

Reduced-precision computing for Earth-system modelling

Sam Hatfield

samuel.hatfield@ecmwf.int

“More accuracy with less precision”

Received: 6 May 2021 | Revised: 4 September 2021 | Accepted: 28 September 2021 | Published on: 21 October 2021

DOI: 10.1002/qj.4181

RESEARCH ARTICLE

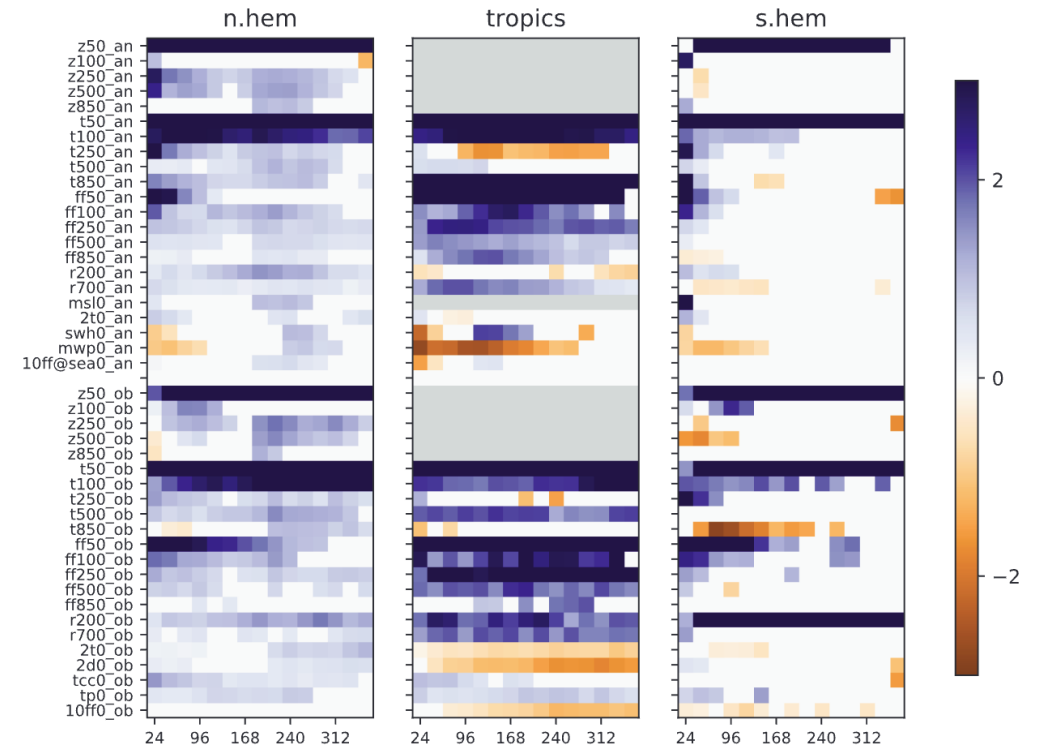
Quarterly Journal of the Royal Meteorological Society | RMetS

More accuracy with less precision

Simon T. K. Lang¹ | Andrew Dawson¹ | Michail Diamantakis¹ |
Peter Dueben¹ | Samuel Hatfield¹ | Martin Leutbecher¹ | Tim Palmer² |
Fernando Prates¹ | Christopher D. Roberts¹ | Irina Sandu¹ | Nils Wedi¹

