2020-03-11

1 The best of all possible worlds

Last time, we discussed three methods of solving $f(x) = 0$: Newton, modified Newton, and bisection. Newton is potentially faster than bisection; bisection is more reliable. Ideally, we would like something that is both fast *and* robust. We consider two different approaches to this problem: Brent's method (1D), and globalized Newton (which will generalize).

1.1 Newton with line search

One difficulty with Newton iteration is that sometimes the step is in the right direction, but has the wrong magnitude. We can get around this with a *line search* strategy: we propose a Newton step, but accept the step only if it reduces $|f(x)|$. If $|f(x)|$ goes up, we go in the same direction but by a smaller amount; a typical choice is to cut the step in half. In code, we have the following.

```
function simple_newton(f, df, x; atol=1e-6, maxiter=100)
    fx = f(x)
    dfx = df(x)
    dx = -fx/dfx
    for fevals = 1:maxiter
        if abs(fx) < atol
            return x, true
        end
        xnew = x+dx
        fxnew = f(xnew)
        if abs(fxnew) < abs(fx)
            x   = xnew
            fx  = fxnew
            dfx = df(x)
            dx  = -fx/dfx
        else
            # If we increase |f(x)|, try a smaller update
            dx = dx/2;
        end
    end
    return x, false
end
```

Newton with line search converges more frequently than Newton with no guards, but it can still go astray. There is nothing in the setup for a guarded Newton iteration that guarantees the function f even has a zero; and even if it does, it is possible to set up functions where Newton always heads in the wrong direction. In order to go from "works better than Newton" to "works all the time," we need another trick.

1.2 Secant iteration and Brent's method

One of the annoying properties of Newton's method is that it requires that we compute the derivative of f. In some cases, we may not have this derivative in closed form, but we can always estimate using finite differences:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}.$$

In the setting of a root finding iteration, it is natural to use a derivative approximation based on the last two steps of the iteration; this gives us the *secant iteration*

$$x_{k+1} = x_k - \frac{f(x_k)(x_k - x_{k-1})}{f(x_k) - f(x_{k-1})}.$$

The secant iteration is superlinearly convergent, though not quadratically convergent[1] Unlike Newton, we need *two* starting points for the iteration; but if we start with an interval $[a, b]$ such that f has a sign change, it is natural to choose a and b as the initial guesses.

Unfortunately, secant iteration can also go astray. Fortunately, we can combine secant iteration with bisection to get both speed and robustness. The basic idea is:

- At each step, maintain an interval $[\alpha, \beta]$ such that f has a sign change between the end points.
- If secant iteration is converging quickly, try taking a new point based on a secant step.
- If the secant step falls out of bounds, or if secant iteration has not improved the bounding interval sufficiently in the past few steps, consider a new point based on a bisection step.

[1] You can read the convergence theory elsewhere.

- Reduce the interval based on the sign of f at the new point (whether from a bisection or a secant point) and repeat.

This combined method is known as *Brent's method*, and it is the usual default root-finder for one-dimensional problems. Unfortunately, it is an intrinsically one-dimensional process — there is no natural generalization with similar robustness properties for solving systems of equations.

Use a routine, or roll your own?

The MATLAB function `fzero` is a fast, reliable black-box root-finding algorithm based on a combination of bisection (for safety) and interpolation-based methods (for speed). If you provide an initial interval containing exactly one zero, and if the root you seek is not too sensitive, `fzero` will find the root you seek to high accuracy (the default relative error tolerance is about $2\epsilon_{\text{mach}}$). I use the function often, and recommend it to you.

That said, there are a few reasons to write your own root-finding algorithms, at least some of the time:

1. Not all the world is MATLAB, and you may sometimes find that you have to write these things yourself.

2. Black box approaches are far less useful for problems involving multiple variables. Consequently, it's worth learning to write Newton-like methods in one variable so that you can learn their properties well enough to work with similar algorithms in more than one variable.

