Lecture Notes: Floating-Point Numbers

CS227-Scientific Computing

September 8, 2010

What this Lecture is About

- How computers represent numbers
- How this affects the accuracy of computation

Positional Number Systems

- What do we mean when we write a number like 23708? This is shorthand for

$$2 \times 10^4 + 3 \times 10^3 + 7 \times 10^2 + 8 \times 10^0.$$

- Each digit has a value, but the value is weighted by the *position* the digit is in.
- The weight associated to each position is a power of ten, so this is a *radix ten* or *base ten* positional number system.
- Every positive integer has a unique representation in this scheme, and we only need a fixed repertory of ten digits to represent arbitrarily large numbers.
- Observe that the system practically forces you to have a symbol for zero!

Positional Number Systems

- This system originated in India between the 4th and 6th centuries AD, then spread to the Arab world in the 8th and 9th centuries, and to Europe as early as the 10th century. (This explains the terms *Hindu-Arabic numerals* and *Arabic numerals*.)
- A positional number with sixty as a radix was developed by the Babylonians as early as 3000 BC. The Maya of Central America used a radix twenty positional number system dating from the first century BC.
- Computers use a radix two, or *binary* system.

Integers in Binary

- There are only two digits, 0 and 1. These are called *bits*: "bit" is a contraction of "binary digit".
- For instance

$$110100101$$

  is the binary representation of $2^8 + 2^7 + 2^5 + 2^2 + 2^0$, which is four hundred twenty-one.
- Sometimes, to keep the radices straight, you will see the equation written as

$$110100101_{\text{two}} = 421_{\text{ten}},$$

  or something similar.
- *Everything* in a computer's memory—numbers of many different kinds, text, images, audio, program instructions—is represented as as sequences of bits.
- **Everything.**

Hexadecimal Notation

- It is hard to read something like 110100101 at a glance. *Hexadecimal* notation is binary for humans, designed to solve this problem.
- The idea is to break the bits into groups of four, beginning at the righmost bit.

$$1 \quad 1010 \quad 0101$$

- Each group represents an integer between 0 and fifteen, which we denote by single digit, using A, B, C, D, E, F for the vaues ten through fifteen. So for the number above, we get 1A5:

$$1\mathrm{A}5_{\mathrm{hex}} = 421_{\mathrm{ten}}.$$

- It's really just radix sixteen!

But that is just part of the story, and not how MATLAB usually represents numbers!

"Scientific Notation"

.

- You all know this.
- Distance from Earth to Sun.

$$1.496 \times 10^8 \quad \text{kilometers.}$$

- Gravitational Constant

$$6.673 \times 10^{-11} \quad \frac{m^3}{kg \cdot sec^2}.$$

- U.S. Balance of Trade Deficit

$$3.801 \times 10^9 \quad \text{dollars.}$$

Roundoff and Precision

- Each number is rounded to show just *four significant decimal digits*, or *four decimal digits* of **precision**.
- Another way to look at precision is through the relative error

$$\frac{|\tilde{x} - x|}{|x|},$$

where x is the value we are approximating, and $\tilde{x}$ is the rounded approximation.

- For example, a more accurate measurement of the mean distance to the sun is 149597871 km, so the relative error is

$$\frac{(149600000 - 149597871)}{149597871} \approx 1.42 \times 10^{-5}.$$

(The value 10^{-5} shows this is actually accurate to five digits precision.)

- Note that relative error is not affected by choice of units.

Calculations with fixed precision

- Suppose you perform each operation between two values as accurately as possible, but then round off the result to a fixed precision, say two decimal digits of precision.
- You get some anomalous results, for instance

$$(1/3 + 1/3) - 2/3 \quad \text{“}=\text{”} \quad (0.33 + 0.33) - 0.67 = -0.1$$

$$(1.4 + 0.46) + 0.06 \quad \text{“}=\text{”} \quad 1.9 + 0.06 \quad \text{“}=\text{”} \quad 2.0.$$

- Note the *loss of precision*: the second result is NOT equal to the sum correctly rounded to two digits of precision. On the other hand

$$1.4 + (0.46 + 0.06) \quad \text{“}=\text{”} \quad 1.4 + 0.52 \quad \text{“}=\text{”} \quad 1.9,$$

which is correct.

