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 function 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

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

Below, the function 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),

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

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 function `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 **V** that zeros the vector value of **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,

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

In (17.1.3), the Jacobian matrix **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 vecfunc that calculates **F** by integrating the ODEs. Here is the appropriate routine, called shoot, that is to be passed as the actual argument in newt:

```
#include "nrutil.h"
#define EPS 1.0e-6

extern int nvar;                  Variables that you must define and set in your main pro-
extern float x1,x2;                   gram.

int kmax,kount;                   Communicates with odeint.
float *xp,**yp,dxsav;

void shoot(int n, float v[], float f[])
```

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 routine 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 routine derivs(x,y,dydx) supplies derivative information to the ODE integrator (see Chapter 16). The first set of global variables above receives its values from the main program so that shoot can have the syntax required for it to be the argument vecfunc of newt.

```
{
    void derivs(float x, float y[], float dydx[]);
    void load(float x1, float v[], float y[]);
    void odeint(float ystart[], int nvar, float x1, float x2,
        float eps, float h1, float hmin, int *nok, int *nbad,
        void (*derivs)(float, float [], float []),
        void (*rkqs)(float [], float [], int, float *, float, float,
```

```
        float [], float *, float *, void (*)(float, float [], float [])));
    void rkqs(float y[], float dydx[], int n, float *x,
        float htry, float eps, float yscal[], float *hdid, float *hnext,
        void (*derivs)(float, float [], float []));
    void score(float xf, float y[], float f[]);
    int nbad,nok;
    float h1,hmin=0.0,*y;

    y=vector(1,nvar);
    kmax=0;
    h1=(x2-x1)/100.0;
    load(x1,v,y);
    odeint(y,nvar,x1,x2,EPS,h1,hmin,&nok,&nbad,derivs,rkqs);
    score(x2,y,f);
    free_vector(y,1,nvar);
}
```

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 output variable and feed it to `fdjac` via a global variable.

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 intial 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 function `load(x1,v,y)` which calculates 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 function `score(x2,y,f)` which calculates 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 function `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 effort to obtain expansions of "wrong" solutions near the singular point (those not satisfying the desired boundary condition).

The solution to the above mentioned difficulties is *shooting to a fitting point*. Instead of integrating from x_1 to x_2, we integrate first from x_1 to some point x_f that is *between* x_1 and x_2; and second from x_2 (in the opposite direction) to x_f.

If (as before) the number of boundary conditions imposed at x_1 is n_1, and the number imposed at x_2 is n_2, then there are n_2 freely specifiable starting values at x_1 and n_1 freely specifiable starting values at x_2. (If you are confused by this, go back to §17.1.) We can therefore define an n_2-vector $\mathbf{V}_{(1)}$ of starting parameters at x_1, and a prescription `load1(x1,v1,y)` for mapping $\mathbf{V}_{(1)}$ into a $\mathbf{y}$ that satisfies the boundary conditions at x_1,

$$y_i(x_1) = y_i(x_1; V_{(1)1}, \dots, V_{(1)n_2}) \qquad i = 1, \dots, N \tag{17.2.1}$$

Likewise we can define an n_1-vector $\mathbf{V}_{(2)}$ of starting parameters at x_2, and a prescription `load2(x2,v2,y)` for mapping $\mathbf{V}_{(2)}$ into a $\mathbf{y}$ that satisfies the boundary conditions at x_2,

$$y_i(x_2) = y_i(x_2; V_{(2)1}, \dots, V_{(2)n_1}) \qquad i = 1, \dots, N \tag{17.2.2}$$

We thus have a total of N freely adjustable parameters in the combination of $\mathbf{V}_{(1)}$ and $\mathbf{V}_{(2)}$. The N conditions that must be satisfied are that there be agreement in N components of $\mathbf{y}$ at x_f between the values obtained integrating from one side and from the other,

$$y_i(x_f; \mathbf{V}_{(1)}) = y_i(x_f; \mathbf{V}_{(2)}) \qquad i = 1, \dots, N \tag{17.2.3}$$

In some problems, the N matching conditions can be better described (physically, mathematically, or numerically) by using N different functions F_i, $i = 1 \dots N$, each possibly depending on the N components y_i. In those cases, (17.2.3) is replaced by

$$F_i[\mathbf{y}(x_f; \mathbf{V}_{(1)})] = F_i[\mathbf{y}(x_f; \mathbf{V}_{(2)})] \qquad i = 1, \dots, N \tag{17.2.4}$$