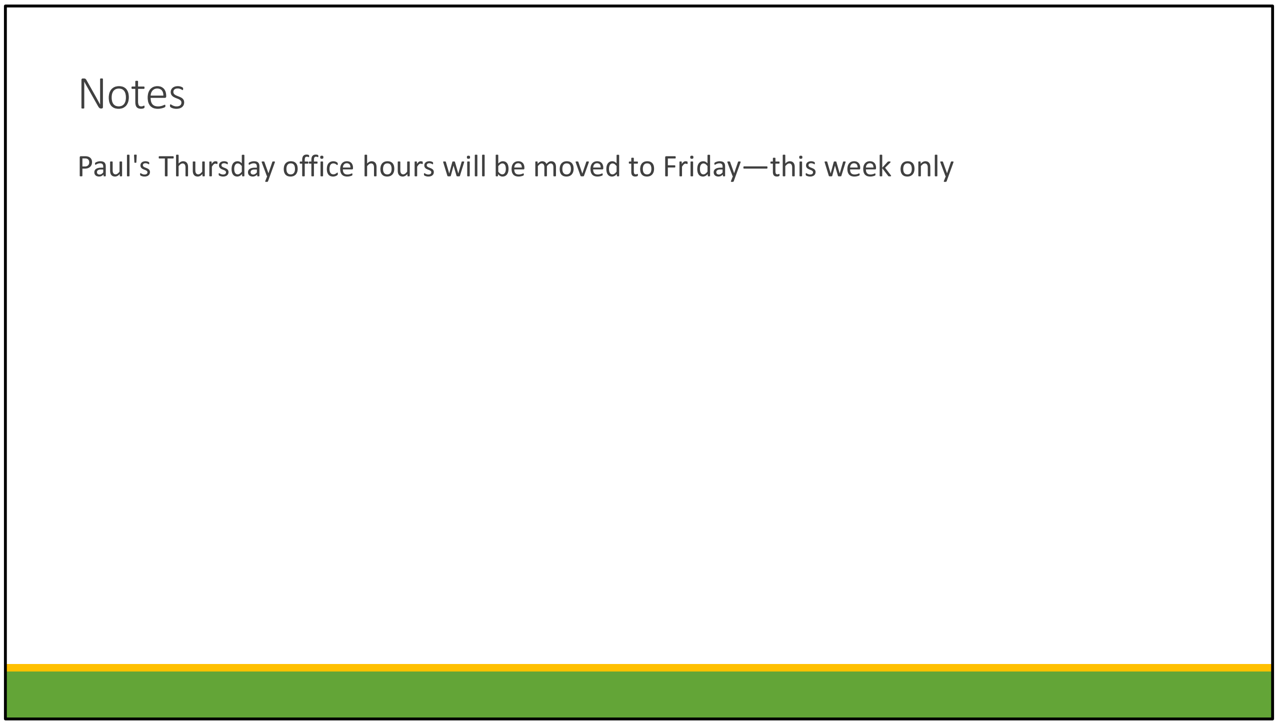


Notes

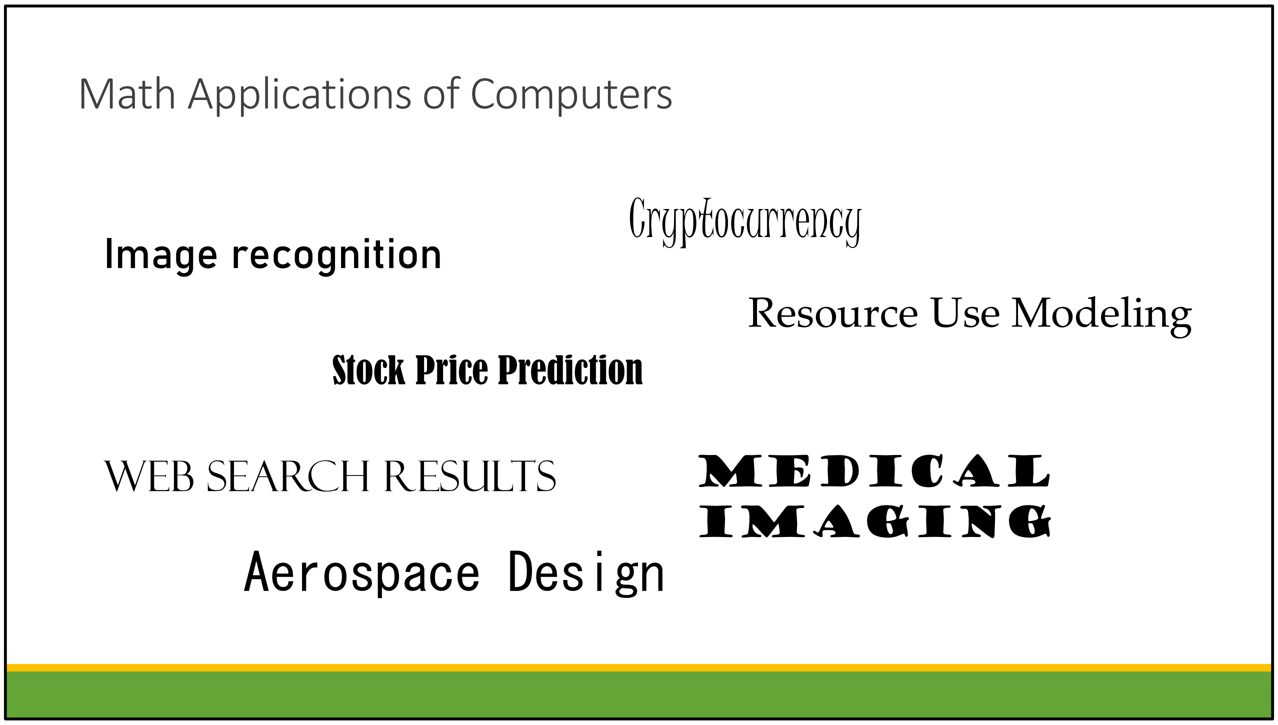
Paul's Thursday office hours will be moved to Friday—this week only

Go over Quiz04. Go over SimpleFloat after lecture!!

Computational Numerical Accuracy

Paul Haskell

CS112



Math Applications of Computers

Image recognition

Cryptocurrency

Resource Use Modeling

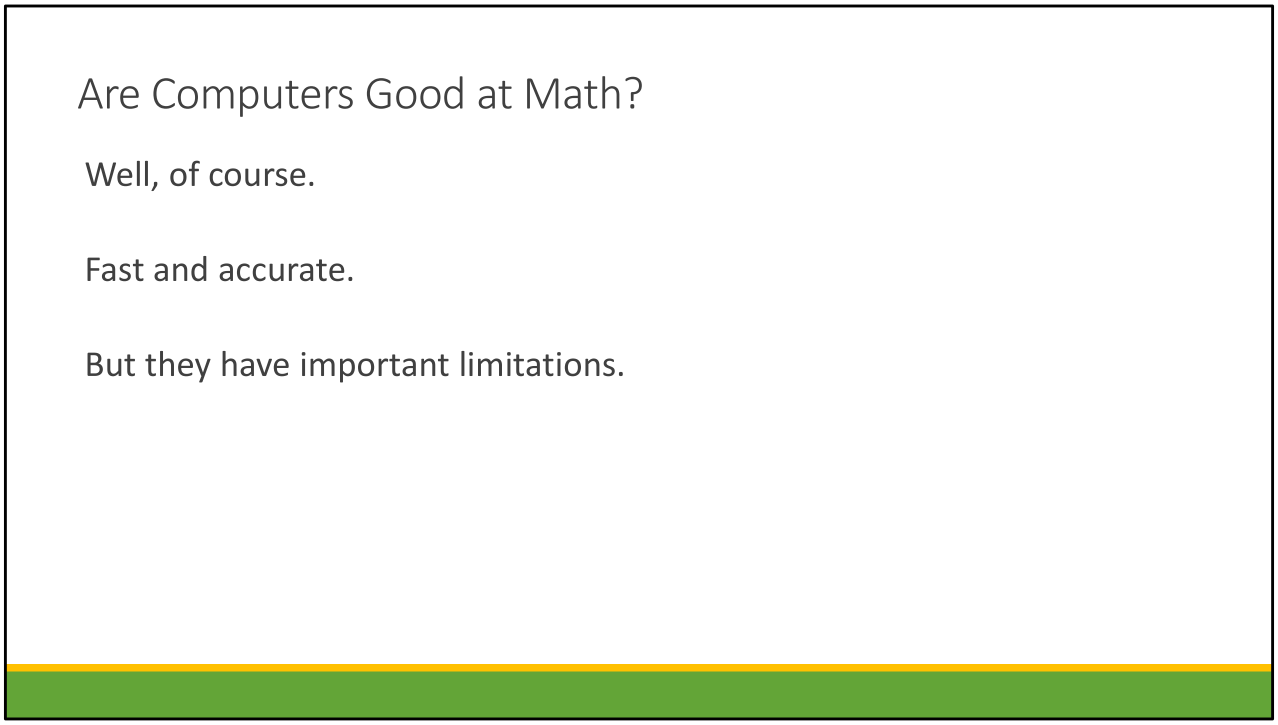
Stock Price Prediction

WEB SEARCH RESULTS

MEDICAL IMAGING

Aerospace Design

Computers used for lots of things: user interfaces, communication, data storage and retrieval. But of course math!
These applications all use computers because they require a lot of Number Crunching



Are Computers Good at Math?

