4.4 Newton-Raphson iteration

Concepts

Quadratic convergence
Newton-Raphson iteration
MSP function **NR**
Example: the cam function
Finding the address of an instance of a template function
Example: bouncing polynomial
MSP hybrid Newton-Raphson/bisection function **NRBisect**
Secant method
MSP function **Secant**
Complex equations

In Section 4.3 you saw how the problem of solving an equation $f(x) = 0$ for a root $x = p$ of a function f can be transformed in various ways into a problem of solving an equivalent equation $x = g(x)$ for a fixpoint $x = p$ of a related function g. If g' is continuous near p, $|g'(p)| \le L < 1$, and your initial approximation x_0 is close enough to p, then the successive approximations $x_1, x_2, \ldots$ computed by fixpoint iteration $x_{n+1} = g(x_n)$ converge linearly to p, and the approximation errors $\varepsilon_n = x_n - p$ satisfy the inequality $|\varepsilon_{n+1}| \le L|\varepsilon_n|$. Clearly, you should choose g so that $|g'(p)|$ is as small as possible. The example

$$A > 0, \quad f(x) = x^2 - A, \quad p = \sqrt{A}, \quad g(x) = \frac{1}{2}\left[x + \frac{A}{x}\right], \quad g'(p) = 0$$

indicated that very rapid convergence might result if you choose g so that $g'(p) = 0$.

In fact, this is generally true. If g'' is continuous on a neighborhood N of p that g maps into itself, then

$$\frac{\varepsilon_{n+1}}{\varepsilon_n^2} = \frac{x_{n+1} - p}{(x_n - p)^2} = \frac{g(x_n) - g(p)}{(x_n - p)^2}$$

$$= \frac{g'(p)(x_n - p) + \frac{1}{2}g''(\xi)(x_n - p)^2}{(x_n - p)^2} = \frac{1}{2}g''(\xi).$$

Here, ξ is a number between x_n and p, provided by Taylor's theorem. If $g'(p) = 0$ and $|g''(\xi)| \le M$ for *all* ξ in N, then $|\varepsilon_{n+1}| \le \frac{1}{2} M |\varepsilon_n|^2$ in general. If the error terms ε_n of a convergent sequence $x_0, x_1, x_2, \ldots$ satisfy an inequality like this for any constant M, then the sequence is said to converge *quadratically* to p. (This concept can also be defined for vector, matrix, and complex sequences: just replace the absolute values by appropriate norms.)

To achieve quadratic convergence, you need a way to transform functions f with roots p into functions g with fixpoints p such that $g'(p) = 0$. The most common method is to set

$$g(x) = x - \frac{f(x)}{f'(x)}.$$

Then you can compute

$$g'(x) = \frac{f(x)f''(x)}{f'(x)^2}$$

$$g''(x) = \frac{f'(x)^2 f''(x) + f(x)f'(x)f'''(x) - 2f(x)f''(x)^2}{f'(x)^3},$$

hence $g'(p) = 0$ and g'' is continuous near p provided $f'(p) \ne 0$ and f''' is continuous near p. Fixpoint iteration with this function g is called *Newton-Raphson* iteration. Under these assumptions, the successive approximations converge quadratically to p, provided your initial estimate x_0 is close enough to p. (A more precise analysis shows that only the continuity of f'' is required for this result [20, Chapter 4].

With $A > 0$ and $f(x) = x^2 - A$, the Newton-Raphson iteration function is

$$g(x) = x - \frac{f(x)}{f'(x)} = x - \frac{x^2 - A}{2x} = \frac{1}{2}\left[x + \frac{A}{x}\right].$$

This is just the example given at the beginning of this section. It's known as *Newton's method* for computing square roots.

Newton-Raphson iteration has a vivid graphical representation, shown in Figure 4.4.1. Consider one of the successive approximations x_n to a root p of a function f. The tangent to the graph of $y = f(x)$ at the point $x, y = x_n, f(x_n)$ has equation $y - f(x_n) = f'(x_n)(x - x_n)$. Set $y = 0$ and solve for the intercept x: you get

$$x = x_n - \frac{f(x_n)}{f'(x_n)}.$$

This is the next Newton-Raphson approximation x_{n+1}. Apparently, if x_n is close enough to p and the graph of $y = f(x)$ is sufficiently steep and smooth near $x = p$, the next approximation will be *much* closer to p. Figure 4.4.1 also shows the *next* approximation x_{n+2}; it's much closer to p than even x_{n+1}! (If x_n had been slightly left of p, the next iteration would have brought x_{n+1} to the right of p as in this example. If the graph of f had been concave *down* near p instead of *up*, you'd have to reverse the roles of left and right in this discussion.)

