Dangers of computer arithmetic

Because computers use a relatively few bits to represent numbers, one must be careful when asking them to carry out certain arithmetical operations.

Adding quantities of very different magnitude: decimal

The problem here is

1. the mantissa of a floating point number can only cover a certain range
2. computers can add and subtract numbers only if they have the same exponent

Before they can add or subtract two values, computers must somehow force the two values to have the same exponent. By convention, they modify the smaller of the values, doing their best to retain its original value. However, the conversion sometimes causes the final value to be different, which leads to an error in the result of the addition or subtraction.

Let's look at an example for a fictional computer which uses base-10 arithmetic. Suppose that it has 4 decimal places for the mantissa and 2 decimal places for the exponent. If we ask it to add the numbers 239,400 and 875, here's what happens:

```
0.2394 x 10^(5)     = 239,400
0.8750 x 10^(2)     =     875
```

The computer can't add two numbers with different exponents, so it converts the smaller number into a form with the same exponent as the big number:

```
      0.8750 x 10^(2)     =     875
shift
      0.0875 x 10^(3)     =     875     okay
shift
      0.0087 x 10^(4)     =     870     uh-oh!
shift
      0.0008 x 10^(5)     =     800     double uh-oh!
```

Whoops! The result of the conversion is 800, not 875. But the computer just keeps going:

```
  0.2394  x  10^(5)          =  239,400
+ 0.0008  x  10^(5)          =      800
=========================================
  0.2402  x  10^(5)          =  240,200
```

So, according to this fictional base-10 computer, 239,400 + 875 = 240,200. That's incorrect, of course: the true sum is 240,275. But with only 4 decimal places in the mantissa, the computer can't keep track of all the digits.

Adding quantities of very different magnitude: binary

Now, let's watch the same procedure carried out by a real computer which uses binary representations. Consider a computer which uses a 4-bit mantissa and a 3-bit exponent to represent numbers in base 2. Let's choose two numbers, 40 and 9 in ordinary base ten.

```
40   =   0.625    x   2^(6)

     =   1010         110      in the computer


 9   =   0.5625   x   2^(4)

     =   1001         100      in the computer
```

Now, we know that when you add these two numbers, you should get

```
40 +  9  =  49
```

But what happens when the computer tries to do it? The computer adds numbers by modifying the smaller number until its exponent matches the larger number's exponent, and then combining the mantissas. If the computer could explain itself, it would say

1. Start with 1010 110 (that's 40 in base 10)
   1001 100 (that's 9 in base 10)

2. Modify the smaller value so that it has the same exponent as the large one. We can do this in steps:

   1001 100 = 0.5626 x 2^(4) is the starting point

   0100 101 = 0.25 x 2^(5) after shifting exponent once

   0010 110 = 0.125 x 2^(6) after shifting exponent twice

Oh, no! The computer has done the best it could to convert the smaller value into a form which has the same exponent as the big one ... but the result is wrong. We started with "1001 100", a value of 9, but the result "0010 110" has a value of 8. The computer doesn't know this; it just keeps going.

3. Now that the two numbers have the same exponent, add the mantissas.

```
  1010 110
+ 0010 110
==========
  1100 110
```

4. The sum has mantissa 1100 = 0.75, exponent 110 = 2^(6), for a total of 0.75 x 2^(6) = 48. Thus,

   computer says 40 + 9 = 48

Obviously, the result is not correct. Adding a small number to a large one yields an answer which is only approximately correct.

The problem can become even worse. Suppose we try to add an even smaller number, 3, to the same big one, 40. In step 2 of its procedure, the computer will try to convert its representation for "3" so that the exponent matches that of its representation of "40".

