Lecture 8

Root finding II

1 Introduction

In the previous lecture we considered the bisection root-bracketing algorithm. It requires only that the function be continuous and that we have a root bracketed to start. Under those conditions it is *guaranteed* to converge with error

$$\epsilon_k = \left(\frac{1}{2}\right)^k \epsilon_0$$

at the k^{th} iteration. Thus bisection provides linear convergence with a rate of convergence of 1/2. Because of its general applicability and guaranteed convergence, bisection has much to recommend it.

We also studied fixed-point iteration

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

where a is a constant. We found that provided we start out "close enough" to a root r the method converges linearly with error

$$\epsilon_k = \left[g'(r)\right]^k \epsilon_0 \tag{1}$$

As this is a root-polishing algorithm, it does not require an initial root bracketing, which might be considered a plus. Still, for one-dimensional functions $f(x)$, fixed-point iteration is not an attractive algorithm. It provides the same linear convergence as bisection without any guarantee of finding a root, even if one exists. However, it can be useful for multidimensional problems. What we want to investigate here is the tantalizing prospect suggested by $g'(r) = 0$ which in light of (1) suggests "superlinear" convergence of some sort.

2 Newton's method

To achieve $g'(r) = 1 + a\, f'(r) = 0$ we take

$$a = -\frac{1}{f'(r)}$$

to arrive at the root polishing iteration formula

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

The only problem is that to compute $f'(r)$ we'd need to know r, and that's what we are searching for in the first place. But if we are "close" to the root, so that $f'(x_k) \approx f'(r)$, then it makes sense to use

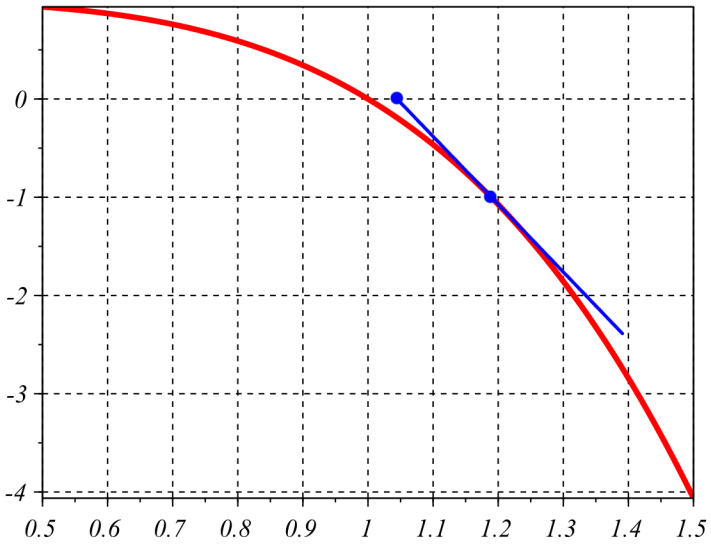

Fig. 1: Newton's method.

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

Let's derive this formula another way. If $f(x)$ is continuous and differentiable, then for small changes in x, $f(x)$ is well approximated by a first-order Taylor series. If we expand the Taylor series about the point $x = x_k$ and take $x_{k+1} = x_k + h$ then, assuming h is "small," we can write

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

and solve for

$$h = -\frac{f(x_k)}{f'(x_k)}$$

This gives us the formula

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \tag{2}$$

which is called *Newton's method*. In Newton's method we "model" the function by the tangent line at the current point $(x_k, f(x_k))$. The root of the tangent line is our next estimate of the root of $f(x)$ (Fig. 1).

Taking $x_k = r + \epsilon_k$ and assuming the error ϵ_k is small enough that the 2nd order Taylor series

$$f(x) \approx f(r) + f'(r)\epsilon + \frac{1}{2}f''(r)\epsilon^2 = f'(r)\epsilon + \frac{1}{2}f''(r)\epsilon^2$$

is accurate, it's straight-forward (but a bit messy) to show that Newton's method converges quadratically with

$$\epsilon_{k+1} = \frac{1}{2}\frac{f''(r)}{f'(r)}\epsilon_k^2$$

Example: Let's take

$$f(x) = x^2 - 2 = 0$$

Since $f'(x) = 2x$, Newton's method gives the iteration formula

$$x_{k+1} = x_k - \frac{x_k^2 - 2}{2x_k}$$

Let's start at $x_0 = 1$. Four iterations produce the sequence of numbers

$$1.5,\ 1.4166667,\ 1.4142157,\ 1.4142136$$

This is very rapid convergence toward the root $\sqrt{2} = 1.4142136\ldots$. In fact the final value (as stored in the computer) turns out to be accurate to about 11 decimal places.