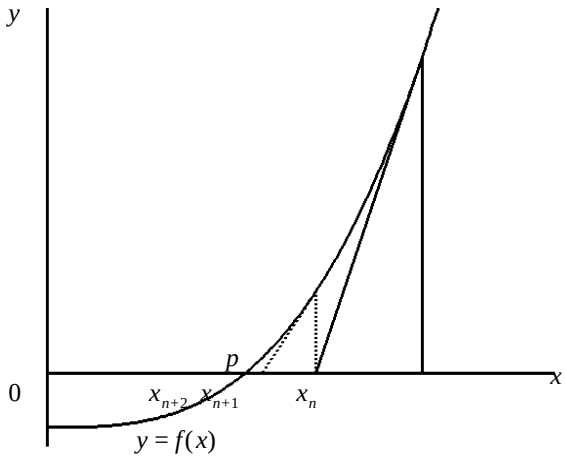

Figure 4.4.1 Newton-Raphson iteration

The MSP implementation of Newton-Raphson iteration is function **NR**, shown in Figure 4.4.2. Since Newton-Raphson iteration is valid for both real and complex scalars, **NR** is a template function. It has parameters **x** and **T** corresponding to the initial estimate x and the tolerance T, and a parameter corresponding to a C++ function **FdF** that computes values of both f and f'. **FdF** must have prototype

```
double  FdF(double  x, double&  dF), or
complex FdF(complex x, complex& dF).
```

It returns the value $f(x)$ after computing **dF** = $f'(x)$. By packaging the code for both f and f' in a single C++ function, you can avoid duplicate computation of expressions common to f and f', as in the next example. An additional **NR** reference parameter **Code** reports the outcome of the computation. **Code** = 0 indicates that the iteration terminated satisfactorily when the difference of two successive approximations became less than the tolerance T, whereas **Code** = −1 signifies that this condition was not attained even after one thousand iterations. (This upper limit was chosen arbitrarily.)

```
template<class Scalar>                  // Find a root of  f  by
  Scalar NR(Scalar FdF(Scalar x,        // Newton-Raphson
                       Scalar& dF),     // iteration.
            Scalar x,                   // Initial estimate.
            double T,                   // Tolerance.
            int&   Code) {              // Status code.
  try {
    int    n = 0;                       // Iteration count.  FdF
    Scalar Fx,dF,dx;                    // must set  dF = f'(x)
    do {                                // and return  f(x) .
        Fx = FdF(x,dF);                      // Newton-Raphson
        if (Fx == Scalar(0)) dx = 0;         // iteration:  do at
          else dx = Fx/dF;                   // most  1000  steps.
        x -= dx; }
      while (++n < 1000 && abs(dx) >= T);
    if (abs(dx) < T) Code = 0;               // Report success, or
      else Code = -1;                        // failure to
    return x; }                              // converge.
    catch(...) {
      cerr << "\nwhile executing  NR ;  now  x,T = "
           << x << ',' << T;
      throw; }}
```

Figure 4.4.2 **Equate1** function **NR**

Figure 4.4.3 shows execution of function **NR** to locate the root of the Figure 4.2.2 cam function. It also contains the code for function **FdFforCam** used to evaluate the cam function and its derivative. You can see that expressions common to $f(x)$ and $f'(x)$ are computed only once. **FdFforCam** is implemented with a template, because later in this section it's used to demonstrate the Newton-Raphson method for a complex equation. The demonstration program's source code includes the template but no explicit **double** or **complex** version of the function definition. **FdFforCam** doesn't point to anything, so you *can't* execute **double x = NR(FdFforCam,0,1e-14,Code)** to produce the output shown in Figure 4.4.3. Instead, the demonstration program included the code

```
double (*FdF)(double,double&) = FdFforCam
double x = NR(*FdF,0,1e-14,Code)
```

to construct a pointer **FdF** to the **double** instance of **FdFforCam** and invoke **NR** with that pointer.

```
Finding a root  x  of the real cam function  f
  by Newton-Raphson iteration

Initial estimate  x :  0
Tolerance         T :  1e-14

Itera-
tion    x       f(x)     f'(x)       dx
  0   0.0     -0.20      0.50     -0.40
  1   0.40    -0.017     0.40     -0.043
  2   0.44    -0.00029   0.39     -0.00076
  3   0.44    -9.4e-08   0.39     -2.4e-07
  4   0.44    -9.6e-15   0.39     -2.5e-14
  5   0.44    -1.6e-17   0.39     -4.0e-17

Root  x ≈  0.443680771333451
Code    =  0
```

