CS 542G: Nonlinear Systems, Quasi-Newton, Gravity

Robert Bridson

November 3, 2008

1 Solving Nonlinear Systems

The version of Newton's method that you probably saw first in a calculus class is the one for solving nonlinear equations, of the form $g(x) = 0$. It's most easily discussed in 1D in terms of the graph of the function $g(x)$. Starting from a guess $x^{(k)}$ at the root, the tangent line to the graph at $x^{(k)}$ is constructed—i.e. the line of slope $g'(x^{(k)})$ passing through the point $(x^{(k)}, g(x^{(k)}))$. Where this line intersects the x-axis defines the next guess, $x^{(k+1)}$.

This extends naturally to higher dimensions (where $x \in \setminus^n$ and $g : \mathbb{R}^n \to \mathbb{R}^n$), though it's harder to visualize. The idea is to replace the difficult g with a simpler model function that's easy to solve with, using Taylor series to approximate g as before. It's pretty hard to solve higher dimensional quadratics or worse, and a constant model function can't be meaningfully solved, so the natural choice is to use the linear Taylor polynomial:

$$g(x^{(k)} + \Delta x) = g(x^{(k)} + \frac{\partial g(x^{(k)})}{\partial x}\Delta x + O(\Delta x^2)$$

Here the matrix $\partial g/\partial x$ is the Jacobian of g, the analogy of the gradient for a scalar function.

Replacing g with this **linearization** in the system of equations gives a linear system:

$$\begin{aligned} g(x^{(k)} + \frac{\partial g(x^{(k)})}{\partial x}\Delta x &= 0 \\ \Leftrightarrow \qquad \frac{\partial g(x^{(k)})}{\partial x}\Delta x &= -g(x^{(k)}) \end{aligned}$$

Assuming the Jacobian matrix is non-singular, we can solve this for Δx, getting a new guess $x^{(k+1)} = x^{(k)} + \Delta x$. When started near enough the solution, this Newton's method also converges quadratically, meaning the number of accurate digits doubles each iteration (and thus we will be confidently done in 3–5 iterations for double precision, after convergence begins).

The connection to the optimization-form of Newton we saw before is simple enough: a local minimum of $f(x)$ satisfies $\nabla f(x) = 0$. Letting $g(x) = \nabla f(x)$ and plugging it into the nonlinear system version of Newton is exactly equivalent to the optimization form of Newton: the Hessian matrix of f is the same as the Jacobian matrix of g.

Similar to the case of optimization, the nonlinear system Newton can be made more robust with ideas such as line search and regularization of the matrix. However, life is rather more difficult in this case. In the case of minimization, we know the matrix has to be symmetric (it's a Hessian), and near the solution has to be positive semi-definite. For a generic nonlinear system, the Jacobian matrix could be unsymmetric and indefinite even at the solution—and there isn't even an obvious analogy to the steepest descent direction since we are not descending. The moral of the story here is that it's a bad idea to throw information away: if you know you're minimizing f, don't throw $g(x) = \nabla f(x)$ into a generic nonlinear system solver, since it can't take advantage of the knowledge that the Jacobian has to be symmetric etc.

2 Root-Finding

It's also worth taking a deeper look at the 1D version of solving a nonlinear equation, a.k.a. **root-finding**. While Newton works wonderfully well when it converges, there's no guarantee it won't in fact diverge from an arbitrary initial guess. In some cases (for example, computing $x = \sqrt{y}$ by solving $x^2 - y = 0$ with Newton) we might have extra knowledge that lets us derive a reliable initial guess, but often we have no idea.

One particularly robust alternative is **binary search**. Binary search requires as input two values $x_{\text{lo}} < x_{\text{hi}}$ where the sign of $g(x_{\text{lo}})$ is different from $g(x_{\text{hi}})$. Assuming g is continuous (as otherwise all bets are off) the intermediate value theorem then guarantees there is at least one root in the interval $(x_{\text{lo}}, x_{\text{hi}})$. (Synonyms for root in this context include "sign crossing" or "zero crossing" for obvious reasons.)

Finding such an interval isn't necessarily easy. If we know that $\lim_{x\to-\infty} g(x)$ and $\lim_{x\to\infty} g(x)$ are of different signs (and may or may not be infinite)—for example, if g is monotonic, the analogy of convexity in minimization problems, and has a root—then **bracketing** can be used. That is, we start with a guess at an interval, and expand it exponentially until we find a sign change. On the other hand, if we don't know anything about g, we might be out of luck.

