Numerics

Geometric Algorithms
Lecture 2

CAS CS 132

Recap (1/2)

Linear equations define hyperplanes

Systems of linear equations define intersections of hyperplanes

We solve systems linear equations by elimination and back substitution

Systems of linear equations can be represented as matrices

Recap (2/2)

elimination and back-substitution can be represented as <u>row operations</u> on matrices

row operations don't change the solution sets

Recap Problem (1/2)

Show that if (s_1, s_2) is a solution to

$$ax + by = c$$
$$dx + ey = f$$

then it is also a solution to

$$ax + by = c$$
$$(a + d)x + (b + e)y = (c + f)$$

Recap Problem (2/2)

Give values of a through f such that

$$(a+d)x + (b+e)y = (c+f)$$

has a solution but

$$ax + by = c$$
$$dx + ey = f$$

does not

don't drop equations when doing replacements

Objectives

1. `number representations`
2. `consequences of floating point representations`
3. `best practices`

Keywords

`floating point numbers`

`IEEE-754`

`relative error`

`numpy.isclose`

`ill-conditioned problems`

let's do a quick demo

Significant Figures (Sig Figs)

Have you ever been docked points in a science class for having incorrect sig figs?

when you use a ruler, you can't do better than ± 1mm, so we can't say anything about nanometer differences

we run into a similar problem with decimal numbers in programs

Number Representations

your computer is a collection of fixed size registers

each register holds a sequence of bits

The Goal. represent numbers so they fit in those registers

this is, of course, ~~a lie~~ an abstraction

Number Representations

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Question. How do we slice up our fixed sequence to represent numbers?

things to consider:

- simple idea (easy to understand)
- maximize coverage (not too redundant)
- simple numeric operations (easy to use)

Unsigned Integers

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

value

binary value (we should know this by now)

e.g. 10001010 represents

$$1(2^7) + 0(2^6) + 0(2^5) + 0(2^4) + 0(2^3) + 1(2^2) + 0(2^1) + 1(2^0)$$

Signed Integers

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

sign value

sign bit + binary value

e.g. 10001010 represents

$$-1 \times \left(0(2^6) + 0(2^5) + 0(2^4) + 0(2^3) + 1(2^2) + 0(2^1) + 1(2^0)\right)$$

Floating-Point Numbers (Some Figures)

`floats in python use` <u>`64 bits`</u>

`That's` 1.8×10^{19} `possible values`

`We can't represent everything. We'll have to choose and then round`

`Question.` `Which ones should we represent?`

Floating-Point Numbers (An Idea)

Integers work because they are **discrete and evenly spaced**

What if we evenly discretize a range of values?

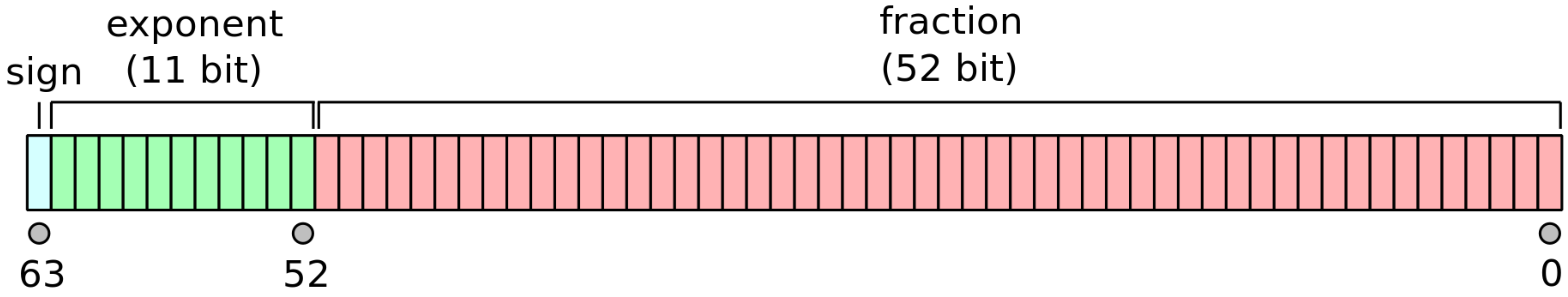
i.e., represent

..., –0.001, 0, 0.0001, 0.002, 0.003, 0.004,...

