Chapter 3

Nonlinear Equations

Nonlinear equations can take one of two forms. In the nonlinear *rootfinding problem*, a function f from $\Re^n$ to $\Re^n$ is given, and one must compute an n-vector x, called a *root* of f, that satisfies

$$f(x) = 0.$$

In the nonlinear *fixed-point problem*, a function g from $\Re^n$ to $\Re^n$ is given, and one must compute an n-vector x, called a *fixed-point* of g, that satisfies

$$g(x) = x.$$

The two forms are equivalent. The rootfinding problem may be recast as a fixed-point problem by letting $g(x) = x - f(x)$; conversely, the fixed-point problem may be recast as a rootfinding problem by letting $f(x) = x - g(x)$.

Nonlinear equations arise in many economic applications. For example, the typical static equilibrium model characterizes market prices and quantities with an equal number of supply, demand, and market clearing equations. If one or more of the equations is nonlinear, a nonlinear rootfinding problem arises. One also encounters a nonlinear rootfinding problem when attempting to maximize a real-valued function by setting its first derivative to zero. Yet another way in which nonlinear equations arise in computational economic applications are as elementary tasks in solution procedures designed to solve more complicated functional equations. For example, the Euler functional equation of a dynamic optimization problem might be solved using a collocation method, which gives rise to a nonlinear equation.

Various practical problems arise with nonlinear equations. In many applications, it is not possible to solve the nonlinear equation analytically. In these instances, the solution is often computed numerically using an iterative method that reduces the nonlinear problem to a sequence of linear problems. Such methods can be very sensitive to initial conditions and inherit many of the potential problems of linear equation methods, most notably rounding error and ill-conditioning. Nonlinear equations also present the added problem that they may have more than one solution.

Over the years, numerical analysts have studied nonlinear equations extensively and have devised algorithms for solving them quickly and accurately. In applied work, one can often rely on Newton and quasi-Newton methods, which use derivatives or derivative estimates to help locate the root or fixed-point of a function. Another technique, the function iteration method, is applicable to a nonlinear equation expressed as a fixed-point problem. Yet another method, the bisection method, is very simple to implement, but is applicable only to univariate problems.

3.1 Bisection Method

The *bisection method* is perhaps the simplest and most robust method for computing the root of a continuous real-valued function defined on an interval of the real line. The bisection method is based on the Intermediate Value Theorem. According to the theorem, if a continuous function defined on an interval assumes two distinct values, then it must assume all values in between. In particular, if f is continuous, and $f(a)$ and $f(b)$ have different signs, then f must have at least one root x in $[a, b]$.

The bisection method is an iterative procedure. Each iteration begins with an interval known to contain or to 'bracket' a root of f. The interval is bisected into two subintervals of equal length. One of two subintervals must contain a root of f. This subinterval is taken as the new interval with which to begin the subsequent iteration. In this manner, a sequence of intervals is generated, each half the width of the preceding one, and each known to contain a root of f. The process continues until the width of the interval known to contain a root of f shrinks below an acceptable convergence tolerance.

The following code segment computes the root of a univariate function f using the bisection method. The code segment assumes that the user has specified a convergence tolerance `tol` and two points, `a` and `b`, at which the function has different signs. It calls a user-defined routine `f` that computes the value of the function at a point and an intrinsic function `sign` that returns -1, 0, or 1 if its argument is negative, zero, or positive, respectively:

```
sa = sign(f(a));
sb = sign(f(b));
if sa=sb, error('same sign at endpoints');
while abs(b-a)>tol;
   x = (a+b)/2;
   sx = sign(f(x));
   if sx == sa;
      a = x;
   else
      b = x;
   end
end
x = (a+b)/2;
```

The bisection method's greatest strength is its robustness: In contrast to other rootfinding methods, the bisection method is guaranteed to compute a root to a prescribed tolerance in a known number of iterations, provided valid data are input. Specifically, the method computes a root to a precision τ in no more than in $\log((b-a)/\tau)/\log(2)$ iterations. The bisection method, however, typically requires more iterations than other rootfinding methods to compute a root to a given precision, largely because it ignores information about the function's curvature. Given its relative strengths and weaknesses, the bisection method is often used in conjunction with other rootfinding methods. In this context, the bisection method is first used to obtain a crude approximation for the root. This approximation then becomes the starting point for a more precise rootfinding method that is sensitive to starting point. The more precise method is then used to compute a sharper, final approximation to the root.

