Paul Klein
January 31, 2022

Further notes on root finding in one dimension

In Kopecky's notes on root finding, she discusses bisection, Newton's method, the secant method and Brent's method. (She also discusses Broyden's method, but that is for multidimensional problems.) Here we will discuss some further methods that are faster than bisection and more robust than the secant method.

Incidentally, what is wrong with Newton's method? The problem is that it is in fact slower than the secant method, if we take into account the number of function evaluations per iteration. To deal with this, Traub (1964) defines an efficiency index as follows:

$$\mathcal{E} := \frac{k}{d}$$

where k is the order of convergence as defined as in Kopecky's notes on root finding and d is the number of function evaluations per iteration. Given that Newton's method requires (at least) two function evaluations per iteration, its efficiency index is 1, or no better than bisection.

Traub (1964) also defines an alternative efficiency index as follows.

$$\mathcal{E}^* := k^{1/d}.$$

According to this criterion, Newton's method has an (alternative) efficiency index of $\sqrt{2}$, which is better than bisection but still worse than the secant method, whose efficiency index is about 1.62 (see below). It is this alternative efficiency index that has caught on in the literature. That is no coincidence. The point is the following. Let's compare Newton's method ($k = 2$, $d = 2$) with a hypothetical method that raises the error to the power $k = \sqrt{2}$ in each iteration but only requires one function evaluation per iteration ($d = 1$). They are equally efficient in the sense that two function evaluations are required to square the error. And indeed they have the same alternative efficiency index $\mathcal{E}^* = \sqrt{2}$.

1 The secant method and its basic problem

The secant method is described as follows. It begins with an interval $[a_0, b_0]$. For any interval $[a_k, b_k]$, a third point c_k is computed as follows.

$$c_k = \frac{a_k f(b_k) - b_k f(a_k)}{f(b_k) - f(a_k)}.$$

Incidentally, this formula is better from a computational point of view than the following more obvious alternative:

$$c_k = b_k - f(b_k) \cdot \frac{b_k - a_k}{f(b_k) - f(a_k)}.$$

The reason why the former is superior to the latter is that the difference $b_k - a_k$ has the property that the difference is much smaller in magnitude than the constituent terms. It can be written, in other words, as $x + \varepsilon - x$, where ε is small relative to x. We know from basic principles of finite-precision arithmetic that this turns into garbage once ε is smaller than x by enough orders of magnitude. Meanwhile, $f(b_k) - f(a_k)$ is not as bad as it looks, because, when things are going well, $f(a_k)$ and $f(b_k)$ have opposite sign and so we are in fact dealing with a sum, not a difference.

In any event, the secant method then proceeds iteratively by setting $a_{k+1} = b_k$ and $b_{k+1} = c_k$. If it converges, then the limiting convergence order is superlinear with $k \approx 1.62$, the golden ratio. Notice that the golden ratio is strictly greater than $\sqrt{2}$, rendering the secant method more efficient than Newton's method according to Traub's alternative efficiency index. To show the result, we begin by invoking Newton's interpolation formula[1]

$$f(x) = f(c_k) + \frac{f(c_k) - f(c_{k-1})}{c_k - c_{k-1}} \cdot (x - c_k) + \frac{1}{2} f''(\xi) \cdot (x - c_{k-1}) \cdot (x - c_k) \qquad (1)$$

for some number ξ in the interval spanned by x, c_{k-1} and c_k. Now call the solution to our equation x^* so that $f(x^*) = 0$ and put $x = x^*$ in Equation (1). We get

$$f(c_k) + \frac{f(c_k) - f(c_{k-1})}{c_k - c_{k-1}} \cdot (x^* - c_k) + \frac{1}{2} f''(\xi) \cdot (x^* - c_{k-1}) \cdot (x^* - c_k) = 0.$$

[1] See, for instance, Dahlquist and Björck (1974), Section 7.3.3.

