Eggleton, P.P. 1971, "The Evolution of Low Mass Stars," *Monthly Notices of the Royal Astronomical Society*, vol. 151, pp. 351–364.

London, R.A., and Flannery, B.P. 1982, "Hydrodynamics of X-Ray Induced Stellar Winds," *Astrophysical Journal*, vol. 258, pp. 260–269.

Stoer, J., and Bulirsch, R. 2002, *Introduction to Numerical Analysis*, 3rd ed. (New York: Springer), §7.3 – §7.4.

18.1 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 $\mathbf{V}$ that lives in a vector space of dimension n_2. Then you, the user, knowing the functional form of the boundary conditions (18.0.2), can write a function or functor that generates a complete set of N starting values $\mathbf{y}$, satisfying the boundary conditions at x_1, from an arbitrary vector value of $\mathbf{V}$ in which there are no restrictions on the n_2 component values. In other words, (18.0.2) converts to a prescription

$$y_i(x_1) = y_i(x_1; V_0, \ldots, V_{n_2-1}) \qquad i = 0, \ldots, N-1 \tag{18.1.1}$$

In the routine `Shoot` below, the function or functor that implements (18.1.1) will be called `load`, but you can pass it as an argument to the routine with any name of your choosing.

Notice that the components of $\mathbf{V}$ might be exactly the values of certain "free" components of $\mathbf{y}$, with the other components of $\mathbf{y}$ determined by the boundary conditions. Alternatively, the components of $\mathbf{V}$ 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 $\mathbf{V}$'s generates (through 18.1.1) all allowed starting vectors $\mathbf{y}$.

Given a particular $\mathbf{V}$, 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 17's `Odeint`). Now, at x_2, let us define a *discrepancy vector* $\mathbf{F}$, also of dimension n_2, whose components measure how far we are from satisfying the n_2 boundary conditions at x_2 (18.0.3). Simplest of all is just to use the right-hand sides

of (18.0.3),

$$F_k = B_{2k}(x_2, \mathbf{y}) \qquad k = 0, \ldots, n_2 - 1 \tag{18.1.2}$$

As in the case of $\mathbf{V}$, however, you can use any other convenient parametrization, as long as your space of $\mathbf{F}$'s spans the space of possible discrepancies from the desired boundary conditions, with all components of $\mathbf{F}$ 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 or functor that uses (18.0.3) to convert an N-vector of ending values $\mathbf{y}(x_2)$ into an n_2-vector of discrepancies $\mathbf{F}$. Inside `Shoot`, this function is called `score`.

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

$$\mathbf{J} \cdot \delta\mathbf{V} = -\mathbf{F} \tag{18.1.3}$$

and then adding the correction back,

$$\mathbf{V}^{\text{new}} = \mathbf{V}^{\text{old}} + \delta\mathbf{V} \tag{18.1.4}$$

In (18.1.3), the Jacobian matrix $\mathbf{J}$ has components given by

$$J_{ij} = \frac{\partial F_i}{\partial V_j} \tag{18.1.5}$$

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

$$\frac{\partial F_i}{\partial V_j} \approx \frac{F_i(V_0, \ldots, V_j + \Delta V_j, \ldots) - F_i(V_0, \ldots, V_j, \ldots)}{\Delta V_j} \tag{18.1.6}$$

This is done automatically for you in the functor `NRfdjac` that comes with `newt`. The only input to `newt` that you have to provide is the routine `vecfunc` that calculates $\mathbf{F}$ by integrating the ODEs. Here is the appropriate routine, a functor called `Shoot`, that is to be passed as the actual argument in `newt`:

shoot.h

```
template <class L, class R, class S>
struct Shoot {
Functor for use with newt to solve a two-point boundary value problem by shooting.
    Int nvar;                       Number of coupled ODEs.
    Doub x1,x2;                     Start and end points.
    L &load;                        Supplies initial values for ODEs from v[0..n2-1].
    R &d;                           Supplies derivative information to the ODE integrator.
    S &score;                       Returns the n2 functions that ought to be zero to satisfy
    Doub atol,rtol;                     the boundary conditions at x2.
    Doub h1,hmin;
    VecDoub y;
    Shoot(Int nvarr, Doub xx1, Doub xx2, L &loadd, R &dd, S &scoree) :
        nvar(nvarr), x1(xx1), x2(xx2), load(loadd), d(dd),
        score(scoree), atol(1.0e-14), rtol(atol), hmin(0.0), y(nvar) {}
    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[0..n2-1], using the user-supplied routine load.
    VecDoub operator() (VecDoub_I &v) {
    This is the functor used by newt. It integrates the ODEs to x2 using an eighth-order Runge-
    Kutta method with absolute and relative tolerances atol and rtol, initial stepsize h1, and
```

```
    minimum stepsize hmin. At x2 it calls the user-supplied routine score and returns the
    n2 functions that ought to be zero. newt uses a globally convergent Newton's method to
    adjust the values of v until the returned functions are in fact zero.
        h1=(x2-x1)/100.0;
        y=load(x1,v);
        Output out;              No output generated by Odeint.
        Odeint<StepperDopr853<R> > integ(y,x1,x2,atol,rtol,h1,hmin,out,d);
        integ.integrate();
        return score(x2,y);
    }
};
```

Note that `Shoot` is templated on the `load`, right-hand side for `Odeint`, and `score` routines. In practice, you will almost always want to write these as functors rather than functions. This makes communicating the various parameters in the problem easy — just pass them as parameters in the constructors.

For some problems the initial stepsize ΔV might depend sensitively upon the initial conditions. It is straightforward to alter `load` to compute a suggested stepsize `h1` as a member variable and feed it fist to `Shoot` and hence to `NRfdjac` when the `Shoot` object is passed to `newt`.

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 (18.1.3) – (18.1.4) 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 high-efficiency eighth-order Runge-Kutta method of §17.2 to integrate the ODEs, but any of the other methods of Chapter 17 could just as well be used.

You, the user, must supply `Shoot` with: (i) a function or functor `load(x1,v)` that returns the `n`-vector `y[0..n-1]` (satisfying the starting boundary conditions, of course), given the freely specifiable variables of `v[0..n2-1]` at the initial point `x1`; (ii) a function or functor `score(x2,y)` that returns the discrepancy vector `f[0..n2-1]` of the ending boundary conditions, given the vector `y[0..n-1]` at the endpoint `x2`; (iii) a starting vector `v[0..n2-1]`; (iv) a function or functor, called `d` in the routine, for the ODE integration; and other obvious parameters as described in the header comment above.

In §18.4 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, DC: Mathematical Association of America).

Keller, H.B. 1968, *Numerical Methods for Two-Point Boundary-Value Problems*; reprinted 1991 (New York: Dover).