Floating Point

CSC207 Fall 2015
Ariane 5 Rocket Launch
Ariane 5 rocket explosion

• In 1996, the European Space Agency’s Ariane 5 rocket exploded 40 seconds after launch.

• During conversion of a 64-bit to a 16-bit format, overflow occurred: the number was too big to store in 16 bits.

• This hadn’t been expected because the data (acceleration reported by sensors) had never been this large before. But this new rocket was faster than its predecessor.

• $7 billion of R&D had been invested in this rocket.

• Reference: http://www.around.com/ariane.html
Example 1

- Perform some simple arithmetic, and check that the laws of mathematics hold.
- Code: Adding.java
Is Java broken?

It’s not only Java. Check this out in Python:

```python
>>> x = 0.1
>>> sum = x + x + x
>>> sum == 0.3
False
>>> sum
0.3000000000000004
>>> bigger = 1.0
>>> s = 1.0e-6
>>> sum1 = s + s + s + s + s + s + s + s + s + s + s + s + bigger
>>> sum2 = bigger + s + s + s + s + s + s + s + s + s + s + s + s
>>> sum1 == sum2
False
>>> sum1
1.00001
>>> sum2
1.0000099999999992
```
Representing numbers

- It all makes sense if you understand how “real” numbers are represented.
- First, consider an int like 42. Hardware doesn’t directly represent 4s or 2s -- everything is binary.
- \[ 42 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \]
- So 42 can be represented by 101010 (base 2).
Representing fractions

- Fractions can be handled using the same approach.

- Example: $0.4375 = 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} + 1 \times 2^{-4}$
  \[= 0/2 + 1/4 + 1/8 + 1/16\]
  \[= 0.25 + 0.125 + 0.0625\]

- So we can represent 0.4375 using 0.0111 (base 2).

- Another example: $0.1 = 0.00011001100110011001100...$

- 0.1 does not have a finite binary representation
Some problem numbers

• You already knew from math that some numbers do not have a finite representation.

• Now we’ve seen that some numbers that have a finite representation in decimal do not in binary!

• Computer systems have finite memory. But we need to represent numbers that take an infinite number of bits.

• Solution?
IEEE-754 Floating Point

- Like a binary version of scientific notation
- 32 bits for a float (64 bits for a double) as follows:
  - 1 bit for the **sign**
  - 8 bits for the **exponent** $e$
  - 23 bits for the **mantissa** (significand) $M$
Allocation of 32 bits

• **1 bit** for the **sign**:
  • 1 for negative and 0 for positive

• **8 bits** for the **exponent** $e$
  • To allow for negative exponents, 127 is added to the exponent.
  • So the range of possible exponents is not 0 to $2^8-1 = 0$ to 255, but $(0-127)$ to $(255-127) = -127$ to 128.

• **23 bits** for the **mantissa** $M$
  • Since the first bit must be 1, we don’t waste space storing it!
  • So we get 24 bits of information into 23 bits.
IEEE-754 Floating Point

$(-1)^s \times (1 + M) \times 2^{e-127}$
\[
(-1)^{\text{sign}} \left( 1 + \sum_{i=1}^{23} b_{23-i} 2^{-i} \right) \times 2^{(e-127)}
\]

- \text{sign} = 0
- \[1 + \sum_{i=1}^{23} b_{23-i} 2^{-i} = 1 + 2^{-2} = 1.25\]
- \[2^{(e-127)} = 2^{124-127} = 2^{-3}\]

Thus:
- \text{value} = 1.25 \times 2^{-3} = 0.15625
Rounding

• If we have to lose some digits, we don’t just truncate, we round.

• In rounding a decimal to a whole number, an issue arises: If we have a 0.5, do we round up or down?

• If we always round up, we are biasing towards higher values.

• “Proper” rounding: round to the nearest even number. E.g., 17.5 is rounded up to 18 but 16.5 is rounded down to 16.

• The IEEE standard uses proper rounding.
Historical aside

• 30 years ago, computer manufacturers each had their own standard for floating point.

• Problem? Writing portable software!

• Advantage to manufacturers? Customers got locked in to their particular computers.

• In the late 1980s, the IEEE produced the standard that now virtually all follow.

• Kahan spearheaded the effort, and won the 1989 Turing Award for it.
Back to the example (Adding.java)

- As we saw, 0.1 cannot be represented exactly in binary, leading to the unexpected result.

- And adding a very small quantity to a very large quantity can mean the smaller quantity falls off the end of the mantissa.

- But if we add small quantities to each other, this doesn’t happen. And if they accumulate into a larger quantity, they may not be lost when we finally add the big quantity in.
Examples 2 and 3

• The previous example seems contrived, but consider these situations that happen all the time:
  
  • Accumulating a value in a loop.  
    Code: Totalling.java
  
  • Adding up a list of doubles.  
    Code: ArrayTotal.java
Lessons

• When adding floating point numbers, add the smallest first.

• More generally, try to avoid adding dissimilar quantities.

• Specific scenario: When adding a list of floating point numbers, sort them first.
Example 4

• Repeat a task for values in a particular range with an increment of 0.1.
  • For example, for values 0.1 to 0.5 with an increment of 0.1.
  • For example, for values 1.1 to 1.5 with an increment of 0.1.

• Code: FunctionValues.java
Lessons

• Don’t use floating point variables to control what is essentially a counted loop.

• Also: Notice that we wrote
  \[ x = 1.0 + i \times 0.1; \]
  instead of initializing \( x \) to 1.0 and then repeatedly adding 0.1.
  Why? Fewer total arithmetic operations means fewer rounding errors are introduced.

• Use fewer arithmetic operations where possible.
Example 5

• A very simple program that just prints the same variable using different formats.

• Code: Examine.java
What happened?

- We shouldn’t be surprised by now to find out that 4/5 can’t be represented exactly in a float. Lots of things can’t.
- But the represented value should be off by a tiny bit. What are all these extra digits??
- \( 4/5 = 1.1001100110011001100110 \times 2^{-1} \)
- It gets rounded to \( 1.10011001100110011001101 \times 2^{-1} \)
- When we ask to print it as a decimal number, it gets converted. The exact equivalent is \( 0.8000000119209289550781250000000 \)
- But only 7 of those digits are significant.
Lesson

• Don’t print more precision in your output than you are holding.
Why does this matter?
Patriot missile accident

• In 1991, an American missile failed to track and destroy an incoming missile. Instead it hit a US Army barracks, killing 28.

• The system tracked time in tenths of seconds. The error in approximating 0.1 with 24 bits was magnified in its calculations.

• At the time of the accident, the error corresponded to 0.34 seconds. A Patriot missile travels about half a km in that time.

• Reference: http://www.ima.umn.edu/~arnold/disasters/patriot.html
Sinking of an oil rig

- In 1992, the Sleipner A oil and gas platform sank in the North Sea near Norway.
- Numerical issues in modelling the structure caused shear stresses to be underestimated by 47%.
- As a result, concrete walls were not built thick enough.
- Cost: $700 million
What should you do?

“95% of folks out there are completely clueless about floating-point.”

James Gosling
Follow the lessons

- Use double instead of float.
- When adding floating point numbers, add the smallest first.
- More generally, try to avoid adding dissimilar quantities.
- Specific scenario: When adding a list of floating point numbers, sort them first.
- Don’t use floating point variables to control what is essentially a counted loop.
- Use fewer arithmetic operations where possible.
- Don’t print more precision in your output than you are holding.
There is lots more to learn!

- Consider taking csc336: Numerical Methods