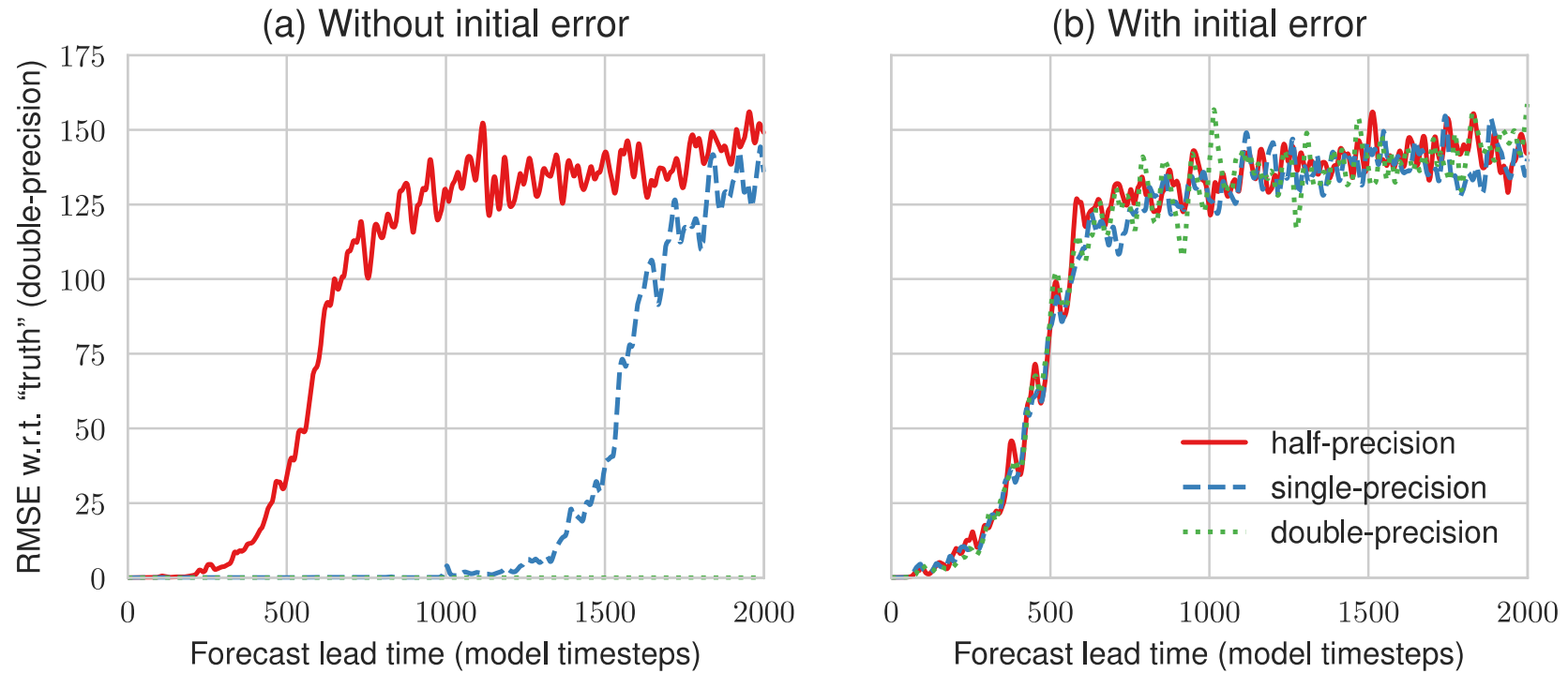
Lang et al. 2021

Switching from double to single precision permitted
vertical resolution increase in ENS **for free**
→ **improves forecast skill**



Scorecard SP vs DP (+ve = SP better)

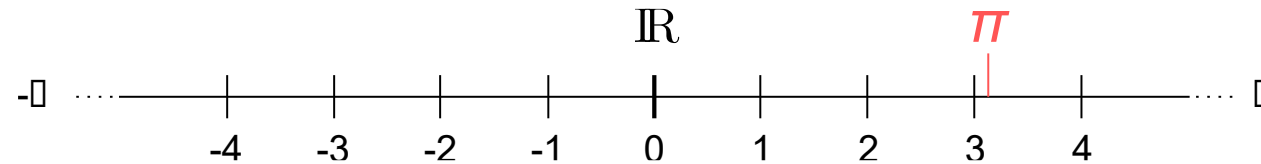
How much does precision matter?



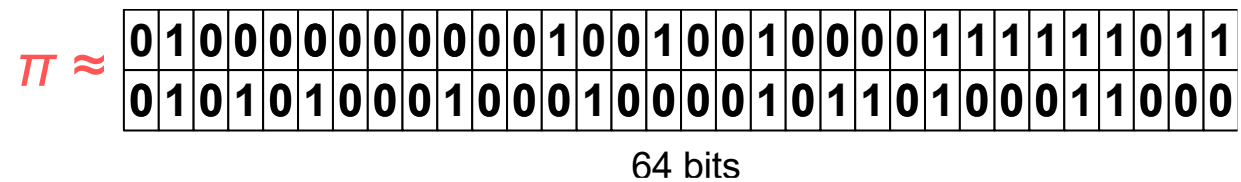
Lorenz '63 example

Real numbers on computers

Numerical models use **real number arithmetic**



Computers deal with **finite bit strings**



How do we map a real number to a string of bits?