Well, of course.

Fast and accurate.

But they have important limitations.

Computers used for lots of things: user interfaces, communication, data storage and retrieval. But of course math!
These applications all use computers because they require a lot of Number Crunching

Consider this Math Problem

- 71x - 43y = -17
- 72x - 42y = -6

Solution? x=4, y=7

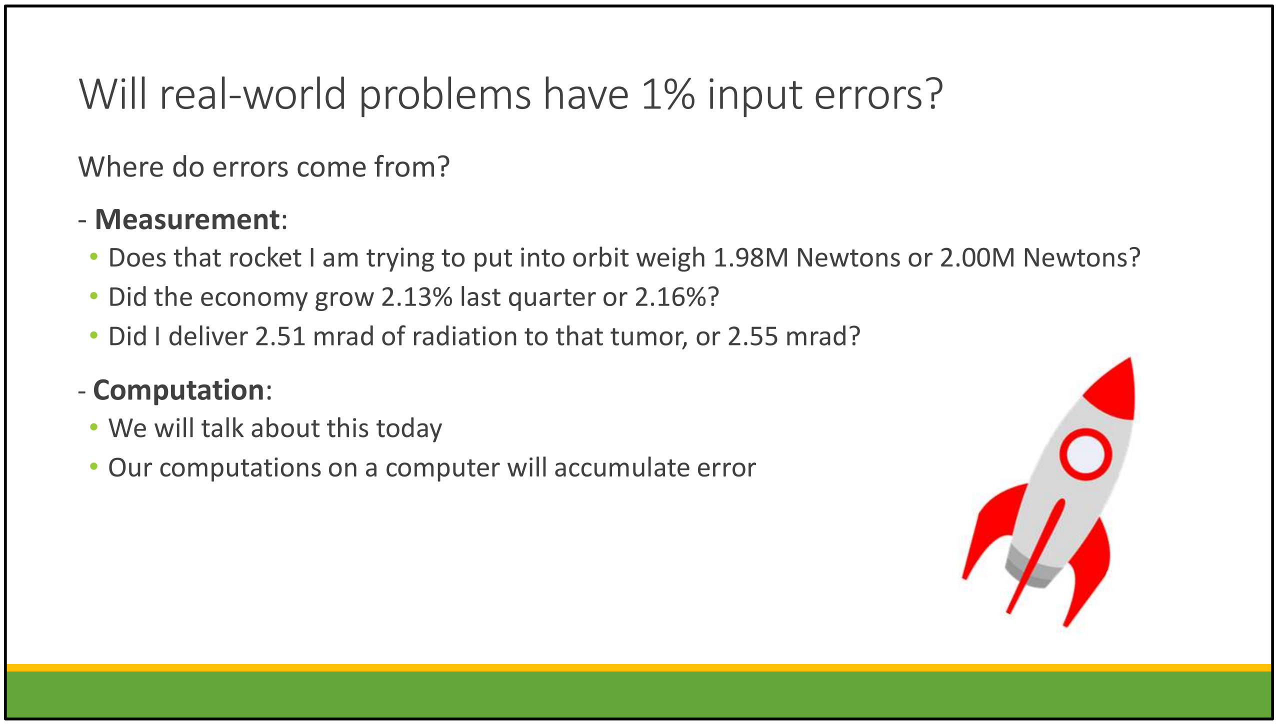
How about this similar problem?

- 71x - 42.5y = -17
- 72x - 42y = -6

Similar answer?

x = 5.8846…, y = 10.2308…

1% change in one input coefficient, 47% change in solution



Will real-world problems have 1% input errors?

Where do errors come from?

- **Measurement**:
  - Does that rocket I am trying to put into orbit weigh 1.98M Newtons or 2.00M Newtons?
  - Did the economy grow 2.13% last quarter or 2.16%?
  - Did I deliver 2.51 mrad of radiation to that tumor, or 2.55 mrad?

- **Computation**:
  - We will talk about this today
  - Our computations on a computer will accumulate error

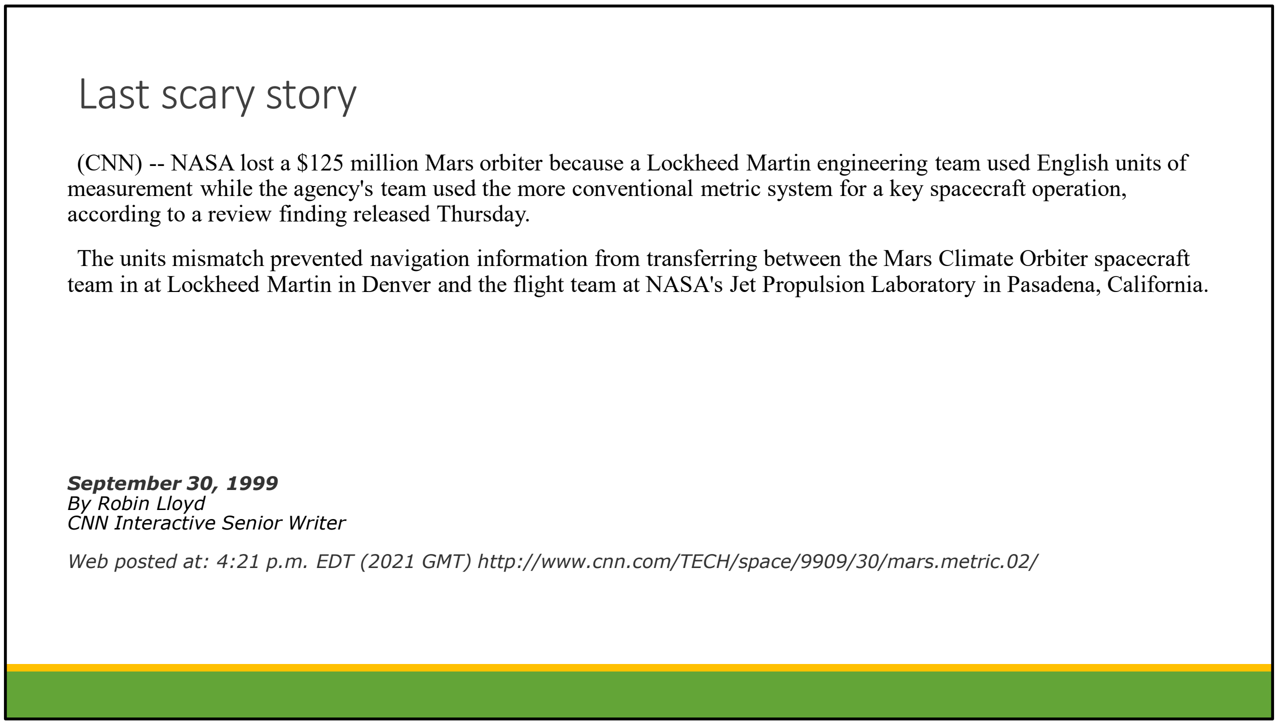
(Almost) Main Point: Any real-world numerical problem we solve with a computer will have some inaccuracy in the problem's numbers.