3.2 Newton's Method

In practice, most nonlinear equations are solved using *Newton's method* or one of its variants. Newton's method is based on the principle of *successive linearization*. Successive linearization calls for a hard nonlinear problem to be replaced with a sequence of simpler linear problems whose solutions converge to the solution of the nonlinear problem. Newton's method is typically formulated as a rootfinding technique, but may be used to solve a fixed-point problem $g(x) = x$ by recasting it as the rootfinding problem $x - g(x) = 0$.

The univariate Newton method is graphically illustrated in figure 1. The algorithm begins with the analyst supplying a guess x_0 for the root of f. The function f is approximated by its first-order Taylor series expansion about x_0, which is graphically represented by the line tangent to f at x_0. The root x_1 of the tangent line is then accepted as an improved estimate for the root of f. The step is repeated, with the root x_2 of the line tangent to f at x_1 taken as an improved estimate for the root of f, and so on. The process continues until the roots of the tangent lines converge.

Figure 3.1: Newton's method.

More generally, the multivariate Newton method begins with the analyst supplying a guess x_0 for the root of f. Given x_k, the subsequent iterate x_{k+1} is computed by solving the linear rootfinding problem obtained by replacing f with its first order Taylor approximation about x_k:

$$f(x) \approx f(x_k) + f'(x_k)(x - x_k) = 0.$$

This yields the iteration rule

$$x_{k+1} \leftarrow x_k - [f'(x_k)]^{-1} f(x_k).$$

In theory, Newton's method converges if f is continuously differentiable and if the initial value of x supplied by the analyst is "sufficiently" close to a root of f at which f' is invertible. There is, however, no generally practical formula for determining what sufficiently close is. Typically, an analyst makes a reasonable guess for the root f and counts his blessings if the iterates converge. If the iterates do not converge, then the analyst must look more closely at the analytic properties of f to find a better starting value, or change to another rootfinding method. Newton's method can be robust to starting value if f is well behaved, for example, if f has monotone second derivatives. Newton's method can be very sensitive to starting value, however, if the function behaves erratically, for example, if f has high derivatives that change sign frequently. Finally, in practice it is not sufficient for f' to be merely invertible at the root. If f' is invertible but ill-conditioned, then rounding errors in the vicinity of the root can make it difficult to compute a precise approximation.

The following code segment computes the root of a function f using Newton's method. It assumes that the user has provided an initial guess `x` for the root, a convergence tolerance `tol`, and an upper limit `maxit` on the number of iterations. It calls a user-supplied routine `func` that computes the value `f` and Jacobian `d` of the function at an arbitrary point `x`. To conserve on storage, only the most recent iterate is stored:

```
for it=1:maxit
   [f,d] = func(x);
   x = x - d\f;
   if norm(f)<tol, break, end;
end
```

If the initial estimate of the root is poor, then Newton's method may diverge. In these instances, the stability of Newton's method can often be enhanced, at a cost, by monitoring the iterates to ensure that they improve rather than deteriorate with each iteration. Since the norm of the function value $||f(x)||$ is precisely zero at a root, one may view an iterate as yielding an improvement if it reduces this norm. If an iterate increases the norm of the function value, then one can cut the step length prescribed by the Newton method in half, and continue cutting it in half, until the revised iterate yields an improvement. Cutting the step length in half, when necessary, prevents Newton's method from taking a large step in the wrong direction, something that can occur early in execution if the starting value is poor or the function is irregular. Newton methods that implement this feature are said to be 'safeguarded'.