The obvious way: fixed-point numbers

Integer representation can be easily modified to represent real numbers

$$10110110_2 = 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 182_{10}$$

The obvious way: fixed-point numbers

Integer representation can be easily modified to represent real numbers

$$10110110_2 = 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 182_{10}$$

$$10110.110_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} = 22.75_{10}$$

The obvious way: fixed-point numbers

Integer representation can be easily modified to represent real numbers

$$10110110_2 = 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 182_{10}$$

$$10110.110_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} = 22.75_{10}$$

Major drawback: limited range = $\left[2^{n_{\text{bin}} - n_{\text{bits}}}, \sum_{i=n_{\text{bin}} - n_{\text{bits}}}^{n_{\text{bin}} - 1} 2^i \right]$

Left offset of
binary point

Total width in bits

A better way: floating-point numbers

Instead we use **floating-point numbers**:

$$x = \underbrace{\text{fixed-point number}}_{\substack{\text{significand/mantissa} \\ \text{(between 1 and 2)}}} \times 2^{\underbrace{\text{integer-bias}}_{\text{exponent}}}$$

A better way: floating-point numbers

Instead we use **floating-point numbers**:

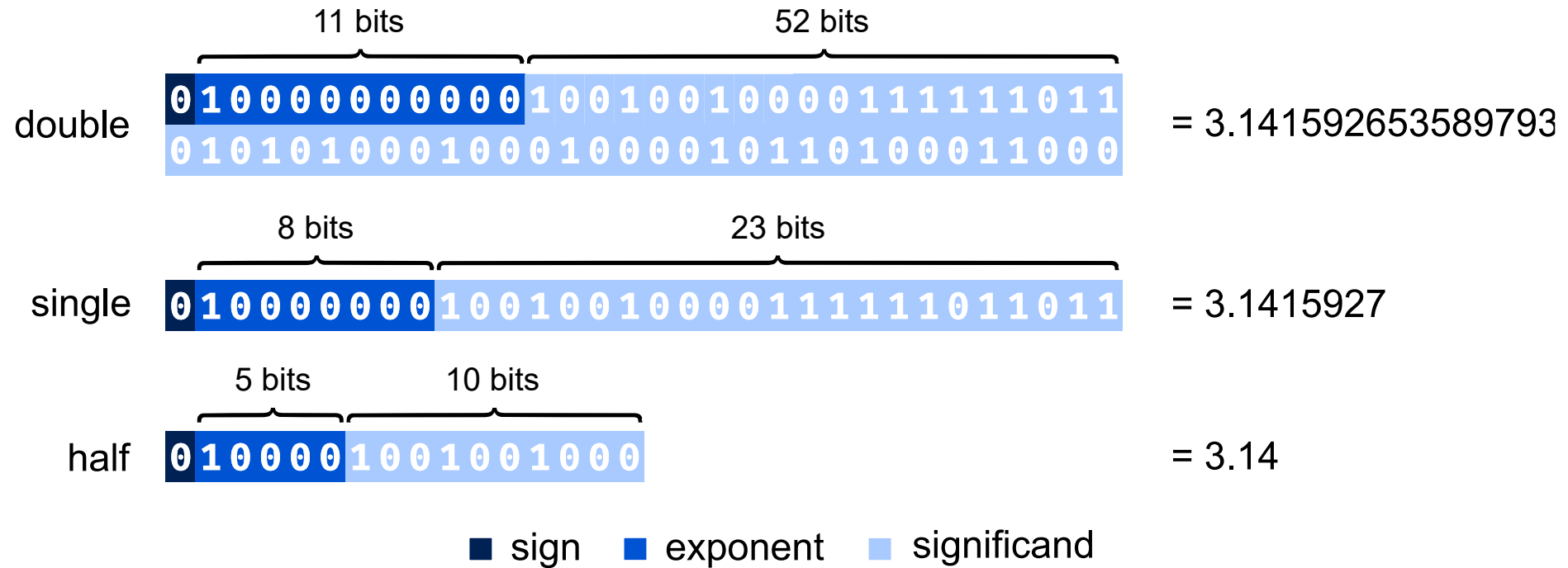
$$x = \underbrace{\text{fixed-point number}}_{\substack{\text{significand/mantissa} \\ \text{(between 1 and 2)}}} \times 2^{\underbrace{\text{integer-bias}}_{\text{exponent}}}$$

$$10110110_2 = (1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 1 \times 2^{-4} + 0 \times 2^{-5}) \times 2^{1 \times 4 + 1 \times 2 + 0 \times 1 - 3} = 13.5_{10}$$

$$11111111_2 = (1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} + 1 \times 2^{-4} + 1 \times 2^{-5}) \times 2^{1 \times 4 + 1 \times 2 + 1 \times 1 - 3} = 31.05_{10}$$