IEEE Double-Precision Floating Point Representation

- MATLAB uses this by now near-universal standard to represent numbers in a kind of binary version of scientific notation.
- To see how this works, let's return our earlier example of four hundred twenty-one. This is

$$110100101_{\text{two}} = 1.10100101_{\text{two}} \times 2^8.$$

- The idea is then to use 64 bits to represent the number.
- The leftmost bit represents the sign (0 for positive, 1 for negative). The next eleven bits represent the *exponent* (in this case 8), and the remaining 52 bits represent the piece .10100101.... Note that there is no point in representing the 1 preceding the "radix point". (We can't call it a *decimal point!*)

IEEE Double-Precision Floating Point Representation

- Here is the result, illustrated with MATLAB:

```
>> num = 421;
>> format hex
>> num
num = 407a500000000000
```

- If we dissect the resulting hexadecimal and write it in binary, we see the three pieces.

$$0010 \quad 0000 \quad 0111 \quad 1010 \quad 0101 \quad 0000 \quad 0000\cdots$$

$$\underbrace{0}_{\text{sign}} \quad \underbrace{010000000111}_{\text{exponent}} \quad \underbrace{101001000000\cdots}_{\text{mantissa}}.$$

- Observe that the representation of the exponent is a bit strange—it's not obvious how this represents 8.

Another Example

- What is one-fifth in binary? Observe that $16 \times \frac{1}{5} = 3 + \frac{1}{5}$, so shifting the radix point four bits to the right should be the same as adding 3. This gives

$$0.00110011001100\cdots = 1.100110011001 \times 2^{-3}.$$

- The mantissa $.100110011001\cdots$ should be $999\cdots$ in hexadecimal but it has to be rounded after 52 bits, which can change the final bits.
- `>> num=-1/5`
  `num = bfc999999999999a`

- We parse this as

$$1011 \quad 1111 \quad 1100 \quad 1001 \quad 1001 \quad \cdots \quad 1010,$$

or

$$\underbrace{1}_{\text{sign}} \quad \underbrace{01111111100}_{\text{exponent}} \quad \underbrace{1001100110011001\cdots10011010.}_{\text{mantissa}}$$

Special Values

- If you start with a positive value x and keep doubling, you will eventually exceed the largest representable value. *(Overflow.)* The result will appear as Inf.
- If instead you keep dividing by 2, you will eventually get a result smaller than the smallest representable positive number. *(Underflow.)* The result will appear as 0. You should note that in the scheme described above, 0 is not actually representable, since it cannot be written in the form $1.xxxx \cdots \times 2^{\exp}$. However the bit pattern $00 \cdots 0$ is reserved to represent this value.
- If you try, say, to evaluate $0/0$, you will not get an error message. Instead the result will be represented by a special bit pattern, which appears as NaN ('Not A Number').
- A measure of the precision of the system is given by the smallest positive number x such that $1 + x$ gives a different value than 1. This is called *machine epsilon*. Note that this is NOT the same thing as the smallest representable positive number.

- In the IEEE standard, the exponent can vary between -2^{10} and $2^{10} - 1$ meaning that the maximum value representable is on the order of $2^{1000} \approx 10^{300}$, and the minimum representable positive value about 10^{-300}. (But there's a catch here—see Assignment 2.)
- Machine epsilon, on the other hand, is $2^{-52} \approx 2 \times 10^{-16}$. If you add two numbers and one is more than 10^{16} times the other, the smaller number will have no effect on the sum.
- See the `floatgui` application in Moler's book for a look at the floating-point system with toy parameters (small precision and small range on the exponent).

Loss of Precision

- When MATLAB performs an arithmetic operation on two floating-point numbers, it computes the result precisely, then rounds it to the nearest value representable with the given precision.
- This leads to some anomalous results. For instance, try evaluating

