Australian National University

COMP1730/COMP6730
Programming for Scientists

Floating point numbers

Outline

- Numbers in binary and other bases
- Floating point numbers
- Error analysis

Representing Integers

Sequential encoding

* A *sequential encoding system* represents each item (words, numbers, etc) by a sequence of symbols; the order (position) of a symbol in the sequence carries meaning, as much as the symbol itself.
* For example,
  - ” representation ” $\neq$ ”interpret as one”
  - $007 \neq 700$

Positional number system

- The position of a digit is the power of the *base* that it adds to the number.
- For example, in base 10:

$$
\begin{array}{llllll}
 & 1864 & & & \\
= & 1 \text{ thousand} & 8 \text{ hundreds} & 6 \text{ tens} & 4 \text{ ones} \\
= & 1 \times 10^3 & + 8 \times 10^2 & + 6 \times 10^1 & + 4 \times 10^0
\end{array}
$$

- The position of the least significant digit is 0. ($b^0 = 1$ for any base b.)
- The representation of any (non-negative integer) number is unique, except for leading zeros.

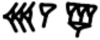
We can count in any base

* For example, in base 3:

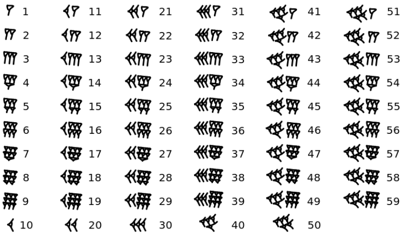
$$\begin{aligned} & 2120001_3 \\ = & \quad 2 \times 3^6 \\ & + 1 \times 3^5 + 2 \times 3^4 + 0 \times 3^3 \\ & + 0 \times 3^2 + 0 \times 3^1 + 1 \times 3^0 \\ = & \ 2 \times 729 + 243 + 2 \times 81 + 1 \\ = & \ 1864 \end{aligned}$$

* Each digit is one of $0, \ldots, b - 1$.
* ("$nnnn_b$" means a number in base b.)

- Ancient Babylonians (*ca* 2,000 BC) counted in base 60.



$$= 31 \times 60^1 + 4 \times 60^0$$
$$= 1864$$



- However, they did not have a symbol for 0:  can mean 1, 60, 3600, $1/60$, etc.

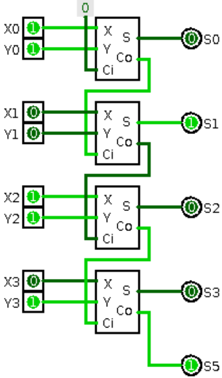
Binary numbers

* Binary numbers are simply numbers in base 2.

$$
\begin{aligned}
& 11101001000_2 \\
= \quad & 1 \times 2^{10} \\
& + 1 \times 2^9 + 1 \times 2^8 + 0 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 \\
& + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\
= \quad & 1024 + 512 + 256 + 64 + 8 \\
= \quad & 1864
\end{aligned}
$$

Bits and bytes

* In the electronic computer, a single binary digit (*bit*) is represented by the presence or absence of current in a circuit element.
* 8 bits make an *octet*, or *byte*.
* Digital hardware works with *fixed-width* number representations ("*words*").
* Common word sizes: 32-bit, 64-bit.



Arithmetic

- Long (multi-digit) addition, subtraction, multiplication, division and comparison (of non-negative numbers) work the same way in any base.

$$\boxed{\begin{array}{l} 0_2 + 0_2 = 0_2 \\ 0_2 + 1_2 = 1_2 \\ 1_2 + 0_2 = 1_2 \\ 1_2 + 1_2 = 10_2 \end{array}}$$

$$\begin{array}{r} {\scriptstyle 1\,1\,1} \\ 0101_2 \\ +\,0111_2 \\ \hline 1100_2 \end{array}$$

$$\begin{array}{lr} & 1001_2 \\ \times & 101_2 \\ \hline & 1001_2 \\ & 00000_2 \\ & 100100_2 \\ \hline & 101101_2 \end{array}$$

Floating point numbers

Representing fractional numbers

* Extend the number system to negative positions; decimal point marks position zero.

$$
\begin{aligned}
& 0.25_{10} \\
= \; & 0 \times 10^0 + 2 \times 10^{-1} + 5 \times 10^{-2} \\
= \; & 0 \times 1 + 2 \times {}^1\!/_{10} + 5 \times {}^1\!/_{100}
\end{aligned}
$$

$$
\begin{aligned}
& 0.01_2 \\
= \; & 0 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} \\
= \; & 0 \times 1 + 0 \times {}^1\!/_2 + 1 \times {}^1\!/_4 \\
= \; & 0.25_{10}
\end{aligned}
$$

* Not every fraction has a finite decimal expansion in a given base.
* For example,
  – $1/3 = 0.3333\ldots$ in base 10
  – $1/5 = 0.001100110011\ldots$ in base 2
  – $1/3 = 0.1$ in base 3.

* Because digital computers work with numbers of fixed width, representation of fractions have *finite precision*.

Floating point representation

* A floating point number in base b,

  $$x = \pm m \times b^e$$

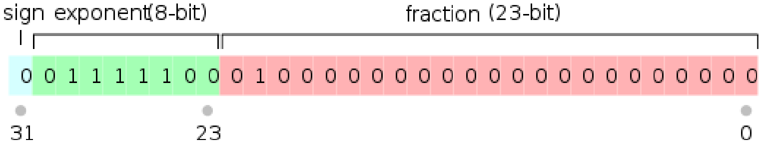
  consists of three components:
  - the sign ($+$ or $-$);
  - the *significand* (m);
  - the *exponent* (e);
* The number is *normalised* iff $1 \leq m < b$.

