CSE 351 Lecture 7 – Floating Point II

IEEE 754 Floating Point Special Cases

Floating point allows for special numbers. These are obtained using specific combinations of the exponent and mantissa fields as follows:

`E`	`M`	Meaning
0b0…0	0b0…0	± 0
0b0…0	non-zero	Denormalized number
everything else	anything	Normalized number
0b1…1	0b0…0	$\pm\infty$
0b1…1	non-zero	Not-a-Number (NaN)

A ***denormalized number*** uses an implicit leading `0` (instead of `1`) and the fixed exponent of `1` – bias (even though the encoding is 0b0…0). This allows for the encoding of smaller numbers (closer to 0).

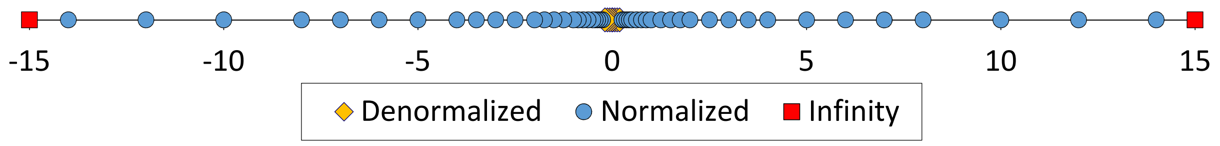
Notice that the addition of these special cases constricts the range of normalized exponents by one on both extremes.

Floating Point Limitations

The limitations of floating point are much more apparent to a programmer than integers due to existence of the special cases and the nature of the types of numbers we are trying to represent. Our chosen representation scheme leaves open three different categories of representation limitations:

1. The *largest* normalized number we can represent is when `E` = 0b1…10 and `M` = 0b1…1. The next largest number we can represent is infinity. When the result/value we are trying to represent is too large, it results in ***overflow*** and the stored result is $\pm\infty$.
2. The *smallest* normalized number we can represent is when `E` = 0b0…0 (denormalized) and `M` = 0b0…01. The next smallest number we can represent is 0. When the result/value we are trying to represent is too small, it results in ***underflow*** and the stored result is ± 0.
3. The limited *precision* afforded us by the fixed width of the mantissa also means that we cannot represent numbers that fall *between* two neighboring representable numbers. In this case, the stored result will be one of the two neighboring representable numbers, determined by ***rounding***.

A rough visualization of the representable floating point numbers can be seen below. Values between two norm or denorm points result in *rounding*. Values between the outermost norm points and infinity result in *overflow*. Values between the innermost denorm points and zero result in *underflow*.



Floating Point Arithmetic

Because of the limitations of floating point representation, there are lots of issues to be aware of when using floating point numbers as a programmer:

- Arithmetic operations on ∞ and NaN will work without warning, sometimes giving unintuitive results and making it difficult to pinpoint the origin of unexpected values.
- *Rounding* breaks the associative, distributive, and cumulative properties of floating point arithmetic.
- Straight equality comparisons (==) may yield unexpected results due to rounding. For example, two close but different values may be rounded to the same stored value and unexpectedly return true.
- Casting between an integral data type and a floating point data type *changes the bit representation*! The value may be changed from rounding, truncation, or overflow during the conversion.