The following code segment computes the root a function using safeguarded Newton's method. It assumes that the user has specified a maximum number of cuts in the step length `maxcut`:

```
[f,d] = func(x);
normf = norm(f);
for it=1:maxit
   xold = x;
   normfold = normf;
   delx = - d\f;
   for ic=1:maxcut;
```

```
         x = xold + delx;
         [f,d] = func(x);
         normf = norm(f);
         if normf > normfold
            delx = delx/2
         else
            break
         end;
      end
      if normf < tol, break, end;
   end
```

In practice, the most common cause of convergence failure in Newton's method is not a poor starting value, but rather a programming error by the analyst. While Newton's method tends to be far more robust to initialization than the underlying theory suggests, the iterates can easily explode or begin to jump around wildly if either the user-supplied function and derivative evaluation routines contain a coding error. For this reason, the analyst should always verify his or her code by comparing the derivatives computed by the derivative evaluation routine with those computed using finite differencing and the function routine. Typically, a programming error in either the function value code or the derivative code will show up clearly in such a comparison.

3.3 Quasi-Newton Methods

Quasi-Newton methods are based on the same successive linearization principle as the Newton method, except that they replace the derivative of f with an estimate that is easier to compute. Quasi-Newton methods are less likely to fail due to programming errors than Newton's method because the analyst need not code the derivative expressions. Quasi-Newton methods, however, often converge more slowly than Newton's method and additionally require the analyst to supply an initial estimate of the function's derivative. Quasi-Newton methods are typically formulated as rootfinding techniques, but can be used to solve a fixed-point problem $g(x) = x$ by recasting it as the equivalent rootfinding problem $x - g(x) = 0$.

The *secant method* is the quasi-Newton method most commonly used to solve univariate rootfinding problems. The secant method is identical to the univariate Newton method, except that it replaces the derivative of f with a finite-difference approximation constructed from the function values at the two previous iterates:

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

This yields the iteration rule

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

Unlike the Newton method, the secant method requires two, rather than one starting value.

The secant method is graphically illustrated in figure 2. The algorithm begins with the analyst supplying two distinct guesses x_0 and x_1 for the root of f. The function f is approximated using the secant line passing through x_0 and x_1, whose root x_2 is accepted as an improved estimate for the root of f. The step is repeated, with the root x_3 of the secant line passing through x_1 and x_2 taken as an improved estimate for the root of f, and so on. The process continues until the roots of the secant lines converge.

Figure 3.2: Secant method.

Broyden's method is the most popular multivariate generalization of the univariate secant method. Broyden's method generates a sequence of vectors x_k and matrices A_k that approximate the root of f and the derivative f' at the root, respectively. Broyden's method begins with guesses x_0 and A_0 supplied by the analyst. Given x_k and A_k, one solves the rootfinding problem obtained by replacing f with the linear approximation:

$$f(x) \approx f(x_k) + A_k(x - x_k) = 0.$$

This yields the root approximation iteration rule

$$x_{k+1} \leftarrow x_k - A_k^{-1} f(x_k).$$

Broyden's method updates the derivative approximant A_k by making the smallest possible change that is consistent with the finite-difference derivative in the direction of change in x_k. This condition yields the iteration rule

$$A_{k+1} \leftarrow A_k + \frac{f(x_{k+1})\delta x_k'}{\delta x_k' \delta x_k}$$

where $\delta x_k = x_{k+1} - x_k$. Note that $f(x_{k+1})\delta x_k'$ is an outer product of two n-vectors, and thus is an n by n matrix.

In theory, Broyden's method converges if f is continuously differentiable, if the initial values of x is "sufficiently" close to a root of f at which f' is invertible, and if A is "sufficiently" close to the derivative at that root. There is, however, no generally practical formula for determining what sufficiently close is. Typically, an analyst makes a reasonable guess for the root f and initializes A by setting it equal to a rescaled identity matrix. A safer approach is to initialize A by setting it equal to a finite difference estimate of the derivative of f at the initial root estimate. Like Newton's method, the stability of Broyden's method depends on the regularity of f and its derivatives. Broyden's method may also have difficulty computing a precise root estimate if f' is ill-conditioned near the root. And finally, the sequence approximants A_k need not, and typically does not, converge to the derivative of f at the root, even if the x_k converge to a root of f.

