Error in Numerical Methods
Notes for CSCI3656

Liz Bradley
(with help from Jay Kominek and Wayne Vinson)
Department of Computer Science
University of Colorado
Boulder, Colorado, USA 80309-0430

Research Report on Curricula and Teaching CT004-02

1 Introduction

Error — that is, how far an answer is from the true value — can be measured in two different ways: as an absolute value, or as a relative value. *Absolute* error is the difference between the computed (or estimated) answer and the true answer. *Relative* error is the absolute error divided by the true answer. Absolute error is measured in regular units: degrees, inches, seconds. Relative error is measured in *relative* units: percent, parts per million, decibels, etc.

For example, if the temperature outdoors is 75 degrees but your thermometer is reading 96 degrees because it's in the sun, its absolute error is

$$|75 - 96| = 21$$

measured in degrees, and its relative error is

$$\frac{|75 - 96|}{75} = 0.28$$

or 28%.

You can also express error in terms of significant figures, or digits of accuracy; π, for example, is 3.141592654... If I round it to the fourth place after the decimal point, I get 3.1416. There is another way to approximate a number using fewer decimal places' worth of space: one can "chop" it (which means simply discarding the digits after the one you want). π chopped to the fourth decimal place, for instance, is 3.1415. Note that chopping error can be worse than rounding error!

Error comes from a variety of sources:

- Blunders: *Software Engineering: Theory and Practice* (Pfleeger; Prentice-Hall, 2001) cites various depressing statistics for bug frequencies: one bug per...
  - 10,000 lines of code in the avionics system of the space shuttle

  - 5,000 lines of code from "leading edge software companies"
  - 700 lines of code in "critical systems"
  - 20-200 lines of code in military systems

  Note that these are in tested, deployed code. The stuff you write late on monday nights is probably worse.

- Modeling assumptions: the real world is nonlinear and complicated, and any mathematical description of it is almost guaranteed to be wrong. Engineers and scientists not only realize this, but actually take advantage of it. In particular, the idea in modeling is to construct a reasoned, controlled approximation that serves your purposes with minimal complexity. In the first month of freshman physics, for instance, you are taught that a ball dropped from a building accelerates constantly until it hits the ground. In the second month, you are taught that air friction also matters if you want to describe the ball's trajectory more precisely. The idea is to use as simple a model as you can get away with, where "get away with" means that the difference between the model and the system is below the resolution that your application demands. (It would be foolish, for instance, for me to include the effects of the coriolis force induced by the earth's rotation unless I was dropping the ball from a *very* tall building[1].) The upshot of all of this is that any model differs from the real system that it describes. This difference may be intentional and useful, but it is still a source of error, and you need to be aware of it.

- Sensors: real measurement devices always have finite resolution, are often noisy, and sometimes inject time delays. The analog-to-digital converter in one of the standard signal processing chips[2] used to process the audio signals going to and coming from standard consumer phone lines, for example, uses 10 bits (including a sign bit) to capture the gradations in the volume of your voice, with a ±1.5 LSB error. This means that they cannot resolve subtle sound differences — those that create a voltage variation of less than 3 parts (that is, ± 1.5 LSB) in 2^9, which translates to ±0.586% error. Sensors can also distort the quantities that they are supposed to measure. The Hubble Space Telescope mirror, for instance, was installed with a critical washer underneath it, instead of on top of it. This threw off the path of every photon that bounced off the mirror, causing the image that arrived at the telescope's lens to be distorted — and in a complicated, nonlinear way that couldn't simply be subtracted off *post facto*. (Incidentally, this could have been caught with a half-million dollar pre-launch test, but the mission was over budget, so they skipped it. Fixing it required a space shuttle visit, which costs hundreds of millions of dollars. Moral: don't skimp — either time or money — on testing.)

- Truncation: comes from the approximation that is inherent in numerical algorithms. Consider methods that are based on some kind of series. If you only use the first n terms of the series, you have "truncated" the series (and the method). The effects

[1] You'd be surprised, though; this effect curls the ball a meter or so sideways for every 500m in height.