1. Start with 1010 110 (that's 40 in base 10)

1100 010 (that's 3 in base 10)

2. Modify the smaller value so that it has the same exponent as the large one. We can do this in steps:

   1100 010 = 0.75 x 2^(2) is the starting point

   0110 011 = 0.375 x 2^(3) after shifting exponent once

   0011 100 = 0.1875 x 2^(4) after shifting exponent twice

   0001 101 = 0.0625 x 2^(5) after shifting exponent thrice

   0000 110 = 0 x 2^(6) after shifting exponent again

Oh, no, again! Now the smaller value has been turned into zero! When we add this to the big number, we'll get the big number itself as the sum.

3. Now that the two numbers have the same exponent, add the mantissas.

```
  1010 110
+ 0000 110
===========
  1010 100
```

4. The sum has mantissa 1010 = 0.625, exponent 110 = 2^(6), for a total of 0.625 x 2^(6) = 40. Thus,

   computer says 40 + 3 = 40

Calculating the difference of two very large values

A classic error occurs when one subtracts (or compares) two very large values which are close in magnitude. Using the same system as the previous example, in which 4 bits are used for the mantissa and 3 for the exponent, suppose we wish to subtract (base ten) 52 from 72.

1. Start with 1001 111 (that's 72 in base 10)
   1101 110 (that's 52 in base 10)

2. Modify the smaller value so that it has the same exponent as the larger one:

   1101 110 = 0.8125 x 2^(6) is the starting point

   0110 111 = 0.375 x 2^(7) after shifting exponent once

3. Now that the two numbers have the same exponent, subtract the mantissas.

```
  1001 111
- 0110 111
===========
  0011 111
```

4. The sum has mantissa 0011 = 0.1875, exponent 111 = 2^(7), for a total of 0.1875 x 2^(7) = 24. Thus,

   computer says 72 - 52 = 24

This sort of error does not occur if one adds two large numbers (as long as the sum doesn't overflow).

Repeating a (very slightly incorrect) operation many, many times

As you will learn in this course, scientists very frequently program a computer to carry out some computation over and over and over and OVER again, thousands or millions of times. If each individual computation incurs some error -- even a very small one -- then that error can build up to a significant level.

For example, suppose one is simulating the orbit of the Earth around the Sun as a function of time. One might calculate the change in the Earth's position once every

second. If that change has even a tiny fractional error, say, 10^(-10), then the Earth's position will slowly become more and more inaccurate.

After	(seconds)	fractional error is
1 day	86,400	86,400 x 10^(-10) = 8 x 10^(-4) percent
1 month	2,592,000	2,592,000 x 10^(-10) = 0.026 percent
1 year	31,557,600	31,557,600 x 10^(-10) = 0.32 percent
1 decade		3.2 percent
1 century		32 percent

It would be impossible to use a program with even a tiny fractional error in each step to simulate the motions of planets in the Solar System over timescales of millions of years, let alone billions of years.

It's often easy to cook up a simple numerical algorithm with a relatively large fractional error. One can apply that algorithm safely to some problems: those which don't require very large numbers of iterations. If one is lazy or careless, one might mis-apply such an algorithm to some other problem for which it isn't safe -- and get garbage as the result. Sometimes, it's necessary to spend extra time to write a more accurate algorithm in the first place.

There's often a choice:

- use a simple method with a small step, iterate MANY times
    - Pro: quick to write, and the code is likely to be correct
    - Con: susceptible to accumulated errors, can take many minutes (or hours!) of computation
- use a complex method with a large step, iterate few times
    - Pro: yields more accurate answer with less computation
    - Con: takes longer to write the code (and is more likely to have mistakes in the algorithm)

You might want to check out a simple example of accumulated roundoff error.

How can you detect errors?

There are several general approaches to checking for errors in a large set of calculations.

1. Run the program on a simple system to which you KNOW the answer. If you can solve a system analytically, the result will be exact. Compare the exact answer to the result of your computations.
2. Set up two independent efforts: they may be two different people writing separate versions of the same program, or two slightly different versions of the same program (perhaps with different algorithms used in a crucial spot). Run both efforts on the same system, and compare results.

It is significantly easier to **detect** errors than it is to **fix** them...

Source: http://spiff.rit.edu/classes/phys317/lectures/dangers.html