Once such an interval is found, binary search takes the midpoint

$$x_{\text{mid}} = \frac{x_{\text{lo}} + x_{\text{hi}}}{2}$$

and checks $g(x_{\text{mid}})$. If it's close enough to zero (within a user-specified tolerance) then we can stop.

Otherwise we continue with either the interval $(x_{\mathrm{lo}}, x_{\mathrm{mid}})$ or $(x_{\mathrm{mid}}, x_{\mathrm{hi}})$, whichever has a sign change. In this way, binary search halves the length of the interval in which a root must be found every iteration, and thus must converge linearly to the solution, adding an extra bit of accuracy each time. We can of course also stop when the interval is small enough—in particular, we have to stop when we hit the limit of floating point precision.

Binary search has a strong guarantee about its convergence, but can be slow. An algorithm somewhere between Newton and binary search is **secant search**, which has interesting relatives in higher dimensions.

The idea of secant search is to mimic Newton's linearization, but approximate the tangent line instead with a **secant** or **chord**: a line connecting two points we've already seen on the curve. Classically these could be chosen as $(x^{(k-1)}, g(x^{(k-1)}))$ and $(x^{(k)}, g(x^{(k)}))$, though we can gain robustness by instead using the endpoints of an interval containing a root, as in binary search. In this case, the equation of the line connecting the endpoints is:

$$y = \frac{g(x_{\mathrm{hi}}) - g(x_{\mathrm{lo}})}{x_{\mathrm{hi}} - x_{\mathrm{lo}}}(x - x_{\mathrm{lo}}) + g(x_{\mathrm{lo}})$$

which we could also write as

$$g(x_{\mathrm{lo}} + \Delta x) \approx g(x_{\mathrm{lo}}) + \frac{g(x_{\mathrm{hi}}) - g(x_{\mathrm{lo}})}{x_{\mathrm{hi}} - x_{\mathrm{lo}}}\Delta x$$

to make the connection to Newton clearer: the fraction is an estimate of the derivative (slope) of g over the interval. This line has a root at

$$x_{\mathrm{mid}} = x_{\mathrm{lo}} - g(x_{\mathrm{lo}})\frac{x_{\mathrm{hi}} - x_{\mathrm{lo}}}{g(x_{\mathrm{hi}}) - g(x_{\mathrm{lo}})}$$

If g is close to linear in the interval, which if it's smooth and the interval is small enough it must be (thanks to Taylor), then this should be an excellent guess at the root. If it's not quite accurate enough, we can continue as with binary search, on a sub-interval. In fact, we can intermix steps of secant search and binary search to get a hybrid algorithm that's even more robust; we can also try the classic secant step or even Newton if it gives a guess within the current interval, and thus maintain robustness.

Secant search doesn't converge quite as fast as Newton, but when convergence begins it is faster than linear. However, it has a great advantage in that it doesn't need the derivative to be evaluated. As mentioned last time in the context of Gauss-Newton, calculating derivatives is often much more expensive than evaluating the function itself—apart from the special case of polynomials written in standard form—and is also numerically more delicate. For example, if $g(x)$ is a product of k terms, the product-rule of differentiation ends up increasing the complexity of the derivative by a factor of k. This potential savings can mean that secant search will be faster than Newton, despite requiring more iterations.

3 Quasi-Newton Methods

The idea behind secant search, approximating Newton by using values of the function to estimate a derivative instead of differentiating, can also be used in higher dimensional minimization problems. This leads to **Quasi-Newton** methods.

Extending the slope estimate from 1D to higher dimensions gives an approximation to the directional derivative:

$$\begin{aligned} \frac{\partial g}{\partial p}(x) &= \lim_{h \to 0} \frac{g(x+hp) - g(x)}{h} \\ &\approx g(x+p) - g(x) \end{aligned}$$

If we take $p = x^{(k)} - x^{(k-1)}$, the difference between successive guesses in an algorithm which near convergence should get small, we see the difference $g(x^{(k)} - g(x^{k-1})$ is an approximation of the directional derivative of g with respect to p. We can also define the directional derivative in terms of the Jacobian:

$$\frac{\partial g(x)}{\partial p} = \frac{\partial g}{\partial x} p$$

and thus the difference in values of g is telling us some information about the Jacobian.

