Representing Rational Numbers in YANG

Better alternatives to Float

Don Fedyk
dfedyk@labn.net

How did we end up here?

- YANG Example from dot1as YANG
- Two Cases
  - Used for UScaledNs unsigned values of time and time interval in units of 2^-16 ns
    - What is the required range of this value?
  - Other Rational numbers Frequence and Refractivity

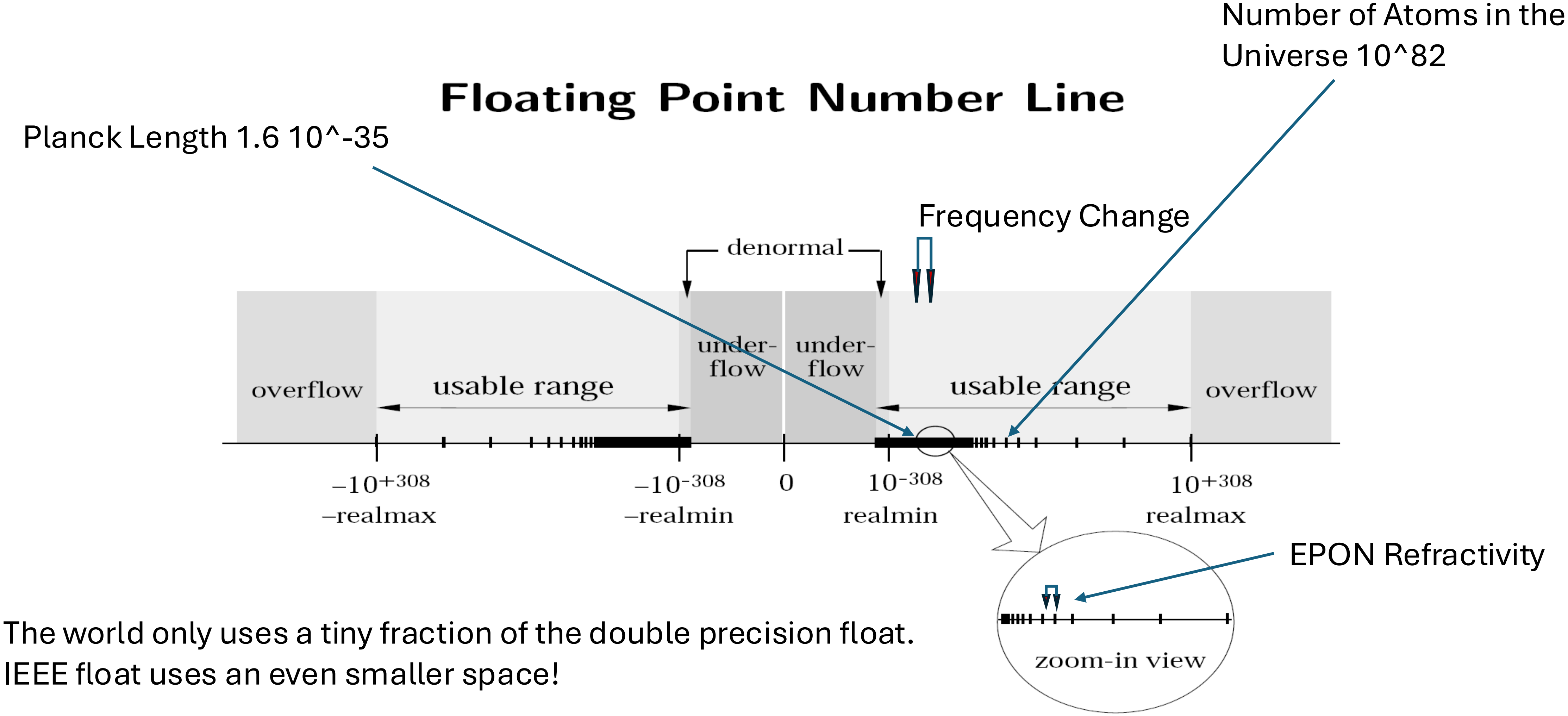
Let's examine these…

Note we have come a long way from when I first made comments,
Also, comments in here are the opinion of the author.