The following code segment computes the root of a multivariate function f using Broyden's method. It assumes that the user has specified an initial guess `x` for the root, an initial guess `A` for the derivative of the function at `x`, a convergence tolerance `tol`, and an upper limit `maxit` on the number of iterations. The routine calls a user-supplied routine `f` that evaluates the function at an arbitrary point. To conserve on storage and computational effort, the routine stores only the most recent iterates `x` and `A`:

```
v = f(x);
for it=1:maxit
   if norm(v)<tol, break, end;
   delx = -A\v;
   x = x + delx;
   v = f(x);
   A = A + v*delx'/(delx'*delx);
end
```

As with Newton's method, the convergence properties of Broyden's method can often be enhanced by cutting the step size in half if the iterate does not yield a reduction in the norm of the function value $||f(x)||$.

3.4 Function Iteration

Function iteration is a relatively simple technique that in many applications can be used to solve a fixed-point problem $g(x) = x$. Function iteration can also be applied to a rootfinding problem $f(x) = 0$ by recasting it as the equivalent fixed-point problem $x - f(x) = x$.

Function iteration begins with the analyst supplying a guess x_0 for the fixed-point of g. Subsequent iterates are generated using the simple iteration rule

$$x_{k+1} \leftarrow g(x_k).$$

Clearly, if g is continuous and the iterates converge, then they converge to a fixed-point of g.

The function iteration method for a univariate function g is graphically illustrated in figure 3. In this example, g possesses an unique fixed-point x^*, which is graphically characterized by the intersection of g and the dashed 45-degree line. The method begins with the analyst supplying a guess x_0 for the fixed-point of g. Starting from x_0, the next iterate x_1 is obtained by projecting upwards to the g function and then rightward to the 45-degree line. Subsequent iterates are obtained by repeating the projection sequence, tracing out a step function. The process continues until the iterates converge.

Figure 3.3: Function iteration.

In theory, function iteration is guaranteed to converge to a fixed-point of g if g is differentiable and if the initial value of x supplied by the analyst is "sufficiently" close to a fixed-point x^* of g at which $||g'(x^*)|| < 1$. Function iteration, however, often converges even when the sufficiency conditions are not met. Given that the method is relatively easy to implement, it is often worth trying before attempting to use a more complex Newton or quasi-Newton method.

The following code segment computes the fixed-point of a function g using function iteration. It assumes that the user has provided an initial guess `x` for the fixed-point, a convergence tolerance `tol`, and an upper limit `maxit` on the number of iterations. It calls a user-supplied routine `g` that computes the value of the function at an arbitrary point:

```
for it=1:maxit;
   xold = x;
   x = g(xold);
   if norm(x-xold) < tol, break, end;
end;
```

3.5 Choosing a Solution Method

Numerical analysts have special terms that they use to classify the rates at which iterative routines converge. Specifically, a sequence of iterates x_k is said to converge to x^* at a rate of order p if there is constant $C > 0$ such that

$$||x_{k+1} - x^*|| \leq C||x_k - x^*||^p$$

for sufficiently large k. In particular, the rate of convergence is said to be *linear* if $C < 1$ and $p = 1$, *superlinear* if $1 < p < 2$, and *quadratic* if $p = 2$.

The asymptotic rates of convergence of the nonlinear equation solution methods discussed earlier are well known. The bisection method converges at a linear rate with $C = 1/2$. The function iteration method converges at a linear rate with C equal to $||f'(x^*)$. The secant and Broyden methods converge at a superlinear rate, with $p \approx 1.62$. And Newton's method converges at a quadratic rate. The rates of convergence are asymptotically valid, provided that the algorithms are given "good" initial data.

