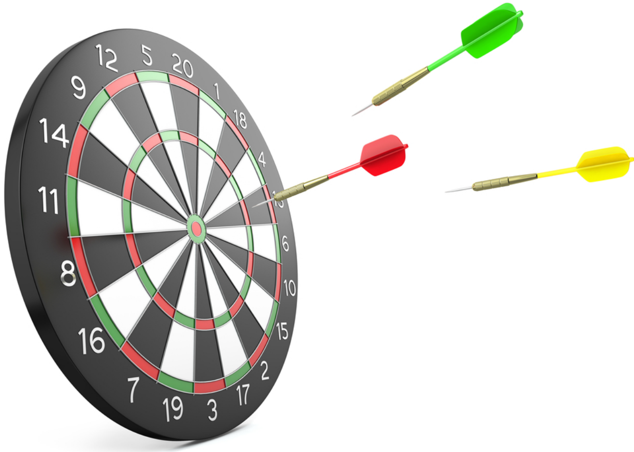


What is difference between *precision* and *accuracy*?

- what is the distance from San Jose to Oakland?
  - 60 km : *accurate but not precise (65.5 km is correct)*
  - 5900.1395 km : *precise but not accurate*
- why do we want to know?
  - precision rocketry
  - about how long to drive there?
- *sometimes the last 10% of precision takes 90% of the time to work out*

Decimal system

- each digit represents a power of 10

10^4	10^3	10^2	10^1	10^0		10^{-1}	10^{-2}	10^{-3}	10^{-4}
↓	↓	↓	↓	↓		↓	↓	↓	↓
6	0	7	2	4	.	3	1	2	5

$$6 \times 10^4 + 0 \times 10^3 + 7 \times 10^2 + 2 \times 10^1 + 4 \times 10^0 + 3 \times 10^{-1} + 1 \times 10^{-2} + 2 \times 10^{-3} + 5 \times 10^{-4} = 60{,}724.3125$$

Binary notation

- each digit (bit) represents a power of 2
- convert binary `10011010` to decimal: $= 154$

Base 10 / Base 2

	2^3	2^2	2^1	2^0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0

Base 10 / Base 2

	2^{-1}	2^{-2}	2^{-3}	2^{-4}
0.0625	0	0	0	1
0.125	0	0	1	0
0.1875	0	0	1	1
0.25	0	1	0	0
0.3125	0	1	0	1
0.375	0	1	1	0
0.4375	0	1	1	1
0.5	1	0	0	0
0.5625	1	0	0	1
0.625	1	0	1	0

- convert decimal 0.1 to binary?
  - $0.000110011 = 0.096609375$

Binary notation

- decimal and binary representations of the same number:

$$\begin{array}{ccccccccc} 10^4 & 10^3 & 10^2 & 10^1 & 10^0 & 10^{-1} & 10^{-2} & 10^{-3} & 10^{-4} \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 6 & 0 & 7 & 2 & 4\,. & 3 & 1 & 2 & 5 \end{array}$$

$$6\times10^4+0\times10^3+7\times10^2+2\times10^1+4\times10^0+3\times10^{-1}+1\times10^{-2}+2\times10^{-3}+5\times10^{-4}=60{,}724.3125$$

$$\begin{array}{cccccccccccccccccccc} 2^{15} & 2^{14} & 2^{13} & 2^{12} & 2^{11} & 2^{10} & 2^9 & 2^8 & 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 & 2^{-1} & 2^{-2} & 2^{-3} & 2^{-4} \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0\,. & 0 & 1 & 0 & 1 \end{array}$$

$$1\times2^{15}+1\times2^{14}+1\times2^{13}+0\times2^{12}+1\times2^{11}+1\times2^{10}+0\times2^9+1\times2^8+0\times2^7+0\times2^6+1\times2^5$$
$$+1\times2^4+0\times2^3+1\times2^2+0\times2^1+0\times2^0+0\times2^{-1}+1\times2^{-2}+0\times2^{-3}+1\times2^{-4}=60{,}724.3125$$

