18.330 Lecture Notes: Nonlinear Root Finding and a Glimpse at Optimization

Homer Reid

March 17, 2015

Contents

1 Overview

Root-finding problems take the general form

$$\text{find } x \text{ such that } f(x) = 0$$

where $f(x)$ will generally be some complicated nonlinear function. (It had better be nonlinear, since otherwise we hardly need numerical methods to solve.)

The multidimensional case

The root-finding problem has an obvious and immediate generalization to the higher-dimensional case:

$$\text{find } \mathbf{x} \text{ such that } \mathbf{f}(\mathbf{x}) = 0 \tag{1}$$

where $\mathbf{x}$ is an N-dimensional vector and $\mathbf{f}(\mathbf{x})$ is an M-dimensional vector-valued function (we do not require $M = N$). Equation (1) is unambiguous; it is asking us to find the origin of the vector space $\mathbb{R}^M$, which is a single unique point in that space.

Root-finding is an iterative procedure

In contrast to many of the algorithms we have seen thus far, the algorithms we will present for root-finding are *iterative:* they start with some initial guess and then repeatedly apply some procedure to improve this guess until it has converged (i.e. it is "good enough.") What this means is that we generally don't know *a priori* how much work we will need to do to find our root. That might make it sound as though root-finding algorithms take a long time to converge. In fact, in many cases the opposite is true; as we will demonstrate, many of the root-finding algorithms we present exhibit dramatically faster convergence than any of the other algorithms we have seen thus far in the course.

1.1 Examples of root-finding problems

Ferromagnets

The mean-field theory of the D-dimensional Ising ferromagnet yield the following equation governing the spontaneous magnetization m:

$$m = \tanh \frac{2Dm}{T} \tag{2}$$

where T is the temperature.[1] For a given temperature, we solve (2) numerically to compute m, which characterizes how strongly magnetized our magnet is.

[1] Measured in units of the nearest-neighbor spin coupling J in the Ising hamiltonian.

Resonance frequencies of structures

A very common application of numerical root-finders is identifying the frequencies at which certain physical structures will resonate. As one example, consider a one-dimensional model of a optical fiber consisting of a slab of dielectric material with thickness T (we might have something like $T = 10\,\mu\text{m}$) and refractive index n (for example, silicon has $n \approx 3.4$). Then from Maxwell's equations it's easy to derive that the following relation

$$\tanh \frac{n\omega T}{c} - \frac{2n}{1+n^2} = 0$$

must hold between T, n, and the angular frequency ω in order for a resonant mode to exist. (Here c is the speed of light in vacuum.)

The Riemann ζ function

The greatest unsolved problem in mathematics today is a root-finding problem. The Riemann ζ ("zeta") function is defined by a contour integral as

$$\zeta(s) = \frac{\Gamma(1-s)}{2\pi i} \oint_{\mathcal{C}} \frac{s^{z-1}}{e^{-z}-1} dz$$

where $\mathcal{C}$ is a certain contour in the complex plane. This function has "trivial" roots at negative even integers $s = -2, -4, -6, \cdots$, as well as nontrivial roots at other values of s. To date many nontrivial roots of the equation $\zeta(s) = 0$ have been identified, but *they all have the property that their real part is* $\frac{1}{2}$. The *Riemann hypothesis* is the statement that in fact *all* nontrivial roots of $\zeta(s) = 0$ have $\text{Re } s = \frac{1}{2}$, and if you can prove this statement (or find a counterexample by producing s such that $\zeta(s) = 0, \text{Re } s \neq \frac{1}{2}$) then the Clay Mathematics Institute in Harvard Square will give you a million dollars.

Linear eigenvalue problems

Let $\mathbf{A}$ be an $N \times N$ matrix and consider the problem of determining eigenpairs $(\mathbf{x}, \lambda)$, where $\mathbf{x}$ is an N-dimensional vector and λ is a scalar. These are roots of the equation

$$\mathbf{A}\mathbf{x} - \lambda\mathbf{x} = 0. \tag{3}$$

Because both λ and $\mathbf{x}$ are unknown, we should think of (3) as an $N+1$-dimensional nonlinear root-finding problem, where the $N+1$-dimensional vector of unknowns we seek is $\begin{pmatrix} \mathbf{x} \\ \lambda \end{pmatrix}$, and where the nonlinearity arises because the $\lambda\mathbf{x}$ term couples the unknowns to each other.

Although (3) is thus a nonlinear problem if we think of it as an $N+1$-dimensional problem, it is separately linear in each of λ and $\mathbf{x}$, and for this reason we call it the "linear eigenvalue problem." The linear eigenvalue problem is *not* typically solved using the methods discussed in these notes; instead,

it is generally solved using a set of extremely well-developed methods of numerical linear algebra (namely, Householder decomposition and QR factorization), which are implemented by LAPACK and available in all numerical software packages including JULIA and MATLAB.

Nonlinear eigenvalue problems

On other other hand, it may be the case that the matrix $\mathbf{A}$ in (3) depends on its own eigenvalues and/or eigenvectors. In this case we have a *nonlinear* eigenvalue problem and the usual methods of numerical linear algebra do not apply; in this case we must solve using nonlinear root-finding methods such as Newton's method.

Nonlinear boundary-value problems

In our unit on boundary-value problems we considered the problem of a particle motion in a time-dependent force field $f(t)$. We considered an ODE boundary-value problem of the form

$$\frac{d^2x}{dt^2} = f(t), \qquad x(t_a) = x(t_b) = 0 \tag{4}$$

and we showed that finite-difference techniques allow us to reduce this ODE to a linear system of equations of the form

$$\mathbf{A}\mathbf{x} = \mathbf{f} \tag{5}$$

where $\mathbf{A}$ is a matrix with constant entries, $\mathbf{x}$ is a vector of (unknown) samples of the particle position $x(t_n)$ at time points t_n, and $\mathbf{f}$ is a vector of (known) samples of the forcing function at those time points:

$$\mathbf{x} = \begin{pmatrix} x(t_1) \\ \vdots \\ x(t_N) \end{pmatrix} \equiv \begin{pmatrix} x_1 \\ \vdots \\ x_N \end{pmatrix} = \textbf{unknown}, \qquad \mathbf{f} = \begin{pmatrix} f(t_1) \\ \vdots \\ f(t_N) \end{pmatrix} = \textbf{known}.$$

Equation (5) may be thought of as a linear root-finding problem, i.e. we seek a root of the N-dimensional linear equation

$$\mathbf{A}\mathbf{x} - \mathbf{f} = 0. \tag{6}$$

This simple problem has the immediate solution

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{f} \tag{7}$$

which may be computed easily via standard methods of numerical linear algebra.

