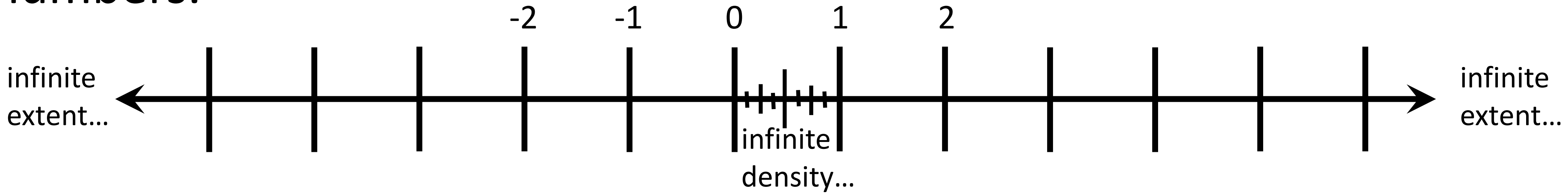
CS370 Lecture 1: Floating Point Systems

Topics

- Floating Point Systems
- Absolute and Relative Error
- Cancellation and Round-off Errors
- Conditioning of Problems
- Stability of Algorithms

Real numbers, $\mathbb{R}$, are…

- Infinite in *extent*: There exists x such that $|x|$ is arbitrarily large.
- Infinite in *density:* Any interval $a \leq x \leq b$ contains infinitely many numbers.



But computers cannot represent such infinite quantities!

The standard (partial) solution is to use ***floating point numbers*** to approximate the reals.

Floating Point Systems

An approximate representation of real numbers using a finite number of bits.

Questions to ponder:

How can we represent real numbers digitally?

How does the resulting “approximate” number system behave?

Why is this important?

Numerical Disasters

Numerical errors can have severe consequences:

- Feb. ’91: A US Patriot missile to failed to stop an incoming Iraqi scud missile, killing 28 soldiers.
- June ‘96: First Ariane 5 rocket exploded shortly after lift-off. Value: $500 million.
- 1982: Vancouver stock exchange was off by factor of about 2 due to rounding error.

More examples: http://ta.twi.tudelft.nl/users/vuik/wi211/disasters.html

Numerical Errors: Toy Example

Consider the sum:

$$12 + \sum_{i=1}^{100} 0.01$$

True answer: 13.

Now, perform the sum one add at a time, retaining two digits of accuracy at each step.

$$((12 + 0.01) + 0.01) + 0.01) + 0.01) + \cdots$$

What is the sum after each step? And at the end?

Numerical answer: 12. Wrong!!

Numerical Errors: Taylor series example

Say we want to evaluate $e^{-5.5}$.

Recall the Taylor series for a function f:

$$f(x) = f(a) + (x-a)f'(a) + \frac{(x-a)^2}{2!}f''(a) + \frac{(x-a)^3}{3!}f'''(a) + \cdots$$

Apply this to $f(x) = e^x$ with $a = 0$, gives

$$e^x = e^0 + (x-0)e^0 + \frac{(x-0)^2}{2}e^0 + \frac{(x-0)^3}{6}e^0 + \cdots$$

Numerical Errors: Taylor series example

So the Taylor series approximation gives: $e^x \approx 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$

With 5 digits of accuracy, we find:

$$e^{-5.5} \approx 1 - 5.5 + 15.125 - 27.730 + \cdots = 0.0026363$$

Wrong!!

But try it on your calculator/phone/computer/abacus…

$$e^{-5.5} = 0.0040868 \text{ (rounded to 5 digits)}$$

Numerical Errors: Taylor series example

We could also note that: $e^{-x} = \frac{1}{e^x} = \frac{1}{1+x+\frac{x^2}{2!}+\frac{x^3}{3!}+\cdots}$

Again with 5 digits accuracy, we eventually get: $\mathrm{e}^{-5.5} \approx 0.0040865$.
This is *much* closer to the desired solution, $\mathrm{e}^{-5.5} = 0.0040868$.

Why was our first result so much worse?

Numerical Errors: Recurrence example

Consider the integration problem

$$I_n = \int_0^1 \frac{x^n}{x+\alpha}\,dx$$

where α is some fixed parameter.

What went wrong?

Floating point numbers often don’t *quite* behave like true real numbers. This can lead to *subtle* (yet huge) errors!

To be useful, our numerical algorithms must work effectively (accurately, safely, correctly, efficiently) under *floating point arithmetic*.

Let’s examine the floating point representation.

Normalized Digital Expansions of Reals

We can express a real number as an *infinite expansion* relative to some *base*.

e.g., consider $\frac{73}{3} \approx 24.3333\ldots$

In base 10:

$$\frac{73}{3} = 0.243333\ldots \times 100 = 2 \times 10^1 + 4 \times 10^0 + 3 \times 10^{-1} + 3 \times 10^{-2} + \cdots$$

In base 2:

$$\frac{73}{3} = 0.11000010101\ldots \times 2^5 = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + \cdots$$

Normalized Form of a Real number

Thus the **normalized form** of a real number has the form

$$0.d_1d_2d_3d_4 \ldots \times \beta^p$$

Where…

- d_i are digits in base β, i.e., $0 \leq d_i < \beta$.
- *normalized* implies $d_1 \neq 0$.
- exponent p is an integer.

How can we limit things to avoid storing infinite data?

Careful: You may see other normalization conventions outside this class, e.g., $d_1.d_2d_3d_4 \ldots \times \beta^p$

Floating Point

Density (or *precision)* is bounded by limiting the number of digits, t.
Extent (or *range)* is bounded by limiting the range of values for p.

Our floating point representation then has the form:

$$\pm 0.d_1 d_2 .. d_t \ldots \times \beta^p$$

for $L \leq p \leq U$, and $d_1 \neq 0$.

or

0 as a special case.

The four integer parameters $\{\beta, t, L, U\}$ characterize a specific floating point system, F.

Overflow/underflow

- If the exponent p is too big (>U) or too small (<L), our system *cannot represent the number*.
- When arithmetic operations generate such a number, this is called overflow or underflow, respectively.

Example #1:

Express 253.9 in floating point with base $\beta = 10, t = 6$ digits, $L = -5, U = 5$.

Process:

1) Express in base $\beta = 10$: 253.9

2) Shift to get leading 0, set exponent p: 0.2539$\times 10^3$, $p = 3$.

3) Pad or round/truncate to t digits: 0.253900×10^3.

Example #2:

Express 8.25 in floating point with base $\beta = 2, t = 7$ digits, $L = -5, U = 5$.

Process:

1) Express in base $\beta = 2$: 1000.01

2) Shift to get leading 0, set exponent p: $0.100001 \times 2^4, p = 4$.

3) Pad or round/truncate to t digits: 0.1000010×2^4.

Floating Point Standards

The two most common standardized floating point systems are:

IEEE single precision (32 bits): $\{\beta = 2, t = 24, L = -126, U = 127\}$
IEEE double precision (64 bits): $\{\beta = 2, t = 53, L = -1022, U = 1023\}$

Almost always implemented directly in (CPU/GPU/etc.) hardware.

Prior to 1980's, lack of FP standards contributed to additional (programming) errors in numerical computing.

Summary

- Floating point numbers are an approximation to the real numbers.
- We limit the range (“size”) and precision (“digits”) to keep things finite.
- Treating FP representations *carelessly* can lead to major problems.