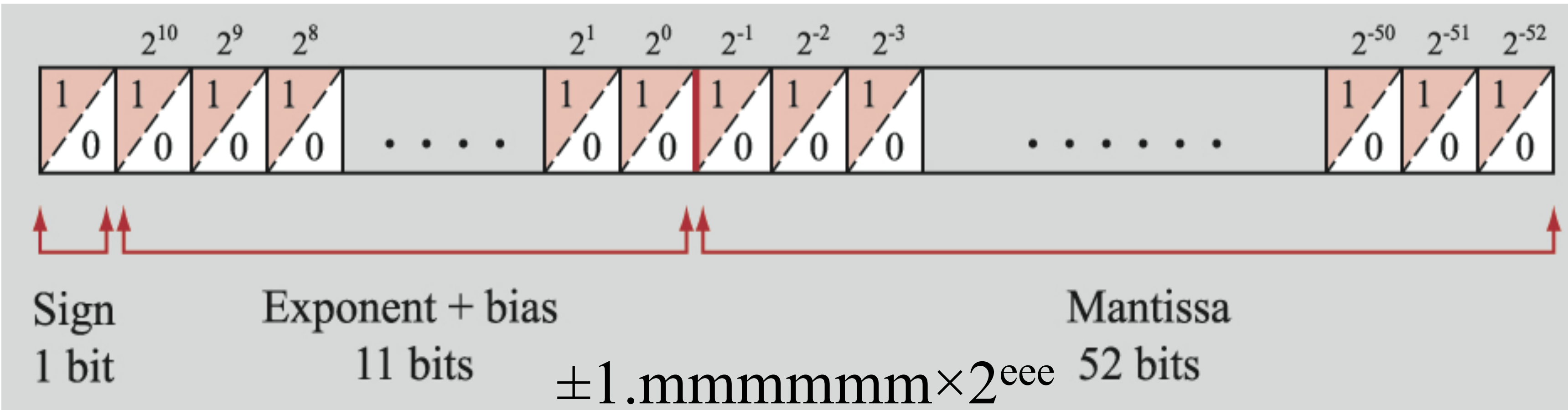
Floating point representation (scientific notation)

- $\#.\#\#\#\# \times \mathbf{base}^{p}$
  - #.#### is mantissa
  - p is exponent (or characteristic)
- base 10 : 8.45648×10^{7}
- base 2 : 1.25×2^{2}

  $= \mathtt{1.01} \times \mathtt{10}^{\mathtt{10}}$
- **precision** is number of (binary) digits in mantissa and exponent

Computer storage of floating-point numbers

- double precision example: 64 bits (8 bytes) :
  - 1 bit for sign, 11 bits for exponent, 52 for mantissa



- single precision has 32 bits (4 bytes: *don't use!*)
- exponent **bias** allows storage of big **and** small numbers:
  1 stored as 2^{1023} not 2^0
  $\sim 10^{-308}$ to $\sim 10^{308}$ rather than 10^0 to 10^{616}
  (down to $\sim 10^{-323}$ from denormalizing/subnormalizing)
- equal numbers of numbers stored from 0.1 to 1, to 10, 10 to 100
- binary representation has limitations: `0.1 + 0.2 = 0.30000000000000004`

Python: formats and types

- Python "float" is double-precision (64-bit)
- use **`type()`** to see type of a variable
- can "cast" (convert) explicitly to other types, e.g.:

`>>> vard = float(1)`	
`>>> varg = single(1.0)`	NumPy 32-bit float
`>>> vari = int16(1.0)`	NumPy 16-bit integer
`>>> varst = str(1)`	string

- convert implicitly ('coercion') by mixing types:

```
>>> new_vard = 1 * 2.0
```

Python: variables

- general variable types (objects):
  - numerical, string, list, tuple, set, dictionary
  - types assigned dynamically (not like C)
- numerical variable types:
  - integer: $|n| < 2^{31}$
  - long integer: `1234567890123456789L`
  - floating point real: `1.2e10` , `1.3E-11`
  - complex: `1.0 + 2.3j`
- assign values:
  - `a = b = c = 1`
  - `a, b, c = 1, 8.5, "test"`
  - `a,b = b,a` (swap values)
  - `a += b` (same as `a = a + b`)
  - `del a` (delete variable)

Physics equations in Python

- examples:

$$E = mc^2 \qquad x(t) = x_0 + \mathrm{v}_0 t + \frac{1}{2} g t^2$$

$$T = \frac{1}{2}\, m\mathrm{v}^2$$

- use some Python variables as "constants," others as "variables"

Python: math functions

- be careful with integer division in v2:

