Floating Point Precision

Excursus

Representation of Numbers

- Unsigned integers are traditionally represented as a string of zeroes and ones in the 2-adic system
  - E.g. 0100 0111 = $1 \times 2^6 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 64 + 4 + 2 + 1 = 71$
- Integers need to incorporate a sign ±
  - Explicit sign ("signed magnitude representation") leads to inefficient hardware addition / subtraction
  - Use two's complement or one's complement

Representation of Numbers

- In hardware, integers take up 16, 32, 64, or even 128 bits
- Limits range to
  - 32768 (16 bits)
  - 2147483648 (32 bits)
  - 9223372036854775808 (64 bits)
  - 170141183460469231731687303715884105728 (128b)
- But larger integers are used

Representation of Numbers

- Overflow:
  - An result of an addition / subtraction / multiplication exceeds the range
  - Depending on platform, can be misinterpreted:
    - E.g.: Adding two large numbers results in a negative number

Representation of Numbers

- Arbitrary precision integers
  - Overflow happens because results of calculations do not fit into the number of bits assigned for integers
    - This can be a function of the architecture
    - Arbitrary precision integers combine storage for several integers to store a single integer
- Python uses arbitrary precision integers

Representation of Numbers

- Example:

```
================================ RESTART: Shell ================================
>>> 2**2**2**2**2
    Squeezed text (247 lines).
>>>
```

Representation of Numbers

- Example:
  - Double click on the message

```
================================ RESTART: Shell ================================
>>> 2**2**2**2**2
    200352993040684646497907235156025575044782547556975141926501697371089405955631148
    950613088093334810103823434290726318182294938211881266886950636476154702916504l
    ...
```

- We just calculate 2^{65536}.

Representation of Numbers

- Python does this automatically
  - Result: Integer calculation in Python are always exact
- **Nota Bene:**
  - There are Python modules that do not use arbitrary precision integers

Representation of Floating Point Numbers

- Rational numbers are represented as floating point numbers
  - Stored as sign significant $\times$ base$^{\text{exponent}}$
  - You should know this as the scientific notation for the decimal numbers
    - E.g. Planck's constant $6.62607004 \times 10^{-34} \frac{\text{m}^2\text{kg}}{\text{sec}}$

Representation of Floating Point Numbers

- The significant and the exponent can store limited information
  - This means that some numbers cannot be represented exactly
  - In the decadic system:
    - 1/17 is 0.076923076923076913076923076923076923
    - with an infinite repetition of the same pattern
    - Or $\pi = 3.141592653589793\ldots$

Representation of Floating Point Numbers

- Computers (almost universally) use the binary system
  - But we have the same phenomena
- Consequences:
  - Normal mathematical identities are no longer true
    - E.g. $x(y-z) = xy - xz$
    - E.g. $(\sqrt{x})^2 = x$

```
>>> (3**0.5)**2
2.9999999999999996
```

Representation of Floating Point Numbers

- Python (Cython):
  - Uses 8 bytes or 64 bits to represent a floating point number
    - Industry standard for high precision floating point numbers
  - There are packages for C++ or Java available for higher precision numbers
  - Python similarly has wrapper modules that make higher precision available

Representation of Floating Point Numbers

- In theory, it is impossible to use exact precision for floating point calculations