numbers

2.3 Posit numbers

2.4 Logarithmic fixed-point numbers

This section will be extended with an unbiased rounding mode for logarithmic fixed-point numbers.

This section will include an analysis on the conservation of the analytically conserved quantities in the shallow water system (mass and tracer volume), which is important to evaluate the stochastic rounding and an unbiased rounding mode for logarithmic fixed-point numbers.

This section will be completed with an optimized version of the Jenks classification with verifications on data sets.

This section will be completed with the application of sonums in the shallow water model. For this a training based on bitpattern histograms needs to be developed and tests will be performed to assess sonums trained to maximise entropy in comparison to minimise the rounding-error.

5 Precision-resilient algorithms

5.1 Rescaling equations

5.2 A semi-Lagrangian advection scheme for 16-bit

5.3 Systematically identifying rounding errors: Sherlogs

Examples will be provided for the shallow water model how sherlogs can be applied in more complex models.

5.4 Algorithmic information entropy

The algorithmic information entropy of different models (Lorenz systems and shallow water) will be investigated and presented in this section.

6 ? Data compression

This is a potential chapter based on a proposal on climate data compression for the ECMWF Summer of Weather Code. This chapter depends on the acceptance of the proposal and contents will be based on project outcomes.

7 ? Oceananigans

This is a potential chapter based on a planned research visit to MIT in Summer 2020, which was postponed from Spring 2020 due to Covid-19. Oceananigans is an ocean model that is developed by MIT within the Clima project. This chapter anticipates to document a low precision version of Oceananigans, which is currently the most complex model in climate science written in Julia. Currently planned is to use Oceananigans that already runs on graphic processing units to investigate 32 or even 16-bit arithmetics in a more complex model to what was used in this study. As it is written in Julia, most of my developments can be directly used. This chapter depends on the outcomes of the work with Oceananigans.

7.2 Timeline

May 2020	Paper submission on weather and climate models in 16-bit arithmetic
May-Aug 2020	Data compression project (pending)
May-Aug 2020	Paper preparation and submission on sonums and sherlogs
Aug-Sep 2020	Research visit at MIT (planned)
Oct-Dec 2020	Paper preparation and submission on stochastic rounding
Jan-May 2021	Documentation and writing up of remaining projects
Jan-May 2021	Paper preparation on data compression (possibly)
Jun-Sep 2021	Thesis writing
Aug-Sep 2021	Thesis revision and submission

7.3 Transferable skills

Programming Developed many open-source software packages in the Julia language, including parallel and distributed computing approaches, MPI, and metaprogramming.

Version control Extensive use of git, including publishing of software packages frequently on GitHub and code maintenance.

Software development Acquired skills on high quality software development standards, including continuous integration, software licensing, testing and documentation

Presentation skills Created many presentations with animations, videos and interactive code evaluation via Jupyter notebooks. Professional poster creation with Inkscape.

Appendix

A.1 Open-source software developments

SoftPosit.jl

- Authors: M Kloewer, M Giordano, C Leong
- URL: github.com/milankl/SoftPosit.jl
- License: MIT
- Version: 0.3.0

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.

StochasticRounding.jl

- Authors: M Kloewer
- URL: github.com/milankl/StochasticRounding.jl
- License: MIT
- Version: 0.1.0

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. Xoroshio128Plus is used as a high-performance random number generator.

ShallowWaters.jl

- Authors: M Kloewer
- URL: github.com/milankl/ShallowWaters.jl
- License: MIT
- Version: 0.3.0

ShallowWaters.jl is a shallow water model with a focus on type-flexibility and 16bit number formats, which allows for integration of the shallow water equations with

arbitrary number formats as long as arithmetics and conversions are implemented. ShallowWaters also allows for mixed-precision and reduced precision communication.

ShallowWaters is fully-explicit with an energy and enstrophy conserving advection scheme and a Smagorinsky-like biharmonic diffusion operator. Tracer advection is implemented with a semi-Lagrangian advection scheme. Runge-Kutta 4th-order is used for pressure, advective and Coriolis terms and the continuity equation. Semi-implicit time stepping for diffusion and bottom friction. Boundary conditions are either periodic (only in x direction) or non-periodic super-slip, free-slip, partial-slip, or no-slip. Output via NetCDF.