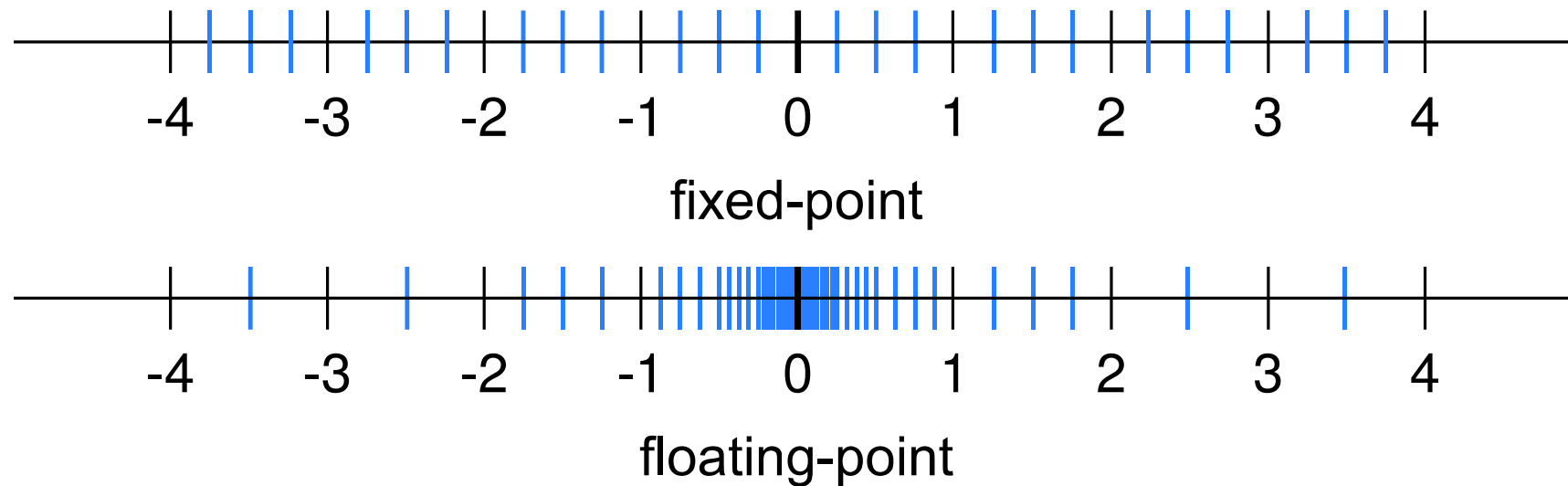
$$00000001_2 = (1 \times 2^0 + 0 \times 2^{-1} + 0 \times 2^{-2} + 0 \times 2^{-3} + 0 \times 2^{-4} + 0 \times 2^{-5}) \times 2^{0 \times 4 + 0 \times 2 + 1 \times 1 - 3} = 0.25_{10}$$

Boring but important: standardisation



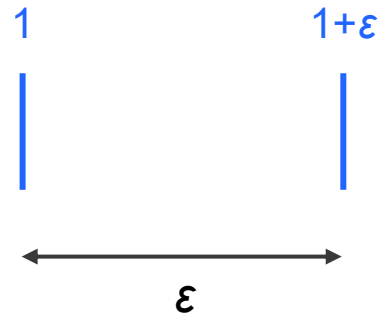
IEEE 754

Fixed- vs. floating-point number distribution



Machine precision

The difference between 1 and the next largest representable number is called the *machine precision/epsilon*



$$\varepsilon = 2^{-\text{number of significant bits}}$$

The relative error of a floating-point assignment will be *at most* $\varepsilon / 2$

“Subnormal” numbers

Remember:

$$00000001_2 = (1 \times 2^0 + 0 \times 2^{-1} + 0 \times 2^{-2} + 0 \times 2^{-3} + 0 \times 2^{-4} + 0 \times 2^{-5}) \times 2^{0 \times 4 + 0 \times 2 + 1 \times 1 - 3} = 0.25_{10}$$



This system cannot represent zero because of this guy.


“Subnormal” numbers

Remember:

$$00000001_2 = (1 \times 2^0 + 0 \times 2^{-1} + 0 \times 2^{-2} + 0 \times 2^{-3} + 0 \times 2^{-4} + 0 \times 2^{-5}) \times 2^{0 \times 4 + 0 \times 2 + 1 \times 1 - 3} = 0.25_{10}$$


This system cannot represent zero because of this guy.