But first…scary stories

Patriot Missile Failure

On February 25, 1991, during the Gulf War, an American Patriot Missile battery in Dharan, Saudi Arabia, failed to intercept an incoming Iraqi Scud missile. The Scud struck an American Army barracks and killed 28 soldiers. A report of the General Accounting office, GAO/IMTEC-92-26, entitled *Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia* reported on the cause of the failure. It turns out that the cause was an inaccurate calculation of the time since boot due to computer arithmetic errors. Specifically, the time in tenths of second as measured by the system's internal clock was multiplied by 1/10 to produce the time in seconds. This calculation was performed using a 24 bit fixed point register. In particular, the value 1/10, which has a non-terminating binary expansion, was chopped at 24 bits after the radix point. The small chopping error, when multiplied by the large number giving the time in tenths of a second, lead to a significant error. Indeed, the Patriot battery had been up around 100 hours, and an easy calculation shows that the resulting time error due to the magnified chopping error was about 0.34 seconds…A Scud travels at about 1,676 meters per second, and so travels more than half a kilometer in this time.

This description is adapted from The Patriot Missile Failure by Douglas N. Arnold



Last scary story

(CNN) -- NASA lost a $125 million Mars orbiter because a Lockheed Martin engineering team used English units of measurement while the agency's team used the more conventional metric system for a key spacecraft operation, according to a review finding released Thursday.

The units mismatch prevented navigation information from transferring between the Mars Climate Orbiter spacecraft team in at Lockheed Martin in Denver and the flight team at NASA's Jet Propulsion Laboratory in Pasadena, California.

September 30, 1999
By Robin Lloyd
CNN Interactive Senior Writer

Web posted at: 4:21 p.m. EDT (2021 GMT) http://www.cnn.com/TECH/space/9909/30/mars.metric.02/

Did NASA forget to convert miles to km? No. They converted with INSUFFICIENT ACCURACY.

Poorly conditioned problems…

There's something special about the first problem

- 71x - 43y = -17
- 72x - 42y = -6

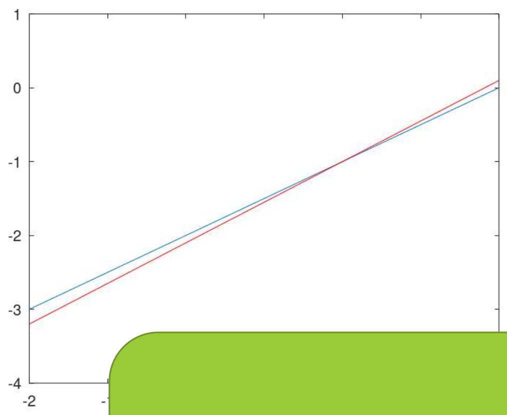
A problem in which small changes in input lead to big changes in output is called "poorly conditioned".

We define a "condition number"

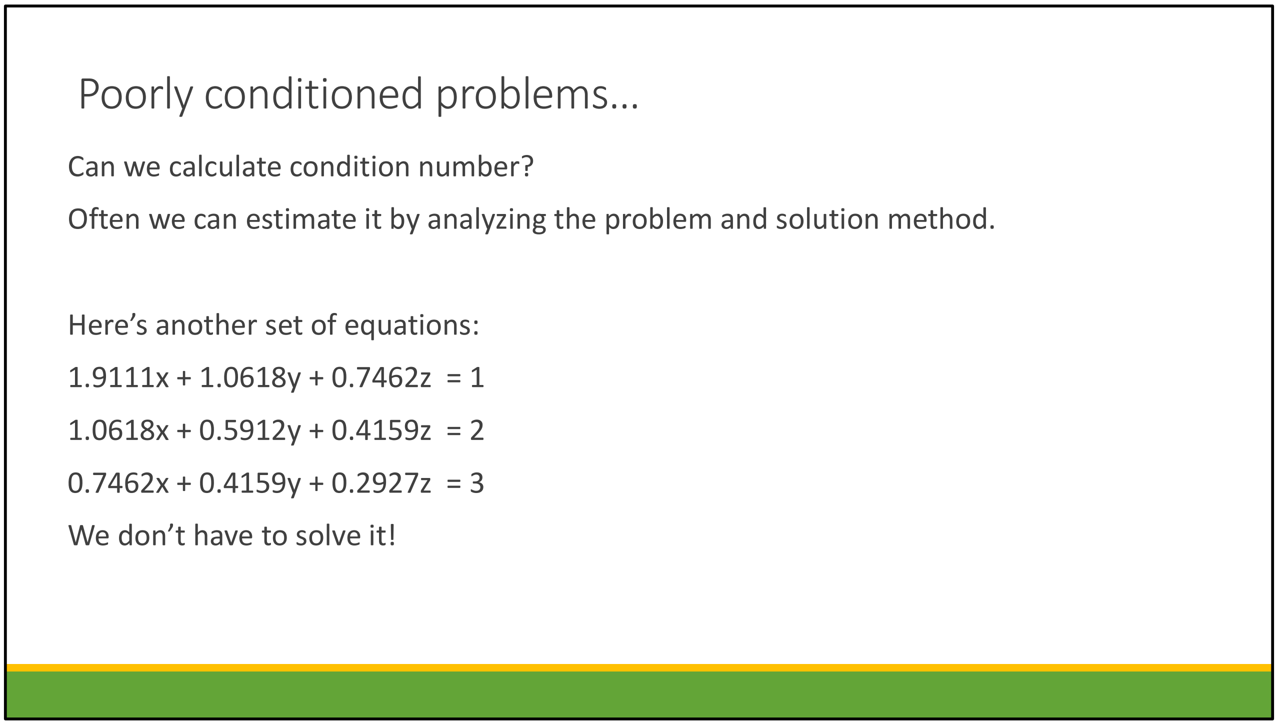
- max$\left|\frac{Relative\ change\ in\ result}{Relative\ change\ in\ input}\right|$

"Relative" change?

- $\frac{\Delta x}{x}$



For our problem, condition number = 177



Poorly conditioned problems…

Can we calculate condition number?

Often we can estimate it by analyzing the problem and solution method.

Here’s another set of equations:

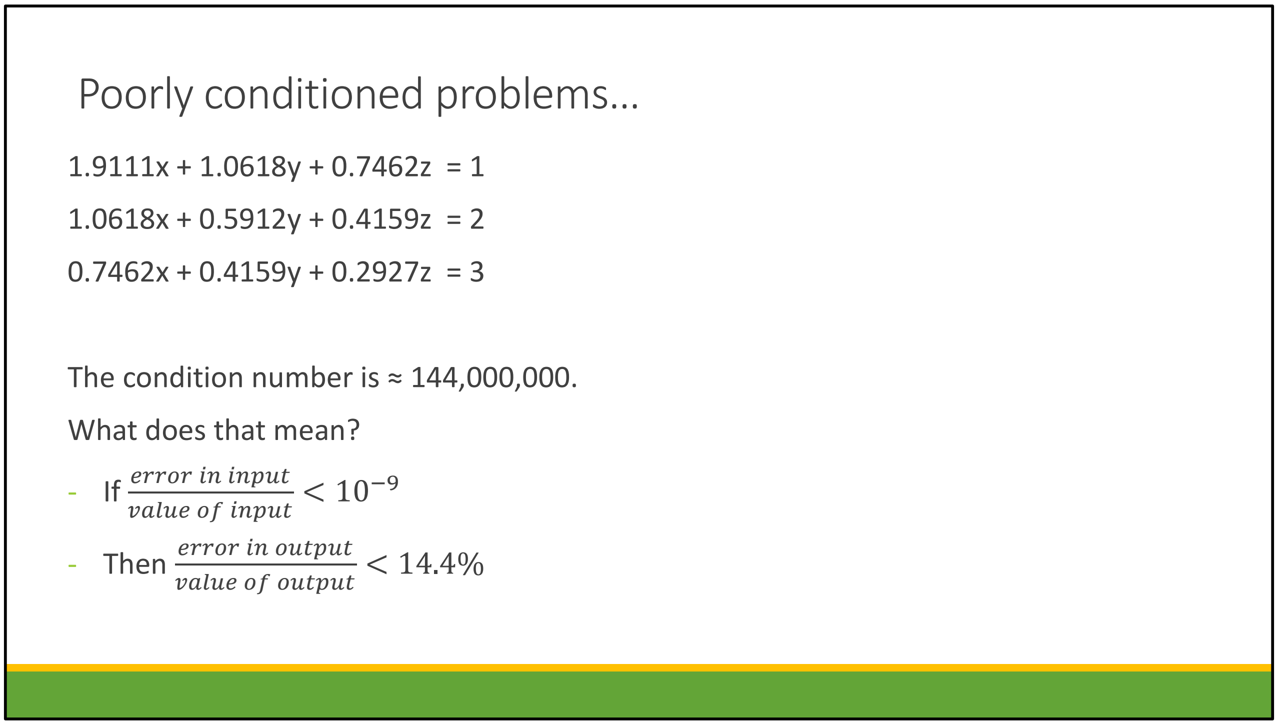
$1.9111x + 1.0618y + 0.7462z = 1$

$1.0618x + 0.5912y + 0.4159z = 2$

$0.7462x + 0.4159y + 0.2927z = 3$

We don’t have to solve it!