But now consider the case of particle motion in a *position-dependent* force field $f(x)$. (For example, in a 1D gravitational-motion problem we would have $f(x) = -\frac{GM}{x^2}$.) The ODE now takes the form

$$\frac{d^2x}{dt^2} = f(x), \qquad x(t_a) = x(t_b) = 0. \tag{8}$$

Again we can use finite-difference techniques to write a system of equations analogous to (5):

$$\mathbf{A}\mathbf{x} = \mathbf{f} \tag{9}$$

However, the apparent similarity of (9) to (5) is deceptive, because the RHS vector in (9) now depends on the unknown vector $\mathbf{x}$! More specifically, in equation (9) we now have

$$\mathbf{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_N \end{pmatrix} = \textbf{unknown}, \qquad \mathbf{f} = \begin{pmatrix} f(x_1) \\ \vdots \\ f(x_N) \end{pmatrix} = \textbf{also unknown}!.$$

Thus equation (9) defines a *nonlinear* root-finding problem,

$$\mathbf{A}\mathbf{x} - \mathbf{f}(\mathbf{x}) = 0 \tag{10}$$

and no immediate solution like (7) is available; instead we must solve iteratively using nonlinear root-finding techniques.

2 One-dimensional root-finding techniques

2.1 Bisection

The simplest root-finding method is the *bisection* method, which basically just performs a simple binary search. We begin by *bracketing* the root: this means finding two points x_1 and x_2 at which $f(x)$ has different signs, so that we are guaranteed[2] to have a root between x_1 and x_2. Then we bisect the interval $[x_1, x_2]$, computing the midpoint $x_m = \frac{1}{2}(x_1+x_2)$ and evaluating f at this point. We now ask whether the sign of $f(x_m)$ agrees with that of $f(x_1)$ or $f(x_2)$. In the former case, we have now bracketed the root in the interval $[x_m, x_2]$; in the latter case, we have bracketed the root in the interval $[x_1, x_m]$. In either case, we have shrunk the width of the interval within which the root may be hiding by a factor of 2. Now we again bisect this new interval, and so on.

Case Study

As a simple case study, let's investigate the convergence of the bisection method on the function $f(x) = \tanh(x - 5)$. The exact root, to 16-digit precision, is x=`5.000000000000000`. Suppose we initially bracket the root in the interval `[3.0,5.8]` and take the midpoint of the interval to be our guess as to the starting value; thus, for example, our initial guess is x_0 = `4.4`. The following table of numbers illustrates the evolution of the method as it converges to the exact root.

n	Bracket	x_n
1	[3.00000000e+00, 5.80000000e+00]	`4.400000000000000e+00`
2	[4.40000000e+00, 5.80000000e+00]	`5.100000000000000e+00`
3	[4.40000000e+00, 5.10000000e+00]	`4.750000000000000e+00`
4	[4.75000000e+00, 5.10000000e+00]	`4.925000000000000e+00`
5	[4.92500000e+00, 5.10000000e+00]	`5.012499999999999e+00`
6	[4.92500000e+00, 5.01250000e+00]	`4.968750000000000e+00`
7	[4.96875000e+00, 5.01250000e+00]	`4.990625000000000e+00`
8	[4.99062500e+00, 5.01250000e+00]	`5.001562499999999e+00`
9	[4.99062500e+00, 5.00156250e+00]	`4.996093750000000e+00`
10	[4.99609375e+00, 5.00156250e+00]	`4.998828124999999e+00`
11	[4.99882812e+00, 5.00156250e+00]	`5.000195312499999e+00`
12	[4.99882812e+00, 5.00019531e+00]	`4.999511718749999e+00`
13	[4.99951172e+00, 5.00019531e+00]	`4.999853515624999e+00`
14	[4.99985352e+00, 5.00019531e+00]	`5.000024414062499e+00`
15	[4.99985352e+00, 5.00002441e+00]	`4.999938964843748e+00`
16	[4.99993896e+00, 5.00002441e+00]	`5.000081689453124e+00`

[2] Assuming the function is continuous. We will not consider the ill-defined problem of root-finding for discontinuous functions.

The important thing about this table is that the number of correct (red) digits grows approximately *linearly* with n. This is what we call *linear convergence*.[3] Let's now try to understand this phenomenon analytically.

Convergence rate

Suppose the width of the interval within which we initially bracketed the root was $\Delta_0 = x_2 - x_1$. Then, after one iteration of the method, the width of the interval within which the root may be hiding has shrunk to $\Delta_1 = \frac{1}{2}\Delta_0$ (note that this is true *regardless* of which subinterval we chose as our new bracket – they both had the same width). After two iterations, the width of the interval within which the root may be hiding is $\Delta_2 = \frac{1}{2}\Delta_1 = \frac{1}{4}\Delta_0$, and so on. Thus, after N iterations, the width of the interval within which the root may be hiding (which we may alternatively characterize as the absolute error with which we have pinpointed a root) is

$$\epsilon_N^{\text{bisection}} = 2^{-N}\Delta_0 \tag{11}$$

In other words, the bisection method converges *exponentially* rapidly. (More specifically, the bisection method exhibits *linear convergence*; the number of correct digits grows linearly with the number of iterations. If we have 6 good digits after 10 iterations, then we need to do 10 more iterations to get the next 6 digits, for a total of 12 good digits).

Note that this convergence rate is faster than anything we have seen in the course thus far: faster than any Newton-Cotes quadrature rule, faster than any ODE integrator, faster than any finite-difference stencil, all of which exhibit errors that decay *algebraically* (as a power law) with N.

