Notes for 2015-01-21

Logistics

- Welcome to CS 4220/5223 / MATH 4260!
- Please make sure you can access CMS and Piazza
- PS1 is posted and will be due in one week

Overview

Scientific computing involves mathematics, computation, and applications. We will be undertaking all three, and have students in the class who are majoring in all three (math majors, CS majors, and majors in various science and engineering disciplines). I have put prerequisites on the syllabus, and we are also doing a short assessment at the start of class. The quiz is not graded, but if you have trouble with these questions, you may want to come talk to me about whether this is the right class for you.

My intention is that you should read the book or other readings in advance, so that we can spend as much class time as possible working problems and getting past any points of confusion. Bring laptops to class! Also bring paper and a writing instrument. You'll need both.

A shot across the bow

Consider a cannon fired on flat ground. We can control the angle of θ, but not the muzzle velocity V_0. We want to hit a designated target.

- What is a model we can analyze computationally?
- What modeling assumptions are you making?
- How do you compute the target angle?
- What could go wrong with the computation?
- What are the sources of error? How sensitive are you to them?

Of course, if we had infinite ammunition, we could start shooting and adjust the angle based on how close we are to the target. The simplest version of this is *bisection*: figure out angles that overshoot and undershoot (a bracketing interval), then test an angle halfway between. This will give a new bracketing interval based on whether we overshot or undershot.

An alternate approach is to build a simplified local model of our model, something sufficiently simple that we can solve it. Each time we solve, we do another simplification. This leads to Newton and secant iterations.

Overview: Nonlinear equations

After this week (and the associated problems), you should come away with some understanding of

- *Algorithms* for equation solving, particularly bisection, Newton, secant, and fixed point iterations.
- *Analysis* of error recurrences in order fo find rates of convergence for algorithms; you should also understand a little about analyzing the sensitivity of the root-finding problem itself.
- *Application* of standard root-finding procedures to real problems. This frequently means some sketches and analysis done in advance in order to figure out appropriate rescalings and changes of variables, handle singularities, and find good initial guesses (for Newton) or bracketing intervals (for bisection).

A little long division

Let's begin with a question: Suppose I have a machine with hardware support for addition, subtraction, multiplication, and scaling by integer powers of two (positive or negative). How can I implement reciprocation? That is, if $d > 1$ is an integer, how can I compute $1/d$ without using division?

This is a linear problem, but as we will see, it presents many of the same issues as nonlinear problems.

```
function x = reciprocal_division(d, n)
  % Approximate x = 1/d by n steps of binary long division.

  r = 1;      % Current remainder
  x = 0;      % Current reciprocal estimate
  bit = 0.5;  % Value of a one in the current place

  for k = 1:n
    if r > d*bit
      x = x + bit;
      r = r - d*bit;
    end
    bit = bit/2;
  end
```

Figure 1: Approximate $1/d$ by n steps of binary long division.

```
function x = reciprocal_bisect(d, n)
  % Approximate x = 1/d by n steps of bisection
  % At each step f(x) = dx-1 is negative at the lower
  % bound, positive at the upper bound.

  hi = 1;   % Current upper bound
  lo = 0;   % Current lower bound

  for k = 1:n
    x  = (hi+lo)/2;
    fx = d*x-1;
    if fx > 0
      hi = x;
    else
      lo = x;
    end
  end
  x = (hi+lo)/2;
```

Figure 2: Approximate $1/d$ by n steps of bisection.

Method 1: From long division to bisection

Maybe the most obvious algorithm to compute $1/d$ is binary long division (the binary version of the decimal long division that we learned in grade school). To compute a bit in the kth place after the binary point (corresponding to the value 2^{-k}), we see whether $2^{-k}d$ is greater than the current remainder; if it is, then we set the bit to one and update the remainder. This algorithm is shown in Figure 1.

At step k of long division, we have an approximation $\hat{x}$, $\hat{x} \leq 1/d < \hat{x}+2^{-k}$, and a remainder $r = 1 - d\hat{x}$. Based on the remainder, we either get a zero bit (and discover $\hat{x} \leq 1/d < \hat{x} + 2^{-(k+1)}$), or we get a one bit (i.e. $\hat{x}+2^{-(k+1)} \leq 1/d < \hat{x}+2^{-k}$). That is, the long division algorithm is implicitly computing interals that contain $1/d$, and each step cuts the interval size by a factor of two. This is characteristic of *bisection*, which finds a zero of any continuous function $f(x)$ by starting with a bracketing interval and repeatedly cutting those intervals in half. We show the bisection algorithm in Figure 2.

Method 2: Almost Newton

You might recall *Newton's method* from a calculus class. If we want to estimate a zero near x_k, we take the first-order Taylor expansion near x_k and set that equal to zero:

$$f(x_{k+1}) \approx f(x_k) + f'(x_k)(x_{k+1} - x_k) = 0.$$

With a little algebra, we have

$$x_{k+1} = x_k - f'(x_k)^{-1} f(x_k).$$

Note that if x_* is the actual root we seek, then Taylor's formula with remainder yields

$$0 = f(x_*) = f(x_k) + f'(x_k)(x_* - x_k) + \frac{1}{2} f''(\xi)(x_* - x_k)^2.$$

Now subtract the Taylor expansions for $f(x_{k+1})$ and $f(x_*)$ to get

$$f'(x_k)(x_{k+1} - x_*) + \frac{1}{2} f''(\xi)(x_k - x_*)^2 = 0.$$

This gives us an iteration for the error $e_k = x_k - x_*$:

$$e_{k+1} = -\frac{1}{2}\frac{f''(\xi)}{f'(x_k)}e_k^2.$$

Assuming that we can bound $f''(\xi)/f(x_k)$ by some modest constant C, this implies that a small error at e_k leads to a *really* small error $|e_{k+1}| \leq C|e_k|^2$ at the next step. This behavior, where the error is squared at each step, is *quadratic convergence*.

