Dynamic Float Compression (DFC)

Specification Version 1.0, 2023/01/10 – Owen Cook

Extended illustrations found in “dfcPage.pdf”

Table of contents:

I’m quite new to math expressions so if I get something wrong in any of the expressions, that’s for sure my bad. If you see an issue with any of the document, feel free to post a GitHub issue. (All expressions are math representations of the code functions anyway.)

Introduction:

Roughly eight months ago, while working on my game engine framework, I wanted to use the glTF model format. After writing the header info needed as small as I could, I found the main problem was the values that represent the model data itself rather than the header info. Today I present a dynamic truncation technique that encodes the floating-point IEEE 754 standard in a much smaller memory footprint in most cases.

Exponent packing:

The exponent section of a float is 8 bits wide spanning +128 to -127. In most cases the exponent isn't at all in this vast range. I encode as many bits as the value needs with twos complement in mind for negative exponents.

Note: the floats exponent is biased by 127 in storage, each equation is expecting the exponent to be unbiased. Hence x-127

For the length of the exponent in bits, the following equation can be used.

$$explen(x-127) = \begin{cases} 0, \; x = 0 \\ \log_2(|x|) + 1\,, \; otherwise \end{cases}$$ (|x| absolute(x) | log2 binary logarithm)

For the value of the exponent, the following equation can be used for packing:

$$\exp(x-127) = \begin{cases} 0, & x = 0 \\ |x| \vee \left(\left(0.5 - \frac{sign(x)}{2}\right) \ll (explen(x) - 1)\right), & otherwise \end{cases}$$

(|x| absolute(x) | V OR operator | << bit shift left)

Exponent unpacking:

Unpacking the exponent from an encoded value is decoded as such.

$$\exp(x,l) = \begin{cases} 127, & l = 0 \\ (2 * (x \gg l) - 1) * \left(x \wedge ((1 \ll l) - 1)\right) + 127, & otherwise \end{cases}$$

(x: exponent | l: exponent length in bits | >> bit shift right | << bit shift left | ^ AND operator)

Mantissa/fraction packing:

The mantissa, or fraction part of the float, is the most important part therefore the longest codec function. As we go from the 23rd bit to the 0th bit, the fractional component gets smaller and smaller meaning the precision curve is less and less. Before we evaluate any part of the fraction, some “literal” cases should be handled before the iterative check approach is used.

- Zero literal:

  Given a ± precision input (`float zeroThresh`) a boundary to consider the input as a zero literal is evaluated. Expressed as such below, zero can be encoded as tag 0xFC or decimal equivalent 252.

  Note: Zero isn’t tagged as 0 because having no exponent, run, and real segments is also a valid and quite often encoded value. Tag value 0 encodes to 1.0f

  $isZero(x, \mu) = |x| < \mu$

  (x: input float | μ: zero threshold)

- Infinity literal:

  Infinity is encoded without signature. Tagged 0xFD or decimal 253 gets encoded if the exponent of the input’s exponent is 255 (2^{128}) and the fraction is 0.

- NaN literal:

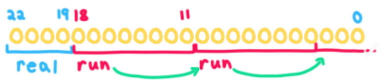
  NaN or “Not a Number” is encoded without signature. Tagged 0xFE or decimal 254 gets encoded if the exponent is 255 (2^{128}) and the fraction is **not** 0.

- Exponential rounding:

  After all literals are tested, masking the signature and (exponent ± 1) from the float is evaluated as an encoding possibility. The float value of both results is tested against the max error (`float maxErr`). If the increased or decreased exponent values are less than the max error, the value is encoded as such.

After testing each encodable literal, the iterative approach is used to find the best truncated storage medium. From here a “real” and “run” component are defined. Real is the number of “real” bits there are from the 23rd bit to a maximum of the 16th bit being stored 1:1. Run explains how many bits (0/4/8/12 bits) are run length encoded after the real bits. For example, a run of 2 and real of 3 make the 23rd to the 20th bit stored directly and from the 20th to the 12th bit run length encoded until the 0th bit.

Preview of example given next.



If none of the run/real encodings sufficed given the max error, real:23 or 1:1 copy of the fraction is encoded instead.

Mantissa/Fraction unpacking:

Decoding the fraction value of an encoded DFC value is mainly two steps.

