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

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

In the program below, the user-supplied function `score(xf,y,f)` is supposed to map an input N-vector **y** into an output N-vector **F**. In most cases, you can dummy this function as the identity mapping.

Shooting to a fitting point uses globally convergent Newton-Raphson exactly as in §17.1. Comparing closely with the routine `shoot` of the previous section, you should have no difficulty in understanding the following routine `shootf`. The main differences in use are that you have to supply both `load1` and `load2`. Also, in the calling program you must supply initial guesses for `v1[1..n2]` and `v2[1..n1]`. Once again a sample program illustrating shooting to a fitting point is given in §17.4.

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

extern int nn2,nvar;            Variables that you must define and set in your main pro-
extern float x1,x2,xf;          gram.

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

void shootf(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` and `x2` to a fitting point `xf`. Initial values for the `nvar` ODEs at `x1` (`x2`) are generated from the `n2` (`n1`) coefficients `v1` (`v2`), using the user-supplied routine `load1` (`load2`). The coefficients `v1` and `v2` should be stored in a single array `v[1..n1+n2]` in the main program by statements of the form `v1=v;` and `v2 = &v[n2];`. The input parameter `n` = `n1` + `n2` = `nvar`. The routine integrates the ODEs to `xf` using the Runge-Kutta method with tolerance EPS, initial stepsize h1, and minimum stepsize `hmin`. At `xf` it calls the user-supplied routine `score` to evaluate the `nvar` functions `f1` and `f2` that ought to match at `xf`. The differences `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`. Set `nn2` = `n2` in the main program.

```
{
    void derivs(float x, float y[], float dydx[]);
    void load1(float x1, float v1[], float y[]);
    void load2(float x2, float v2[], 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 i,nbad,nok;
    float h1,hmin=0.0,*f1,*f2,*y;

    f1=vector(1,nvar);
    f2=vector(1,nvar);
    y=vector(1,nvar);
    kmax=0;
    h1=(x2-x1)/100.0;
    load1(x1,v,y);                  Path from x1 to xf with best trial values v1.
    odeint(y,nvar,x1,xf,EPS,h1,hmin,&nok,&nbad,derivs,rkqs);
    score(xf,y,f1);
    load2(x2,&v[nn2],y);            Path from x2 to xf with best trial values v2.
    odeint(y,nvar,x2,xf,EPS,h1,hmin,&nok,&nbad,derivs,rkqs);
    score(xf,y,f2);
```

```
    for (i=1;i<=n;i++) f[i]=f1[i]-f2[i];
    free_vector(y,1,nvar);
    free_vector(f2,1,nvar);
    free_vector(f1,1,nvar);
}
```

There are boundary value problems where even shooting to a fitting point fails — the integration interval has to be partitioned by several fitting points with the solution being matched at each such point. For more details see [1].

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).

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §§7.3.5–7.3.6. [1]

17.3 Relaxation Methods

In *relaxation methods* we replace ODEs by approximate *finite-difference equations* (FDEs) on a grid or mesh of points that spans the domain of interest. As a typical example, we could replace a general first-order differential equation

$$\frac{dy}{dx} = g(x, y) \tag{17.3.1}$$

with an algebraic equation relating function values at two points $k, k-1$:

$$y_k - y_{k-1} - (x_k - x_{k-1})\, g\left[\tfrac{1}{2}(x_k + x_{k-1}), \tfrac{1}{2}(y_k + y_{k-1})\right] = 0 \tag{17.3.2}$$

The form of the FDE in (17.3.2) illustrates the idea, but not uniquely: There are many ways to turn the ODE into an FDE. When the problem involves N coupled first-order ODEs represented by FDEs on a mesh of M points, a solution consists of values for N dependent functions given at each of the M mesh points, or $N \times M$ variables in all. The relaxation method determines the solution by starting with a guess and improving it, iteratively. As the iterations improve the solution, the result is said to *relax* to the true solution.

While several iteration schemes are possible, for most problems our old standby, multidimensional Newton's method, works well. The method produces a matrix equation that must be solved, but the matrix takes a special, "block diagonal" form, that allows it to be inverted far more economically both in time and storage than would be possible for a general matrix of size $(MN) \times (MN)$. Since MN can easily be several thousand, this is crucial for the feasibility of the method.

Our implementation couples at most pairs of points, as in equation (17.3.2). More points can be coupled, but then the method becomes more complex. We will provide enough background so that you can write a more general scheme if you have the patience to do so.

Let us develop a general set of algebraic equations that represent the ODEs by FDEs. The ODE problem is exactly identical to that expressed in equations (17.0.1)–(17.0.3) where we had N coupled first-order equations that satisfy n_1 boundary conditions at x_1 and $n_2 = N - n_1$ boundary conditions at x_2. We first define a mesh or grid by a set of $k = 1, 2, \ldots, M$ points at which we supply values for the independent variable x_k. In particular, x_1 is the initial boundary, and x_M is the final boundary. We use the notation $\mathbf{y}_k$ to refer to the entire set of