The bisection method is extremely robust; if you can succeed in bracketing the root to begin with, then you are guaranteed to converge to the root. The robustness stems from the fact that, as long as f is continuous and you can succeed in initially bracketing a root, you are *guaranteed* to have a root somewhere in the interval, while the error in your approximation of this root cannot help but shrink inexorably to zero as you repeatedly halve the width of the bucket in which it could be hiding.

On the other hand, the bisection method is not the most rapidly-convergent method. Among other things, the method only uses minimal information about the values of the function at the interval endpoints–namely, only its *sign*, and not its magnitude. This seems somehow wasteful. A method that takes better advantage of the function information at our disposal is the *secant* method, described next.

[3] As emphasized in the lecture notes on convergence terminology, linear convergence is not to be confused with "first-order convergence," which is when the error decreases like $1/n$, and hence the number of correct digits grows like $\log_{10}(n)$.

2.2 Secant

The idea of the secant method is to speed the convergence of the bisection method by using information about the magnitudes of the function values at the interval endpoints in addition to their signs. More specifically, suppose we have evaluated $f(x)$ at two points x_1 and x_2. We plot the points $(x_1, y_1 = f(x_1))$ and $(x_2, y_2 = f(x_2))$ on a Cartesian coordinate system and draw a straight line connecting these two points. Then we take the point x_3 at which this line crosses the x-axis as our updated estimate of the root. In symbols, the rule is

$$x_3 = x_2 - \frac{x_2 - x_1}{f(x_2) - f(x_1)} f(x_2)$$

Then we repeat the process, generating a new point x_4 by looking at the points $(x_2, f(x_2))$ and $(x_3, f(x_3))$, and so on. The general rule is

$$x_{n+1} = x_n - \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} f(x_n) \tag{12}$$

As we might expect, the error in the secant method decays more rapidly than that in the bisection method; the number of correct digits grows roughly like the number of iterations to the power $p = \frac{1+\sqrt{5}}{2} \approx 1.6$.

One drawback of the secant method is that, in contrast to the bisection method, it does not maintain a bracket of the root. This makes the method less robust than the bisection method.

2.3 Newton-Raphson

Take another look at equation (12). Suppose that x_{n-1} is close to x_n, i.e. imagine $x_{n-1} = x_n + h$ for some small number h. Then the quantity multiplying $f(x_n)$ in the second term of (12) is something like the inverse of the finite-difference approximation to the derivative of f at x_n:

$$\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} \approx \frac{1}{f'(x_n)}$$

If we assume that this approximation is trying to tell us something, we are led to consider the following modified version of (12):

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{13}$$

This prescription for obtaining an improved root estimate from a initial root estimate is called *Newton's method* (also known as the *Newton-Raphson* method).

Alternative derivation of Newton-Raphson

Another way to understand the Newton-Raphson iteration (13) is to expand the function $f(x)$ in a Taylor series about the current root estimate x_n:

$$f(x) = f(x_n) + (x - x_n) f'(x_n) + \frac{1}{2}(x - x_n)^2 f''(x_n) + \cdots \tag{14}$$

If we evaluate (14) at the actual root x_0, then the LHS is zero (because $f(x_0) = 0$ since x_0 is a root), whereupon we find

$$0 = f(x_n) + (x_0 - x_n)f'(x_n) + O[(x - x_n)^2]$$

If we neglect the quadratic and higher-order terms in this equation, we can solve immediately for the root x_0:

$$x_0 = x_n - \frac{f(x_n)}{f'(x_n)} \tag{15}$$

This reproduces equation (13).

To summarize: Newton's method approximates $f(x)$ as a linear function and jumps directly to the point at which this linear function is zeroed out. From this, we can expect that the method will work well in the vicinity of a single root (where the function really is approximately linear) but less well in the vicinity of a multiple root and perhaps not well at all when we aren't in the vicinity of a root. We will quantify these predictions below.

Convergence of Newton-Raphson

Suppose we have run the Newton-Raphson algorithm for n iterations, so that our best present estimate of the root is x_n. Let x_0 be the actual root. As above, let's express this root using the Taylor-series expansion of $f(x)$ about the point $x = x_n$:

$$f(x_0) = 0 = f(x_n) + f'(x_n)(x_0 - x_n) + \frac{1}{2}f''(x_n)(x_0 - x_n)^2 + O\Big((x_0 - x_n)^3\Big)$$

Divide both sides by $f'(x_n)$ and rearrange a little:

$$\underbrace{x_0 - x_n + \frac{f(x_n)}{f'(x_n)}}_{x_0 - x_{n+1}} = -\frac{1}{2}\frac{f''(x_n)}{f'(x_n)}(x_0 - x_n)^2 + O\Big((x_0 - x_n)^3\Big)$$

But now the quantity on the LHS is telling us the distance between the root and x_{n+1}, the next iteration of the Newton method. In other words, if we define the error after n iterations as $\epsilon_n = |x_0 - x_n|$, then

$$\epsilon_{n+1} = C\epsilon_n^2$$

(where C is some constant). In other words, the error *squares* on each iteration. To analyze the implications of this fact for convergence, it's easiest to take logarithms on both sides:

$$\begin{aligned} \log \epsilon_{n+1} &\sim 2 \log \epsilon_n \\ &\sim 4 \log \epsilon_{n-1} \end{aligned}$$

and so on, working backwards until we find

$$\log \epsilon_{n+1} \sim 2^{n+1} \log \epsilon_0$$

where ϵ_0 is the error in our initial root estimate. Note that the *logarithm* of the error decays exponentially with n, which means that the error itself decays *doubly exponentially* with n: we have something like

$$\epsilon_n \sim e^{-Ae^{Bn}} \tag{16}$$

for positive constants A and B.

Another way to characterize (16) is to say that the number of correct digits uncovered by Newton's method grows quadratically with the number of iterations; we say Newton's method exhibits *quadratic convergence*.

