Low-precision climate computing: Nobody needed those bits anyway

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 thesis submitted for the degree Doctor of Philosophy

Oxford, September 2021

DON’T PANIC

Abstract

Who needs those bits anyway?

Contents

1 Introduction

2 General methods

2.1 Binary number formats

Integers

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 [**?**]. For example in the 4-bit signed integer format (Int4), $1110_{\text{Int4}} = 1010\_ = -2$. 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

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 [**?**]. 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 [**?**]. However, any result beyond will lead to disastrous results, as integer overflow usually returns a negative value following a wrap-around behaviour.

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 floatin-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.

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

$$x = (-1)^{sign\ bit} \cdot 2^e \tag{2.5}$$

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).

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. **??**). 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 $\text{round}_{\log_2}(x)$ be the round-to-nearest function in log2-space and x be drawn N-times from a random uniform distribution $U(1,2)$, then

$$\frac{1}{N}\sum_i^N x_i = 1.5 \neq \frac{1}{N}\sum_i^N \text{round}_{\log_2}(x_i) = 3 - \sqrt{2} = 1.586... \tag{2.6}$$

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

Posit numbers

Posit numbers arise from a projection of the real axis onto a circle (Fig. **??**), 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. **??**b). Regimes are defined by $useed^{\pm 1}$, $useed^{\pm 2}$, $useed^{\pm 3}$, etc. To encode these regimes into bits, posit numbers extend floating-point arithmetic by introducing regime bits that are responsible for the dynamic range of representable numbers. Instead of having a fixed length, regime bits are defined as the sequence of identical bits after the sign bit, which are eventually terminated by an opposite bit. 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 [**???**] (negative posit numbers are converted first to their two's complement, see Eq. 2.9)

$$p = (-1)^{sign\ bit} \cdot useed^k \cdot 2^e \cdot (1+f) \tag{2.7}$$

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. 2.7. 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 [**??**]. 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. $useed = 4$)

$$57 \approx 01110111_{\text{Posit}(8,1)} = (-1)^0 \cdot 4^2 \cdot 2^1 \cdot (1 + 2^{-1} + 2^{-2}) = 56 \tag{2.8}$$

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 (**?**, denoted with an underscore subscript) by flipping all bits and adding 1,

$$\begin{aligned} -0.28 \approx 11011110_{\text{Posit}(8,1)} &= 10100010\_ \\ &= (-1)^1 \cdot 4^{-1} \cdot 2^0 \cdot (1 + 2^{-3}) = -0.28125. \end{aligned} \tag{2.9}$$

After the conversion to the two's complement, the bits are interpreted in the same way as in Eq. 2.8.

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 [**?**]. 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 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 **?**. In order to use posits on a conventional CPU we developed for the Julia programming language [**?**] the posit emulator *SoftPosit.jl* [**?**], which is a wrapper for the C-based library SoftPosit [**?**]. The type-flexible programming paradigm, facilitated by the Julia language, is outlined in **??**.

Self-organizing numbers

2.2 Rounding modes

Round to nearest

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. 2.6 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

$$\mathrm{round}_{\mathrm{nearest}}(x) = \begin{cases} x_1 & \text{if } x - x_1 < x_2 - x, \\ x_1 & \text{if } x - x_1 = x_2 - x \text{ and } x1 \text{ even}, \\ x_2 & \text{else}\,. \end{cases} \tag{2.10}$$

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

$$\mathrm{round}_{\mathrm{stoch}}(x) = \begin{cases} x_1 & \text{with probability} \quad 1 - \delta^{-1}(x - x_1) \\ x_2 & \text{with probability} \quad \delta^{-1}(x - x_1). \end{cases} \tag{2.11}$$

This behaviour is illustrated in Fig. **??**. 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

$$\lim_{N\to\infty} \frac{1}{N} \sum_{i}^{N} \mathrm{round}_{\mathrm{stoch}}(x) = x \tag{2.12}$$

as follows by inserting Eq. 2.11. 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. 2.11, 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. 2.11 can then be rewritten as

$$\mathrm{round}_{\mathrm{stoch}}(x) = \begin{cases} \mathrm{round}_{\mathrm{nearest}}(x + \frac{\delta}{2}(r - \frac{x - x_1}{\delta}) & \text{if } x_1 = 2^n \text{ and } x - x_1 < \frac{\delta}{4} \\ \mathrm{round}_{\mathrm{nearest}}(x + \delta(r - \frac{1}{2})) & \text{else} \,. \end{cases} \tag{2.13}$$

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.