* Compact (small) representation of numbers far from the decimal point.

  - $1.08 \times 10^9 = 1\overbrace{080000000}^{\rightarrow\times 9}.0$
  - $6.44 \times 10^{-7} = 0.\overbrace{0000006}^{\leftarrow\times 7}44$
  - $1.0000001_2 \times 2^{11110_2} =$

    $1\overbrace{000000100000000000000000000000}^{\rightarrow\times 30}{}_2$

* Floating point types, as implemented in computers, use *fixed-width* binary integer representation of the significand and exponent.
* In a normalised binary number the first digit is 1, so only the fraction is represented ($m = 1.f$).
* The exponent is biased by a negative constant.
* IEEE standard formats:
  – single: 23-bit fraction, 8-bit exponent.
  – double: 52-bit fraction, 11-bit exponent.
* Standard also specifies how to represent 0, $+\infty$, $-\infty$ and `nan` ("not a number").



$$
\begin{aligned}
x &= (-1)^s \, (1.f)_2 \, 2^{(e-127)} \\
&= (-1)^0 \, (1.01)_2 \, 2^{01111100_2 - 127} \\
&= (1 + 1 \cdot 2^{-2}) \, 2^{(64+32+16+8+4)-127} \\
&= (1.25) 2^{-3} = (1.25)/8 = 0.15625
\end{aligned}
$$

(Image from `wikipedia.org`)

* Type `float` can represent infinity:

```
>>> 1 / 1e-320
inf
```

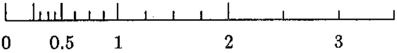
* Most math functions raise an error rather than return `inf`.
  - For example, `1 / 0`, or `math.log(0)`.
* `nan` ("not a number") is a special value used to indicate errors or undefined results.

```
>>> (1 / 1e-320) - (1 / 1e-320)
nan
```

* `math.isinf` and `math.isnan` functions.

Floating point number systems

- A floating point number system (b, p, L, U) is defined by four parameters:
  - the base (b);
  - the *precision*: number of digits in the fraction of the significand (p); and
  - the lower (L) and upper (U) limit of the exponent.
- IEEE double-precision is $(2, 52, -1023, 1024)$ (with some tweaks).

- The numbers that can be represented (exactly) in a floating point number system are not evenly distributed on the real line.
- E.g., $(2, 2, -2, 1)$: 

- E.g., in a $(2, 52, -1023, 1024)$ system,
  - the smallest number > 0 is $2^{-1023} \approx 10^{-308}$,
  - (Actual IEEE double standard can represent numbers down to $\approx 4 \cdot 10^{-324}$.)
  - the smallest number > 1 is $1 + 2^{-52}$ $\approx 1 + 2 \cdot 10^{-16}$.
- Rounding the significand to $p + 1$ digits causes a discrepancy, called the rounding error.

* Because of rounding, mathematical laws do not always hold for floating point arithmetic.

```
>>> a = 11111113.0
>>> b = -11111111.0
>>> c = 7.51111111
>>> (a + b) + c == a + (b + c)
False
>>> ((a + b) + c) - (a + (b + c))
4.488374116817795e-10
```

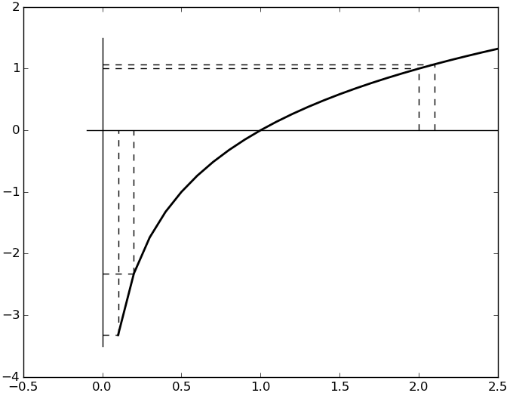
Example from Punch & Enbody

* *(Almost) never compare* `float`*s with* `==`.

Error analysis

* Let x be the true value and $\hat{x}$ the approximate (measured or representable) number.
  - The *absolute error* is $\Delta x = |x - \hat{x}|$.
  - The *relative error* is $\dfrac{\Delta x}{x} = \dfrac{|x - \hat{x}|}{|x|}$.
* Rounding to $p + 1$ digits in base b,
  - the absolute error is $\leq {}^1\!/_2 b^{-p} \cdot b^e$, and
  - the relative error is $\leq {}^1\!/_2 b^{-p}$.

Error propagation



* The absolute error $|f(x) - f(\hat{x})|$ is approximately proportional to $\frac{df}{dx}(x)|x - \hat{x}|$.

- IEEE standard specifies that floating point arithmetic operations (and some other math functions, e.g., $\sqrt{}$) are exact, except for the rounding error in the result.
  - This does *not* mean errors do not propagate.
- If $y = x_1 + x_2$, then $\Delta y = \Delta x_1 + \Delta x_2$
  - Also if either x_1 or x_2 is negative.
- If $y = x_i \times x_2$, then $\Delta y = x_2 \times \Delta x_1 + x_1 \times \Delta x_2$ $+\Delta x_1 \times \Delta x_2$.

- Example, continued:
  - $a = 1.1111113 \cdot 10^7$, $b = -1.1111111 \cdot 10^7$ and $c = 7.51111111$.
  - $y = b + c = -1.11111108511111111 \cdot 10^7$.
  - $\Delta y \leq 2^{-53} \cdot 10^7 \approx 1.1 \cdot 10^{-9}$ (assuming double precision and no error other than rounding).
  - $a + (b + c) = a + y \pm \Delta y$ (plus rounding error).
- When adding floating point numbers, the absolute rounding error is proportional to the magnitude of the largest number that is rounded.