[2] The Texas Instruments TMS320

of those ignored terms are called truncation error. For instance, a three-term Taylor series approximation to $f(x) = e^x$ near $x = 0$, which is $p_2(x) = 1 + x + x^2/2$, involves a trucation error of $x^3/6 + x^4/24 + ...$

- Computer Arithmetic: computers have finite-width words, so their precision is finite. Floating-point numbers can only take on discrete values; working with computer arithmetic is much like walking on a sidewalk and only stepping on the cracks. Moreover, computer arithmetic systems have bounds, and you risk over- and underflow if you exceed them — e.g., if you evaluate 100! on your pocket calculator. Lastly, the exact behavior of the error will depend on whether the computer rounds or chops numbers to get them to fit into its finite-precision world.

These types of errors can combine. Imagine that you construct a model (that is, an equation) that describes how a ball moves through the air, but you neglect air friction. Then, you approximate that equation with a two-term Taylor series and evaluate it on a calculator with five decimal places. In this case, you have trucation error (the higher-order terms of the Taylor series that you didn't include), modeling error (that neglected friction), and finite-precision arithmetic error (all the digits past the fifth place, which the calculator loses).

Error can also *propagate*. Numerical methods that feed their outputs back to their inputs are particularly prone to this. The best example I've seen of this is a simulation of the solar system by E. Hairer of the University of Geneva. He (she? don't know the first name) used a numerical algorithm to predict the position of each planet at the next time step. This algorithm simply looked at the positions of the other planets, computed their gravitational force on the planet it was simulating, and then used that force to figure out where that planet would go next. That new position then became the jumping-off point for the next step of the simulation, and so on. (We'll do these kinds of algorithms in the last month of the semester.) The issue here is that the algorithm's answer is used as its input on the next round of simulation. If that answer is wrong — if the planet is an inch too far to the left — the algorithm will faithfully figure out where it will go from *that* (wrong) position. As you can imagine, things can go rapidly bad from there, much as a snowball accretes more snow as it rolls downhill, making it bigger, which makes it accrete more snow, and so on.

This phenomenon is known variously as dynamic error, propagating error, multiplicative error, etc. The amount of error that takes place in each individual step of such a process is sometimes called *local error*; the accumulated results of those local errors over the whole run is sometimes called *global error*. Global, propagating error is different from *additive* error, such as the distortion introduced by a badly focused lens, which only comes in once, and is not fed back around into the input of the method. Dynamic error is much nastier because of this feedback loop. We will cover this in more detail later in this course.

Algorithm-induced error can come from other sources as well; these notes give only a brief introduction. Any numerical approach involves some sort of mathematical approximation, and hence some error. Implementation matters, too: how you choose to set up the termination condition on your algorithm will obviously affect the resulting error.

2 The Next-Term and Next-Step Heuristics

Calculating the error is all very well in theory, when you know the answer and can write down how far off you are. In practice, of course, things aren't so easy: you invoke some numerical method, you get back an answer (or a series of answers), and you'd like to be able to look at that information and draw some conclusions about how good your answer is.

Numerical methods practitioners use a variety of techniques for this. The first one that we'll discuss is based, again, in the notion of a series. If your algorithm uses a series — and many, many of them do — and you truncate that series at some point, then the terms that you chopped off are a good estimate of the error. (cf., the discussion of truncation error, above.) Furthermore, if that series is a nice, converging one, then each successive term in it is smaller. Putting those two arguments together yields the *next-term rule*: that the error in a numerical method that is based on a series is approximately equal to the first unused term in the series. The reason this is useful is that you *know* (or can look up), for each method, what kind of series it uses. That means you *know* what that next term is, and can sensibly use it as an estimate of the error. (This is exactly the same idea, by the way, as in the derivation of the $R(n)$ in the Taylor series discussion.)

A related way to estimate error is to take advantage of how a numerical method converges to an answer. If your iterates are settling down — that is, if the difference between successive iterates is getting smaller — intuition suggests that you're getting closer to the answer. The *next-step rule* formalizes this: it estimates the error at step n as the difference between the answer at step n and the answer at step $n + 1$. (Note that you have to do one more step's worth of work to calculate this.) To get the *relative* error in this case, you divide the absolute error by the answer at step n. Some authors advocate dividing by the answer at step $n + 1$ instead; either way is fine with me.

