Week 1: Wednesday, Jan 26

Problems du jour

1. We can write 1/3 as a repeating decimal representation: $0.3333\ldots$. How would we write 1/3 as a repeating binary representation?

2. Suppose p and q are relatively prime. Under what circumstances can p/q be represented as a finite decimal representation? A finite binary representation?

3. *For fun*: Suppose p and q are relatively prime, $p < q$. What is the maximum number of digits in the decimal or binary representation of p/q before one sees repetition? Once you start repeating, what is the maximum length of the (minimal) repeating sequence that occurs?

Logistics

- The Google group is by invitation only. If you received an invitation but still can't sign on, it may be because the invitation address was not linked to your Google account. To remedy this:

  1. Go to your Google Groups page (one of the menu links in iGoogle).
  2. Click on the "my account" link at the top right.
  3. There should be a list of email addresses listed under the "personal settings" panel. Click the edit button at the bottom, and walk through the next page to add your Cornell email address.

- If you drop or add the class, please send us a note so that we can keep the CMS site and Google Groups enrollment up to date.

- Remember: HW 1 is due next Wednesday, Feb 2. It mostly involves material we will cover in class today.

- We will start basic concepts of numerical linear algebra next week.

Absolute and relative errors

Remember from last time:

- Absolute error is the difference between approximation and truth.
- Relative error is the absolute error scaled by the size of the true value.
- We usually care more about relative error than absolute error.

We generally can't compute the exact error. If we could, we would have the true value! Instead, we will usually try to bound the magnitude of errors.

Relative error is easy to define when we talk about scalar-valued functions. When we talk about vector-valued functions, we have to clarify what we mean by the "size". We will talk more about measuring vectors (via norms) when we talk about numerical linear algebra.

Sources of error

There are several common sources of error in scientific computation. In the order we encounter them

Roundoff error: IEEE floating point arithmetic is essentially binary scientific notation. The number 1/3 cannot be represented exactly as a finite decimal. It can't be represented exactly in a finite number of binary digits, either. We can, however, approximate 1/3 to a very high degree of accuracy.

Termination of iterations: Nonlinear equations, optimization problems, and even linear systems are frequently solved by iterations that produce successively better approximations to a true answer. At some point, we decide that we have an answer that is good enough, and stop.

Truncation error: We frequently approximate derivatives by finite differences and integrals by sums. The error in these approximations is frequently called truncation error.

Stochastic error: Monte Carlo methods use randomness to compute approximations. The variance due to randomness typically dominates the error in these methods.

In addition to error incurred during computation, of course, there is also frequently error in the problem inputs.

Condition numbers and sensitivity

Suppose we want to evaluate a differentiable function $f(x)$. Ordinary calculus tells us how to estimate the error in the function value due to a small error in the input:

$$|f(x+e) - f(x)| = |f'(x)||e| + o(e).$$

That is, the absolute error in the output is roughly $|f'(x)|$ times the size of the absolute error in the input. What if we want to know how much *relative* error is amplified by a calculation rather than *absolute* error? Using first-order Taylor expansions and a little algebra, we have

$$\begin{aligned} \text{relative error in } f &= \frac{|f(x(1+\delta)) - f(x)|}{|f(x)|} \\ &\approx \left| \frac{xf'(x)}{f(x)} \right| |\delta| \\ &= \kappa_{f(x)} \times \text{relative error in } x \end{aligned}$$

The multiplier $\kappa_{f(x)}$ is called the *condition number* of the calculation. If this number is very large, the problem of evaluating f is called *ill-conditioned*. Note that the condition number is *a feature of the problem formulation and not a feature of the numerical method used to solve it*. Very ill-conditioned problems are typically hard to solve on the computer, since no matter how carefully the computation is coded, tiny relative errors in the inputs (e.g. due to rounding) can lead to huge relative errors in the outputs.

Digression: Conditioning of a root of a quadratic

Question: Recall that in our computation of π from lecture 1, we repeatedly computed the smaller solution x of the quadratic equation

$$x^2 - x + z = 0$$

as a function of z. What is the approximate condition number for this problem when z (and therefore x) is small?

We can approach this question in several ways.

Answer (version 1): The input is z; the output is the small root x. Therefore, the condition number is

$$\kappa = \left| \frac{z(dx/dz)}{x} \right|.$$

We can compute $dx/dz = 1/(1-2x)$ by implicit differentiation:

$$(2x-1)\frac{dx}{dz} + 1 = 0.$$

Therefore, if we also rearrange the quadratic to see $z = x - x^2$, we have

$$\kappa = \left| \frac{(x-x^2)}{x(1-2x)} \right| = \left| \frac{1-x}{1-2x} \right| = \left| 1 + x + O(x^2) \right|$$

When x is very small, $\kappa = 1 + x + O(x^2)$ is very close to 1.