x=5.0391e6, y=-2.8177e7, z=2.7189e7



Poorly conditioned problems…

1.9111x + 1.0618y + 0.7462z = 1

1.0618x + 0.5912y + 0.4159z = 2

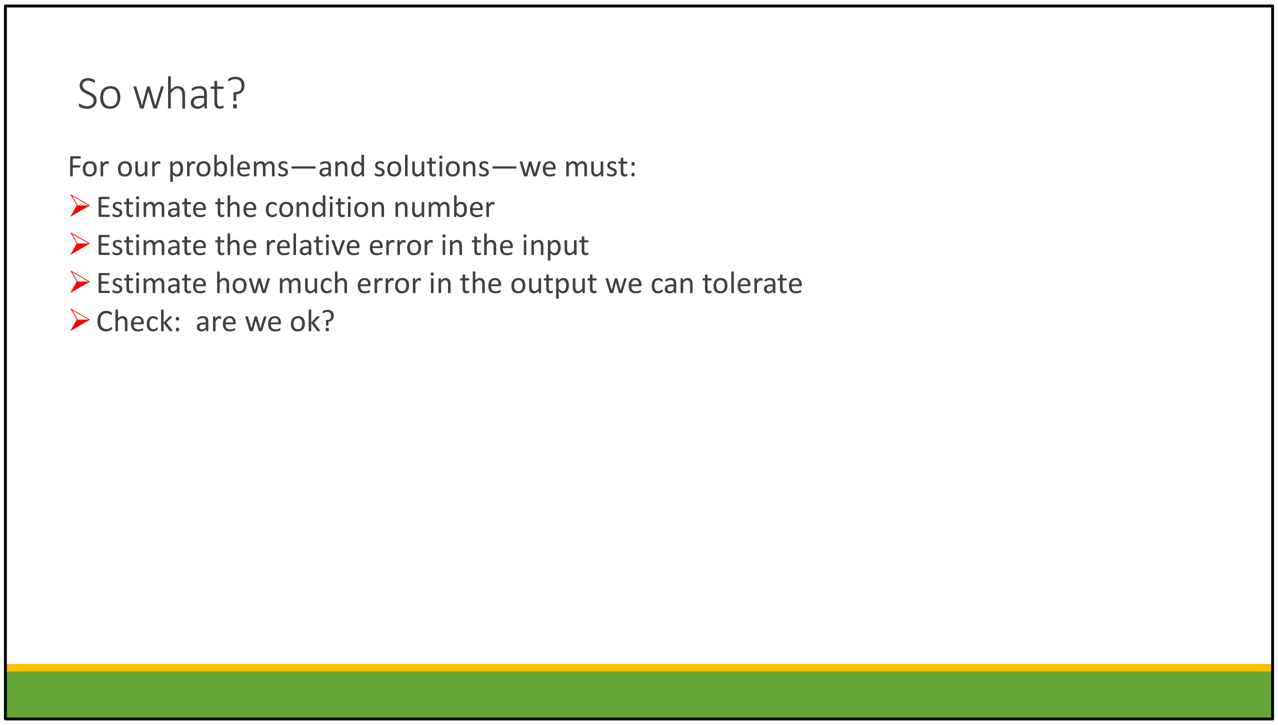
0.7462x + 0.4159y + 0.2927z = 3

The condition number is ≈ 144,000,000.

What does that mean?

- If $\frac{error\ in\ input}{value\ of\ input} < 10^{-9}$
- Then $\frac{error\ in\ output}{value\ of\ output} < 14.4\%$

The source of error could be measurement. Or numerical inaccuracy in our computer's math computations.



So what?

For our problems—and solutions—we must:

- Estimate the condition number
- Estimate the relative error in the input
- Estimate how much error in the output we can tolerate
- Check: are we ok?

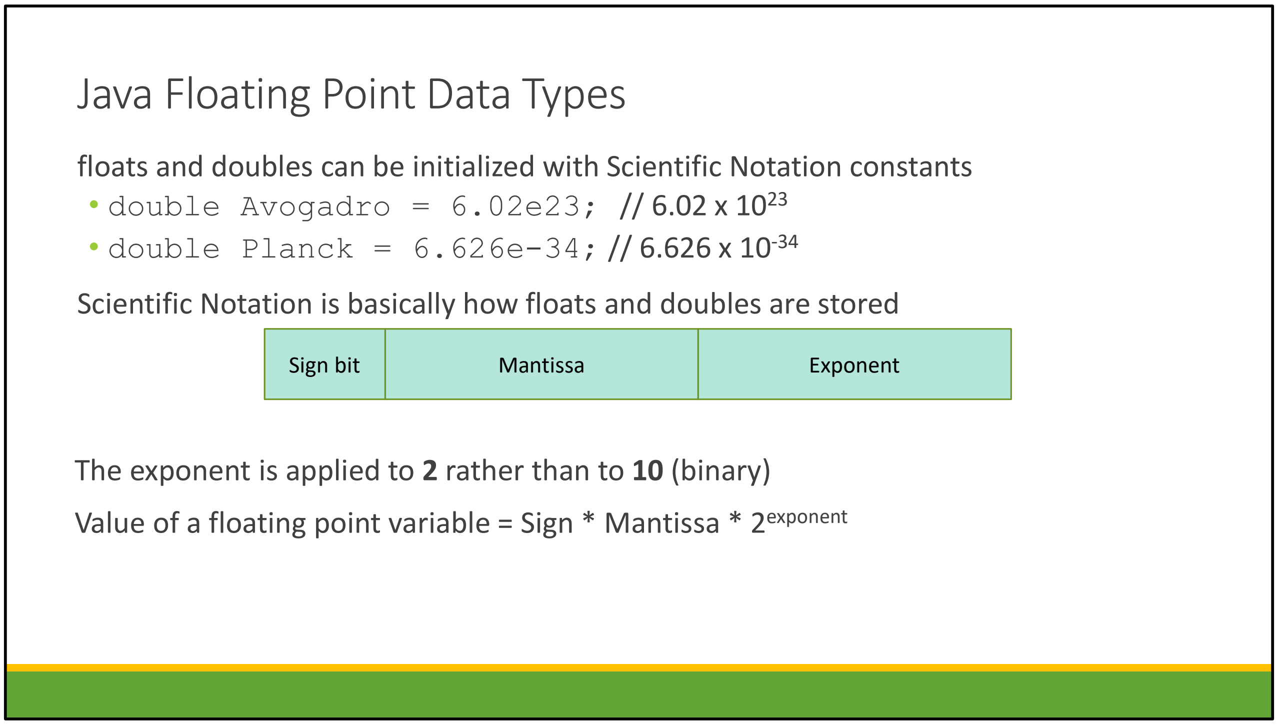
MAIN Point: If we are doing numerical calculations, we MUST understand input error, condition number, output error, and our TOLERANCE for output error.
Is a 14% error too big? Yes if we are flying to Mars. No if we are buying hot dogs for a picnic.

Remember…

One source of error is inaccuracy in the numbers you are given – “measurement error”

Another source comes from actual computation. Usually this is << than measurement errors. But not in our two examples!