Case study

Let's apply Newton's method to find a root of the function $\tanh(x-5)$. The exact root, to 16-digit precision, is x=`5.000000000000000`. We will start the method at an initial guess of $x_1 =$ `4.4` and iterate using (13). This produces the following table of numbers, in which correct digits are printed in red:

n	x_n
1	`4.400000000000000`
2	`5.154730677706086`
3	`4.997518482593209`
4	`5.000000010187351`
5	`5.000000000000000`

After 3 iterations, I have 4 good digits; after 4 iterations, 8 good digits; after 5 iterations, 16 good digits. This is *quadratic convergence.*

Double roots

What happens if $f(x)$ has a *double root* at $x = x_0$? A double root means that both $f(x_0) = 0$ and $f'(x_0) = 0$. Since our error analysis above assumed $f'(x_0) \neq 0$, we might expect it to break down if this condition is not satisfied, and indeed in this case Newton's method exhibits only *linear* convergence.

3 Newton's method in higher dimensions

One advantage of Newton's method over simple methods like bisection is that it extends readily to multidimensional root-finding problems. Consider the problem of finding a root $\mathbf{x}_0$ of a vector-valued function:

$$\mathbf{f}(\mathbf{x}) = 0 \tag{17}$$

where x is an N-dimensional vector and $\mathbf{f}$ is an N-dimensional vector of functions. (Although in the introduction we stated that root-finding problems may be defined in which the dimensions of $\mathbf{f}$ and $\mathbf{x}$ are different, Newton's method only applies to the case in which they are the same.)

The linear case

There is one case of the system (17) that you already know how to solve: the case in which the system is *linear*, i.e. $\mathbf{f}(\mathbf{x})$ is just matrix multiplication of $\mathbf{x}$ by a matrix with $\mathbf{x}$-independent coefficients:

$$\mathbf{f}(\mathbf{x}) = \mathbf{A}\mathbf{x} = 0 \tag{18}$$

In this case, we know there is always the one (trivial) root $\mathbf{x} = 0$, and the condition for the existence of a *nontrivial* root is the vanishing of the determinant of $\mathbf{A}$. If $\det \mathbf{A} \neq 0$, then there is no point trying to find a nontrivial root, because none exists. On the other hand, if $\det \mathbf{A} = 0$ then $\mathbf{A}$ has a zero eigenvalue and it's easy to solve for the corresponding eigenvector, which is a nontrivial root of (18).

The nonlinear case

The vanishing-of-determinant condition for the existence of a nontrivial root of (18) is very nice: it tells us exactly when we can expect a nontrivial solution to exist.

For more general nonlinear systems there is no such nice condition for the existence of a root[4], and thus it is convenient indeed that Newton's method for root-finding has an immediate generalization to the multi-dimensional case. All we have to do is write out the multidimensional generalization of (14) for the Taylor expansion of a multivariable function around the point $\mathbf{x}$:

$$\mathbf{f}(\mathbf{x} + \boldsymbol{\Delta}) = \mathbf{f}(\mathbf{x}) + \mathbf{J}\boldsymbol{\Delta} + O(\boldsymbol{\Delta}^2) \tag{19}$$

[4] At least, this is the message they give you in usual numerical analysis classes, but it is not quite the whole truth. For *polynomial* systems it turns out there is a beautiful generalization of the determinant known as the *resultant* that may be used, like the determinant, to yield a criterion for the existence of a nontrivial root. I hope we will get to discuss resultants later in the course, but for now you can read about it in the wonderful books *Ideals, Varieties, and Algorithms* and *Using Algebraic Geometry*, both by Cox, Little, and O'Shea.

where the *Jacobian* matrix $\mathbf{J}$ is the matrix of first partial derivatives of $\mathbf{f}$:

$$\mathbf{J}(\mathbf{x}) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_N} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_N}{\partial x_1} & \frac{\partial f_N}{\partial x_2} & \cdots & \frac{\partial f_N}{\partial x_N} \end{pmatrix}$$

where all partial derivatives are to be evaluated at $\mathbf{x}$.

Now suppose we have an estimate $\mathbf{x}$ for the root of nonlinear system $\mathbf{f}(\mathbf{x})$. Let's compute the increment $\boldsymbol{\Delta}$ that we need to add to $\mathbf{x}$ to jump to the exact root of the system. Setting (19) equal to zero and ignoring higher-order terms, we find

$$\begin{aligned} 0 &= \mathbf{f}(\mathbf{x} + \boldsymbol{\Delta}) \\ &\approx \mathbf{f}(\mathbf{x}) + \mathbf{J}\boldsymbol{\Delta} \end{aligned}$$

or

$$\boldsymbol{\Delta} = -\mathbf{J}^{-1}\mathbf{f}(\mathbf{x})$$

In other words, if $\mathbf{x}_n$ is our best guess as to the location of the root after n iterations of Newton's method, then our best guess after $n+1$ iterations will be

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{J}^{-1}\mathbf{f}(\mathbf{x}) \tag{20}$$

This is an immediate generalization of (13); indeed, in the 1D case $\mathbf{J}$ reduces simply to f' and we recover (13).

However, computationally, (20) is more expensive than (13): it requires us to solve a linear system of equations on each iteration.

Example

As a case study in the use of Newton's method in multiple dimensions, consider the following two-dimensional nonlinear system:

$$\mathbf{f}(\mathbf{x}) = \begin{pmatrix} x_1^2 - \cos(x_1 x_2) \\ e^{x_1 x_2} + x_2 \end{pmatrix}$$

The Jacobian matrix is

$$\mathbf{J}(\mathbf{x}) = \begin{pmatrix} 2x_1 + x_2 \sin(x_1 x_2) & x_1 \sin(x_1 x_2) \\ x_2 e^{x_1 x_2} & x_1 e^{x_1 x_2} + 1 \end{pmatrix}$$

This example problem has a solution at

$$\mathbf{x}_0 = \begin{pmatrix} \texttt{0.926175} \\ \texttt{-0.582852} \end{pmatrix}$$

Here's a JULIA routine called `NewtonSolve` that computes a root of this system. Note that the body of the `NewtonStep` routine is only three lines long.

```
function f(x)
   x1=x[1];
   x2=x[2];
   [x1^2 - cos(x1*x2); exp(x1*x2) + x2];
end

function J(x)
   x1=x[1];
   x2=x[2];
   J11=2*x1+x2*sin(x1*x2)
   J12=x1*sin(x1*x2)
   J21=x2*exp(x1*x2)
   J22=x1*exp(x1*x2)+1;
   [ J11 J12; J21 J22]
end

function NewtonStep(x)
  fVector = f(x)
  jMatrix = J(x)
  x - jMatrix \ fVector;
end

function NewtonSolve()
  x=[1; 1]; # random initial guess
  residual=norm(f(x))
  while residual > 1.0e-12
     x=NewtonStep(x)
     residual=norm(f(x))
  end
  x
end
```

4 Newton's method is a *local* method

Newton's method exhibits outstanding *local* convergence, but terrible *global* convergence. One way to think of this is to say that Newton's method is more of a root-*polisher* than a root-*finder*: If you are already near a root, you can use Newton's method to zero in on that root to high precision, but if you aren't near a root and don't know where to start looking then Newton's method may be useless.