Some use Double Precision Float in YANG

- How do we display ? - Double precision float – typedef float64
  - 1 bit sign 11 bits exponent and 52 bits of mantissa
- 1 Nanosecond is = 0x3E112E0BE826D695 Hex (8 octets Float64)
- 1 UscaledNs = 0x3D112E0BE826D695 Hex (8 octets Float64)
  - At one time we used this for UScaled-ns now have another type for Uscaled-ns (more on that).
- In a Regex HEX string as 3E-11-2E-0B-E8-26-D6-95
- I don’t think this is very good because readability is poor, and the float range is terribly large

IEEE 784 Double Precision Range.



The world only uses a tiny fraction of the double precision float.
IEEE float uses an even smaller space!

Floating point in YANG

- YANG does not support floating point directly
- YANG supports strings, binary, integers and fixed-point decimal
  - Ranges and some validation can happen on some these items
  - Integer and fixed-point decimal are understood by directly by YANG
- IEEE and IETF Standards have incorporated Floating-point in some Modules
  - Either as:
    - Strings in regex (Mantissa and Sign)
    - Hex representations of IEEE 754 Float
    - ~~Binary Base64~~ – Not used anymore Yeah!
  - These representations have limited validation options and are not easily human comprehensible. (we can read them but can't easily compare them for example).
  - Floats don’t Render Well. All representations of floating point numbers are UGLY.
- This presentation suggest an alternative to floating point that leverages fixed-point decimal as a better alternative for the objects represented.

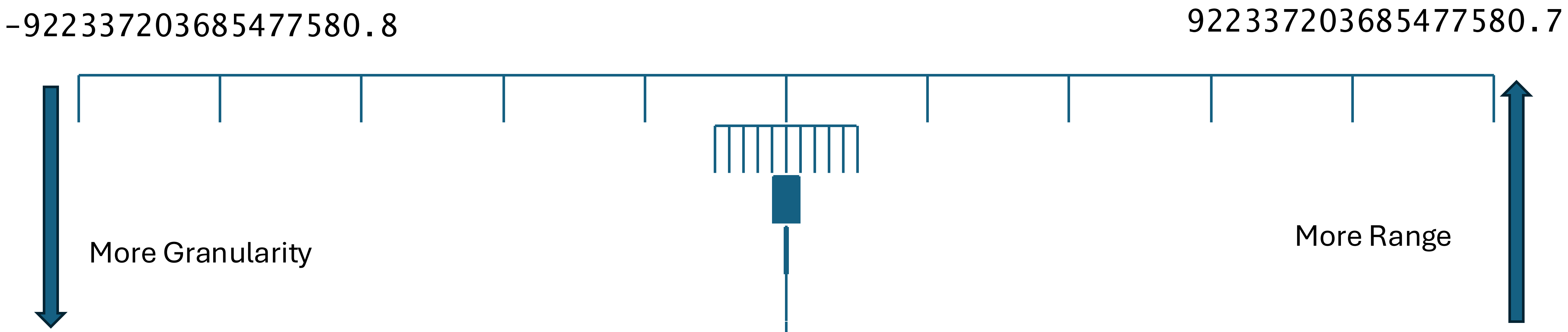
UScaled Nanoseconds

- Current representation is 96 bits of binary.
- This is 96 bits represented as Hexadecimal
- 1 Uscaled NS = 00-00-00-00-00-00-00-00-00-00-00-01
    - 1/65536 of a nanosecond.
    - BTW Who though mixing radix 2 and radix 10 was a good idea?
- 1 NS = 00-00-00-00-00-00-00-00-00-01-00-00
- 1second = 00-00-00-00-00-00-3B-9A-CA-00-00-00
- 1 year = 00-00-2B-CB-83-00-04-63-00-00-00-00
- Max = 38,398,547.532 years.
- I don’t think this is terrible. Probably 64 bits of Binary was sufficient. 96 bits makes it more complicated.