Let's dive in



Java Floating Point Data Types

floats and doubles can be initialized with Scientific Notation constants

- `double Avogadro = 6.02e23;` // 6.02 x 10^{23}
- `double Planck = 6.626e-34;` // 6.626 x 10^{-34}

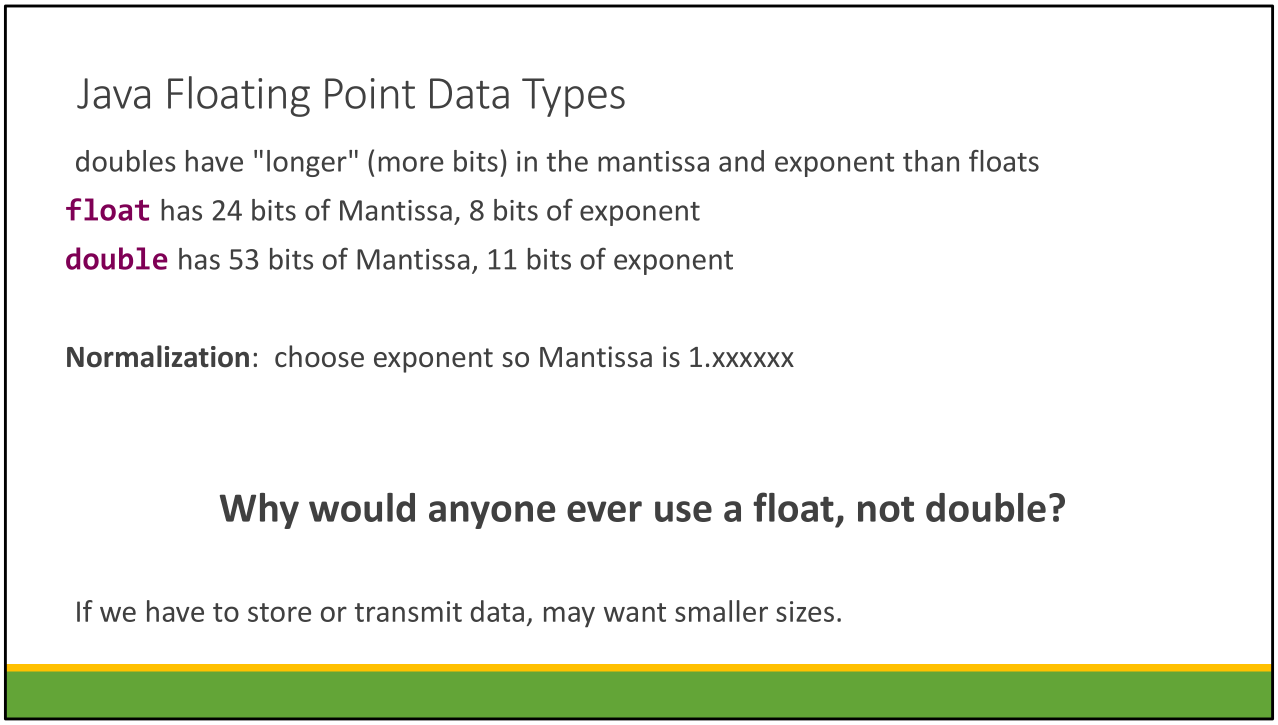
Scientific Notation is basically how floats and doubles are stored

Sign bit	Mantissa	Exponent

The exponent is applied to **2** rather than to **10** (binary)

Value of a floating point variable = Sign * Mantissa * 2^{exponent}

Mantissa is not-signed binary number. Unlike 2's complement
This format is used by every computer language I know about, not just Java.



Java Floating Point Data Types

doubles have "longer" (more bits) in the mantissa and exponent than floats

`float` has 24 bits of Mantissa, 8 bits of exponent

`double` has 53 bits of Mantissa, 11 bits of exponent

Normalization: choose exponent so Mantissa is 1.xxxxxx

Why would anyone ever use a float, not double?

If we have to store or transmit data, may want smaller sizes.

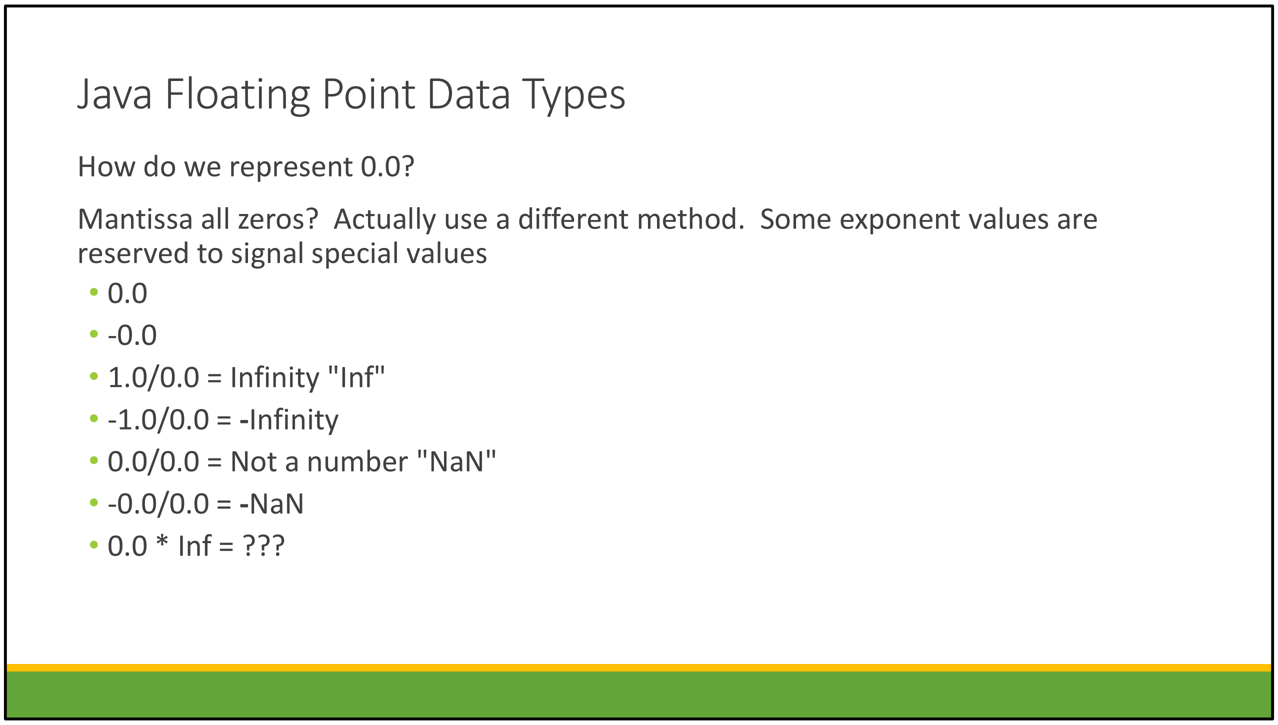
My first computer was a Commodore 64. Anyone know what the 64 meant?

Scott and multi GB files
Me and video processing: use bytes

Java Floating Point Data Types

This structure lets us use standard math rules to perform +, -, *, /

CPUs perform floating point math in hardware, just like with "fixed point"



Java Floating Point Data Types

How do we represent 0.0?

Mantissa all zeros? Actually use a different method. Some exponent values are reserved to signal special values

- 0.0
- -0.0
- 1.0/0.0 = Infinity "Inf"
- -1.0/0.0 = -Infinity
- 0.0/0.0 = Not a number "NaN"
- -0.0/0.0 = -NaN
- 0.0 * Inf = ???