By convention, when the **exponent** is zero, the (implicit) leading bit is 0, not 1

$$00000000_2 = (0 \times 2^0 + 0 \times 2^{-1} + 0 \times 2^{-2} + 0 \times 2^{-3} + 0 \times 2^{-4} + 0 \times 2^{-5}) \times 2^{0 \times 4 + 0 \times 2 + 0 \times 1 - 3} = 0.0_{10}$$


Numbers with a zeroed **exponent** are called **subnormal numbers**

Flops

There are four elementary arithmetic operations:

+ - × ÷

Carrying out one of these **operations** on two **floating-point** numbers constitutes one ***flop***

See also: sqrt and FMA

Flop/s: an imperfect measure

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	El Capitan - HPE Cray EX255a, AMD 4th Gen EPYC 24C 1.8GHz, AMD Instinct MI300A, Slingshot-11, TOSS, HPE DOE/NNSA/LLNL United States	11,340,000	1,809.00	2,821.10	29,685
2	Frontier - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE Cray OS, HPE DOE/SC/Oak Ridge National Laboratory United States	9,066,176	1,353.00	2,055.72	24,607
3	Aurora - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	1,980.01	38,698
4	JUPITER Booster - BullSequana XH3000, GH Superchip 72C 3GHz, NVIDIA GH200 Superchip, Quad-Rail NVIDIA InfiniBand NDR200, RedHat Enterprise Linux, EVIDEN EuroHPC/FZJ Germany	4,801,344	1,000.00	1,226.28	15,794
5	Eagle - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Azure United States	2,073,600	561.20	846.84	

Rmax (PFlop/s)

Measured performance of LINPACK

Rpeak (PFlop/s)

Theoretical peak performance

*Top500 supercomputer rankings
November 2025*

Floating-point numbers are weird: demonstration 1

```
julia> 0.1 + 0.2 == 0.3  
false # WTF?
```

What's going on?

Floating-point numbers are weird: demonstration 1

```
julia> 0.1 + 0.2 == 0.3  
false # WTF?
```

What's going on?

Floating-point numbers are *binary* – each number is a sum of powers of two

Floating-point numbers are weird: demonstration 1

```
julia> 0.1 + 0.2 == 0.3  
false # WTF?
```

What's going on?

Floating-point numbers are *binary* – each number is a sum of powers of two

0.1 is actually 0.09999999

0.2 is actually 0.19999999

Floating-point numbers are weird: demonstration 1

```
julia> 0.1 + 0.2 == 0.3  
false # WTF?
```

What's going on?

Floating-point numbers are *binary* – each number is a sum of powers of two

0.1 is actually 0.09999999

0.2 is actually 0.19999999

0.1 + 0.2 is actually 0.29988888

Floating-point numbers are weird: demonstration 1

```
julia> 0.1 + 0.2 == 0.3  
false # WTF?
```

What's going on?

Floating-point numbers are *binary* – each number is a sum of powers of two

0.1 is actually 0.09999999

0.2 is actually 0.19999999

0.1 + 0.2 is actually 0.29988888

0.3 is actually 0.30004888

Floating-point numbers are weird: demonstration 1

```
julia> 0.1 + 0.2 == 0.3  
false # WTF?
```

What's going on?

Floating-point numbers are *binary* – each number is a sum of powers of two

0.1 is actually 0.09999999

0.2 is actually 0.19999999

0.1 + 0.2 is actually 0.29988888

0.3 is actually 0.30004888

hence

0.1 + 0.2 != 0.3

Floating-point numbers are weird: demonstration 2

The harmonic series

$$1 + \frac{1}{2} + \frac{1}{3} + \dots$$

diverges when calculated
with infinite precision

With (finite-precision)
floating-point arithmetic, it
converges!

Float16

$$1 + \frac{1}{2} + \frac{1}{3} + \dots = 7.0859$$

Float32

$$1 + \frac{1}{2} + \frac{1}{3} + \dots = 15.404$$

Float64

$$1 + \frac{1}{2} + \frac{1}{3} + \dots = 34.122$$

Swamping

The harmonic series converges because of *swamping*

```
julia> Float16(2500.0) + Float16(1.0)
Float16(2500.0)
```

This can occur when doing big number + small number. Why?

```
julia> nextfloat(Float16(2500.0))
Float16(2502.0) # There is no Float16(2501.0)!
```

Floating-point numbers are weird: demonstration 3

Pathological case:

*a, b are arrays of **50,000** elements of all **1s** except the first which is **50***

Answer with Float32:
52499.0

Answer with Float16:
2500.0

```
function inner_product(a, b)
    sum = a[1] * b[1]

    for (a_i, b_i) in zip(a[2:end], b[2:end])
        sum += a_i * b_i
    end

    sum
end
```

Floating-point numbers are weird: demonstration 3

Pathological case:

*a, b are arrays of **50,000** elements of all **1s** except the first which is **50***

Answer with Float32:
52499.0

Answer with Float16:
52499.0

```
function inner_product_mixed(a, b)
    sum = Float32(a[1] * b[1])

    for (a_i, b_i) in zip(a[2:end], b[2:end])
        sum += Float32(a_i * b_i)
    end

    sum
end
```

Floating-point numbers are weird: demonstration 3

Pathological case:

*a, b are arrays of **50,000** elements of all **1s** except the first which is **50***

Answer with Float32:
52499.0

Answer with Float16:
52500.0

```
function inner_product_compensated(a, b)
    sum = a[1] * b[1]
    c = convert(typeof(a[1]), 0.0)

    for (a_i, b_i) in zip(a[2:end], b[2:end])
        y = (a_i * b_i) - c
        t = sum + y
        c = (t - sum) - y
        sum = t
    end

    sum
end
```

Floating-point numbers are weird: demonstration 3

Base type	Algorithm	Answer	FLOPs
Float32	Basic	52499.0	$2n-1$ Float32
Float16	Basic	2500.0	$2n-1$ Float16
Float16	Mixed	52499.0	n Float16 + $n-1$ Float32
Float16	Compensated	52500.0	$5n-4$ Float16

Floating-point numbers: *recap*

- Floating-point numbers have a **significand** and an **exponent**

$$x = \underbrace{\text{fixed-point number}}_{\text{significand}} \times 2^{\underbrace{\text{integer-bias}}_{\text{exponent}}}$$

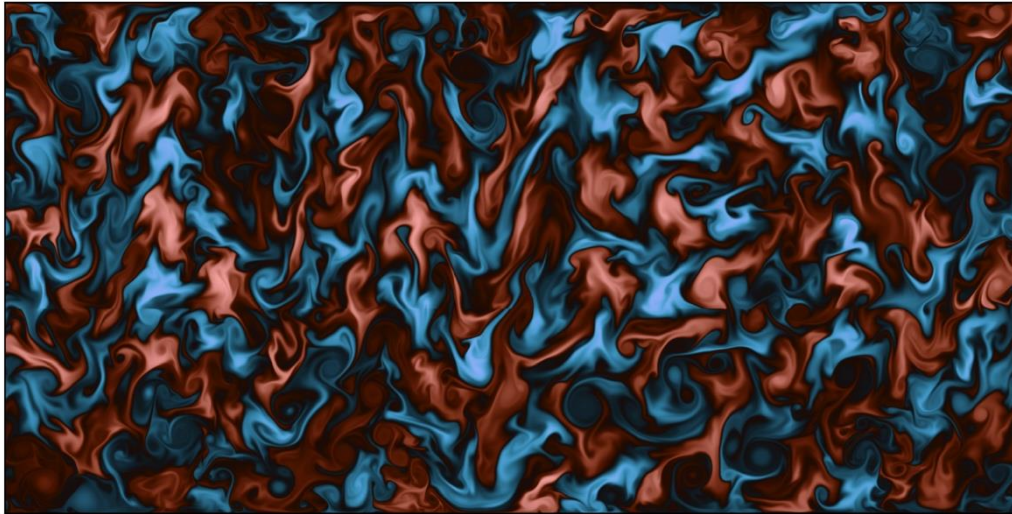
- Their precision is determined by the **machine epsilon**
- They have a **normal** range and a **subnormal** range
- “Computational work” is measured in **flops**, speed in **flop/s**
- Only numbers decomposable into power-two sums are perfectly representable
- Caution required when adding **big numbers** and **small numbers**