Consider a simple example. The function $g(x) = x^{0.5}$ has an unique fixed-point $x^* = 1$. Function iteration may be used to compute the fixed-point. One can also compute the fixed-point by applying either Newton's method or the secant method to the equivalent rootfinding problem $x - x^{0.5}(x) = 0$. After algebraic reduction, the iteration rules for these three methods are:

Function iteration	$x_{k+1} \leftarrow x_k^{0.5}$
Secant method	$x_{k+1} \leftarrow x_k^{0.5} x_{k-1}^{0.5} / (x_k^{0.5} + x_{k-1}^{0.5} - 1)$
Newton's method	$x_{k+1} \leftarrow x_k / (2x_k^{0.5} - 1)$

Starting from $x_0 = 0.5$, and using a finite difference derivative for the first secant method iteration, the approximation error $|x_k - x^*|$ afforded by the three iteration methods are:

k	Function Iteration	Secant Method	Newton's Method
1	$0.3 \times 10^{+00}$	0.9×10^{-01}	0.9×10^{-01}
2	$0.2 \times 10^{+00}$	0.1×10^{-01}	0.2×10^{-02}
3	0.8×10^{-01}	0.3×10^{-03}	0.9×10^{-06}
4	0.4×10^{-01}	0.9×10^{-06}	0.2×10^{-12}
5	0.2×10^{-01}	0.6×10^{-10}	No Change
6	0.1×10^{-01}	No Change	No Change
7	0.5×10^{-02}	No Change	No Change
8	0.3×10^{-02}	No Change	No Change
9	0.1×10^{-02}	No Change	No Change
10	0.7×10^{-03}	No Change	No Change
15	0.2×10^{-04}	No Change	No Change
20	0.7×10^{-06}	No Change	No Change
25	0.2×10^{-07}	No Change	No Change

This simple experiment generates convergence patterns that are typical for the various iterative nonlinear equation solution algorithms used in practice. Newton's method converges in fewer iterations than the quasi-Newton method, which in turn converges in fewer iterations than function iteration. Both the Newton and quasi-Newton methods converge to machine precision very quickly, in this case 5 or 6 iterations. As the iterates approach the solution, the number of significant digits in the Newton and quasi-Newton approximants begin to double with each iteration.

The rate of convergence, however, is only one determinant of the computational efficiency of a solution algorithm. Algorithms differ in the number of arithmetic operations, and thus the computational effort, required per iteration. For multivariate problems, function iteration requires only a function evaluation, Broyden's method requires a function evaluation and the solution of a linear equation, and Newton's method requires a function evaluation, a derivative evaluation, and the solution of a linear equation. In practice, function iteration tends to require the most overall computational effort to achieve a given accuracy than the other two methods. However, whether Newton's method or Broyden's method requires the most overall computational effort to achieve convergence in a given application depends largely on the dimension of x and complexity of the derivative. Broyden's method will tend to be computationally more efficient than Newton's method if the derivative involves many complex and irregular expressions.

An important factor that must be considered when choosing a nonlinear equation solution method is developmental effort. Developmental effort is the effort exerted by the analyst to produce a viable, convergent computer code—this includes the effort to write the code, the effort to debug and verify the code, and the effort to find suitable starting values. Function iteration and quasi-Newton methods involve the least developmental effort because they do not require the analyst to correctly code the derivative expressions. Newton's method typically requires more developmental effort because it additionally requires the analyst to correctly code derivative expressions. The developmental cost of Newton's method can be quite high if the derivative matrix involves many complex or irregular expressions.

Experienced analysts use certain rules of thumb when selecting a nonlinear equation solution method. If the nonlinear equation is of small dimension, say univariate or bivariate, *or* the function derivatives follow a simple pattern and are relatively easy to code, then development costs will vary little among the different methods and computational efficiency should be the main concern, particularly if the equation is to be solved many times. In this instance, Newton's method is usually the best first choice.

If the nonlinear equation involves many complex or irregular function derivatives, or if the derivatives are expensive to compute, then the Newton's method it less attractive. In such instances, quasi-Newton and function iteration methods may make better choices, particularly if the nonlinear equation is to be solved very few times. If the nonlinear equation is to be solved many times, however, the faster convergence rate of Newton's method may make the development costs worth incurring.