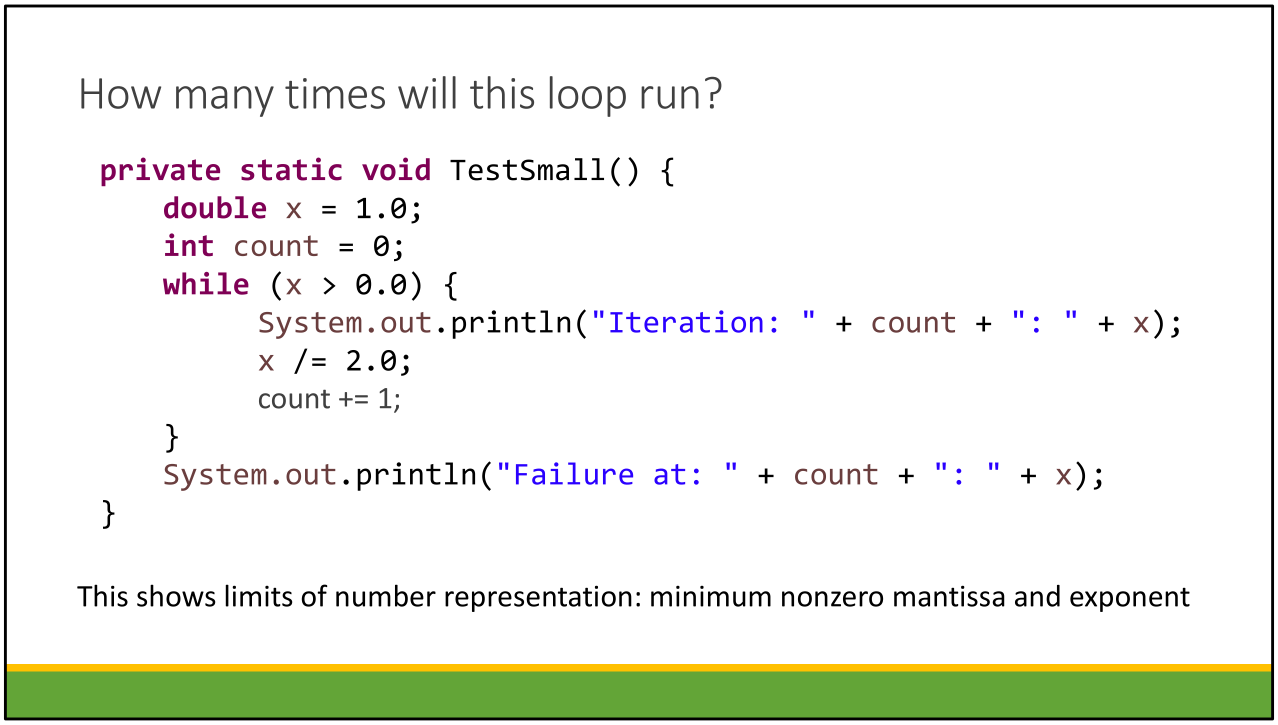
What is NaN? 0/0, 0*inf

Floating Point Number Representation

Example:

- $+8.0 = (-1)^0 \times$ ‘1.00000000…’ $\times 2^3$
- $-8.5 = (-1)^1 \times$ ‘1.00010000…’ $\times 2^3$
- +5.0?

$+5 = (-1)^0 \times$ ‘1.0100000…’ $\times 2^2$

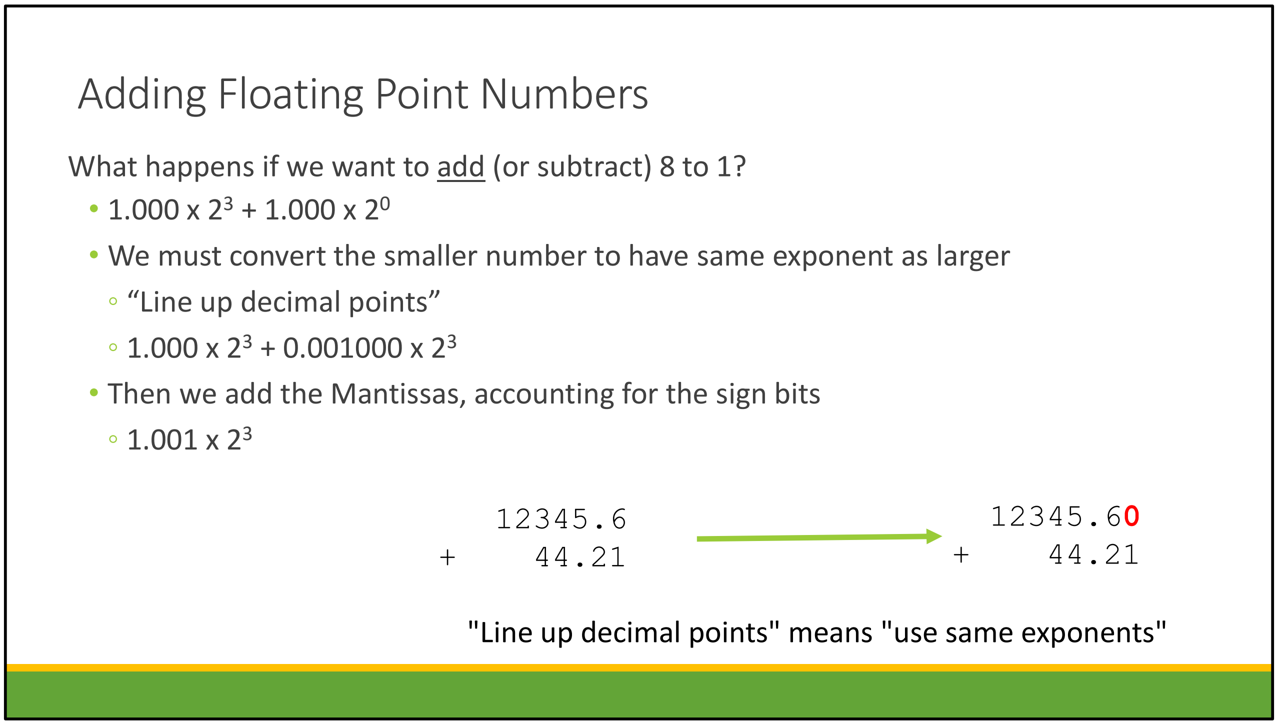


How many times will this loop run?

```
private static void TestSmall() {
    double x = 1.0;
    int count = 0;
    while (x > 0.0) {
          System.out.println("Iteration: " + count + ": " + x);
          x /= 2.0;
          count += 1;
    }
    System.out.println("Failure at: " + count + ": " + x);
}
```

This shows limits of number representation: minimum nonzero mantissa and exponent

Poll the class: 1-10? 10-100? 100-1000? Infinity?
Run on Eclipse: MathLimits.java test #1



Adding Floating Point Numbers

What happens if we want to add (or subtract) 8 to 1?

- $1.000 \times 2^3 + 1.000 \times 2^0$
- We must convert the smaller number to have same exponent as larger
  - “Line up decimal points”
  - $1.000 \times 2^3 + 0.001000 \times 2^3$
- Then we add the Mantissas, accounting for the sign bits
  - 1.001×2^3

```
   12345.6                12345.60
+     44.21   ------>   +    44.21
```

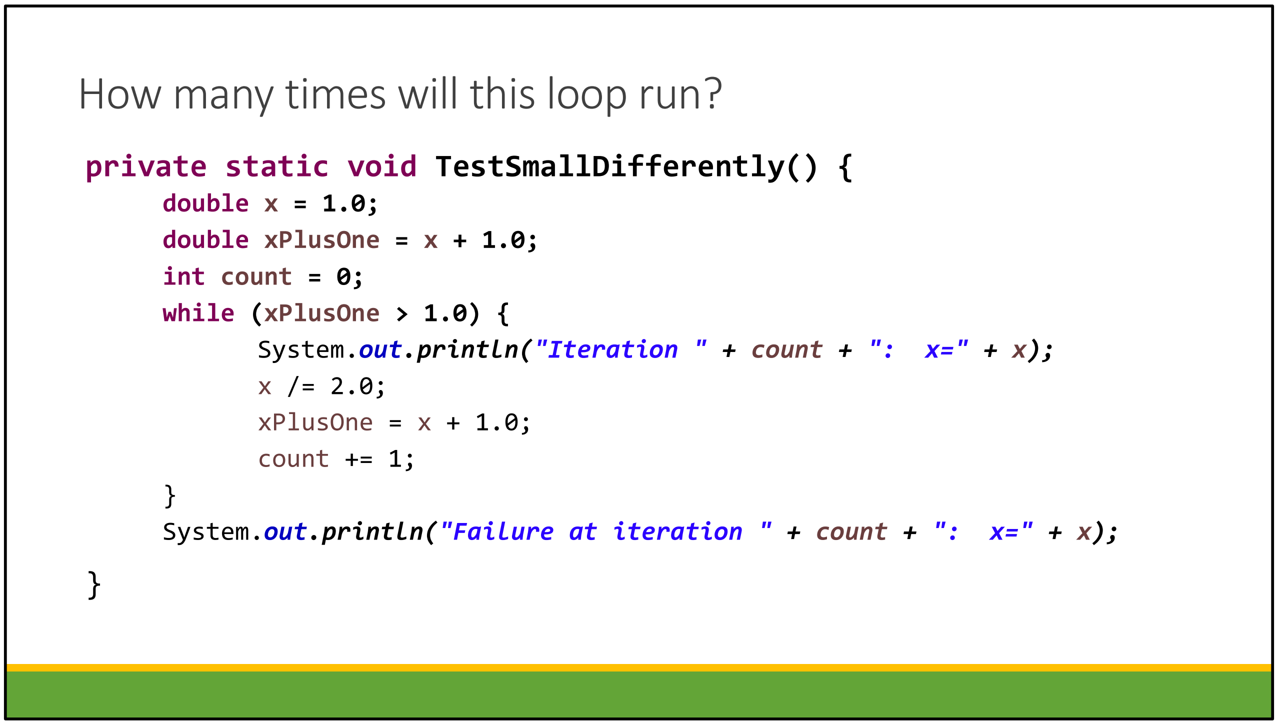
"Line up decimal points" means "use same exponents"

Multiplying Floating Point Numbers

Multiplying floating point numbers is a bit simpler.

