# How are real numbers represented by a computer?

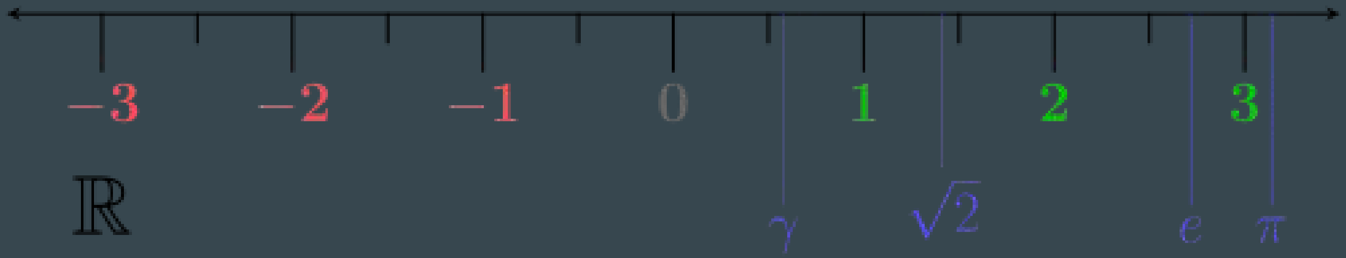
...

# Overview

- Problem
- Floating Point Format
- Properties
- Python/C++
- Links

# Real Numbers

“Real numbers can be thought of as points on an infinitely long line”



# Problem:

- There are an infinite number of real numbers
- Many do not have finite representation: □, sqrt(2)

- But computers have a finite amount of memory
  - How are computers able store such numbers?

# Scientific Notation

- Quite familiar with standard scientific notation of numbers. Able to represent a wide range of numbers
  - $1.528535047 \times 10^5$ - orbital period of Jupiter's moon Io
  - $6.67408 \times 10^{-11}$ - Newton's gravitation constant
- More generally a number can be written as:
  - `significand x base`$^{\texttt{exponent}}$
- `Quite used to working in base 10 but computers only understand binary (base-2)`

# Floating Point & IEEE

- Floating point is most common way to represent real numbers
  - Name comes from the fact you have a decimal point that can “float”
- IEEE standardizes the idea of a floating point formats:
  - binary32, binary64, binary128, binary256 …
- An evolving standard: IEEE 754-1985, IEEE 854-1987, IEEE 754-2008
  - Portable and provably consistent
  - To be conformant many mathematical identities must hold true
  - Every floating point number must be unique
  - Every floating point number must have an opposite
  - Specifies algorithms for addition, subtraction etc

# Floating Point Format

- Most common for us are `float` (binary32) & `double` (binary64)
- 32/64 indicate the amount of memory used to store a variable of that type
- Take `float` as an example. What can we do with 32 bits?
- Format is similar to scientific notation. A `float` is comprised
  - Sign = 1 bit: determines the sign
  - Exponent = 8 bits: sets the scale
  - Mantissa = 23 bits (actually gives 24 bit precision): determines the precision

$$x = -1^{s} \times 2^{e} \times 1.m$$

# Floating Point Format Details: Normalization

- A floating point number is considered normalized when the integer part of the mantissa is 1
- Example: `13.25= 1101.01 (binary). Normalize this`
  - `  1101.01 * (2^0)`
  - `= 110.101 * (2^1)`
  - `= 11.0101 * (2^2)`
  - `= 1.10101 * (2^3)` → normalized form
- By always storing normalized numbers we can avoid storing the leading 1 and therefore a 23-bit mantissa actually represents 24-bits of precision

# Exponents: Special Numbers

- How would you express 1.0 in this format?
  - Naively, $(-1)^0$ $(2^0)$ 1.(0) = [0] [00000000] 1.[00000000000000000000000]
  - But this looks a lot like how you would store 0! (remember the 1. is implicit)
  - Solution is to modify how the exponent part is stored and treat 0.0 as a special case
  - Exponent in binary32 is encoded by shifting it by `-127`.
  - → 1.0 = [0][01111111] 1.[00000000000000000000000]
- Special cases:
  - `0 (all exponent & mantissa bits 0)`
  - `Inf = 1/0 (all exponent bits 1, all mantissa bits 0)`
  - `NaN = 0/0, 0 x ∞ (all exponent bits 1, any mantissa bits non zero)`
- Special cases reduce the range of representable numbers slightly

# Floating Point Properties

- Cannot exactly represent all numbers within a fixed amount of memory
  - Calculations most often produce unrepresentable numbers → rounding error
- We have seen 1.0 represented as [0][01111111] 1.[00000000000000000000000]
- Next representable-number is found by flipping least-significant bit in mantissa
  - [0][01111111] 1.[00000000000000000000001] = 1.0000001192092896
  - The difference between this and 1.0 is defined as `epsilon`
  - Useful for programming “almost equals” calculations
- Other properties:
  - min/max ~ $[1.18 \times 10^{-38}, 1.7 \times 10^{+38}]$
  - significant decimal digits: ~7

# Floating Point Properties

- Representation is not uniform between numbers, i.e.
  - each number is not `epsilon` apart
  - Most precision between `0.0 - 1.0`
  - precision falls away after this
- This can be a good reason to scale calculations to 0.0-1.0 especially for long running computations

# Double Precision

- Most programming languages also support the binary64 float or `double`
- Defined simply as using 64 bits to store a “real” number
- Properties:
  - Sign = 1 bit
  - Exponent = 11 bits
  - Mantissa = 52 bits (actually gives 53 bit precision)
- Many more bits used for mantissa to extend the precision and sacrifice range:
  - epsilon<double> ~ $2.22 \times 10^{-16}$
  - range ~ $[2.22 \times 10^{-308}, 1.78 \times 10^{308}]$

# C++/Python

- C++ defines numeric traits various types in `<numeric_limits>`
  - http://en.cppreference.com/w/cpp/types/numeric_limits
  - e.g. `std::numeric_limits<double>::epsilon`, `std::numeric_limits<double>::min`, `std::numeric_limits<double>::max`
- Python has some definitions in `sys` module:
  - https://docs.python.org/3/library/sys.html
  - `sys.float_info` - returns a struct holding similar information to that in <numeric_limits>
  - `numpy` also has similar information in `numpy.finfo`

# Comments

- Beware computations where exponents are very different
- If you know the scale of various values try to perform computation in sections where you keep numbers of ~ equivalent size together
- Order of floating point arithmetic can matter
    - Multi-threading can have an effect here if the order of calculations is not guaranteed to be the same

# Links

- https://en.wikipedia.org/wiki/Single-precision_floating-point_format
  - Good examples of how to convert between decimal and binary
- Demystifying Floating Point - https://www.youtube.com/watch?v=k12BJGSc2Nc
- What Every Computer Scientist Should Know About Floating-Point Arithmetic - https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html