Question

Discuss the advantages and disadvantages of this approach

Floating-Point Numbers (IEEE-754)



`like scientific notation, but binary`

<u>`the equation:`</u>

$$(-1)^{\text{sign}} \times \left(1 + \frac{\text{fraction}}{2^{52}}\right) \times 2^{\text{exponent}-(2^{10}-1)}$$

`it's an accepted standard, not perfect, but it works well`

<u>`image source`</u>

Question

$$(-1)^{\text{sign}} \times \left(1 + \frac{\text{fraction}}{2^{52}}\right) \times 2^{\text{exponent}-(2^{10}-1)}$$

Any ideas why this is better/worse?

Also, why the additive 1?

And why not have a sign bit for the exponent?

Step Size

$$(-1)^{\text{sign}} \times \left(1 + \frac{\text{fraction}}{2^{52}}\right) \times 2^{\text{exponent}-(2^{10}-1)}$$

⟵ step size increases with magnitude ⟶

Definition. <u>step size</u> is the space between two floating-point representations

for fixed exponent n two numbers are at least

$$0.00\ldots001 \times 2^n = 2^{-52} \times 2^n$$

away (why?)

Step size <u>doubles</u> for each exponent

<u>image source</u>

What to Keep in Mind

IEEE-754 defines a subset of decimal numbers

operations on floating point numbers attempt to give you the closest to the actual value, though there will be errors.

we can assume when we write down a number like '0.3' we get the closest IEEE-754 value

Relative Error

Observation. ± 0.001 is *tiny* error for 10^{20} but *massive* for 10^{-20}

Relative Error.

$$\mathrm{err}_{\mathrm{rel}} = \frac{\mathrm{err}}{\mathrm{val}}$$

IEEE-754 keeps relative error <u>small</u>

Relative Error (Calculation)

$$(-1)^{\text{sign}} \times \left(1 + \frac{\text{fraction}}{2^{52}}\right) \times 2^{\text{exponent}-(2^{10}-1)}$$

`(fix an exponent` n`)`

`error is determined by step-size`

$$\text{err} \leq 2^{-52} \times 2^{n}$$

Relative Error (Calculation)

$$(-1)^{\text{sign}} \times \left(1 + \frac{\text{fraction}}{2^{52}}\right) \times 2^{\text{exponent}-(2^{10}-1)}$$

(fix an exponent n)

the smallest number we can represent at least 1.0×2^n

$$\text{val} \geq 1.0 \times 2^n$$

(why do we care about a lower bound on val?)

Relative Error (Calculation)

$$(-1)^{\text{sign}} \times \left(1 + \frac{\text{fraction}}{2^{52}}\right) \times 2^{\text{exponent}-(2^{10}-1)}$$

`(fix an exponent` n`)`

`the relative error is` *`small`*

$$\text{val} \geq 1.0 \times 2^n$$

$$\text{err} \leq 2^{-52} \times 2^n$$

$$\text{err}_{\text{rel}} = \frac{\text{err}}{\text{val}} \leq \frac{2^{-52} \times 2^n}{1.0 \times 2^n} = 2^{-52} \approx 10^{-16}$$

$\approx$16 digits of accuracy

Not bad, but also not great

let's do a quick demo

example from the notes

The Takeaways

operations on floating-point numbers are not exact

properties like $(ab)c = a(bc)$ (commutativity) may not hold

it's a trade-off for large range and low relative error

What do we do about it?

Best Practices

1. `don't compare floating points for equality`
2. `be aware of ill-conditioned problems`
3. `be aware of small differences`

Principle 1: Closeness

When doing floating-point calculations in a program, define an error margin and use that for equality checking

In Practice.

```
Replace    x == y
with       numpy.isclose(x, y)
```

demo

Principle 2: Ill-Conditioned Problems

Make sure your problem is not sensitive to small errors.

In Practice. for example, don't divide by numbers much smaller than your error tolerance

demo

Principle 3: Small Differences

Make sure you understand your error tolerance when looking that the small differences of large numbers.

In Practice. Don't expect $a - b$ to have 16 digits of accuracy even if a and b do

demo

One Last Note: Special Numbers

```
0       (we can't already represent 0?)

nan     stands for not a number, .e.g, sqrt(-2)

inf     symbolic infinity, behaves as expected
```

Summary

floating point numbers are represented in your computer

floating point operations are not exact

this can have unintended consequences

we get 16 digits of accuracy