Sherlogs.jl

- Authors: M Kloewer
- URL: github.com/milankl/Sherlogs.jl
- License: MIT
- Version: 0.1.0

Sherlogs.jl provides a number format Sherlog64 that behaves like Float64, but inspects your code by logging all arithmetic results into a 16bit bitpattern histogram during calculation. Sherlogs can be used to identify the largest or smallest number occuring in your functions, and where algorithmic bottlenecks are that limit the ability for your functions to run in low precision. A 32bit version is provided as Sherlog32, which behaves like Float32. A 16bit version is provided as Sherlog16T, which uses T for computations as well as for logging.

Sonums.jl

- Authors: M Kloewer
- URL: github.com/milankl/Sonums.jl
- License: MIT
- Version: 0.2.0

Sonums.jl is a software emulator for Sonums - the Self-Organizing NUMbers. A number format that learns from data. Sonum8 is the 8bit version, Sonum16 for 16bit computations. The package exports number types, conversions and arithmetics. Sonums conversions are based on binary tree search, and arithmetics are based on table lookups. Training can be done via maximum entropy or minimising the rounding error.

Float8s.jl

- Authors: M Kloewer, J Sarnoff
- URL: github.com/milankl/Float8s.jl
- License: MIT
- Version: 0.1.0

Float8s.jl is a software emulator for a 8bit floating-point format, 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.

LogFixPoint16s.jl

- Authors: M Kloewer
- URL: github.com/milankl/LogFixPoint16s.jl
- License: MIT
- Version: 0.1.0

LogFixPoint16s.jl is a software emulator for 16-bit logarithmic fixed-point numbers with 7 signed integer bits and 8 fraction bits. The package provides the `LogFixPoint16` number type, as well as arithmetic operations, conversions and additional functionality. The software emulator is based on either integer addition or look-up tables and is therefore a comparably fast emulator.

Lorenz96.jl

- Authors: M Kloewer
- URL: github.com/milankl/Lorenz96.jl
- License: MIT
- Version: 0.3.0

Lorenz96.jl is a type-flexible one-level Lorenz 1996 model, which supports any number type, as long as conversions to and from Float64 and arithmetics are defined. 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.

Lorenz63.jl

- Authors: M Kloewer
- URL: github.com/milankl/Lorenz63.jl
- License: MIT
- Version: 0.2.0

Lorenz63.jl is a type-flexible Lorenz 1963 model, which supports any number type, 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.

Jenks.jl

- Authors: M Kloewer
- URL: github.com/milankl/Jenks.jl
- License: MIT
- Version: 0.1.0

Jenks.jl is the Jenks Natural Breaks Optimization, a data clustering method to minimise in-class variance or L1 rounding error. Jenks provides a data classification algorithm that groups one dimensional data to minimize an in-class error norm from the class mean but maximizes the same error norm between different classes.

Acknowledgements

I am very grateful for the support and very fruitful discussions with my supervisors Tim Palmer and Peter Düben, and especially for the freedom to develop my own ideas.

I gratefully acknowledge funding from the European Research Council under grant number 741112 *An Information Theoretic Approach to Improving the Reliability of Weather and Climate Simulations* and from the UK National Environmental Researc Council (NERC) under grant number NE/L002612/1.

I would like to thank the whole Julia community for an uncountable effort to develop a very modern high performance computing language that is high-level, easy to learn and was proven to be increadibly useful for reduced precision simulations. I also would like to thank everybody who developed the matplotlib plotting library, which was used for every figure in this report.