- "Add" sign bits
- Multiply Mantissas
- Add Exponents

- $(-1)^0 \times 1.10000 \times 2^2 \times (-1)^1 \times 1.00000 \times 2^1 = (-1)^1 \times 1.10000 \times 2^3$



How many times will this loop run?

```java
private static void TestSmallDifferently() {
      double x = 1.0;
      double xPlusOne = x + 1.0;
      int count = 0;
      while (xPlusOne > 1.0) {
             System.out.println("Iteration " + count + ":  x=" + x);
             x /= 2.0;
             xPlusOne = x + 1.0;
             count += 1;
      }
      System.out.println("Failure at iteration " + count + ":  x=" + x);
}
```

Run on Eclipse: MathLimits.java test #2

“Machine epsilon”

What is smallest ε such that $(1 + \varepsilon) > 1$?

This is a measure of **computational accuracy**

In our test, $\varepsilon \approx 2 \times 10^{-16}$

In our earlier problem, condition number = 144×10^{6}

So we expect $|\text{relative output error}| \leq |\text{input error} \cdot \text{condition number}| \approx$ **2.9×10^{-8}**

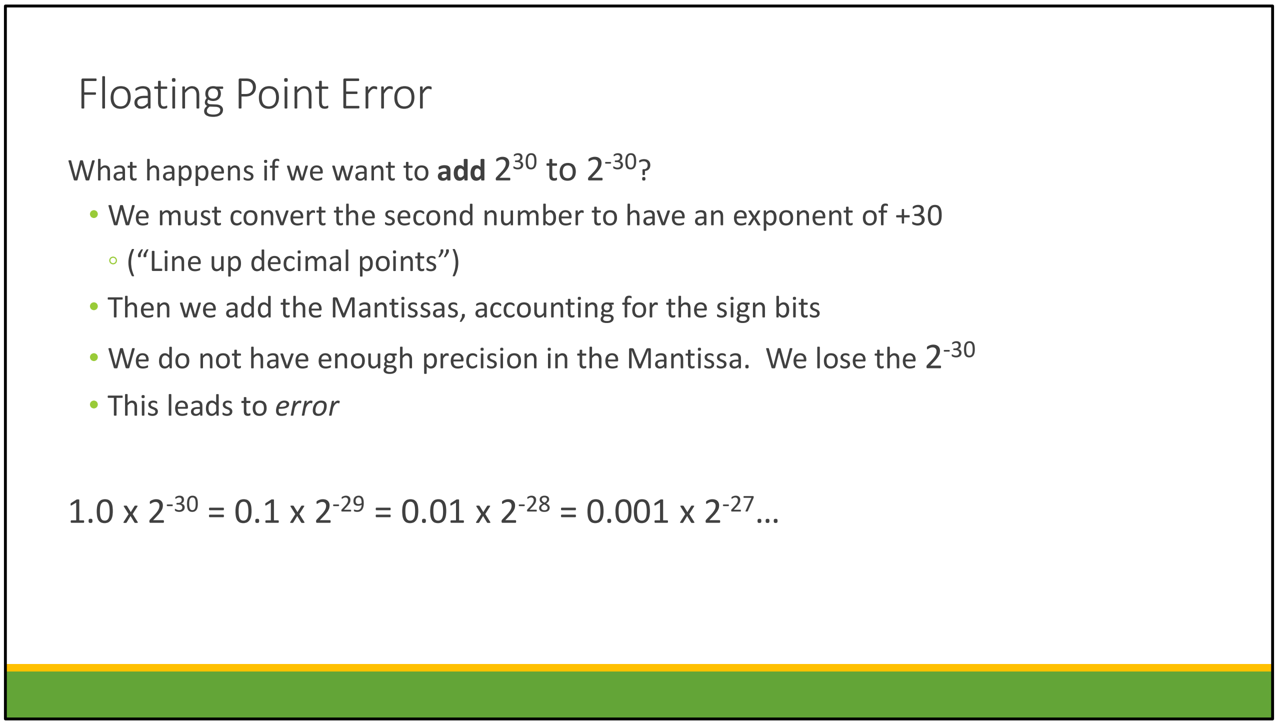
(Remember, there might be **much bigger** measurement errors also.)

Floating Point Error

For example $(1 + 10^{-30})\cdot(1 + 10^{-30}) = (1 + 10^{-30})^2 = 1 + 2\cdot10^{-30} + 10^{-60}$

Because of the finite Mantissa, the 10^{-60} is lost, causing *error.*

Errors *accumulate* with every operation



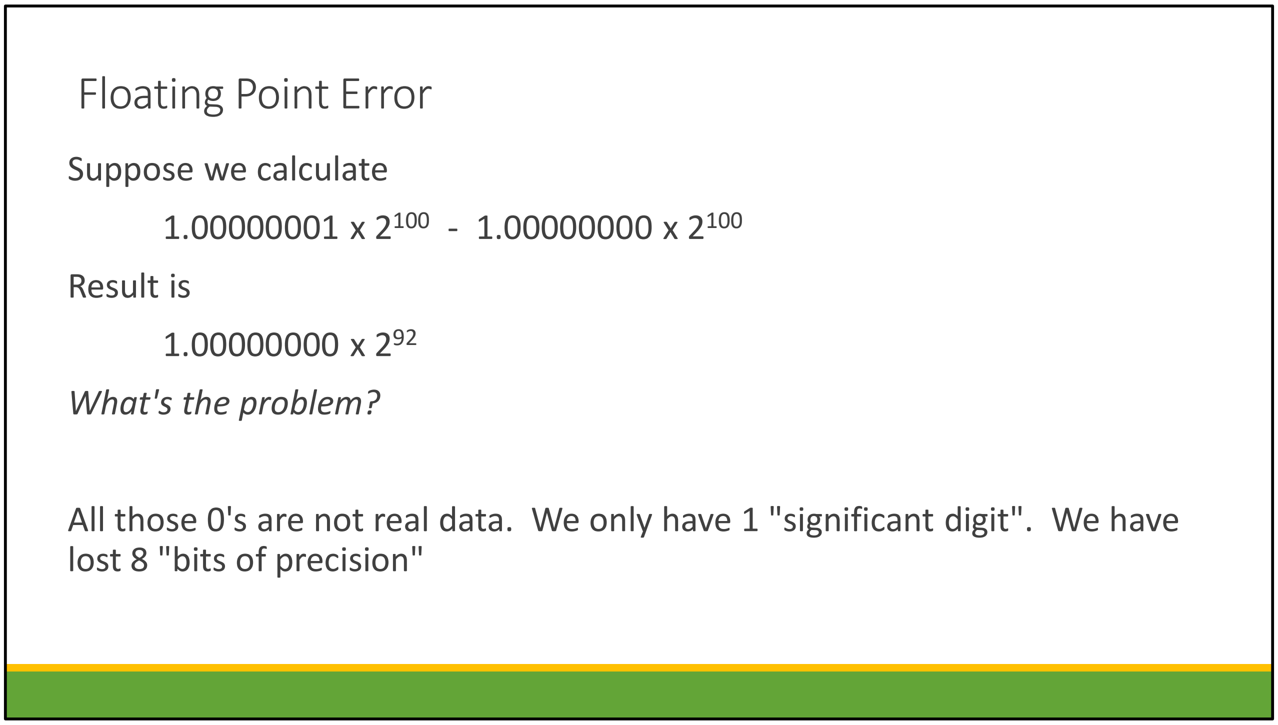
Floating Point Error

What happens if we want to **add** 2^{30} to 2^{-30}?