By definition of the secant method, we have

$$f(c_k) + (c_{k+1} - c_k) \cdot \frac{f(c_k) - f(c_{k-1})}{c_k - c_{k-1}} = 0$$

so that

$$\frac{f(c_k) - f(c_{k-1})}{c_k - c_{k-1}} \cdot (x^* - c_{k+1}) + \frac{1}{2} f''(\xi) \cdot (x^* - c_{k-1}) \cdot (x^* - c_k) = 0.$$

According to the mean-value theorem, there is an $\xi' \in [c_{k-1}, c_k]$ such that

$$f'(\xi') = \frac{f(c_k) - f(c_{k-1})}{c_k - c_{k-1}}.$$

Defining $e_k := x^* - c_k$, we conclude that

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

Now conjecture that $\frac{f''(\xi)}{2 \cdot f'(\xi')}$ converges. Then when k is large, we have, for some positive number C,

$$|e_{k+1}| \approx C|e_k||e_{k-1}|. \tag{3}$$

Suppose $|e_{k+1}| \approx K|e_k|^p$ for all sufficiently large k; if true, then the convergence order would be p. Substituting this, and $|e_k| \approx K|e_{k-1}|^p$ or, rather, $|e_{k-1}| \approx K^{-1/p}|e_k|^{1/p}$, into Equation (3), we get

$$K|e_k|^p \approx CK^{-1/p}|e_k||e_k|^{1/p}$$

and this can only hold if

$$p = 1 + 1/p.$$

This is of course the equation of the golden ratio. (Fortunately, we can ignore the smaller solution.)

Anyhow, the fundamental problem with the secant method is that it can easily escape from the initially given interval. If we know that the solution is bracketed by our initial interval, then this is very bad. It is particularly bad if the function is not even well-defined everywhere outside the given interval. For instance, consider the function $f(x) := 1 - 1/x^5$. We know that a root is bracketed by $[0.5, 1.5]$. Starting from this interval, the subsequent interval is $[1.5, 1.47]$. So far, so good. (The fact

that the interval is oriented backwards is strange but not obviously problematic.) But the next interval is $[1.47, -0.37]$. This is a wider interval than we started with, and it contains the highly problematic point $x = 0$. Further iterations on the secant method diverge.

The secant method can be salvaged by alternating between bisection and the secant method, but a more obvious approach is the ancient method of false position, or *regula falsi*.

2 The method of regula falsi

The false position method computes c_k just as in the secant method, but it then proceeds in either of two ways, depending on circumstances. If the convergence criterion has not been met, but $f(c_k)$ and $f(a_k)$ are of opposite sign (equivalently, if $f(c_k)$ and $f(b_k)$ have the same sign), the algorithm concludes that the solution lies between a_k and c_k. It therefore updates according to $a_{k+1} = a_k$ and $b_k = c_k$. Alternatively, if $f(c_k)$ and $f(a_k)$ have the *same* sign, the conclusion is that the solution lies between c_k and b_k and hence the updating proceeds according to $a_{k+1} = c_k$ and $b_{k+1} = b_k$.

Notice that this approach, though it looks a lot like bisection, does *not* guarantee that the interval $[a_k, b_k]$ converges to a width of zero even if f is strictly monotonic on the interval $[a_0, b_0]$, and $f(a_0)$ and $f(b_0)$ are of opposite sign. On the contrary, under these assumptions, if f is increasing and convex, then b_k never moves and if f is increasing and concave, then a_k never moves. In either case, $f(c_k)$ never changes sign. This undesirable property slows down the algorithm spectacularly, reducing the order of convergence to linear, so that it is no better than bisection. This is of course pretty disastrous given all the effort we invested in learning an algorithm that was supposed to be faster than bisection.