Answer (version 2): If x is small, then a good approximation (relative error x!) is $x^2 - x \approx -x$. So

$$0 = x^2 - x + z/4 \approx -x + z/4.$$

This yields $x \approx z/4$, and

$$x(1+\delta) \approx -z(1+\delta)/4.$$

Answer (version 3): Remember the Taylor series for square roots around 1:

$$\sqrt{1+y} = 1 + \frac{1}{2}y + O(y^2).$$

Using this in the quadratic formula gives

$$x = \frac{1}{2}\left(1 - \sqrt{1-z}\right) = \frac{1}{2}\left(1 - (1 - z/2 + O(z^2))\right) = \frac{z}{4} + O(z^2)$$

From here, the analysis is the same as in version 2!

Answer (version 4): We know $g(x) = x^2 - x + z/4$ has a root close to zero when z is small. Near zero, we can use the linear approximation

$$0 = g(x) \approx g(0) + g'(0)x = z/4 - x$$

This linear approximation again gives $x \approx z/4$. You may recognize this as one step of Newton's method (which we will discuss more later in the class).

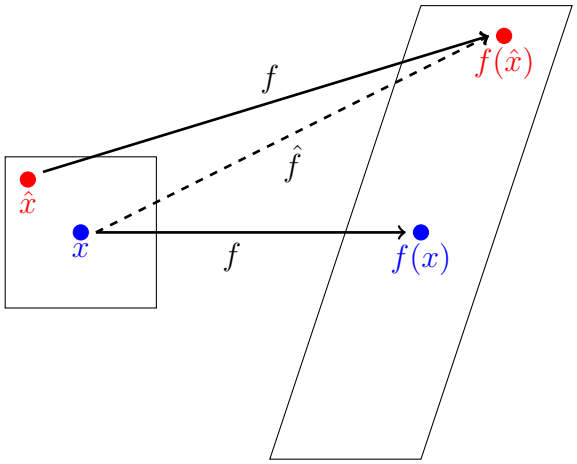

Figure 1: The difference between $f(x)$ had $\hat{f}(x)$ is *forward error*. If we can interpret $\hat{f}(x)$ as $f(\hat{x})$ for some $\hat{x}$, then the difference between x and $\hat{x}$ is *backward error*.

Forward and backward error

We are used to thinking about error as the difference between a computed value $\hat{f}(x)$ and a true value $f(x)$. This is *forward* error. For ill-conditioned problems, though, even the best algorithms can return results with bad forward error, if only because of approximation of the input values.

We can also think about *backward* error. That is, instead of writing our result in terms of an approximate function ($\hat{f}(x)$), we write a result in terms of an approximate *input* ($f(\hat{x})$). Numerical methods that always have small relative backward error are sometimes called *backward stable*. An advantage of using backward stable algorithms is that we can reason about their forward error properties in terms of the condition number.

The relationship between forward and backward error is illustrated in Figure 1.

Question: In the Archimedes example from lecture 1, was the standard quadratic formula always backward stable?

IEEE floating point arithmetic

I ran out of time, so this is doubtless one of the shortest summary of floating point that I have ever given or will ever give, and it leaves out some details. We'll do this more thoroughly on Monday.

The IEEE 754 floating point standard defines a set of normalized double-precision floating point numbers of the form:

$$(-1)^s \times 1.b_1 b_2 \ldots b_p \times 2^E, \quad E = e - \text{bias}$$

In double precision, we have $p = 52$ bits, 11 bits for the exponent, and a bias of 1023. There is also a single precision format (with 23 fraction bits and 8 bits for the exponent), but MATLAB uses double precision by default, and we will henceforth assume double precision unless otherwise stated.

The rule for floating point arithmetic is to return *the exact result, correctly rounded*. Usually, "correctly rounded" means "rounded to the nearest floating point number." It is hard to analyze errors using the "exact result, correctly rounded" characterization of floating point. Instead, we usually analyze floating point computations using a standard *model* for the behavior of normalized floating point numbers in terms of a bound on relative error. The model is that for $\otimes \in \{+, -, \times, /\}$, the floating point value $\text{fl}(x \otimes y)$ generated by computing $x \otimes y$ in floating point is

$$\text{fl}(x \otimes y) = (x \otimes y)(1 + \delta), \quad |\delta| \leq \epsilon_{\text{mach}}.$$

In double precision, the *machine epsilon*[1] is $\epsilon_{\text{mach}} = 2^{-53} \approx 10^{-16}$.

[1] For the purposes of this class, *machine epsilon* is the largest value δ such that $1 + \delta$ gets rounded to 1. According to some other authors, the machine epsilon is the distance between 1 and the next largest floating point number.