A Scilab implementation of Newton's is given in the Appendix. We require two functions as arguments: $f(x)$ and $f'(x)$. Newton's method fails if $f'(x_k) = 0$ and more generally performs poorly if $|f'(x_k)|$ is very small. It can also fail if we start far away from a root.

With root-bracketing methods, such as bisection, we know that the error in our root estimate is less than or equal to half the last bracketing interval. This gives us a clear termination condition ; we stop when the maximum possible error is less than some defined tolerance. With root-polishing methods, such as Newton's method, we don't have any rigorous bound on the error in our root. Deciding when to stop iterating involves some guess work. A simple criterion is

$$|x_k - x_{k-1}| \leq \text{tol} \tag{3}$$

that is, we terminate when the change in root estimates is less than some specified tolerance. This is a reasonable way to estimate the actual uncertainty in our root estimate, but keep in mind that it is not a *rigorous* bound on the actual error, as is the case with bisection. Since Newton's method is not guaranteed to converge – it may just "bounce around" for ever – it is a good idea to also terminate if the number of iterations exceeds some maximum value.

It's natural to look for methods that converge with order $q = 3, 4, \ldots$. *Householder's method* generalizes Newton's method to higher orders. The $q = 3$ version is called *Halley's method*. It requires calculation of $f(x_k), f'(x_k), f''(x_k)$ at every step. The higher-order methods likewise require ever higher-order derivatives to be calculated. For practical purposes, the increased order of convergence is more than offset by the added calculations and complexity. One application where this is not necessarily the case is polynomials. It is very easy to calculate a polynomial's derivatives of all orders, and higher-order methods do find application in polynomial root finding.

If started "close enough" to a simple root Newton's method generally performs very well and is the "method of choice" in many applications. It does, however, require that we compute both

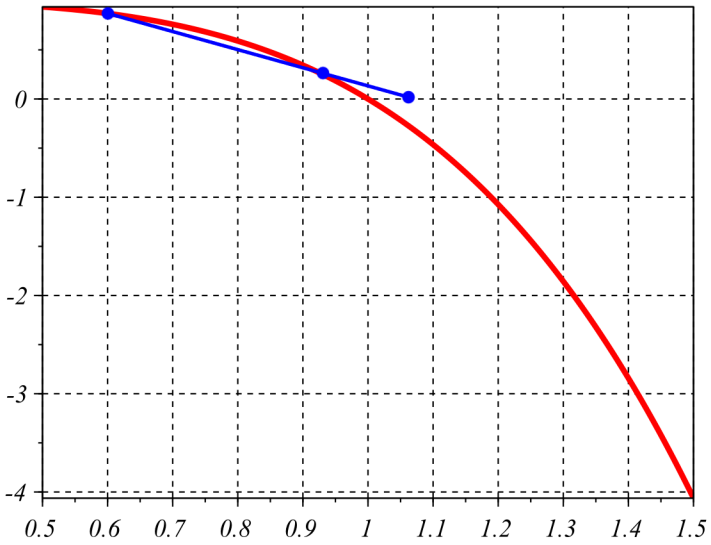

Fig. 2: Secant method

$f(x)$ and $f'(x)$ at each iteration. In some cases it may not be possible or practical to compute $f'(x)$ explicitly. We would like a method that gives us the rapid convergence of Newton's method without the need of calculating derivatives.

3 Secant method

In the *secant method* we replace the derivative appearing in Newton's method by the approximation

$$f'(x_k) \approx \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}$$

Substituting this into (2) results in

$$x_{k+1} = x_k - f(x_k)\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})} \tag{4}$$

Another way to view the secant method is as follows. Suppose we have evaluated the function at two points $(x_k, f_k = f(x_k))$ and $(x_{k-1}, f_{k-1} = f(x_{k-1}))$. Through these two points we can draw a line, the formula for which is

$$y = f_k + \frac{f_k - f_{k-1}}{x_k - x_{k-1}}(x - x_k)$$

Setting this formula equal to zero and solving for *x* we obtain (4). In the secant method we model the function $f(x)$ by a line through our last two root estimates (Fig. 2). Solving for the root of that line provides our next estimate.