To give just one example, consider the function $\tanh(x-5)$ that we considered above. Suppose we didn't know that this function had a root at $x=5$, and suppose we started looking for a root near $x=0$. Setting $x_1=0$ and executing one iteration of Newton's method yields

$$\begin{aligned} x_2 &= x_1 - \frac{f(x_1)}{f'(x_1)} \\ &= 0 - \frac{\texttt{tanh(-5)}}{\texttt{sech(-5))\^{}2}} \\ &= \texttt{5506.61643} \end{aligned}$$

Newton's method has sent us completely out of the ballpark! What went wrong??

What went wrong here is that the function $\tanh(x-5)$ has very gentle slope at $x=0$ – in fact, the function is almost flat there (more specifically, its slope is $\texttt{sech}^2(x-5) \approx 2\cdot 10^{-4}$) – and so, when we approximate the function as a line with that slope and jump to the point at which that line crosses the x axis, we wind up something like 5,000 units away. This is what we get for attempting to use Newton's method with a starting point that is not close to a root.

Newton's method applied to polynomials

We get particularly spectactular examples of the sketchy global convergence properties of Newton's method when we apply the method to the computation of roots of polynomials.

One obvious example of what can go wrong is the use of Newton's method to compute the roots of

$$P(x) = x^2 + 1 = 0. \tag{21}$$

The Newton iteration (13) applied to (21) yields the sequence of points

$$x_{n+1} = x_n - \frac{x_n^2+1}{2x_n}. \tag{22}$$

If we start with any real-valued initial guess x_1, then the sequence of points generated by (22) is guaranteed to remain *real-valued* for all n, and thus we can never hope to converge to the correct roots $\pm i$.

Newton fractals

We get a graphical depiction of phenomena like this by plotting, in the complex plane, the set of points $\{z_0\}$ at which Newton's method, when started at z_0 for a function $f(z)$, converges to a specific root. [More specifically: For each point z in some region of the complex plane, we run Newton's method on the function f starting at z. If the function converges to the mth root in N iterations, we plot a color whose RGB value is determined by the tuple (m, N).] You can generate plots like this using the JULIA function `PlotNewtonConvergence`, which takes as its single argument a vector of the polynomial coefficients sorted in decreasing order. Here's an example for the function $f(z) = z^3 - 1$.

```
julia> PlotNewtonConvergence([1 0 -1])
```

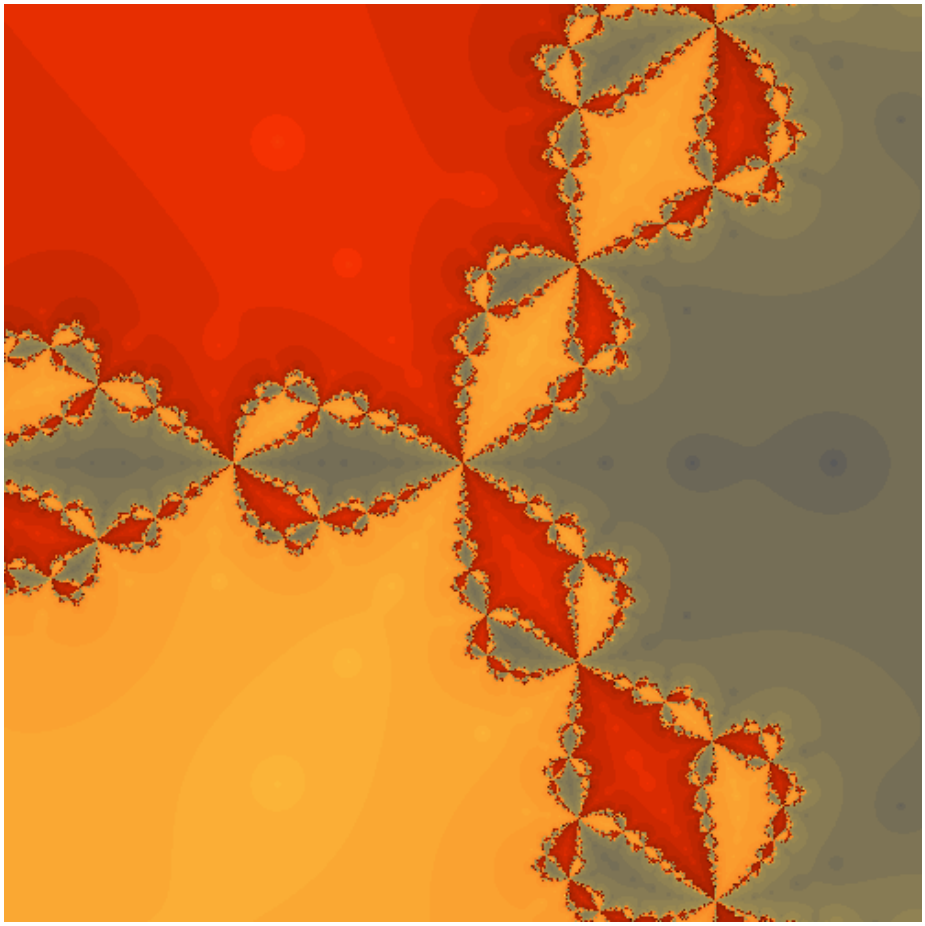

Figure 1: Newton fractal for the function $f(x) = z^3 - 1$.

