Geosci. Model Dev. Discuss.,
https://doi.org/10.5194/gmd-2020-239-SC1, 2020

Geoscientific
Model Development
Discussions

Interactive comment on "A note on precision-preserving compression of scientific data" *by* Rostislav Kouznetsov

Milan Klöwer

milan.kloewer@physics.ox.ac.uk

Received and published: 30 July 2020

The author discusses shortcomings of the some rounding methods for floating-point numbers used in data compression techniques implemented in the NCO software library. The points adressed are worthwhile as the default rounding mode in NCO, so-called BitGrooming, introduces artifacts that can be avoided with other techniques, as shown in this manuscript. The author proposes a round-to-nearest mode as a default rounding mode and provides evidence for its advantages over the other rounding modes such as BitGrooming, -Shaving, and halfshave. Methods to remove the shortcomings of bitgrooming from a previously groomed dataset are presented additionally.

1 General remarks

1.1 Tie rules

The author introduces a round-to-nearest mode that is based on 1 floating-point addition, 1 multiplication and two shave operations, which will be called the 2u-v method from now on, as follows from the underlying equation in the manuscript. The manuscript is currently lacking a discussion on the tie rules, i.e. the behaviour of the 2u-v rounding when a float is exactly between two representable quantums. If the distance between those is 1 ULP than I am referring to floats at 1/2 ULP. The 2u-v method is equivalent to round-to-nearest tie away from zero, which is bias-free for data that is symmetrically distributed around 0, but introduces a bias for data that is predominantly positive or negative. I therefore propose to include this information in the manuscript to distinguish the 2u-v method from the IEEE round-to-nearest tie-to-even (aka half to even) standard. The author can make a point though, that in the case of data compression rounding errors usually don't accumulate as they can in e.g. simulations of dynamical systems, and therefore might be negligible. In any case, it is proposed to include a discussion around this bias (where it is also important to clarify that a tie-bias is different to the bias introduced by bit-shaving for example, which scales with ULP) and whether it is important in the typical use-cases of NCO.

I suggest to discuss how your method is different from round-to-nearest tie-to-even which is already mentioned in IEEE-754 from 1985 as "An implementation of this standard shall provide round to nearest as the default rounding mode. In this mode therepresentable value nearest to the infinitely precise result shall be delivered; if the two nearest representable values areequally near, the one with its least significant bit zero [which is the "even" number] shall be delivered." (page 5 therein)

To illustrate this remark, consider the following float32 numbers. round refers here to round-to-nearest tie-to-even ("half to even") and kouzround to the 2u-v method introduced by the author.

```
julia> a
5-element Array{Float32,1}:
 0.09765625
 0.0041503906
 0.0061035156
 0.028320312
 0.033203125
```

In its bit-representation they are (split into sign,exponent and significant bits)

```
julia> bitstring.(a,:split)
5-element Array{String,1}:
 "0 01111011 10010000000000000000000"
 "0 01110111 00010000000000000000000"
 "0 01110111 10010000000000000000000"
 "0 01111001 11010000000000000000000"
 "0 01111010 00010000000000000000000"
```

Rounding these numbers with round-to-nearest tie-to-even (round towards zero in this case) and keeping 3 significant bits yields

```
julia> bitstring.(round(a,3),:split)
5-element Array{String,1}:
 "0 01111011 10000000000000000000000"
 "0 01110111 00000000000000000000000"
 "0 01110111 10000000000000000000000"
```

```
 "0 01111001 11000000000000000000000"
 "0 01111010 00000000000000000000000"
```

In contrast, the 2u-v method (aka "kouzround") rounds these ties away from zero

```
julia> bitstring.(kouzround(a,3),:split)
5-element Array{String,1}:
 "0 01111011 10100000000000000000000"
 "0 01110111 00100000000000000000000"
 "0 01110111 10100000000000000000000"
 "0 01111001 11100000000000000000000"
 "0 01111010 00100000000000000000000"
```

In the case where the even quantum is away from zero, both methods are indeed identical

```
julia> b
0.030273438f0

julia> bitstring(b,:split)
"0 01111001 11110000000000000000000"

julia> bitstring(round(b,3),:split)
"0 01111010 00000000000000000000000"

julia> bitstring(kouzround(b,3),:split)
"0 01111010 00000000000000000000000"
```

Note how the carry bit carries correctly into the exponent bits, something that bitgrooming, shaving and setting would never do.

1.2 Implementing round-to-nearest tie-to-even with bitwise operations is "difficult"

Although I agree that the 2u-v method is fairly simple to understand conceptually and therefore intuitively easier to implement, a bitwise round-to-nearest (tie-to-even) still only requires 1 shift, 2 bitwise-AND, and 2 integer-ADD. I would therefore argue that its implementation is of even lower algorithmic complexity than 2u-v rounding, which involves, despite hidden, way more bitwise operations within calls to float-multiply and float-subtract. Assume you want to keep 7 significant bits of a float32 number, intepreted here as a unsigned integer ui, meaning there are 16 tailing bits. The algorithm then reads

```
ntb = 16                                # number of tailing bits, 16
shavemask = 0x0000_ffff        # cover all tailing bits
mask2 = 0x0000_7fff              # cover all tailing bits except the

# account for the carry bit
ui += mask2 + ((ui >> ntb) & 0x0000_0001)
ui &= shavemask                    # shave tailing bits
```

Which can be easily generalised for various numbers of keepbits. It is still very much worthwhile to introduce your method, such that libraries similar to NCO can make use of it. However, if you introduce a rounding method that does not

2 Minor remarks

(i) L.48-49

the least-significant bits (LSBs) of mantissa contain arbitrary, often chaotic information,which makes lossless-compression algorithms inefficient"

in comparison to L.19-20

non-significant bits of the value scontain arbitrary numbers with high entropy, that are difficult to compress

I agree with you that the tailing bits contain seemingly random or arbitrary bits, which have a high entropy and are therefore difficult to compress. I use the words "seemingly random" as these bits don't have to be truly random but at least somewhat irregular. I find it problematic to use "chaotic" as also non-chaotic systems can produce those. E.g. even the exp-function will produce

```
julia> bitstring.(exp.(Float32.(1:10)))
10-element Array{String,1}:
 "01000000001011011111100001010100"
 "01000000111011000111001100100110"
 "01000001101000001010111100101110"
 "01000010010110100110010010000010"
 "01000011000101000110100111000101"
 "01000011110010011011011011100011"
 "01000100100010010001010001000011"
 "01000101001110100100111101010100"
 "01000101111111010011100010101100"
 "01000110101011000001010011101110"
```

As e is transcendental, so is e^2 etc. which also means that in binary they will have an irregular sequence of bits (at least in the significant). I therefore suggest to change your wording slightly to give the reader a better understanding of what "difficult to compress" means: (i) Seemingly random, such that a loss-less algorithm can’t identify a pattern and (ii) high entropy, such that entropy-encodings will not yield any benefits.

Interactive comment on Geosci. Model Dev. Discuss., https://doi.org/10.5194/gmd-2020-239, 2020.