When the secant method works it converges with order $q \approx 1.6$, superlinear but not quadratic. Comparison of Figs. 1 and 2 suggest why this is. The secant method will tend to underestimate or

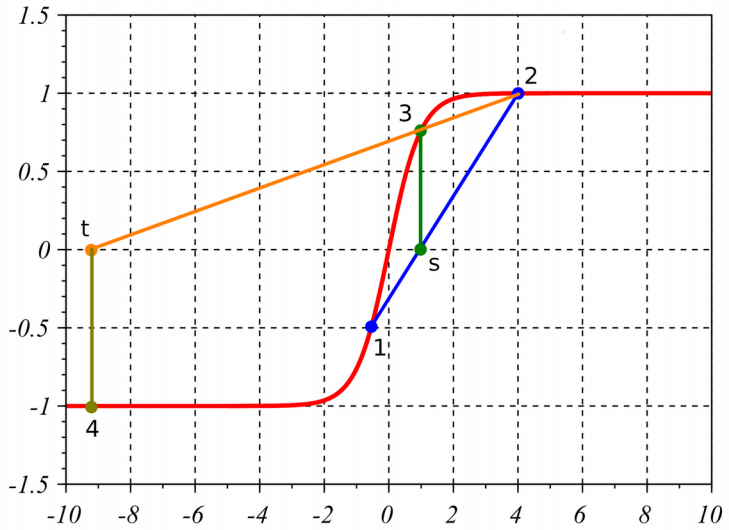

Fig. 3: Failure of the secant method

overestimate (depending on the 2nd derivative of the function at *r*) the actual slope at the point . We are effectively using an average of the slopes at $(x_k, f(x_k))$ and $(x_{k-1}, f(x_{k-1}))$ whereas we use the true slope at $(x_k, f(x_k))$ in Newton's method. It can be shown [2] that the secant method converges as

$$\epsilon_{k+1} = \frac{1}{2}\frac{f''(r)}{f'(r)}\epsilon_k \epsilon_{k-1}$$

or

$$|\epsilon_{k+1}| \approx \left|\frac{1}{2}\frac{f''(r)}{f'(r)}\right|^{0.618} |\epsilon_k|^{1.618}$$

so the secant method is of order $q \approx 1.6$, which is superlinear but less than quadratic. A Scilab implementation of the secant method is given in the appendix.

As is so often the case with numerical methods we are presented a trade off. The secant method does not require explicit calculation of derivatives while Newton's method does. But, the secant method does not converge as fast as Newton's method. As with all root-polishing methods, deciding when to stop iterating involves some guess work. Criterion (3) is an obvious choice.

As illustrated in Fig. 3, the secant method can fail, even when starting out with a bracketed root. There we start with points 1 and 2 on the curve. The line through those points crosses the *x* axis at *s*. The corresponding point on the curve is point 3. Now we draw a line through points 2 and 3. This gives the root estimate *t*. The corresponding point on the curve is point 4. We are actually moving *away* from the root.

In the case illustrated points 1 and 2 bracket a root while points 2 and 3 do not. Clearly if we have a root bracketed we should never accept a new root estimate that falls outside that bracket. The *false position* method is a variation of the secant method in which we use the last two points

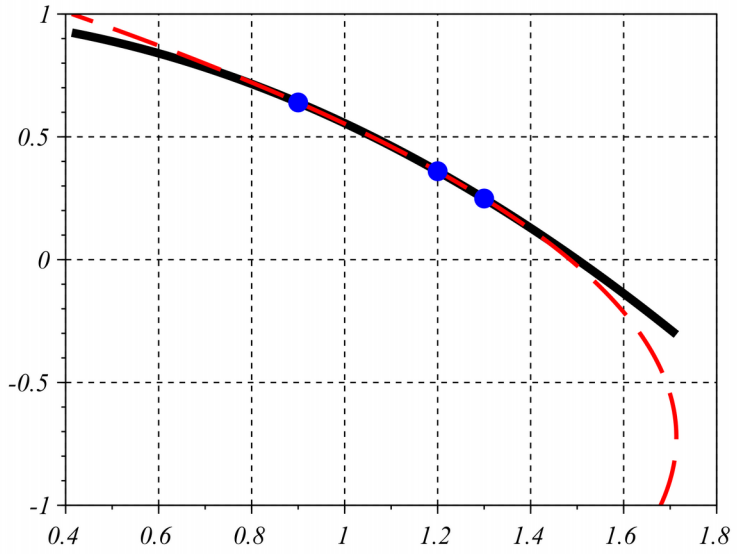

Fig. 4: Inverse quadratic interpolation. Solid line is the function. Dotted line is x represented as a quadratic function of y passing through three given points on the curve.

which bracket a root to represent our linear approximation. In that case we would have drawn a line between point 3 and point 1. Of course, as in the bisection method, this would require us to start with a bracketed root. Moreover in cases where both methods would converge the false position method does not converge as fast as the secant method.