If we apply Newton iteration to $f(x) = dx - 1$, we get

$$x_{k+1} = x_k - \frac{dx_k - 1}{d} = \frac{1}{d}.$$

That is, the iteration converges in one step. But remember that we wanted to avoid division by d! This is actually not uncommon: often it is inconvenient to work with $f'(x_k)$, and so we instead cook up some approximation. In this case, let's suppose we have some $\hat{d}$ that is an integer power of two close to d. Then we can write a modified Newton iteration

$$x_{k+1} = x_k - \frac{dx_k - 1}{\hat{d}} = \left(1 - \frac{d}{\hat{d}}\right) x_k + \frac{1}{\hat{d}}.$$

Note that $1/d$ is a *fixed point* of this iteration:

$$\frac{1}{d} = \left(1 - \frac{d}{\hat{d}}\right) \frac{1}{d} + \frac{1}{\hat{d}}.$$

If we subtract the fixed point equation from the iteration equation, we have an iteration for the error $e_k = x_k - 1/d$:

$$e_{k+1} = \left(1 - \frac{d}{\hat{d}}\right) e_k.$$

So if $|d - \hat{d}|/|d| < 1$, the errors will eventually go to zero. For example, if we choose $\hat{d}$ to be the next integer power of two larger than d, then $|d - \hat{d}|/|\hat{d}| < 1/2$, and we get at least one additional binary digit of accuracy at each step.

When we plot the error in long division, bisection, or our modified Newton iteration on a semi-logarithmic scale, the decay in the error looks (roughly) like a straight line. That is, we have *linear* convergence. But we can do better!

Method 3: Actually Newton

We may have given up on Newton iteration too easily. In many problems, there are multiple ways to write the same nonlinear equation. For example, we can write the reciprocal of d as x such that $f(x) = dx - 1 = 0$, or we can write it as x such that $g(x) = x^{-1} - d = 0$. If we apply Newton iteration to g, we have

$$x_{k+1} = x_k - \frac{g(x_k)}{g'(x_k)} = x_k + x_k^2(x_k^{-1} - d) = x_k(2 - dx_k).$$

As before, note that $1/d$ is a fixed point of this iteration:

$$\frac{1}{d} = \frac{1}{d}\left(2 - d\frac{1}{d}\right).$$

Given that $1/d$ is a fixed point, we have some hope that this iteration will converge — but when, and how quickly? To answer these questions, we need to analyze a recurrence for the error.

We can get a recurrence for error by subtracting the true answer $1/d$ from both sides of the iteration equation and doing some algebra:

$$\begin{aligned}
e_{k+1} &= x_{k+1} - d^{-1} \\
&= x_k(2 - dx_k) - d^{-1} \\
&= -d(x_k^2 - 2d^{-1}x_k + d^{-2}) \\
&= -d(x_k - d^{-1})^2 \\
&= -de_k^2
\end{aligned}$$

In terms of the relative error $\delta_k = e_k/d^{-1} = de_k$, we have

$$\delta_{k+1} = -\delta_k^2.$$

If $|\delta_0| < 1$, then this iteration converges — and once convergence really sets in, it is ferocious, roughly doubling the number of correct digits at each step. Of course, if $|\delta_0| > 1$, then the iteration diverges with equal ferocity. Fortunately, we can get a good initial guess in the same way we got a good guess for the modified Newton iteration: choose the first guess to be a nearby integer power of two.

On some machines, this sort of Newton iteration (intelligently started) is actually the preferred method for division.

The big picture

Let's summarize what we have learned from this example (and generalize slightly to the case of solving $f(x) = 0$ for more interesting f):

- *Bisection* is a general, robust strategy. We just need that f is continuous, and that there is some interval $[a, b]$ so that $f(a)$ and $f(b)$ have different signs. On the other hand, it is not always easy to get a bracketing interval; and once we do, bisection only halves that interval at each step, so it may take many steps to reach an acceptable answer. Also, bisection is an intrinsically one-dimensional construction.

- *Newton iteration* is a standard workhorse based on finding zeros of successive linear approximations to f. When it converges, it converges ferociously quickly. But Newton iteration requires that we have a derivative (which is sometimes inconvient), and we may require a good initial guess.

- A *modified Newton iteration* sometimes works well if computing a derivative is a pain. There are many ways we can modify Newton method for our convenience; for example, we might choose to approximate $f'(x_k)$ by some fixed value, or we might use a secant approximation.

- It is often convenient to work with *fixed point iterations* of the form

  $$x_{k+1} = g(x_k),$$

  where the number we seek is a fixed point of g ($x_* = g(x_*)$). Newton-like methods are an example of fixed point iteration, but there are others. Whenever we have a fixed point iteration, we can try to write an iteration for the error:

  $$e_{k+1} = x_{k+1} - x_* = g(x_k) - g(x_*) = g(x_* + e_k) - g(x_*).$$

  How easy it is to analyze this error recurrence depends somewhat on the properties of g. If g has two derivatives, we can write

  $$e_{k+1} = g'(x_*)e_k + \frac{1}{2}g''(\xi_k)e_k^2 \approx g'(x_*)e_k.$$

  If $g'(x_*) = 0$, the iteration converges *superlinearly.* If $0 < |g'(x_*)| < 1$, the iteration converges linearly, and $|g'(x_*)|$ is the rate constant. If $|g'(x_*)| > 1$, the iteration diverges.