3 The Illinois algorithm

Fortunately, the regula falsi method can be salvaged. The so-called *Illinois* algorithm is a very minor modification of the false position method, and it solves the problem described above. The modification involves halving the retained function value whenever the new function value has the same sign as the function value at the current upper bound. The purpose of this is to avoid retaining an endpoint indefinitely. As demonstrated in Dowell and Jarratt (1971), the Illinois algorithm exhibits superlinear convergence with (alternative) efficiency index $\mathcal{E}^* = 3^{1/3} \approx 1.44$, only slightly worse than the secant method. Moreover, it is as robust as bisection or regula falsi.

Here is a more explicit description. Denote the current left endpoint by x_0, the current right endpoint by x_1, the current function value at the left endpoint by y_0 and the current function value at the right endpoint by y_1. Now define

$$x := \frac{x_0 y_1 - x_1 y_0}{y_1 - y_0}$$

as in the secant method and $y := f(x)$. Then if $y \cdot y_1 < 0$, replace (x_0, y_0) with (x_1, y_1). Otherwise, replace y_0 with $0.5 \cdot y_0$ and don't update x_0. Either way, replace (x_1, y_1) with (x, y).

Incidentally, there is of course no guarantee that $x_0 < x_1$ and so the phrase "left endpoint" is misleading. (Obviously the algorithm is not affected by the use of this misleading language.) If we are pedantic, we can redescribe the algorithm in the following way. Let x_{n-1} and x_n be given. Define $y_{n-1} := f(x_{n-1})$ and $y_n := f(x_n)$. Let

$$x := \frac{x_0 y_1 - x_1 y_0}{y_1 - y_0}$$

and $y := f(x)$. If $y \cdot y_n < 0$, then $x_{n+1} = x$. Otherwise,

$$x_{n+1} = \frac{x_{n-2} y_n - \frac{1}{2} x_n y_{n-2}}{y_n - \frac{1}{2} y_{n-2}}.$$

Either way, $y_{n+1} = y$.

4 The Pegasus algorithm

Rather than multiply the retained function value by a half, as in the Illinois algorithm, the so-called *Pegasus* algorithm multiplies it by $\frac{y_1}{y_1 + y}$. Dowell and Jarratt (1972) demonstrate that this approach implies an asymptotic (alternative) efficiency index of about $\mathcal{E}^* \approx 1.64$, so even better than the secant method.

5 The Anderson-Björck algorithm

According to Galdino (2011), the state of the art algorithm is the one in Anderson and Björck (1973). Instead of multiplying y_0 by $\frac{y_1}{y_1 + y}$, it multiplies it by either $\frac{y_1 - y}{y_1}$ if that is strictly positive, or by 0.5 otherwise.

This algorithm has an asymptotic (alternative) efficiency index between $\mathcal{E}^* = 8^{1/4} \approx 1.68$ and $\mathcal{E}^* = 5^{1/3} \approx 1.71$. In the worst case, it is still the best of the algorithms presented here.

6 Horse race

Consider again the function $f(x) := 1 - 1/x^5$, suppose our initial interval is $[0.5, 1.5]$ and suppose we require a precision of 10^{-12} both for the function value and for the solution. Then bisection requires $\mathsf{ceil}(\log_2(10^{12})) = 40$ iterations, the Illinois algorithm uses 14, the Pegasus algorithm 12 and the Anderson-Björck algorithm takes 10 iterations. In terms of execution time, the Illinois algorithm is about 3 times faster than bisection; the Pegasus method about 3.5 times faster. Finally, the Anderson-Björck algorithm is about 4 times faster than bisection.

Another example is $f(x) := 1 - 1/x$ with the same initial interval and tolerances as before. Then, as usual, the bisection algorithm takes 40 iterations. The Illinois algorithm takes 9 iterations, the Pegasus method takes 8, and, finally, the Anderson-Björck algorithm takes just 3 iterations. In terms of execution time, the Anderson-Björck method is more than 8.5 times faster than bisection, while the Pe-

gasus method and the Illinois methods are both about 4 times faster than bisection. All of these execution time results are very sensitive to the software that you use. However, the superiority of the Anderson and Björck (1973) method is robust.

