17.1 The Shooting Method

In this section we discuss "pure" shooting, where the integration proceeds from x_1 to x_2, and we try to match boundary conditions at the end of the integration. In the next section, we describe shooting to an intermediate fitting point, where the solution to the equations and boundary conditions is found by launching "shots" from both sides of the interval and trying to match continuity conditions at some intermediate point.

Our implementation of the shooting method exactly implements multidimensional, globally convergent Newton-Raphson (§9.7). It seeks to zero n_2 functions of n_2 variables. The functions are obtained by integrating N differential equations from x_1 to x_2. Let us see how this works:

At the starting point x_1 there are N starting values y_i to be specified, but subject to n_1 conditions. Therefore there are $n_2 = N - n_1$ *freely specifiable* starting values. Let us imagine that these freely specifiable values are the components of a vector **V** that lives in a vector space of dimension n_2. Then you, the user, knowing the functional form of the boundary conditions (17.0.2), can write a subroutine that generates a complete set of N starting values **y**, satisfying the boundary conditions at x_1, from an arbitrary vector value of **V** in which there are no restrictions on the n_2 component values. In other words, (17.0.2) converts to a prescription

$$y_i(x_1) = y_i(x_1; V_1, \ldots, V_{n_2}) \qquad i = 1, \ldots, N \tag{17.1.1}$$

Below, the subroutine that implements (17.1.1) will be called `load`.

Notice that the components of **V** might be exactly the values of certain "free" components of **y**, with the other components of **y** determined by the boundary conditions. Alternatively, the components of **V** might parametrize the solutions that satisfy the starting boundary conditions in some other convenient way. Boundary conditions often impose algebraic relations among the y_i, rather than specific values for each of them. Using some auxiliary set of parameters often makes it easier to "solve" the boundary relations for a consistent set of y_i's. It makes no difference which way you go, as long as your vector space of **V**'s generates (through 17.1.1) all allowed starting vectors **y**.

Given a particular **V**, a particular $\mathbf{y}(x_1)$ is thus generated. It can then be turned into a $\mathbf{y}(x_2)$ by integrating the ODEs to x_2 as an initial value problem (e.g., using Chapter 16's `odeint`). Now, at x_2, let us define a *discrepancy vector* **F**, also of dimension n_2, whose components measure how far we are from satisfying the n_2 boundary conditions at x_2 (17.0.3). Simplest of all is just to use the right-hand sides of (17.0.3),

$$F_k = B_{2k}(x_2, \mathbf{y}) \qquad k = 1, \ldots, n_2 \tag{17.1.2}$$

As in the case of **V**, however, you can use any other convenient parametrization, as long as your space of **F**'s spans the space of possible discrepancies from the desired boundary conditions, with all components of **F** equal to zero if and only if the boundary conditions at x_2 are satisfied. Below, you will be asked to supply a user-written subroutine `score` which uses (17.0.3) to convert an N-vector of ending values $\mathbf{y}(x_2)$ into an n_2-vector of discrepancies **F**.

Now, as far as Newton-Raphson is concerned, we are nearly in business. We want to find a vector value of $\mathbf{V}$ that zeros the vector value of $\mathbf{F}$. We do this by invoking the globally convergent Newton's method implemented in the routine `newt` of §9.7. Recall that the heart of Newton's method involves solving the set of n_2 linear equations

$$\mathbf{J} \cdot \delta\mathbf{V} = -\mathbf{F} \tag{17.1.3}$$

and then adding the correction back,

$$\mathbf{V}^{\rm new} = \mathbf{V}^{\rm old} + \delta\mathbf{V} \tag{17.1.4}$$

In (17.1.3), the Jacobian matrix $\mathbf{J}$ has components given by

$$J_{ij} = \frac{\partial F_i}{\partial V_j} \tag{17.1.5}$$

It is not feasible to compute these partial derivatives analytically. Rather, each requires a *separate* integration of the N ODEs, followed by the evaluation of

$$\frac{\partial F_i}{\partial V_j} \approx \frac{F_i(V_1, \ldots, V_j + \Delta V_j, \ldots) - F_i(V_1, \ldots, V_j, \ldots)}{\Delta V_j} \tag{17.1.6}$$