This template function computed $f(x)$ *and* $f'(x)$:

```
template<class Scalar>
  Scalar FdFforCam(Scalar  x,
                   Scalar& dF) {
    Scalar T = 2*M_PI;
    Scalar E = .5*exp(-x/T);
    Scalar S = sin(x);
    Scalar C = cos(x);
    dF       = E*(C - S/T);
    return -.2 + E*S; }
```

Figure 4.4.3 Executing function **NR** with function **FdFforCam**

The difference in accuracy between the three equation-solving methods discussed so far is remarkable:

Method	$\lvert x_5 - x_4 \rvert$	$\lvert f(x_5) \rvert$
Section 4.1 Bisection	3.1×10^{-2}	2.4×10^{-3}
Section 4.2 Fixpoint	9.7×10^{-7}	2.8×10^{-8}
Section 4.3 Newton-Raphson	4.0×10^{-17}	1.6×10^{-17}

(For the Fixpoint method, $\lvert f(x_5) \rvert$ was computed by hand.)

The rapid convergence of Newton-Raphson iteration for ideal cases, such as that shown in Figure 4.4.3, is attained at some cost: it's not so general, nor so predictable as the bisection method. Is it ever as slow as that method? When does it fail altogether? Clearly, the iteration will halt with approximation x_n if $f'(x_n) = 0$ or fails to exist. Moreover, the condition derived earlier for quadratic convergence requires that the initial estimate be sufficiently close to the root p, f'' be continuous near p, and $f'(p) \neq 0$. When the latter conditions fail, convergence slows. For example, if $f(x) = x^2$, so that $p = f(p) = f'(p) = 0$, then

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

Thus, in this case the ratio of successive approximation errors is always ½, so the approximations converge linearly, not quadratically. That generally happens when $f'(p) = 0$ but the other conditions hold. If p is a root of order n, then the ratio is $1 - 1/n$ [30, Section 2.3B]. For high-order roots, this ratio makes convergence so slow that numerical analysis software must sometimes use very special mathematical techniques.

A more serious failure can occur when the initial estimate is too far from p. The sequence of Newton-Raphson approximations may approach a different root. Or, if there's a local extremum at a point x near p, the approximations may cycle indefinitely or at least "bounce around" for a while before converging. For example, execute **NR** with initial estimate $x = -12$ to find the root $p \approx 1.46$ (the only real root) of the polynomial $f(x) = x^5 - 8x^4 + 17x^3 + 8x^2 - 14x - 20$ shown in Figure 4.4.4. **NR**
will iterate more than 50 times before settling down to converge rapidly to p. When an approximation x_n falls near the local maximum or minimum, x_{n+1} jumps far away, as shown by the solid tangent lines. Several steps are then needed to return to the region near the root. (The dotted line is referred to later.)

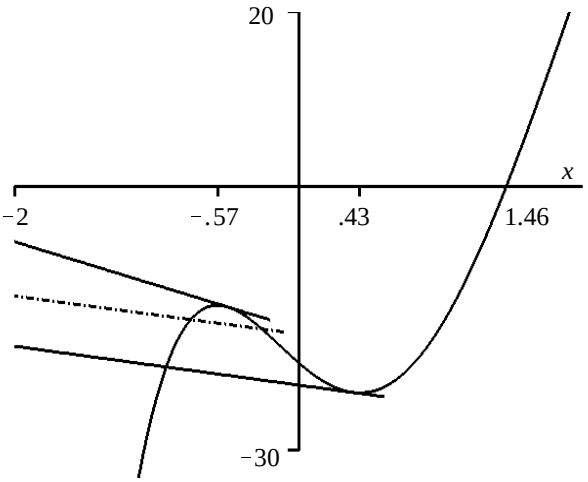

Figure 4.4.4

$$f(x) = x^5 - 8x^4 + 17x^3 + 8x^2 - 14x - 20$$

Figure 4.4.5 shows MSP function **NRBisect**, a hybrid root finder that combines features of the Newton-Raphson and bisection methods to combat this "bouncing" problem. You invoke it like the bisection method, with bracketing estimates x_L and x_R of a root p of a function f: $[x_L, x_R]$ must contain p, and $f(x_L)$ and $f(x_R)$ must differ in sign. **NRBisect** uses the interval midpoint as an initial estimate x, and successively improves x by performing either Newton-Raphson or bisection steps. After each step it adjusts x_L and x_R as in the bisection method so that $[x_L, x_R]$ becomes a smaller bracketing interval, but doesn't necessarily shrink by half. If a Newton-Raphson iteration would throw the next approximation x outside the interval, **NRBisect** sets x equal to the interval midpoint. The iteration terminates when a Newton-Raphson step changes x by an amount less than the specified tolerance T, or when a bisection step yields the midpoint of a bracketing interval shorter than $2T$.