YANG Decimal64

- “The decimal64 built-in type represents a subset of the real numbers, which can be represented by decimal numerals. The value space of decimal64 is the set of numbers that can be obtained by multiplying a 64-bit signed integer by a negative power of ten, i.e., expressible as "i x 10^-n" where i is an integer64 and n is an integer between 1 and 18, inclusively.” RFC 7450
- The internal representation is effectively a 63 bit mantissa with fixed point **decimal**. division by 10, 100 up to 10^18.
- This covers a linear space within a range.

Decimal 64 Range



```
+----------------+----------------------+----------------------+
| fraction-digit | min                  | max                  |
+----------------+----------------------+----------------------+
| 1              | -922337203685477580.8 | 922337203685477580.7 |
| 2              | -92233720368547758.08 | 92233720368547758.07 |
| 3              | -9223372036854775.808 | 9223372036854775.807 |
| 4              | -922337203685477.5808 | 922337203685477.5807 |
| 5              | -92233720368547.75808 | 92233720368547.75807 |
| 6              | -9223372036854.775808 | 9223372036854.775807 |
| 7              | -922337203685.4775808 | 922337203685.4775807 |
| 8              | -92233720368.54775808 | 92233720368.54775807 |
| 9              | -9223372036.854775808 | 9223372036.854775807 |
| 10             | -922337203.6854775808 | 922337203.6854775807 |
| 11             | -92233720.36854775808 | 92233720.36854775807 |
| 12             | -9223372.036854775808 | 9223372.036854775807 |
| 13             | -922337.2036854775808 | 922337.2036854775807 |
| 14             | -92233.72036854775808 | 92233.72036854775807 |
| 15             | -9223.372036854775808 | 9223.372036854775807 |
| 16             | -922.3372036854775808 | 922.3372036854775807 |
| 17             | -92.23372036854775808 | 92.23372036854775807 |
| 18             | -9.223372036854775808 | 9.223372036854775807 |
+----------------+----------------------+----------------------+
```

Note Current 96bit UScaled-ns also uses a linear range than encompasses all of this space.

Decimal64 YANG
Is it sufficient when using 18 fractional digits ?

- 1 UScaledNs = 0.0000000000000152587890625
- In Decimal64 (fraction-digits 18) = 0.000000000000015259
- Error = 0.000013824
- 1 NS = 0.000000001
- Error is zero for any number that fits exactly in 18 decimal fraction digits or less.
- Lowest zero error value is an attosecond or quintillionth of a second. 10^-18 or 0.000000000000000001

What does Decimal64 fraction-digits 18 look like in YANG?

- 1 UScaledNs is 0.000000000000015259
- 1 Nanosecond is 0.000000001
- 1 Microsecond is 0.000001
- The representation is treated as decimal (radix 10) and Integer decimal math can be used.
- It is displayed as decimal.
- It is human comprehensible !

Summary Small Number YANG Evaluation.

Numerical Representation	Range	Presentation	Human Readable	Integer Math	YANG Ranges ?	Error	Efficiency
Float64	Insane	HEX - String	Not really	No	No	Extremely Low	Low
Binary 64	Practical	Int or HEX string	Yes Mostly	Yes	Yes	Very Low	High
Binary 96	Ridiculous	Hex -String	Sort of	Yes	No	Extremely Low	Medium
Decimal64	Practical	Decimal Value	Absolutely	Yes	Yes	Very Low	High

Almost any case where float is considered Decimal 64 is better
For Uscaled-ns it's a toss up because it mixes radix 2 and radix 10
Although using Binary96 versus binary64 needs justification.