The three roots of $f(z)$ are $1, e^{2\pi i/3}, e^{4\pi i/3}$. The variously colored regions in the plot indicate points in the complex plane for which Newton's method converges to the various roots; for example, red points converge to $e^{2\pi i/3}$, and yellow points converge to $e^{4\pi i/3}$. What you see is that for starting points in the immediate vicinity of each root, convergence to that root is guaranteed, but elsewhere in the complex plane all bets are off; there are large red and yellow regions that lie nowhere near the corresponding roots, and the fantastically intricate boundaries of these regions indicate the exquisite sensitivity of Newton's method to the exact location of the starting point.

This type of plot is known as a *Newton fractal*, for obvious reasons. Thus Newton's method applied to the global convergence of polynomial root-finding yields beautiful pictures, but not a very happy time for actual numerical root-finders.

5 Computing roots of polynomials

In the previous section we observed that Newton's method exhibits spectacularly sketchy global convergence when we use it to compute roots of polynomials. So what *should* you do to compute the roots of a polynomial $P(x)$? For an arbitrary Nth-degree polynomial with real or complex coefficients, the fundamental theorem of algebra guarantees that N complex roots exist, but on the other hand Galois theory guarantees for $N > 5$ that there is no nice formula expressing these roots in terms of the coefficients, so finding them is a task for numerical analysis. Although specialized techniques for this problem do exist (one such is the "Jenkins-Traub" method), a method which works perfectly well in practice and requires only standard tools is to find a matrix whose characteristic polynomial is $P(x)$ and compute the eigenvalues of this polynomial using standard methods of numerical linear algebra.

The companion matrix

Such a matrix is called the *companion matrix,* and for a monic[5] polynomial $P(x)$ of the form

$$P(x) = x^n + C_{n-1}x^{n-1} + C_{n-2}x^{n-2} + \cdots + C_1 x + C_0$$

the companion matrix takes the form.

$$\mathbf{C}_P = \begin{pmatrix} 0 & 0 & 0 & \cdots & -C_0 \\ 1 & 0 & 0 & \cdots & -C_1 \\ 0 & 1 & 0 & \cdots & -C_2 \\ 0 & 0 & 1 & \cdots & -C_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & -C_{n-1} \end{pmatrix}$$

Given the coefficients of P_N, it is a simple task to form the matrix C_P and compute its eigenvalues numerically. You can find an example of this calculation in the `PlotNewtonConvergence.jl` code mentioned in the previous section.

[5] A *monic* polynomial is one for which the coefficient of the highest-degree monomial is 1. If your polynomial is not monic (suppose the coefficient of its highest-order monomial is $A \neq 1$), just consider the polynomial obtained by dividing all coefficients by A. This polynomial is monic and has the same roots as your original polynomial.

6 A glimpse at numerical optimization

A problem which bears a superficial similarity to that of root-finding, but which in many ways is quite distinct, is the problem of *optimization*, namely, given some complicated nonlinear function $f(x)$, we ask to

$$\text{find } x \text{ such that } f(x) \text{ has an extremum at } x$$

where the extremum may be a global or local maximum or minimum. This problem also has an obvious generalization to scalar-valued functions of vector-valued variables, i.e.

$$\text{find } \mathbf{x} \text{ such that } f(\mathbf{x}) \text{ has an extremum at } \mathbf{x}.$$

Numerical optimization is a huge field into which we can't delve very deeply in 18.330; what follows is only the most cursory of overviews, although the point at the end regarding the accuracy of root-finding vs. optimization is an important one.

6.1 Derivative-free optimization of 1D functions

Golden-Section Search The *golden-section search* algorithm, perhaps the simplest derivative-free optimization method for 1D functions, is close in spirit to the bisection method of root finding. Recall that the bisection method for finding a root of a function $f(x)$ begins by finding an initial interval $[a_0, b_0]$ within which the root is known to lie; the method then proceeds to generate a sequence of pairs, i.e.

$$\begin{array}{c} [a_0, b_0] \\ \Downarrow \\ [a_1, b_1] \\ \Downarrow \\ \vdots \\ [a_n, b_n] \\ \Downarrow \\ \vdots \end{array}$$

with the property that the root is always known to be contained within the interval in question, i.e. with the property

$$\text{sign } f(a_n) \neq \text{sign } f(b_n)$$

preserved for all n.

Golden-section search does something similar, but instead of generating a sequence of *pairs* $[a_n, b_n]$ it produces a sequence of *triples* $[a_n, b_n, c_n]$, i.e.

$$\begin{array}{c} [a_0, b_0, c_0] \\ \Downarrow \\ [a_1, b_1, c_1] \\ \Downarrow \\ \vdots \\ [a_n, b_n, c_n] \\ \Downarrow \\ \vdots \end{array}$$

with the properties that $a_n < b_n < c_n$ and each triple be guaranteed to *bracket* the minimum, in the sense that $f(b_n)$ is always lower than either of $f(a_n)$ or $f(c_n)$, i.e. the properties

$$f(a_n) > f(b_n) \qquad \text{and} \qquad f(b_n) < f(c_n) \tag{23}$$

is preserved for all n.

To start the golden-section search algorithm, we need to identify an initial triple $[a_0, b_0, c_0]$ satisfying property (23). Then, we iterate the following algorithm that inputs a bracketing triple $[a_n, b_n, c_n]$ and outputs a new, smaller, bracketing triple $[a_{n+1}, b_{n+1}, c_{n+1}]$:

1. Choose[6] a new point x that lies a fraction γ of the way into the larger of the intervals $[a_n, b_n]$ and $[b_n, c_n]$.
2. Evaluate f at x.
3. We now have four points and four function values, and we want to discard either the leftmost point (a_n) or the rightmost point (c_n) and take the remaining three points—in ascending order—to be our new triple. More specifically, this amounts to the following four-way decision tree based on the relative positions of $\{x, b_n\}$ and $\{f(x), f(b_n)\}$:

$$[a_{n+1}, b_{n+1}, c_{n+1}] = \begin{cases} [a_n, x, b_n], & \text{if } x < b_n \text{ and } f(x) < f(b_n) \\ [b_n, x, c_n], & \text{if } x > b_n \text{ and } f(x) < f(b_n) \\ [x, b_n, c_n], & \text{if } x < b_n \text{ and } f(x) > f(b_n) \\ [a_n, b_n, x], & \text{if } x > b_n \text{ and } f(x) > f(b_n) \end{cases}$$

Do you see how this works? The decision tree in Step 3 guarantees the preservation of property (23), while meanwhile the shrinking of the intervals in Step

[6] A more specific description of this step is that we set

$$x = \begin{cases} b_n + \gamma(c_n - b_n), & \text{if } (c_n - b_n) > (b_n - a_n) \\ b_n + \gamma(a_n - b_n), & \text{if } (c_n - b_n) < (b_n - a_n). \end{cases}.$$

1 guarantees that our bracket converges inexorably to a smaller and smaller interval within which the minimum could be hiding.