4 Inverse quadratic interpolation

Given two points on a curve (x_k, f_k) and (x_{k-1}, f_{k-1}) the secant method approximates the function by a line $y = f(x) \approx c_1 + c_2 x$ (Fig. 2). This suggests that if we have a third point (x_{k-2}, f_{k-2}) we might draw a parabola through the three points to obtain a quadratic approximation to the function. Setting that approximation equal to zero

$$y = f(x) \approx c_1 + c_2 x + c_3 x^2 = 0$$

might provide a better root approximation than the secant method. Unfortunately we would have to solve a quadratic equation in this case. An alternative approach is *inverse quadratic interpolation* where we represent x as a quadratic function of y

$$x = f^{-1}(y) = c_1 + c_2 y + c_3 y^2$$

Setting $y = f(x) = 0$ we simply have $x = c_1$ (Fig. 4). It turns out there is an explicit formula for this value

$$x = \frac{x_1 f_2 f_3}{(f_1 - f_2)(f_1 - f_3)} + \frac{x_2 f_3 f_1}{(f_2 - f_3)(f_2 - f_1)} + \frac{x_3 f_1 f_2}{(f_3 - f_1)(f_3 - f_2)}$$

(We will understand this formula when we study Lagrange interpolation.) Inverse quadratic

interpolation allows us to exploit information about both the first derivative (slope) and second derivative (curvature) of the function.

Like the secant method, inverse quadratic interpolation can also fail. But when it works it converges with order $q \approx 1.8$. Still not as rapid as Newton's method, but it does not require evaluation of derivatives. It converges faster than the secant method ($q \approx 1.6$) but at the cost of more bookkeeping and a more complicated recursion formula.

5 Hybrid methods

In general, numerical root finding is a difficult problem. We are presented with various trade offs, such as that between the guaranteed converge of the bisection method and the faster convergence of the Newton, secant or inverse quadratic methods. Consequently people have developed *hybrid methods* that seek to combine the best of two or more simpler methods. One of the most widely used is *Brent's method* [1] (used in the Matlab `fzero` function). This method combines the bisection, secant and inverse quadratic methods. Brent's method starts off with a bracketed root. If we don't have a bracket then we have to search for one. With those two initial points, Brent's method applies the secant method to get a third point. From there it tries to use inverse quadratic interpolation for rapid convergence, but applies tests at each iteration to see if it is actually converging superlinearly. If not, Brent's method falls back to the slow-but-sure bisection method. At the next iteration it again tries inverse quadratic interpolation. Another example is *Powell's hybrid method*. (used in the Scilab `fsolve` function).

There is no single agreed-upon "one size fits all" algorithm for root finding. Hybrid methods seek to use the fastest algorithm that "seems to be working" with the option to fall back to slower, but surer methods as a backup. These methods are recommended for most root-finding applications. However there may be specific applications where a particular method will be superior. Newton's method is hard to beat if it is possible to directly calculate both $f(x)$ and $f'(x)$ and a reasonably accurate initial guess to a root is available. It can be coded very compactly, so it's easy to incorporate directly into a program.

6 The fsolve command (Scilab)

As with nearly all non-trivial numerical algorithms, Scilab employs state-of-the-art methods developed by numerical computation researchers in it's library of functions. For solving $f(x)=0$ Scilab provides the `fsolve` function. The simplest use of this is

```
r = fsolve(x0,f);
```

Here `x0` is an initial guess at a root of the function `f(x)`. The returned value `r` is the estimated root. For example

```
-->deff('y=f(x)','y=cos(x)');

-->r = fsolve(1,f)
 r  =

    1.5707963
```

The Matlab equivalent is called `fzero`, although it has a slightly different syntax. Note, however, that `fsolve` may fail to find a root but may still return a value `r`. For example

```
-->deff('y=f(x)','y=2+cos(x)');

-->r = fsolve(1,f)
 r  =

    3.1418148
```

The function $2+\cos(x)$ is never 0 yet Scilab returned a value for *r*. This value is actually where the function gets closest to the *x* axis; it's where $|f(x)|$ is a minimum. Therefore, you should always check the value of the function at the reported "root." Running the command using the syntax

```
[r,fr] = fsolve(x0,f);
```

returns the value of the function at `r`, `fr=f(r)`. For example

```
-->[r,fr] = fsolve(1,f)
 fr  =

    1.
 r  =

    3.1418148
```

shows us that *r* is not actually a root since $f(r)=1$. On the other hand

```
-->deff('y=f(x)','y=0.5+cos(x)');

-->[r,fr] = fsolve(1,f)
 fr  =
    2.220D-16
 r  =
    2.0943951
```

makes it clear that *r* is a root in this case. By default, `fsolve` tries to find the root to within an estimated tolerance of 10^{-10} . You can specify the tolerance explicitly as in

```
[r,fr] = fsolve(x0,f,tol);
```

Because of limitations due to round-off error it is not recommended to use a smaller tolerance than the default. There may be situations where you don't need much accuracy and using a larger tolerance might save a few function calls.