- We must convert the second number to have an exponent of +30
  - (“Line up decimal points”)
- Then we add the Mantissas, accounting for the sign bits
- We do not have enough precision in the Mantissa. We lose the 2^{-30}
- This leads to *error*

$1.0 \times 2^{-30} = 0.1 \times 2^{-29} = 0.01 \times 2^{-28} = 0.001 \times 2^{-27}$…

We have to shift the ‘1’ in the mantissa of 2^-30 by 60 places to the right. Oops, we only have 53 places in the Mantissa



Floating Point Error

Suppose we calculate

$1.00000001 \times 2^{100} - 1.00000000 \times 2^{100}$

Result is

1.00000000×2^{92}

What's the problem?

All those 0's are not real data. We only have 1 "significant digit". We have lost 8 "bits of precision"

This matters with measurement error also, as we all learned in H.S. Chemistry

Types of Numerical Error

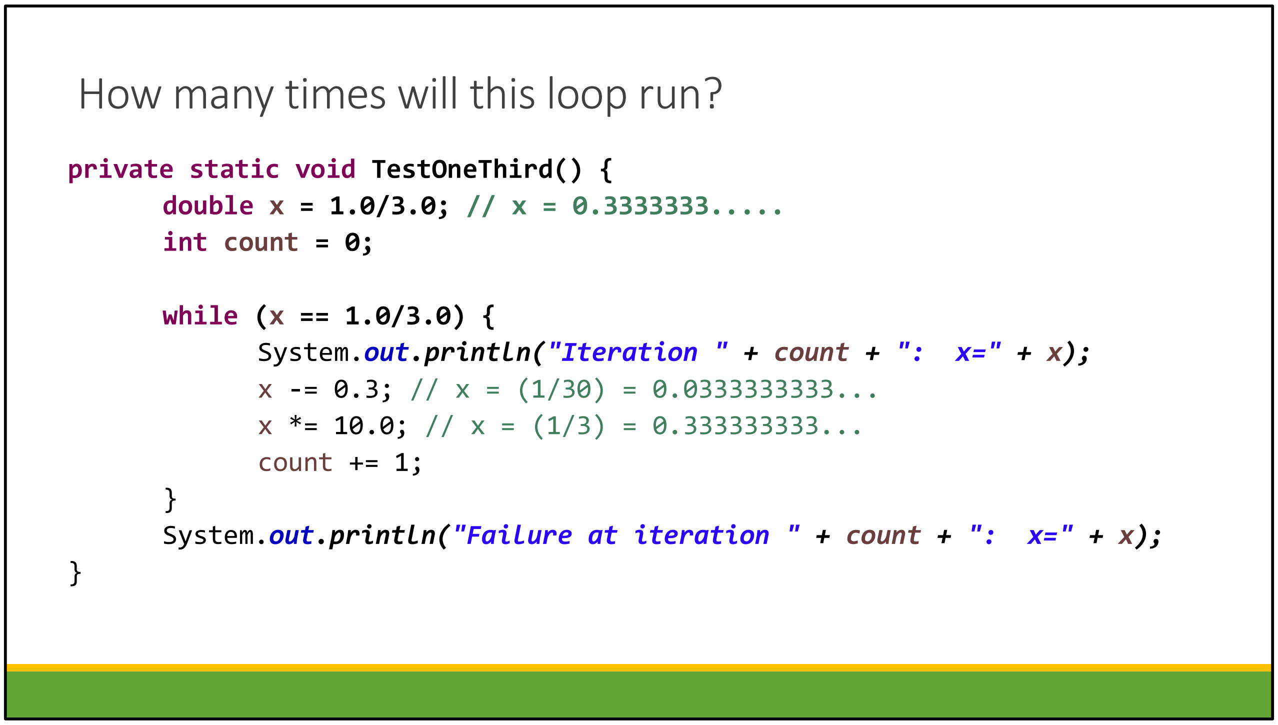
Gradual accumulation: epsilon per operation

Absorption: small numbers are completely lost when combining with bigger numbers

- Solution: add together groups of small #s first, then add the groups together. (But this leads to more gradual accumulation!)

Loss of precision: subtracting or dividing large #s with close-but-not-identical values. Lose precision

A last problem!



How many times will this loop run?

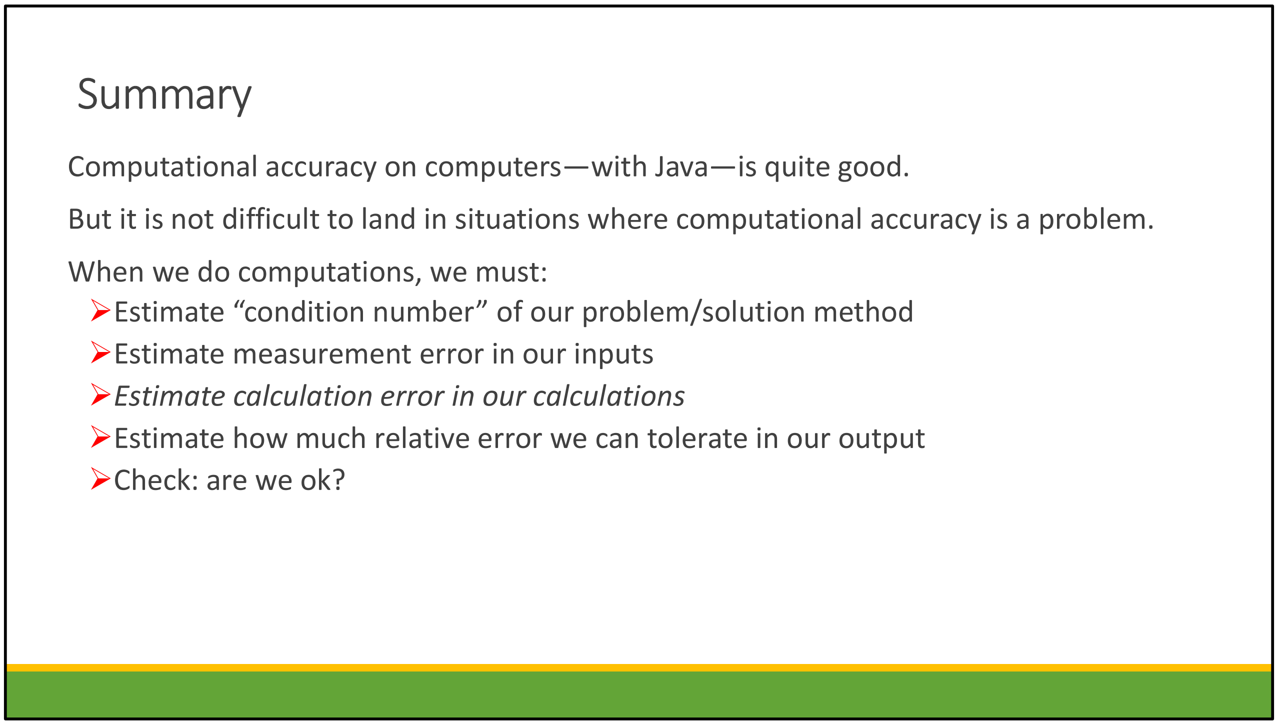
```
private static void TestOneThird() {
      double x = 1.0/3.0; // x = 0.3333333.....
      int count = 0;

      while (x == 1.0/3.0) {
            System.out.println("Iteration " + count + ":  x=" + x);
            x -= 0.3; // x = (1/30) = 0.0333333333...
            x *= 10.0; // x = (1/3) = 0.333333333...
            count += 1;
      }
      System.out.println("Failure at iteration " + count + ":  x=" + x);
}
```

Run on Eclipse: MathLimits.java test #4
Only runs once!

What’s the lesson here?

```
double x = 1.0/3.0; // x = 0.3333333.....
double y = x;
y -= 0.3;
y *= 10.0;
double diff = abs(x – y) / max(abs(x), abs(y))
```

Don’t check two values for equality, check for “close enough”.



Summary

Computational accuracy on computers—with Java—is quite good.

But it is not difficult to land in situations where computational accuracy is a problem.

When we do computations, we must:

- Estimate “condition number” of our problem/solution method
- Estimate measurement error in our inputs
- *Estimate calculation error in our calculations*
- Estimate how much relative error we can tolerate in our output
- Check: are we ok?

MAIN Point again…