References

Bezanson, J., Edelman, A., Karpinski, S. & Shah, V.B. (2017). Julia: A Fresh Approach to Numerical Computing. *SIAM Review*, **59**, 65–98. 9, 13

Blackman, D. & Vigna, S. (2019). Scrambled Linear Pseudorandom Number Generators. *arXiv:1805.01407 [cs]*. 28

Burgess, N., Milanovic, J., Stephens, N., Monachopoulos, K. & Mansell, D. (2019). Bfloat16 Processing for Neural Networks. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, 88–91. 2

Chaurasiya, R., Gustafson, J., Shrestha, R., Neudorfer, J., Nambiar, S., Niyogi, K., Merchant, F. & Leupers, R. (2018). Parameterized Posit Arithmetic Hardware Generator. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, 334–341, IEEE, Orlando, FL, USA. 2

Chen, J., Al-Ars, Z. & Hofstee, H.P. (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. 2, 6

Choo, H., Muhammad, K. & Roy, K. (2003). Two's complement computation sharing multiplier and its applications to high performance DFE. *IEEE Transactions on Signal Processing*, **51**, 458–469. 4, 7, 31

Dawson, A. & Düben, P.D. (2017). Rpe v5: An emulator for reduced floating-point precision in large numerical simulations. *Geoscientific Model Development*, **10**, 2221–2230. 2

Düben, P.D. (2018). A New Number Format for Ensemble Simulations. *Journal of Advances in Modeling Earth Systems*, **10**, 2983–2991. 1

Düben, P.D., McNamara, H. & Palmer, T. (2014). The use of imprecise processing to improve accuracy in weather & climate prediction. *Journal of Computational Physics*, **271**, 2–18. 1, 2

Glaser, F., Mach, S., Rahimi, A., Gürkaynak, F.K., Huang, Q. & Benini, L. (2017). An 826 MOPS, 210 uW/MHz Unum ALU in 65 nm. *arXiv:1712.01021 [cs]*. 2

Gupta, S., Agrawal, A., Gopalakrishnan, K. & Narayanan, P. (2015). Deep Learning with Limited Numerical Precision. *arXiv:1502.02551 [cs, stat]*. 2

GUSTAFSON, J. (2017). Posit Arithmetic. 2, 6, 7, 8, 9, 10

GUSTAFSON, J.L. & YONEMOTO, I. (2017). Beating Floating Point at its Own Game: Posit Arithmetic. *Supercomputing Frontiers and Innovations*, **4**, 16. 2, 6, 10, 31

HATFIELD, S., DÜBEN, P., CHANTRY, M., KONDO, K., MIYOSHI, T. & PALMER, T. (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. 37

HATFIELD, S., CHANTRY, M., DÜBEN, P. & PALMER, T. (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. 2

HATFIELD, S., MCRAE, A., PALMER, T. & DÜBEN, P. (2020). Single-Precision in the Tangent-Linear and Adjoint Models of Incremental 4D-Var. *Monthly Weather Review*, **148**, 1541–1552. 1

JEFFRESS, S., DÜBEN, P. & PALMER, T. (2017). Bitwise efficiency in chaotic models. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, **473**, 20170144. 1, 3

JENKS, G.F. & CASPALL, F.C. (1971). Error on Choroplethic Maps: Definition, Measurement, Reduction. *Annals of the Association of American Geographers*, **61**, 217–244, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8306.1971.tb00779.x. 34

JOUPPI, N.P., YOUNG, C., PATIL, N., PATTERSON, D., AGRAWAL, G., BAJWA, R., BATES, S., BHATIA, S., BODEN, N., BORCHERS, A., BOYLE, R., CANTIN, P.L., CHAO, C., CLARK, C., CORIELL, J., DALEY, M., DAU, M., DEAN, J., GELB, B., GHAEMMAGHAMI, T.V., GOTTIPATI, R., GULLAND, W., HAGMANN, R., HO, C.R., HOGBERG, D., HU, J., HUNDT, R., HURT, D., IBARZ, J., JAFFEY, A., JAWORSKI, A., KAPLAN, A., KHAITAN, H., KILLEBREW, D., KOCH, A., KUMAR, N., LACY, S., LAUDON, J., LAW, J., LE, D., LEARY, C., LIU, Z., LUCKE, K., LUNDIN, A., MACKEAN, G., MAGGIORE, A., MAHONY, M., MILLER, K., NAGARAJAN, R., NARAYANASWAMI, R., NI, R., NIX, K., NORRIE, T., OMERNICK, M., PENUKONDA, N., PHELPS, A., ROSS, J., ROSS, M., SALEK, A., SAMADIANI, E., SEVERN, C., SIZIKOV, G., SNELHAM, M., SOUTER, J., STEINBERG, D., SWING, A., TAN, M., THORSON, G., TIAN, B., TOMA, H., TUTTLE, E., VASUDEVAN, V., WALTER, R., WANG, W., WILCOX, E. & YOON, D.H. (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. 1, 2