3. Actually walking through the internals of a root-finding algorithm can be a terrific way to gain insight into how to formulate your problems so that a standard root finder can solve them.

Sensitivity and error

Suppose we want to find x_* such that $f(x_*) = 0$. On the computer, we actually have $\hat{f}(\hat{x}_*) = 0$. We'll assume that we're using a nice, robust code like `fzero`, so we have a very accurate zero of $\hat{f}$. But this still leaves the question: how well do $\hat{x}_*$ and x_* approximate each other? In other words, we want to know the sensitivity of the root-finding problem.

If $\hat{x}_* \approx x_*$, then

$$f(\hat{x}_*) \approx f'(x_*)(\hat{x}_* - x_*).$$

Using the fact that $\hat{f}(\hat{x}_*) = 0$, we have that if $|\hat{f} - f| < \delta$ for arguments near x_*, then

$$|f'(x_*)(\hat{x}_* - x_*)| \lesssim \delta.$$

This in turn gives us

$$|\hat{x}_* - x_*| \lesssim \frac{\delta}{f'(x_*)}.$$

Thus, if $f'(x_*)$ is close to zero, small rounding errors in the evaluation of f may lead to large errors in the computed root.

It's worth noting that if $f'(x_*) = 0$ (i.e. if x_* is a multiple root), that doesn't mean that x_* is completely untrustworthy. It just means that we need to take more terms in a Taylor series in order to understand the local behavior. In the case $f'(x_*) = 0$, we have

$$f(\hat{x}_*) \approx \frac{1}{2} f''(x_*)(\hat{x}_* - \hat{x}_*),$$

and so we have

$$|\hat{x}_* - \hat{x}_*| \leq \sqrt{\frac{2\delta}{f''(x_*)}}.$$

So if the second derivative is well behaved and δ is on the order of around 10^{-16}, for example, our computed $\hat{x}$ might be accurate to within an absolute error of around 10^{-8}.

Understanding the sensitivity of root finding is not only important so that we can be appropriately grim when someone asks for impossible accuracy. It's also important because it helps us choose problem formulations for which it is (relatively) easy to get good accuracy.

Choice of functions and variables

Root-finding problems are hard or easy depending on how they are posed. Often, the initial problem formulation is not the most convenient. For example, consider the problem of finding the positive root of

$$f(x) = (x+1)(x-1)^8 - 10^{-8}.$$

This function is terrifyingly uninformative for values close to 1. Newton's iteration is based on the assumption that a local, linear approximation provides a good estimate of the behavior of a function. In this problem, a linear approximation is terrible. Fortunately, the function

$$g(x) = (x+1)^{1/8}(x-1) - 10^{-1}$$

has the same root, which is very nicely behaved.

There are a few standard tricks to make root-finding problems easier:

- Scale the function. If $f(x)$ has a zero at x_*, so does $f(x)g(x)$; and sometimes we can analytically choose a scaling function to make the root finding problem easier.

- Otherwise transform the function. For example, in computational statistics, one frequently would like to maximize a likelihood function

  $$L(\theta) = \prod_{j=1}^{n} f(x_j; \theta)$$

  where $f(x;\theta)$ is a probability density that depends on some parameter θ. One way to do this would be find zeros of $L'(\theta)$, but this often leads to scaling problems (potential underflow) and other numerical discomforts. The standard trick is to instead maximize the log-likelihood function

  $$\ell(\theta) = \sum_{j=1}^{n} \log f(x_j; \theta),$$

  often using a root finder for $\ell'(\theta)$. This tends to be a much more convenient form, both for analysis and for computation.

- Change variables. A good rule of thumb is to pick variables that are naturally *dimensionless*[2] For difficult problems, these dimensionless variables are often very small or very large, and that fact can be used to simplify the process of coming up with good initial guesses for Newton iteration.