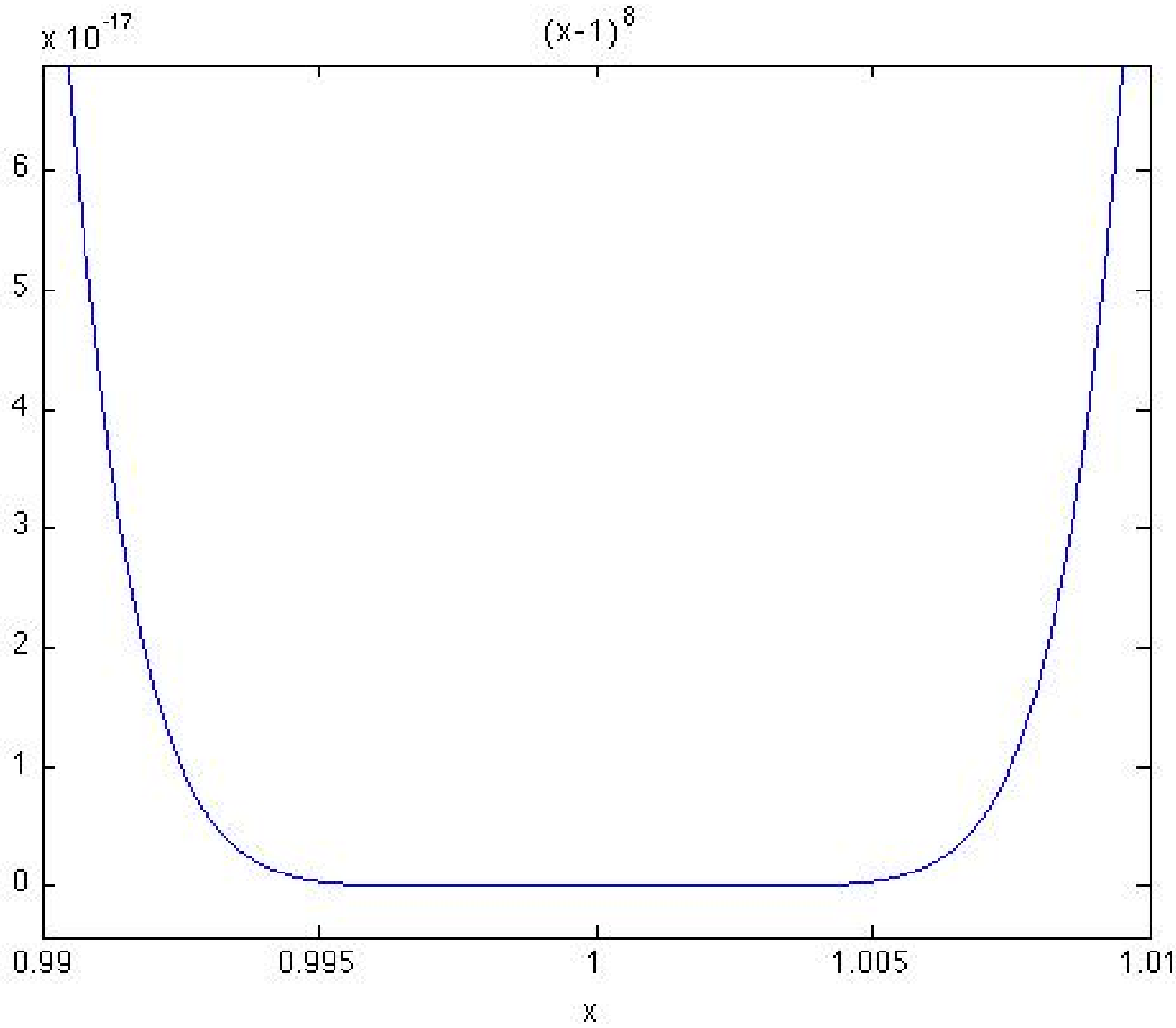
  `1/10+1/10+1/10-3/10`

  in MATLAB.
- After many iterations, precision can erode. One particular situation to watch out for is addition of two quantities of very different magnitudes. Most of the bits of precision of the smaller value will be lost.
- Another is *catastrophic cancellation*, when you take the difference of two quantities of nearly equal precision.

Example of Catastrophic Cancellation

- Let's plot the polynomial $(x-1)^8$ in an interval about 1:
- `>> ezplot(@(x)(x-1).^8,[0.99,1.01])`

This is what you'd expect.



Example of Catastrophic Cancellation

- Suppose you expanded the polynomial and wrote it in the form:

$$x^8 - 8x^7 + 28x^6 - 56x^5 + 70x^4 - 56x^3 + 28x^2 - 8x + 1$$

- ```
  >> f=@(x)(x.^8-8*x.^7+28*x.^6-56*x.^5+70*x.^4-56*x.^3+
  28*x.^2-8*x+1);
  >> ezplot(f,[0.99,1.01])
  ```
  ```

The wild oscillation in the graph is due entirely to magnification of roundoff error when we find the difference between two nearly equal large quantities.