Efficient bitwise implementations

2.3 Error norms

Mean, absolute and relative error

Decimal error and precision

The decimal precision is defined as [**??**]

$$\text{decimal precision} = -\log_{10} |\log_{10}(\frac{x_{\text{repr}}}{x_{\text{exact}}})| \tag{2.14}$$

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. **??** 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.

2.4 Type-flexibility through code composability

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 [**?**]. 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.

frame=lines, fontsize=, framesep=2mm]julia

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. **??**). 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 integer and conversion to float are defined for `T`.

frame=lines, fontsize=, framesep=2mm]julia

The harmonic sum converges after 513 elements when using Float16 (Fig. **??**). 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 **??**. Using Posit(16,1), the sum only converges after 1024 terms, due to the higher decimal precision of posits between 1 and 10.

Analysis number formats

2.5 Information theory

Entropy

Mutual information

Preserved information

3 Information-preserving compression

3.1 Introduction

3.2 Methods

Linear quantization

Logarithmic quantization

Lossless compression

Compressor performances

3.3 Drawbacks of current compression methods

3.4 Bitwise real information content

3.5 Compressing only the real information

3.6 Multi-dimensional climate data compression

3.7 A data compression Turing test

3.8 A roadmap for climate data compression

4 Periodic orbits in chaotic systems

Contributions This chapter is largely based on the following publication

M Klöwer, P Coveney, EA Paxton, and TN Palmer, 2021. *On periodic orbits in chaotic systems simulated at low precision*, submitted.

with the following author contributions. Conceptualisation: MK, PC, EAP. Data curation: MK. Formal Analysis: MK. Methodology: MK. Visualisation: MK. Writing – original draft: MK. Writing – review and editing: MK, PC, EAP, TNP. The contributions of Peter, Adam and Tim are highly appreciated.

4.1 Introduction

Once upon a time

4.2 Methods

Wasserstein distance

An improved random number generator for uniformly distributed floats

Monte Carlo orbit search

Efficient orbit search with distributed computing

4.3 Revisiting the generalised Bernoulli map

The special $\beta = 2$ case

Effects of stochastic rounding

4.4 Orbits in the Lorenz 1996 system

The Lorenz 1996 system

Longer orbits with more variables

More variables instead of higher precision

4.5 Discussion

5 A 16-bit shallow water model

Contributions This chapter is largely based on the following publications

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

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

with the following author contributions. Conceptualisation: MK, PDD. Data curation: MK. Formal Analysis: MK. Methodology: MK. Visualisation: MK. Writing – original draft: MK. Writing – review and editing: MK, PDD, TNP. The contributions of Peter and Tim are highly appreciated.

5.1 Introduction

... which was often debated as a step in the wrong direction.

5.2 Methods

The shallow water model

Scaling and reordering the shallow water equations

Mixed precision

Compensated time integration

Reduced precision communication for distributed computing

A 16-bit semi-Lagrangian advection scheme

5.3 Impact of low-precision on the physics

Error growth

Mean and variability

Geostrophy

Gravity waves

Mass and tracer conservation

5.4 Discussion

6 Running on 16-bit hardware

6.1 Introduction

6.2 Methods

Choosing a scale with Sherlogs

Identifying subnormals with DrWatson

6.3 Squeezing ShallowWaters.jl into Float16

6.4 Approaching 4x speedups on A64FX

6.5 Discussion

7 Conclusions

7.1 Summary

7.2 Discussion

7.3 Outlook

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 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.

Funding

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.