For decoding the real segment of the buffer, the expression below can be used.

$$frac_{real}(x,\tau) = \big((x \gg (\tau+1)) \wedge ((1 \ll \tau) - 1)\big) \ll (23-\tau)$$

(x: encoded real buffer | τ: real bits)

For decoding the run segment of the buffer, a much longer expression can be used but its code structure is quite simple. *The math does look a little scary though :)*

$$let\ m = \ [0, 0x111111, 0x10101, 0x1001]$$

$$frac_{run}(x,\tau,\omega) = \Big(\Big(\big((x \gg (23 - 4*\omega - \tau)) \wedge \big((1 \ll (4*\omega)) - 1\big) * m[\omega]\big) \gg (\tau+1)\Big) \wedge \big((1 \ll (4*\omega)) - 1\big)$$

(x: encoded run buffer | τ: real bits | ω: run value | >> bit shift right | << bit shift left | m array given above | ^ AND operation)

The full fraction can then be constructed with

$$frac(x,\tau,\omega,\delta) = frac_{real}(x \gg (\delta+1), \tau) \vee frac_{run}(x \gg (\delta+1), \tau, \omega) \vee ((x \wedge 1) \ll 31)$$

(x: buffer | τ: real bits | ω: run value | δ: exponent length | << bit shift left | >> bit shift right | V OR operation | ^ AND operation)

Tag layout:

The tag layout combines the exponent, run, and real information into a one-byte long tag. Three bits represent the exponent length, neglecting the sign bit, giving 0-8 bits exponent. The run component is stored as [0/4/8/12 bit runs] with enumeration states in 2 bits. Lastly, the real component is expressed as the last 3 bits given (0-7 real bits). From left to right, most significant bit to least, **[exp][real][run] 000-000-00** is the tag layout.

For array encodings of multiple DFC values, an Adaptive Huffman Tree (AHT), or really any other compression algorithm, may be used to encode tag values that are used more often with less bits.
The AHT is initialized with the furthest leaves from root, having the same left-right address encoding as their representational value. (3 would be root/right/right/[3] as mapped 0b11) Of course all parts of the tag can be externalized outside of the tag component itself. For instance, in a model encoder implementation, the float exponents might be a good sorting delimiter. Having the exponent stored in the tag might end up being more data than having it stored outside.

Optimal and exhaustive encoding extensions:

The values for the encoded run and real segments are basically your entire float. Having said that, testing as many things as possible on the encoder side is highly advised in this codec since your evaluation might be close but not quite there.

- LSB in real segment inversion:

The least exhaustive while being the most driving error changer is the last bit in the real segment. This is because bits 0-7 are crucial to the error derivative when comparing the encoded DFC value against the input value. Flipping the state of the last bit and trying again is equivalent to turning (23-tag.real) bits off or on again without stating explicitly.

$$flip(x,\tau) = x \oplus (1 \ll (23 - \tau))$$

(x: encoded buffer | τ: real bits | $\oplus$: XOR operation | << bit shift left)

If the flip has less error than the current evaluation float, overwrite the current error with the flipped error and update the buffer with the bit flipped.

- Hyper exhaustive RLE:

Exhaustive RLE isn’t advised in time constrained applications but is still presented as an option. For this extension you try absolutely every possible run within the given run length and build the buffer for testing. To be implemented the naïve way, it would require [4=16 tests, 8=256 tests, 12=4096 tests] per run evaluation. You could precompute the run lengths ahead of time amounting to just 4096 buffer creations, but you would still have to test a lot more than if this extension isn’t used in the first place. TLDR if you have the time in the encoder, you can do it, but I recommend some form of SIMD if applicable.

- Cross referencing:

Cross referencing may be used when multiple DFC values are encoded together. This means encode all DFC values first, then test all values against the current value to see if other values can be re-encoded as the current. If there are multiple values that can be encoded to this current value, make a global DFC table and store either in the tag or the bit stream some delimiter to override normal decoding with an index to said table.

$$indexCount(x,v,\mu) = \{(\forall x) \mid (|x - v_i| \leq \mu)\}$$

(x: encoded values | v: input values | l: value count | μ: v max error | $(\forall x)$: foreach(x) | $(|x - v_i| \leq \mu)$: x can replace the encoded v value)