Figure 4.4.6 shows execution of **NRBisect** to find the root of the Figure 4.4.4 polynomial that caused Newton-Raphson iteration to "bounce around" so much. The tabulated **dx** values are either the size of a Newton-Raphson step from the previous approximation, or the length of the current interval. **NRBisect** takes Newton-Raphson steps through Iteration 10. Convergence is somewhat slow because the graph is terribly steep and nonlinear. Then the hybrid function performs three bisections in a row, to avoid bouncing. After that, the approximation is close enough to the root for Newton-Raphson iteration to take over again and converge quickly.

Another device for avoiding "bouncing" is simply to limit the size of the Newton-Raphson steps, relative to the size of the current approximation. That's particularly appropriate for complex polynomial equations; it's implemented in Section 6.4 as function **NRHStep**. Operating on the Figure 4.4.6 polynomial with the same initial estimate, it required 21 iterations—a few more than **NRBisect**.

```
double NRBisect(double FdF(double,     // Find a root of  f.
                           double&),
                double xL,             // Left estimate.
                double xR,             // Right estimate.
                double T,              // Tolerance.
                int&   Code) {         // Status code.
  try {
    Boolean Done;
    int n = 0;                         // Iteration count.
    double x,h,yL,yLp,y,yp;
    x = (xL + xR)/2;                   // The initial approxima-
    h = x - xL;                        // tion is the midpoint.
    yL = FdF(xL,yLp);
    do {                               // Repeat until  h , the
        y = FdF(x,yp);                 // Newton-Raphson
        Done = (h < T);                // correction or half the
        if (!Done) {                   // current interval
          if (Sign(yL) == Sign(y)) {   // length, is  < T .
              xL = x;  yL = y; }
            else                       // Adjust the bracketing
              xR = x;                  // interval.
          h = (y == 0 ? 0 : y/yp);     // Try the Newton-Raphson
          x -= h;                      // correction  h .
          h  = fabs(h);
          if (x < xL || xR < x) {      // If  x  would go out of
            x = (xL + xR)/2;           // bounds, bisect instead.
            h = x - xL; }}}            // Do at most  1000  iter-
      while (++n < 1000 && !Done);     // ations.  Set  Code  to
    Code = -!Done;                     // report success/failure.
    return x; }
  catch(...) {
    cerr << "\nwhile executing  NRBisect ;  now  xL,xR,T = "
         << xL << ',' << xR << ',' << T;
    throw; }}
```

Figure 4.4.5 `Equate1` function `NRBisect`

```
Finding a root of the bouncing function  f
  by the hybrid Newton-Raphson bisection method

Left  estimate  xL :  -26
Right estimate  xR :    2
Tolerance       T  :  .01

Itera-
tion       xL             x              xR          f(x)       dx
    0   -26.00000      -12.00000       2.000000     -4.4e+05   14.
    1   -12.00000      -9.334385       2.000000     -1.4e+05   2.7
    2   -9.334385      -7.215922       2.000000     -4.7e+04   2.1
           :              :               :            :         :
    9   -1.311093      -0.9262393      2.000000     -20.       0.38
   10   -0.9262393     -0.4665007      2.000000     -14.       0.46
   11   -0.4665007      0.7667497      2.000000     -21.       1.2
   12    0.7667497      1.383375       2.000000     -3.3       0.62
   13    1.383375       1.466876       2.000000      0.087     0.084
   14    1.383375       1.464772       1.466876      4.9e-05   0.0021

Root  x ≈  1.465
Code    =  0
```

Figure 4.4.6 Executing function **NRBisect**

One disadvantage of the Newton-Raphson method and its hybrid is their requirement for a value of the derivative $f'(x_n)$ at each approximation x_n of the root p of f. That might be difficult or impossible to provide, especially if values of f are given only by a table, or by the solution of some other substantial problem like a differential equation or a multiple integral. One way to avoid this problem is to replace $f'(x_n)$ in the Newton-Raphson iteration formula

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

by its approximation

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

You can check that the resulting expression

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

is the formula for the x intercept of the secant line through points $x, y = x_{n-1}, f(x_{n-1})$ and $x_n, f(x_n)$. For this reason, the resulting root finding algorithm is called the *secant method*. It requires *two* initial estimates x_0 and x_1. They need not bracket the root.