This is done automatically for you in the routine `fdjac` that comes with `newt`. The only input to `newt` that you have to provide is the routine `funcv` that calculates $\mathbf{F}$ by integrating the ODEs. Here is the appropriate routine:

```
C   SUBROUTINE shoot(n2,v,f) is named "funcv" for use with "newt"
    SUBROUTINE funcv(n2,v,f)
    INTEGER n2,nvar,kmax,kount,KMAXX,NMAX
    REAL f(n2),v(n2),x1,x2,dxsav,xp,yp,EPS
    PARAMETER (NMAX=50,KMAXX=200,EPS=1.e-6)    At most NMAX coupled ODEs.
    COMMON /caller/ x1,x2,nvar
    COMMON /path/ kmax,kount,dxsav,xp(KMAXX),yp(NMAX,KMAXX)
C   USES derivs,load,odeint,rkqs,score
        Routine for use with newt to solve a two point boundary value problem for nvar coupled
        ODEs by shooting from x1 to x2. Initial values for the nvar ODEs at x1 are generated
        from the n2 input coefficients v(1:n2), using the user-supplied routine load. The routine
        integrates the ODEs to x2 using the Runge-Kutta method with tolerance EPS, initial stepsize
        h1, and minimum stepsize hmin. At x2 it calls the user-supplied subroutine score to
        evaluate the n2 functions f(1:n2) that ought to be zero to satisfy the boundary conditions
        at x2. The functions f are returned on output. newt uses a globally convergent Newton's
        method to adjust the values of v until the functions f are zero. The user-supplied subroutine
        derivs(x,y,dydx) supplies derivative information to the ODE integrator (see Chapter
        16). The common block caller receives its values from the main program so that funcv
        can have the syntax required by newt. The common block path is included for compatibility
        with odeint.
    INTEGER nbad,nok
    REAL h1,hmin,y(NMAX)
    EXTERNAL derivs,rkqs
    kmax=0
    h1=(x2-x1)/100.
    hmin=0.
    call load(x1,v,y)
    call odeint(y,nvar,x1,x2,EPS,h1,hmin,nok,nbad,derivs,rkqs)
    call score(x2,y,f)
    return
    END
```

For some problems the initial stepsize ΔV might depend sensitively upon the initial conditions. It is straightforward to alter `load` to include a suggested stepsize `h1` as another returned argument and feed it to `fdjac` via a common block.

A complete cycle of the shooting method thus requires $n_2 + 1$ integrations of the N coupled ODEs: one integration to evaluate the current degree of mismatch, and n_2 for the partial derivatives. Each new cycle requires a new round of $n_2 + 1$ integrations. This illustrates the enormous extra effort involved in solving two point boundary value problems compared with initial value problems.

If the differential equations are *linear*, then only one complete cycle is required, since (17.1.3)–(17.1.4) should take us right to the solution. A second round can be useful, however, in mopping up some (never all) of the roundoff error.

As given here, `shoot` uses the quality controlled Runge-Kutta method of §16.2 to integrate the ODEs, but any of the other methods of Chapter 16 could just as well be used.

You, the user, must supply `shoot` with: (i) a subroutine `load(x1,v,y)` which returns the `n`-vector `y(1:n)` (satisfying the starting boundary conditions, of course), given the freely specifiable variables of `v(1:n2)` at the initial point `x1`; (ii) a subroutine `score(x2,y,f)` which returns the discrepancy vector `f(1:n2)` of the ending boundary conditions, given the vector `y(1:n)` at the endpoint `x2`; (iii) a starting vector `v(1:n2)`; (iv) a subroutine `derivs` for the ODE integration; and other obvious parameters as described in the header comment above.

In §17.4 we give a sample program illustrating how to use `shoot`.

CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America).

Keller, H.B. 1968, *Numerical Methods for Two-Point Boundary-Value Problems* (Waltham, MA: Blaisdell).

17.2 Shooting to a Fitting Point

The shooting method described in §17.1 tacitly assumed that the "shots" would be able to traverse the entire domain of integration, even at the early stages of convergence to a correct solution. In some problems it can happen that, for very wrong starting conditions, an initial solution can't even get from x_1 to x_2 without encountering some incalculable, or catastrophic, result. For example, the argument of a square root might go negative, causing the numerical code to crash. Simple shooting would be stymied.

A different, but related, case is where the endpoints are both singular points of the set of ODEs. One frequently needs to use special methods to integrate near the singular points, analytic asymptotic expansions, for example. In such cases it is feasible to integrate in the direction *away* from a singular point, using the special method to get through the first little bit and then reading off "initial" values for further numerical integration. However it is usually not feasible to integrate *into* a singular point, if only because one has not usually expended the same analytic