How do we choose the optimal shrinking fraction γ? One elegant approach is to choose γ to ensure that the ratio of the lengths of the two subintervals $[a_n, b_n]$ and $[b_n, c_n]$ remains constant even as the overall width of the bracketing interval shrinks toward zero. With a little effort you can show that this property is ensured by taking γ to be the *golden ratio*,

$$\gamma = \frac{3-\sqrt{5}}{2} = \texttt{0.381966011250105}$$

and a γ-fraction of an interval is known as the *golden section* of that interval, which explains the name of the algorithm.

Here's a JULIA implementation of the golden-section search algorithm:

```
function GSSearch(f,a,c)

  Gamma = ( 3.0-sqrt(5.0) ) / 2.0;   # golden ratio
  MachineEps = eps();                # machine precision

  ######################################################
  # initial bracket [a,b,c]
  ######################################################
  fa=f(a);
  fc=f(c);
  b = a + Gamma*(c-a);
  fb=f(b);
  xbest=b;
  for n=1:1000

    ######################################################
    # choose x to be a fraction Gamma of the way into
    # the larger of the two intervals; also,
    # possibly relabel b<->x to ensure that the points
    # are labeled in such a way that a<b<x<c
    ######################################################
    if (c-b) > (b-a)
       x = b + Gamma*(c-b);
       fx=f(x);
    else
       Newb = b + Gamma*(a-b);
       x = b
       b = Newb
       fx=fb;
       fb=f(b);
    end
```

```
    ####################################################
    # choose new bracketing triple (a,b,c)
    ####################################################
    if fx < fb
      a=b;
      b=x;
      fa=fb;
      fb=fx;
      xbest=x;
      fbest=fb;
    else
      c=x;
      fc=fx;
      xbest=b;
      fbest=fx;
    end

    @printf("%3i   %+.16e   %.2e\n",n,xbest,(c-a));

    ####################################################
    # stop when the width of the bracketing triple
    # is less than roughly machine precision
    # (really we might as well stop much sooner than this,
    #  for reasons explained in the lecture notes)
    ####################################################
    if abs(c-a) < 3*MachineEps
      return xbest;
    end

  end
  return xbest;

end
```

6.2 Roots can be found more accurately than extrema

An important distinction between numerical root-finding and derivative-free numerical optimization is that the former can generally be done much more accurately. Indeed, if a function $f(x)$ has a root at a point x_0, then in many cases we will be able to approximate x_0 to roughly machine precision—that is, to 15-decimal-digit accuracy on a typical modern computer. In contrast, if $f(x)$ has an extremum at x_0, then in general we will only be able to pin down the value of x_0 to something like the *square root* of machine precision—that is, to just 8-digit accuracy! This is a huge loss of precision compared to the root-finding case.

To understand the reason for this, suppose f has a minimum at x_0, and let the value of this minimum be $f_0 \equiv f(x_0)$. Then, in the vicinity of x_0, f has a Taylor-series expansion of the form

$$f(x) = f_0 + \frac{1}{2}(x - x_0)^2 f''(x_0) + O\Big((x - x_0)^3\Big) \tag{24}$$

where the important point is that the linear term is absent because the derivative of f vanishes at x_0.

Now suppose we try to evaluate f at floating-point numbers lying very close to, but not exactly equal to, the nearest floating-point representation of x_0. (Actually, for the purposes of this discussion, let's assume that x_0 is exactly floating-point representable, and moreover that the magnitudes of x_0, f_0, and $f''(x_0)$ are all on the order of 1. The discussion could easily be extended to relax these assumptions at the expense of some cluttering of the ideas.) In 64-bit floating-point arithmetic, where we have approximately 15-decimal-digit registers, the floating-point numbers that lie closest to x_0 without being equal to x_0 are something like[7] $x_{\text{nearest}} \approx x_0 \pm 10^{-15}$. We then find

$$f(x_{\text{nearest}}) = \underbrace{f_0}_{\sim 1.0} + \frac{1}{2}\underbrace{(x - x_0)^2}_{\sim 1.0\text{e-}30} f''(x_0) + O\Big(\underbrace{(x - x_0)^3}_{\sim 1.0\text{e-}45}\Big).$$

Since x_{nearest} deviates from x_0 by something like 10^{-15}, we find that $f(x_{\text{nearest}})$ deviates from $f(x_0)$ by something like 10^{-30}, i.e. the digits begin to disagree in the 30th decimal place. *But our floating-point registers can only store 15 decimal digits,* so the difference between $f(x_0)$ and $f(x_{\text{nearest}})$ is completely lost; the two function values are *utterly indistinguishable* to our computer.

Moreover, as we consider points x lying further and further away from x_0, we find that $f(x)$ *remains* floating-point indistinguishable from $f(x_0)$ over a wide interval near x_0. Indeed, the condition that $f(x)$ be floating-point distinct from $f(x_0)$ requires that $(x - x_0)^2$ fit into a floating-point register that is also storing $f_0 \approx 1$. This means that we need[8]

$$(x - x_0)^2 \gtrsim \epsilon_{\text{machine}} \tag{25}$$

or

$$(x - x_0) \gtrsim \sqrt{\epsilon_{\text{machine}}} \tag{26}$$

This explains why, in general, we can only pin down minima to within the square root of machine precision, i.e. to roughly 8 decimal digits on a modern computer.

[7] This is where the assumption that $|x_0| \sim 1$ comes in; the more general statement would be that the nearest floating-point numbers not equal to x_0 would be something like $x_0 \pm 10^{-15}|x_0|$.