The secant method attains greater generality than the Newton-Raphson method at a cost. Two initial estimates are required, and convergence is somewhat slower. Its behavior in difficult situations is sometimes problematic, as shown in the next paragraph. Its convergence conditions are similar to those of Newton-Raphson iteration, but the rate of convergence falls between linear and quadratic: if p is a simple root, then the approximation errors $\varepsilon_1, \varepsilon_2, \varepsilon_3, \ldots$ satisfy the equation

$$\lim_{n \to \infty} \frac{\varepsilon_{n+1}}{\varepsilon_n^r} = K, \qquad r = \frac{1 + \sqrt{5}}{2} \approx 1.6$$

for some constant K [20, §4.2].

While the secant method is conceptually similar to Newton-Raphson iteration, its implementation involves an extra complication: you need two initial estimates and you must keep two approximations current. You can see how this is done by inspecting the MSP implementation, function **Secant**, shown in Figure 4.4.7. Executing it to find the root of the cam function as in Figure 4.4.3 produces results comparable to the Newton-Raphson method: with initial estimates 0 and 1, or 0 and 0.1, the secant method achieves the same accuracy with five or six steps. With the troublesome Figure 4.4.4 polynomial f, the results are mixed. With initial estimates −12 and 5 bracketing the root, **Secant** attains the accuracy 0.01 after only seven iterations. But with initial estimates −12 and −11 and tolerance 0.01 it falsely reports a root $x \approx -0.34$ after 16 iterations, even though $f(-0.34) \approx -15$. This happens because two successive approximations x_{14} and x_{15} straddle the local maximum, so that $f(x_{14}) \approx f(x_{15})$. The resulting secant—the dotted line in Figure 4.4.4—is nearly horizontal. Thus x_{16} jumps away from the region shown and $f(x_{16})$ is so large that the secants determined by x_{15} and x_{16} and by x_{16} and x_{17} are nearly vertical. Their intercepts x_{17} and x_{18} are closer than the tolerance, so **Secant** terminates successfully and returns x_{18} even though that's not close to the root. With the smaller tolerance 10^{-6} this

anomaly disappears, though the computation takes longer: **Secant** computed the root appropriately after 27 iterations.

```
template<class Scalar>                  // Find a root of  f .
  Scalar Secant(Scalar f(Scalar),
                Scalar x0,              // Initial
                Scalar x1,              // estimates.
                double T,               // Tolerance.
                int&   Code) {          // Status code.
    try {
      int    n  = 0;                    // Iteration count.
      Scalar dx = x1 - x0;
      Scalar y0 = f(x0);
      Scalar y1,dy;
      do {                              // Secant method
          y1  = f(x1);                  // iteration.  Do at
          dy  = y1 - y0;                // most  1000  steps.
          dx  = -y1*dx/dy;
          x0  = x1;  y0  = y1;
          x1 += dx; }
        while (++n < 1000 && abs(dx) >= T);      // Report
      if (abs(dx) < T) Code = 0;                 // success,
        else Code = -1;                          // or failure
      return x1; }                               // to converge.
    catch(...) {
      cerr << "\nwhile executing  Secant ;  now  x0,x1,T = "
           << x0 << ',' << x1 << ',' << T;
      throw; }}
```

Figure 4.4.7 **Equate1** function **Secant**

Some variations on the secant method are easy to consider. First, you could construct its hybrid with the bisection method, analogous to function **NRBisect**. Second, you could remove the need for two initial estimates of the root of function f by selecting x_1 close to

x_0, but so that $f(x_0)$ and $f(x_1)$ aren't too close. (However, this would prevent you from using bracketing estimates, which might be safer, if you have them at hand.)

The secant and Newton-Raphson methods apply to complex equations, too, because the mathematics that underlies their convergence properties generalizes to complex functions. You can check that functions **`Secant`** and **`NR`** work with **`complex`** scalars. With initial estimate $100 + 100i$, **`NR`** will find the root $111.87 + 17.582i$ of the Figure 4.4.3 cam function. With estimates $\pm 5 \pm 5i$ it will find the roots $3.969 \pm 1.430i$ and -0.701 ± 0.524 of the Figure 4.4.4 bouncing function. You can test **`Secant`** similarly—it may be hard to find successful initial estimates. (For either method to converge to a nonreal root, you *must* start with a nonreal initial estimate, else the algebra will result only in successive real approximations.)

Most complex equations that you'll encounter involve polynomials like the bouncing function. Complex polynomial roots are so important in practice that Newton-Raphson iteration is specially adapted for that case in Section 6.3. That process, in turn, is applied to eigenvalue problems in Section 8.8.