Now let $g(x)$ be $\nabla f(x)$ for some scalar objective function f we are trying to minimize, and thus the Jacobian of g is just the Hessian H of f. Our result is:

$$H\,(x^{(k)} - x^{(k-1)}) \approx \nabla f(x^{(k)}) - \nabla f(x^{(k-1)})$$

Quasi-Newton methods use this to build an approximation to the Hessian (or its inverse, which is of more use in a Newton-like iteration), finding matrices $\bar{H}$ that exactly satisfy the **secant condition**:

$$\bar{H}\,(x^{(k)} - x^{(k-1)}) = \nabla f(x^{(k)}) - \nabla f(x^{(k-1)})$$

or, if the inverse is favoured,

$$x^{(k)} - x^{(k-1)} = \bar{H}^{-1}(\nabla f(x^{(k)}) - \nabla f(x^{(k-1)}))$$

The most successful of the Quasi-Newton methods is one called BFGS (Broyden-Fletcher-Goldfarb-Shanno). BFGS starts with an initial estimate of the inverse of the Hessian, perhaps the inverse at the initial guess or even just the identity matrix, and modifies it every step with a low-rank update which enforces the most recent secant condition—while preserving symmetry and even positive definiteness. We won't go further into details on BFGS, but if you are intrigued there are plenty of good references out there (for example, see the book by Nocedal and Wright on optimization).

Quasi-Newton methods are important mainly for the same reasons we are interested in secant search in 1D: they avoid the need for evaluating the Hessian, which can be expensive, yet retain some of the convergence advantages of Newton. Furthermore, they can be designed to be a bit more robust—in the case of BFGS, maintaining an SPD matrix even when the Hessian isn't, for example.

4 The N-Body Problem

The last topic in the course is partial differential equations (PDEs). We'll take a bit of a sneaky route into them, however, by way of the n-body problem. The particular instance we'll focus on is simulating the gravitational interaction of n point masses. This comes up in astrophysics for example, looking at the evolution of things like galaxies where each point might represent a star. We have skipped over the topic of **time integration**, solving for the positions of the points over time given the forces and masses (using $F = ma$), but the critical step in this is being able to evaluate the forces on all the points.

Let $\{x_i\}_{i=1}^n$ be the locations of the point masses, and $\{m_i\}_{i=1}^n$ their masses. The force on point i is the sum of the forces due to all the other masses:

$$F_i = \sum_{j \neq i} f_{ij}$$

and the force on i due to $j \neq i$ is given by Newton's law of gravitation:[1]

$$f_{ij} = -G \frac{m_i m_j}{\|x_j - x_i\|^3}(x_j - x_i)$$

where G is the gravitational constant, a surprisingly hard number to measure ($G \approx 6.67428 \times 10^{-11} Nm^2/kg^2$ in SI units). Even exploiting the symmetry $\vec{f_{ij}} = -\vec{f_{ji}}$ leads to $O(n^2)$ work just to evaluate the forces on all the points. This ends up being the critical bottleneck for large calculations, such as attempts to simulate galaxy formation, where ideally n might be in the billions.

There isn't any obvious way to re-express those forces with an asymptotically faster formula in general. However, taking a step back and considering the problem as a whole we can see there are going to be many sources of error: error due to assuming stars are ideal point masses, error due to ignoring interstellar gas, error in measurement or otherwise setting up the problem, rounding error from floating point arithmetic, errors in approximating the motion of the points if we simulate the system forward in time. Given all this, there's no reason we *have* to evaluate the forces exactly—we have the liberty of just approximating them as long as we can control the error. This is generally a worthwhile perspective on

[1] We neglect general relativity effects here!

any problem: it's not always worth the effort and computational expense of calculating one part of the problem "exactly" when other errors remain large.

The approximation we have in mind is actually one we have already made in setting up the problem: lumping together matter spread over a small region of space into an idealized point mass. The Sun is obviously a very large body, but it's extremely tiny compared to the distance to other stars (or even the planets in the solar system: Mercury is on the order of 100 times further from the Sun than the Sun's radius). With some hand-waving, that justifies treating it as a point mass in these contexts. Of course, this would be a bad assumption if we are looking at much nearer effects, such as the dynamcis of gases inside the Sun itself, so we will need to replace the hand-waving with something more definite next lecture!

Going further with this intuition, that a point mass can well approximate a small region of matter as long as that region is far away, we see an opportunity for speeding up the n-body force calculations. Look again at the sum of forces F_i on point i. If a subset of the other $n-1$ points can be grouped into a "cluster" of points in a small region of space far away from x_i, we can replace that cluster in the F_i expression with a single term: a single point at the centre of mass of the cluster, with mass equal to the total mass of the cluster. If we can find enough of these clusters, we will be able to significantly cut down on the cost of computing F_i.

Next time we will have to address two things in connection with this idea: how to quantify (and control) the error incurred by replacing a cluster with a single point, and how to efficiently determine these clusters.