[8] This is where the assumptions that $|f_0| \sim 1$ and $|f''(x_0)| \sim 1$ come in; the more general statement would be that we need $(x - x_0)^2 |f''(x_0)| \gtrsim \epsilon_{\text{machine}} \cdot |f_0|$.

On other hand, suppose the function $g(x)$ has a root at x_0. In the vicinity of x_0 we have the Taylor expansion

$$g(x) = (x - x_0)g'(x_0) + \frac{1}{2}(x - x_0)^2 g''(x_0) + \cdots \tag{27}$$

which differs from (24) by the presence of a linear term. Now there is generally no problem distinguishing $g(x_0)$ from $g(x_{\text{nearest}})$ or g at other floating-point numbers lying within a few machine epsilons of x_0, and hence in general we will be able to pin down the value of x_0 to close to machine precision. (Note that this assumes that g has only a *single* root at x_0; if g has a *double* root there, i.e. $g'(x_0) = 0$, then this analysis falls apart. Compare this to the observation we made earlier that the convergence of Newton's method is worse for double roots than for single roots.)

Figures 6.2 illustrates these points. The upper panel in this figure plots, for the function $f(x) = f_0 + (x - x_0)^2$ [corresponding to equation (24) with $x_0 = f_0 = \frac{1}{2}f''(x_0) = 1$], the deviation of $f(x)$ from its value at $f(x_0)$ versus the deviation of x from x_0 *as computed in standard 64-bit floating-point arithmetic.* Notice that $f(x)$ remains *indistinguishable* from $f(x_0)$ until x deviates from x_0 by at least 10^{-8}; thus a computer minimization algorithm cannot hope to pin down the location of x_0 to better than this accuracy.

In contrast, the lower panel of Figure 6.2 plots, for the function $g(x) = (x - x_0)$ [corresponding to equation (27) with $x_0 = g'(x_0) = 1$], the deviation of $g(x)$ from $g(x_0)$ versus the deviation of x from x_0, again as computed in standard 64-bit floating-point arithmetic. In this case our computer is easily able to distinguish points x that deviate from x_0 by as little as $2 \cdot 10^{-16}$. This is why numerical root-finding can, in general, be performed with many orders of magnitude better precision than minimization.

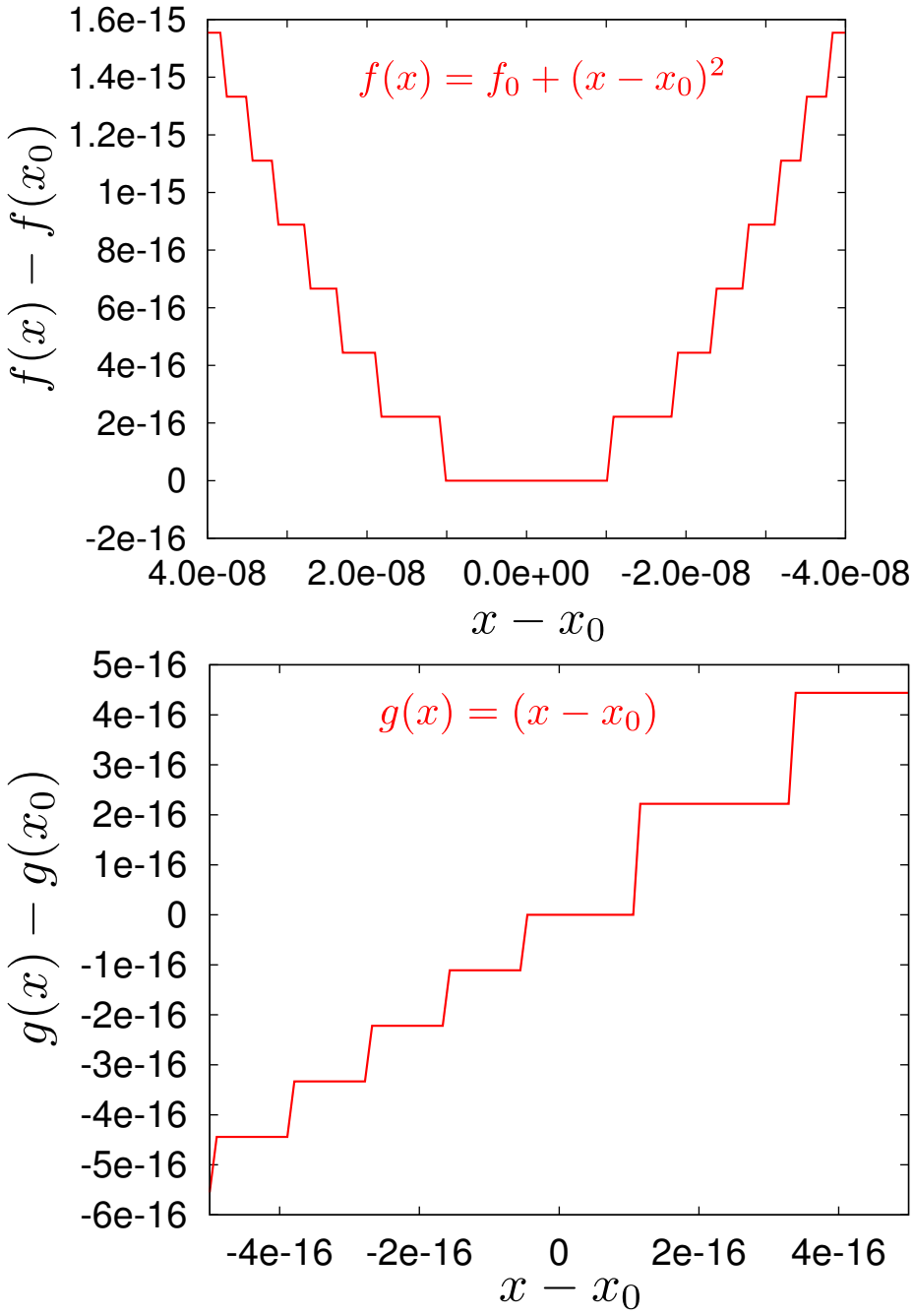

Figure 2: In standard 64-bit floating-point arithmetic, function *extrema* can generally be pinned down only to roughly 8-digit accuracy (upper), while *roots* can typically be identified with close to 15-digit accuracy (lower).