[2]Those of you who are interested in applied mathematics more generally should look up the Buckingham Pi Theorem — it's a tremendously useful thing to know about.

Starting points

All root-finding software requires either an initial guess at the solution or an initial interval that contains the solution. This sometimes calls for a little cleverness, but there are a few standard tricks:

- If you know where the problem comes from, you may be able to get a good estimate (or bounds) by "application reasoning." This is often the case in physical problems, for example: you can guess the order of magnitude of an answer because it corresponds to some physical quantity that you know about.
- Crude estimates are often fine for getting upper and lower bounds. For example, we know that for all $x > 0$,

  $$\log(x) \leq x - 1$$

  and for all $x \geq 1$, $\log(x) > 0$. So if I wanted to $x + \log(x) = c$ for $c > 1$, I know that c should fall between x and $2x - 1$, and that gives me an initial interval. Alternatively, if I know that $g(z) = 0$ has a solution close to 0, I might try Taylor expanding g about zero – including higher order terms if needed – in order to get an initial guess for z.
- Sometimes, it's easier to find local minima and maxima than to find zeros. Between any pair of local minima and maxima, functions are either monotonically increasing or monotonically decreasing, so there is either exactly one root in between (in which case there is a sign change between the local min and max) or there are zero roots between (in which case there is no sign change). This can be a terrific way to start bisection.

Problems to ponder

1. Analyze the convergence of the fixed point iteration

   $$x_{k+1} = c - \log(x_k).$$

   What is the equation for the fixed point? Under what conditions will the iteration converge with a good initial guess, and at what rate will the convergence occur?

2. Repeat the previous exercise for the iteration $x_{k+1} = 10 - \exp(x_k)$.

3. Analyze the convergence of Newton's iteration on the equation $x^2 = 0$, where $x_0 = 0.1$. How many iterations will it take to get to a number less than 10^{-16}?

4. Analyze the convergence of the fixed point iteration $x_{k+1} = x_k - \sin(x_k)$ for x_k near zero. Starting from $x = 0.1$, how many iterations will it take to get to a number less than 10^{-16}?

5. Consider the cubic equation

$$x^3 - 2x + c = 0.$$

Describe a general purpose strategy for finding *all* the real roots of this equation for a given c.

6. Suppose we have some small number of samples $X_1, \ldots, X_m$ drawn from a Cauchy distribution with parameter θ (for which the pdf is)

$$f(x, \theta) = \frac{1}{\pi} \frac{1}{1 + (x - \theta)^2}.$$

The *maximum likelihood estimate* for θ is the function that maximizes

$$L(\theta) = \prod_{j=1}^{m} f(X_j, \theta).$$

Usually, one instead maximizes $l(\theta) = \log L(\theta)$ — why would this make sense numerically? Derive a MATLAB function to find the maximum likelihood estimate for θ by finding an appropriate solution to the equation $l'(\theta) = 0$.

7. The Darcy friction coefficient f for turbulent flow in a pipe is defined in terms of the Colebrook-White equation for large Reynolds number Re (greater than 4000 or so):

$$\frac{1}{\sqrt{f}} = -2 \log_{10} \left(\frac{\epsilon / D_h}{3.7} + \frac{2.51}{\mathrm{Re}\sqrt{f}} \right)$$

Here ϵ is the height of the surface roughness and D_h is the diameter of the pipe. For a 10 cm pipe with 0.1 mm surface roughness, find f for Reynolds numbers of 10^4, 10^5, and 10^6. Ideally, you should use a Newton iteration with a good initial guess.

8. A cable with density of 0.52 lb/ft is suspended between towers of equal height that are 500 ft apart. If the wire sags by 50 ft in between, find the maximum tension T in the wire. The relevant equations are

$$c + 50 = c \cosh\left(\frac{500}{2c}\right)$$
$$T = 0.52(c + 50)$$

Ideally, you should use a Newton iteration with a good initial guess.