Of course, if the problem is pathological, the next term in the series could be bigger than all previous ones, and the next step of the iteration could diverge from all the steps that preceded it. The next-term/next-step rules are heuristics, not algorithms or theorems.

3 Computer Arithmetic Error

Floating-point error is common and dangerous enough to warrant a bit more discussion. In real-world computers, these errors are small, but their effects are still important if the numbers that you're working with are small. Calculations can magnify these effects. Here is a function that demonstrates these problems very nicely:

$$f(x) = \frac{1 - \cos x}{x^2}$$

If you evaluate $f(x)$ near $x = 0$, a bunch of bad things happen. First and foremost, since x is small, the *relative* error introduced by the computer's arithmetic system will be large. Calculating $1/x$ worsens that effect: it's like a magnifying glass for the error. Squaring a small number that contains a large relative error also magnifies that error, and inverting the quantity ($1/x^2$) makes matters even worse. Lastly, $\cos x$ is close to 1 when $x = 0$, so the numerator is also problematic, in that it involves a subtraction of two nearly identical numbers. The upshot of all of this is that the error in the calculated answer $f(x)$ will get worse as $x \to 0$. This becomes painfully obvious if you calculate the value of $f(x)$ near $x = 0$ using a calculator that has only 10 digits of accuracy:

x	$f(x)$: 10 digits	$f(x)$: true value
0.1	0.4995834700	0.4995834722
0.01	0.4999960000	0.4999995833
0.001	0.5000000000	0.4999999583
0.0001	0.5000000000	0.4999999996
0.000001	0.0000000000	0.5000000000

Sometimes you can rearrange a function like this and make it less sensitive to numerical errors. The trick is to look for the places where the "small number effects" described above might creep in, and see if you can write that part of the function in a way that gets around it. (This is related to *scaling*, which we will talk about in a few weeks, in conjunction with linear systems solvers.)

The traditional way to think about a computer arithmetic system's precision is similar to the notion of a calculator with a fixed number of decimal places. In particular, people think about a property called machine ϵ: the smallest number (or difference between two numbers) that the computer can perceive. If a computer used 32 bits to represent floating-point numbers, for instance, and its range were -1000 to 1000, a simplistic arithmetic system might divide up the interval [-1000, 1000] into 2^{32} chunks, and represent a number using the binary code identifying the chunk into which it falls. The quantization steps in this system fall at even gaps of $2000/2^{32} = 4.67 \times 10^{-7}$.

This is not, however, the way computers really do arithmetic. As noted above, a given size error has a stronger effect on calculations that involve smaller numbers. For this reason, computer arithmetic systems try to be more precise about smaller numbers. This means that they spread the representable numbers out in a nonuniform way — specifically, the smaller the range, the more dense the numbers that can be represented. What this means is that the notion of a fixed-size machine ϵ is oversimplified, and the quantization introduced by floating-point arithmetic is not uniform. This is well intentioned and useful: it happens because computer arithmetic systems are designed to minimize the error effects described above. Nonetheless, arithmetic error is still a problem, and it still gets magnified by calculations, and you still have to worry about it.

All of this is addressed further in Lloyd Fosdick's *IEEE Arithmetic Short Reference*, which we will cover next.

4 Conclusion and Reality Checks

Understanding where errors come from is not just an academic exercise. Rather, it allows you to figure out if your computation is right. If you want to know whether the truncation error is an issue, you can use one more term in the series and see of your answer changes. If you're worried about floating-point arithmetic effects, change from single to double-precision numbers (e.g., from `long` to `double` in `c` or `c++`) and see if the answer changes. If your algorithm dices up space or time and you want to know if the quantization is messing things up, halve the spacing and re-run the code. If you don't know whether or not friction matters, include it and see if things change. And so on and so forth. If you don't know what kind of error you have and you want to find out, you can make each of those changes, successively, and see which one(s) affect your answer.