7 References

1. Brent, Richard P. *Algorithms for Minimization Without Derivatives*. Dover Publications. Kindle edition. ASIN: B00CRW5ZTK. 2013 (Originally published 1973)
2. http://www.math.drexel.edu/~tolya/300_secant.pdf

8 Appendix – Scilab code

8.1 Newton's method

```
0001  ////////////////////////////////////////////////////////////////////
0002  // rootNewton.sci
0003  // 2014-06-04, Scott Hudson
0004  // Implements Newton's method for finding a root f(x) = 0.
0005  // Requires two functions: y=f(x) and y=fp(x) where fp(x) is
0006  // the derivative of f(x). Search starts at x0. Root is returned as r.
0007  ////////////////////////////////////////////////////////////////////
0008  function r=rootNewton(x0, f, fp, tol)
0009    MAX_ITERS = 40; //give up after this many iterations
0010    nIters = 1; //1st iteration
0011    r = x0-f(x0)/fp(x0); //Newton's formula for next root estimate
0012    while (abs(r-x0)>tol) & (nIters<=MAX_ITERS)
0013      nIters = nIters+1; //keep track of # of iterations
0014      x0 = r; //current root estimate is last output of formula
0015      r = x0-f(x0)/fp(x0); //Newton's formula for next root estimate
0016    end
0017  endfunction
```

8.2 Secant method

```
0001  ////////////////////////////////////////////////////////////////////
0002  // rootSecant.sci
0003  // 2014-06-04, Scott Hudson
0004  // Implements secant method for finding a root f(x) = 0.
0005  // Requires two initial x values: x1 and x2. Root is returned as r
0006  // accurate to (hopefully) about tol.
0007  ////////////////////////////////////////////////////////////////////
0008  function r=rootSecant(x1, x2, f, tol)
0009    MAX_ITERS = 40; //maximum number of iterations allowed
0010    nIters = 1; //1st iteration
0011    fx2 = f(x2);
0012    r = x2-fx2*(x2-x1)/(fx2-f(x1));
0013    while (abs(r-x2)>tol) & (nIters<=MAX_ITERS)
0014      nIters = nIters+1;
0015      x1 = x2;
0016      fx1 = fx2;
0017      x2 = r;
0018      fx2 = f(x2);
0019      r = x2-fx2*(x2-x1)/(fx2-fx1);
0020    end
0021  endfunction
```

9 Appendix – Matlab code

9.1 Newton's method

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% rootNewton.m
%% 2014-06-04, Scott Hudson
%% Implements Newton's method for finding a root f(x) = 0.
%% Requires two functions: y=f(x) and y=fp(x) where fp(x) is
%% the derivative of f(x). Search starts at x0. Root is returned as r.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function r = rootNewton(x0,f,fp,tol)
  MAX_ITERS = 40; %%give up after this many iterations
  nIters = 1; %%1st iteration
  r = x0-f(x0)/fp(x0); %%Newton's formula for next root estimate
  while (abs(r-x0)>tol) & (nIters<=MAX_ITERS)
    nIters = nIters+1; %%keep track of # of iterations
    x0 = r; %%current root estimate is last output of formula
    r = x0-f(x0)/fp(x0); %%Newton's formula for next root estimate
  end
end
```

9.2 Secant method

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% rootSecant.m
%% 2014-06-04, Scott Hudson
%% Implements secant method for finding a root f(x) = 0.
%% Requires two initial x values: x1 and x2. Root is returned as r
%% accurate to (hopefully) about tol.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function r = rootSecant(x1,x2,f,tol)
  MAX_ITERS = 40; %%maximum number of iterations allowed
  nIters = 1; %%1st iteration
  fx2 = f(x2);
  r = x2-fx2*(x2-x1)/(fx2-f(x1));
  while (abs(r-x2)>tol) & (nIters<=MAX_ITERS)
    nIters = nIters+1;
    x1 = x2;
    fx1 = fx2;
    x2 = r;
    fx2 = f(x2);
    r = x2-fx2*(x2-x1)/(fx2-fx1);
  end
end
```