Half-precision in practice

Half precision already demonstrated in shallow water simulations

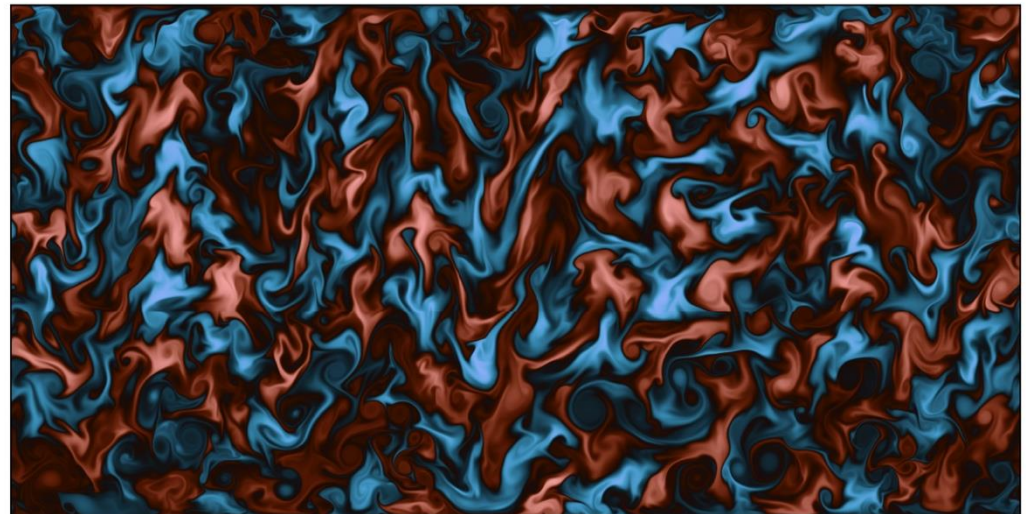
Float64 simulation

a



Float16 simulation

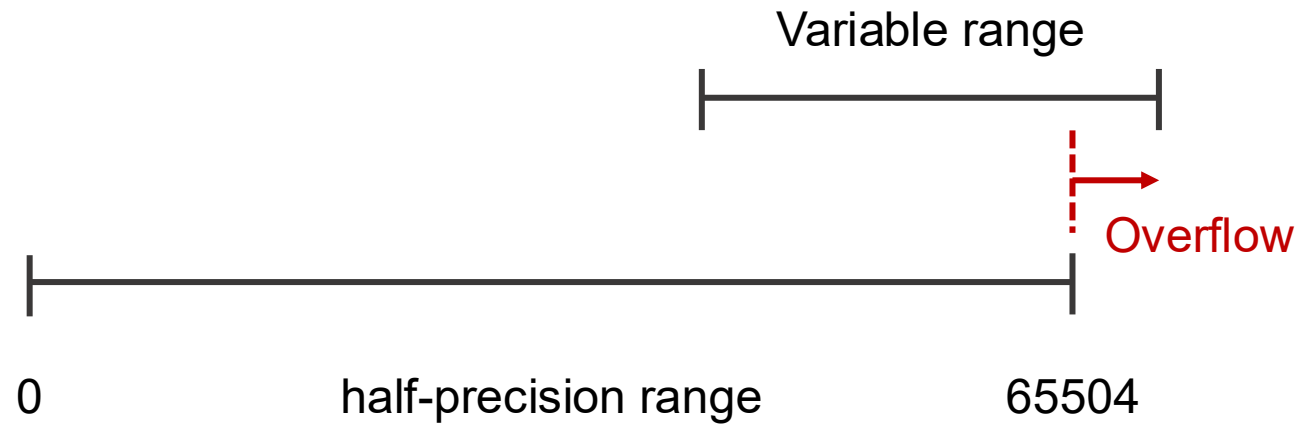
b



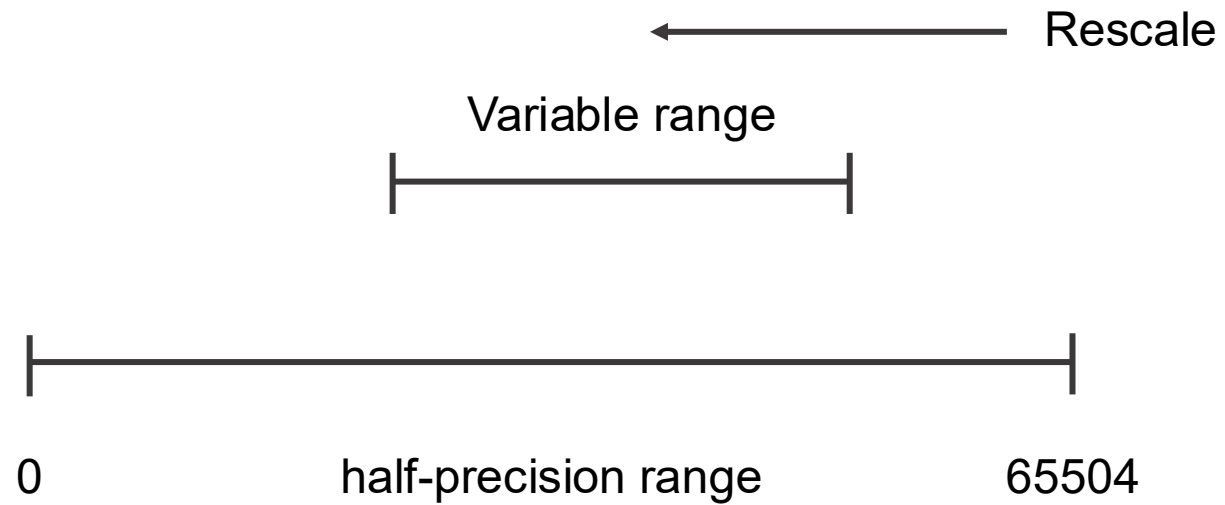
Klöwer et al. 2021

What about in NWP models?

Could we use half precision in NWP?

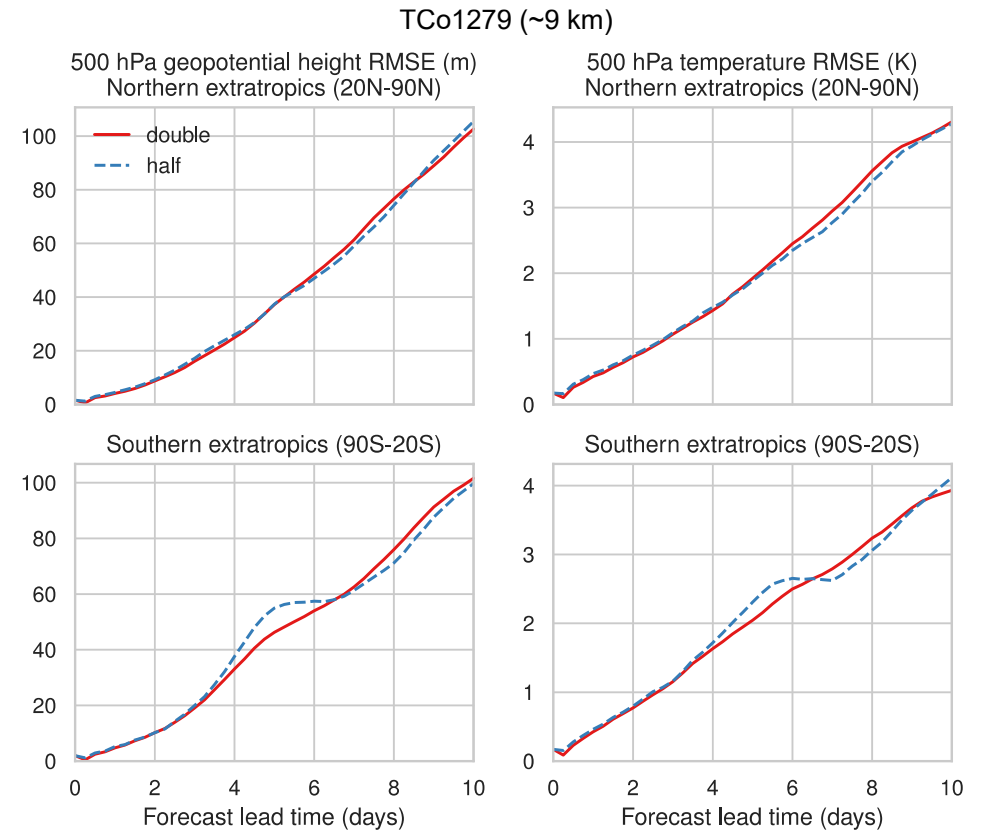


Could we use half precision in NWP?



Could we use half precision in NWP?

It actually works...
(for software-emulated half precision)



Hatfield et al. 2019

What about data assimilation?

Successful convergence of minimization in 4D-Var depends on this equality holding:

$$\underbrace{[\mathbf{M}_{im}(\mathbf{x}_0)\delta\mathbf{x}]^T}_{\text{tangent-linear model}} \delta\mathbf{y} = \delta\mathbf{x}^T \underbrace{[\mathbf{M}_{im}^T(\mathbf{x}_0)\delta\mathbf{y}]}_{\text{adjoint model}}$$

This equality will never hold exactly for floating-point arithmetic!
The higher the precision, the less the inequality

See Hatfield et al. 2020

Sam Hatfield

samuel.hatfield@ecmwf.int