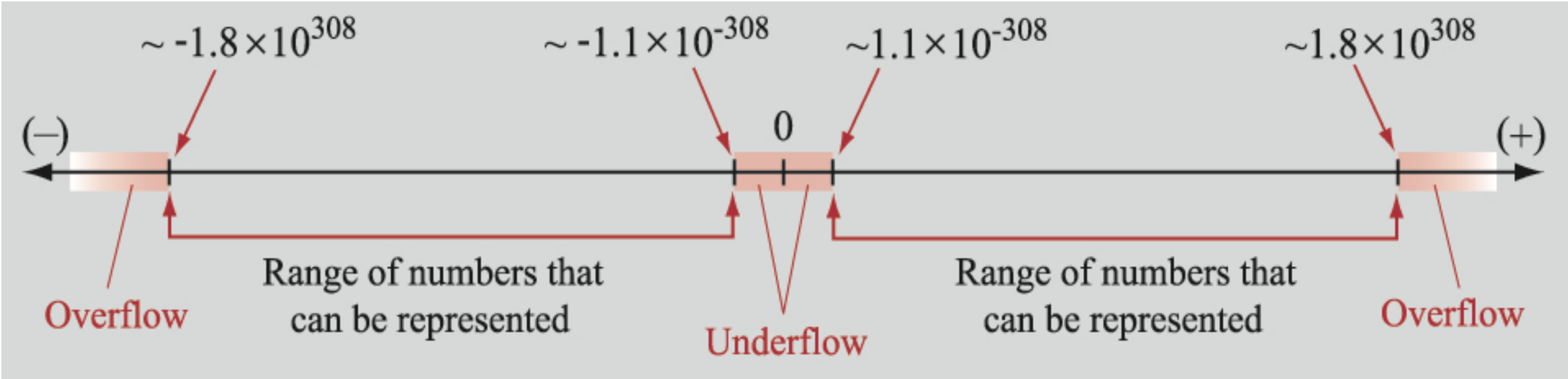
  `>>> 8/3` (“floors” the answer: =2; use `//` in v3)

  `>>> 8.0/3.0` (force a float)

- `>>> a = 12 % 8` (remainder, returns value of 4)
- `>>> b = 2**3` (exponentiation, 2^3)
- basic math functions in math module (numpy better):
  - `from math import *`
  - `sqrt(), sin(), cos(), tan(), asin(), acos(), atan(), sinh(), cosh(), tanh(), exp(), log(), log10(), pi, e, floor(), ceil(), abs(), erf(), erfc()`

Dynamic range of number storage

- computer cannot represent all real numbers



- errors can occur: overflow, underflow, round-off:
  - $10^{308} + 10^{308} =$ **`inf`** (in Python)
  - $10^{-323} - 8\times10^{-324} =$ **`0.0`**
  - smallest value in mantissa is **machine epsilon** :
    - $1+\varepsilon = 1$ (how do we figure out the value of ε ?)
    - $\varepsilon \approx 2^{-52} \approx 10^{-16}$ in double precision
    - smaller **fractional** differences between numbers cannot be computed: $1.0000000000000007 - 1.0000000000000006 = 0$!

see **`sys.float_info`** and **`np.finfo(float).eps`** `(import sys)`

Sources of numerical error

- **truncation error** due to approximation:
  – finite series expansion :
  $$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$
  – happens even with infinite dynamic range

- **total/true error** = truncation + round-off
  – loss of accuracy + precision

- **relative** error more important than absolute:

$$\Delta f = \left| \frac{f_{\text{num}} - f_{\text{true}}}{f_{\text{true}}} \right|$$

Fractional error

- **relative/fractional** error: $\Delta f = \left| \frac{f_{\text{estimate}} - f_{\text{true}}}{f_{\text{true}}} \right|$

- example: leave enough water for your pet on a hot day
  - cow: $V_{\text{correct}} = 10\ \text{L}$, error of $-0.5\ \text{L}$ is $\Delta f = 0.05 = 5\%$, not a problem
  - hamster: $V_{\text{correct}} = 0.5\ \text{L}$, error of $-0.5\ \text{L}$ is $\Delta f = 100\%$, poor hamster!

Mathematical equivalence is not numerical equivalence

- example:
  - $f_1 = x^3 - 6x^2 + 3x - 1$
  - $f_2 = ((x - 6)*x + 3)*x - 1$
    - check numerical difference in Python for $x = 3.82$
    - which is better?
- more arithmetic operations mean more round-off error and longer run-time