Suggestions Going forwards

- Utilizing Decimal64 and Fraction digits.
- One definition with fractions of 18
- This fits for current uses
  - Epon Refractivity - 1.47 is supported directly with 18 or less fraction digits.
  - Restrict Decimal64 to 18 digits of decimal fraction unsigned (or signed)

But what if we need more High-end range greater than 9 seconds along with 18 places of decimal?

  - Make a YANG typedef
  - Fractional part Decimal64 fraction-digits 18 with
    - Range -0. 999999999999999999 .. 0.999999999999999999
  - Integer part either 8 bit, 16bit, 32bit signed.
  - What if the fractional error is still considered too coarse ? The Decimal fraction becomes milliseconds or microseconds.
    - High range is a high order decimal64 with Fraction-digits 3 (or 6).
  - <u>This keeps it Human readable and linear</u> in line with Decimal64 practices.

Final Thoughts

- When looking at operations per second on Float versus Integer on a modern computer the time for a single conversion is extremely small. So why bother?
- I have never experienced embedded devices, routers or bridges where the Engineer said the boot time was too fast or the UI was too responsive. But I have seen plenty of times where the boot time was under pressure, or the UI was sluggish.
- I also have debugged many interfaces over 40 years and having to stop to make a Base 64 or a Float conversion to a number I can understand or compare when there is a better option just seems wrong.

Backup

Operational Complexity

- Decimal64 can be scaled from binary with simple 10 x integer math.
- The 64 bit signed number has an associated power of 10.
- 64 bits + an exponent number 1 – 18
- Float Double precision fits into a 64 bit number. However, to do anything with it you must use floating point operations or convert it to integer.
- Roughly speaking Fixed point operations are about 1/10 as complex as floating point.

Why avoid Floating Point Numbers?
Lets ask a computer AI!

- Precision issues: Floating-point numbers are represented as binary fractions, which can lead to rounding errors and precision issues. This is because many decimal fractions cannot be exactly represented as binary fractions.
- Loss of accuracy: When performing arithmetic operations on floating-point numbers, the results can be subject to rounding errors, which can accumulate and lead to loss of accuracy.
- Comparing floating-point numbers: Due to the imprecision of floating-point numbers, comparing two floating-point numbers for equality can be problematic. This is because two numbers that are mathematically equal may not be exactly equal when represented as floating-point numbers.
- Speed and performance: Floating-point operations can be slower than integer operations, which can impact the performance of your code.
- Portability issues: Floating-point representations can vary across different platforms and architectures, which can lead to portability issues.
- Special values: Floating-point numbers have special values, such as infinity and NaN (Not a Number), which can be tricky to handle and may lead to unexpected behavior.
- Rounding errors: Rounding errors can occur when converting between different floating-point formats, such as when converting from a double to a float.

When to avoid floating-point numbers?
Lets ask a computer AI!

- Financial calculations: In financial calculations, precision is crucial. Avoid using floating-point numbers for calculations involving money, interest rates, or investment returns.
- ➡ Scientific simulations: In scientific simulations, accuracy is critical. Avoid using floating-point numbers for calculations that require high precision, such as in physics or engineering simulations.
- Machine learning: In machine learning, precision is important for accurate predictions. Avoid using floating-point numbers for calculations that require high precision, such as in neural networks or deep learning models.
- ➡ Embedded systems: In embedded systems, resources are limited, and precision is critical. Avoid using floating-point numbers for calculations that require high precision, such as in control systems or sensor data processing.

Alternatives to floating-point numbers:

- Integer arithmetic: Use integer arithmetic when possible, especially for calculations that involve whole numbers or discrete values.
- ➡ Fixed-point arithmetic: Use fixed-point arithmetic when possible, especially for calculations that require a fixed number of decimal places.
- ➡ Decimal arithmetic: Use decimal arithmetic when possible, especially for calculations that require high precision and accuracy, such as in financial calculations.
- Arbitrary-precision arithmetic: Use arbitrary-precision arithmetic when possible, especially for calculations that require high precision and accuracy, such as in scientific simulations or cryptographic applications.