JOUPPI, N.P., YOUNG, C., PATIL, N. & PATTERSON, D. (2018). A domain-specific architecture for deep neural networks. *Communications of the ACM*, **61**, 50–59. 2

KALAMKAR, D., MUDIGERE, D., MELLEMPUDI, N., DAS, D., BANERJEE, K., AVANCHA, S., VOOTURI, D.T., JAMMALAMADAKA, N., HUANG, J., YUEN, H., YANG, J., PARK, J., HEINECKE, A., GEORGANAS, E., SRINIVASAN, S., KUNDU, A., SMELYANSKIY, M., KAUL, B. & DUBEY, P. (2019). A Study of BFLOAT16 for Deep Learning Training. *arXiv:1905.12322 [cs, stat]*. 2

KLÖWER, M. & GIORDANO, M. (2019). SoftPosit.jl - A posit arithmetic emulator. Zenodo. 9

KLÖWER, M., DÜBEN, P.D. & PALMER, T.N. (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. 2, 4, 6, 15, 27, 28, 37, 42, 44

LANGROUDI, H.F., CARMICHAEL, Z. & KUDITHIPUDI, D. (2019). Deep Learning Training on the Edge with Low-Precision Posits. *arXiv:1907.13216 [cs, stat]*. 2

LEONG, S.H. (2020). SoftPosit. Zenodo. 9

LORENZ, E.N. & EMANUEL, K.A. (1998). Optimal Sites for Supplementary Weather Observations: Simulation with a Small Model. *Journal of the Atmospheric Sciences*, **55**, 399–414. 37

MACKAY, D. (2003). *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, 1st edn. 3, 32

MARKIDIS, S., CHIEN, S.W.D., LAURE, E., PENG, I.B. & VETTER, J.S. (2018). NVIDIA Tensor Core Programmability, Performance Precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 522–531. 2

PALMER, T. (2015). Modelling: Build imprecise supercomputers. *Nature News*, **526**, 32. 1

PALMER, T.N. (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. 1

PALMER, T.N. (2019). Stochastic weather and climate models. *Nature Reviews Physics*, **1**, 463–471. 2

RÜDISÜHLI, S., WALSER, A. & FUHRER, O. (2013). COSMO in single precision. *Cosmo Newsletter*, 5–1. 1

RUSSELL, F.P., DÜBEN, P.D., NIU, X., LUK, W. & PALMER, T. (2017). Exploiting the chaotic behaviour of atmospheric models with reconfigurable architectures. *Computer Physics Communications*, **221**, 160–173. 2, 4

SILVER, J.D. & ZENDER, C.S. (2017). The compression–error trade-off for large gridded data sets. *Geoscientific Model Development*, **10**, 413–423. 3

THORNES, T., DÜBEN, P. & PALMER, T. (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. 1

TINTÓ PRIMS, O., ACOSTA, M.C., MOORE, A.M., CASTRILLO, M., SERRADELL, K., CORTÉS, A. & DOBLAS-REYES, F.J. (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. 1

VAN DAM, L., PELTENBURG, J., AL-ARS, Z. & HOFSTEE, H.P. (2019). An Accelerator for Posit Arithmetic Targeting Posit Level 1 BLAS Routines and Pair-HMM. In *Proceedings of the Conference for Next Generation Arithmetic 2019*, CoNGA'19, 1–10, Association for Computing Machinery, Singapore, Singapore. 2

VÁŇA, F., DÜBEN, P., LANG, S., PALMER, T., LEUTBECHER, M., SALMOND, D. & CARVER, G. (2017). Single Precision in Weather Forecasting Models: An Evaluation with the IFS. *Monthly Weather Review*, **145**, 495–502. 1

ZENDER, C.S. (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. 3