References

Anderson, N. and Å. Björck (1973, Sep). A new high order method of regula falsi type for computing a root of an equation. *BIT Numerical Mathematics 13*(3), 253–264.

Dahlquist, G. and Å. Björck (1974). *Numerical Methods*. Dover Publications, Inc., Mineola, New York.

Dowell, M. and P. Jarratt (1971, Jun). A modified regula falsi method for computing the root of an equation. *BIT Numerical Mathematics 11*(2), 168–174.

Dowell, M. and P. Jarratt (1972, Dec). The "Pegasus" method for computing the root of an equation. *BIT Numerical Mathematics 12*(4), 503–508.

Galdino, S. (2011). A family of regula falsi root-finding methods. http://sergiogaldino.pbworks.com/w/file/fetch/66011429/0130-1943543.

Traub, J. F. (1964). *Iterative methods for the solution of equations*. Prentice Hall, Inc., Englewood Cliffs, N. J.

Appendix A: suggested Matlab code for the Illinois algorithm

```
function x = illinois(f,x0,x1,xtol,ftol,maxiter)

% f is a string containing the name of the function
% x0 is a lower bound of the solution
% x1 is an upper bound of the solution
% ftol is the function tolerance
% xtol is the solution tolerance
% maxiter is the maximum number of iterations allowed


xdev = 1;
iter = 0;

y0 = feval(f,x0);
y1 = feval(f,x1);

x = x0;
y = y0;

while (xdev>xtol || abs(y)>ftol) && iter<maxiter
  newx = (x0*y1 - x1*y0)/(y1-y0);
  xdev = abs(x-newx);
  x = newx;
  y = feval(f,x);
  if y*y1<0
   x0 = x1;
   y0 = y1;
  else
   y0 = 0.5*y0;
  end
  x1 = x;
  y1 = y;
  iter = iter + 1;
end
```

Appendix B: suggested Matlab code for the Pegasus algorithm

```
function x = pegasus(f,x0,x1,xtol,ftol,maxiter)

% f is a string containing the name of the function
% x0 is a lower bound of the solution
% x1 is an upper bound of the solution
% ftol is the function tolerance
% xtol is the solution tolerance
% maxiter is the maximum number of iterations allowed


xdev = 1;
iter = 0;

y0 = feval(f,x0);
y1 = feval(f,x1);

x = x0;
y = y0;

while (xdev>xtol || abs(y)>ftol) && iter<maxiter
  newx = (x0*y1 - x1*y0)/(y1-y0);
  xdev = abs(x-newx);
  x = newx;
  y = feval(f,x);
  if y*y1<0
    x0 = x1;
    y0 = y1;
  else
    y0 = y0*y1/(y1+y);
  end
  x1 = x;
  y1 = y;
  iter = iter + 1;
end
```

Appendix C: suggested Matlab code for Anderson-Björck

```
function x = andersonbjorck(f,x0,x1,xtol,ftol,maxiter)

% f is a string containing the name of the function
% x0 is a lower bound of the solution
% x1 is an upper bound of the solution
% ftol is the function tolerance
% xtol is the solution tolerance
% maxiter is the maximum number of iterations allowed


xdev = 1;
iter = 0;

y0 = feval(f,x0);
y1 = feval(f,x1);

x = x0;
y = y0;

while (xdev>xtol || abs(y)>ftol) && iter<maxiter
  newx = (x0*y1 - x1*y0)/(y1-y0);
  xdev = abs(x-newx);
  x = newx;
  y = feval(f,x);
  if y*y1<0
    x0 = x1;
    y0 = y1;
  else
    g = 1.0-y/y1;
    g = 0.5*(g<=0) + g*(g>0);
    y0 = g*y0;
  end
  x1 = x;
  y1 = y;
  iter = iter + 1;
end
```