17.2 Shooting to a Fitting Point

For some problems the initial stepsize $\Delta V$ might depend sensitively upon the initial conditions. It is straightforward to alter load to include a suggested stepsize $h_1$ 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($x_1,v,y$) which returns the $n$-vector $y(1:n)$ (satisfying the starting boundary conditions, of course), given the freely specifiable variables of $v(1:n_2)$ at the initial point $x_1$; (ii) a subroutine score($x_2,y,f$) which returns the discrepancy vector $f(1:n_2)$ of the ending boundary conditions, given the vector $y(1:n)$ at the endpoint $x_2$; (iii) a starting vector $v(1:n_2)$; (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:


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 $V_{(1)}$ of starting parameters at $x_1$, and a prescription $load1(x_1,v1,y)$ for mapping $V_{(1)}$ into a $y$ that satisfies the boundary conditions at $x_1$,

$$y_i(x_1) = y_i(x_1; V_{(1)1},..., V_{(1)n_2}) \quad i = 1,...,N \quad (17.2.1)$$

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

$$y_i(x_2) = y_i(x_2; V_{(2)1},..., V_{(2)n_1}) \quad i = 1,...,N \quad (17.2.2)$$

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

$$y_i(x_f; V_{(1)}) = y_i(x_f; V_{(2)}) \quad i = 1,...,N \quad (17.2.3)$$

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

$$F_i[y(x_f; V_{(1)})] = F_i[y(x_f; V_{(2)})] \quad i = 1,...,N \quad (17.2.4)$$

In the program below, the user-supplied subroutine score$(x_f,y,f)$ is supposed to map an input $N$-vector $y$ into an output $N$-vector $F$. In most cases, you can dummy this subroutine 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.

```fortran
C SUBROUTINE shootf(n,v,f) is named "functv" for use with "newt"
SUBROUTINE functv(n,v,f)
INTEGER n,nvar,nn2,kmax,kount,KMAXX,NMAX
REAL f(n),v(n),x1,x2,xf,dxsav,xp,yp,EPS
PARAMETER (NMAX=50,KMAXX=200,EPS=1.e-6) At most NMAX equations.
COMMON /caller/ x1,x2,xf,nvar,nn2
COMMON /path/ kmax,dsav,xp(KMAXX),yp(NMAX,KMAXX)
C USES derive,load1,load2,odeint,rkqs,score

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
```
17.3 Relaxation Methods

ODEs at \( x_1 \) (\( x_2 \)) are generated from the \( n_2 \) (\( n_1 \)) coefficients \( v_1 \) (\( v_2 \)), using the user-supplied routine \( \text{load1} \) (\( \text{load2} \)). The coefficients \( v_1 \) and \( v_2 \) should be stored in a single array \( v(1:n_1+n_2) \) in the main program by an \texttt{EQUIVALENCE} statement of the form \( \text{(v1(1),v(1)),(v2(1),v(n2+1))} \). The input parameter \( n = n_1+n_2 \) = \( n\text{var} \). The routine integrates the ODEs to \( x_f \) using the Runge-Kutta method with tolerance \( \text{EPS} \), initial stepsize \( h_1 \), and minimum stepsize \( h_{\text{min}} \). At \( x_f \) it calls the user-supplied subroutine \( \text{score} \) to evaluate the \( n\text{var} \) functions \( f_1 \) and \( f_2 \) that ought to match at \( x_f \). The differences \( f \) are returned on output.

\newt \text{uses a globally convergent Newton's method to adjust the values of} v \text{ until the functions} f \text{ are zero. The user-supplied subroutine} \text{derivs}(x,y,dydx) \text{ supplies derivative information to the ODE integrator (see Chapter 16). The common block} \text{caller} \text{receives its values from the main program so that funcv can have the syntax required by newt. Set} \text{nn2 = n2 in the main program. The common block path is for compatibility with odeint.}

\begin{verbatim}
INTEGER i,nbad,nok
REAL h1,hmin,f1(NMAX),f2(NMAX),y(NMAX)
EXTERNAL derivs,rkqs
kmax=0
h1=(x2-x1)/100.
hmin=0.
call load1(x1,v,y) Path from \( x_1 \) to \( x_f \) with best trial values \( v_1 \).
call odeint(y,nvar,x1,xf,\text{EPS},h1,hmin,nok,nbad,derivs,rkqs)
call score(xf,y,f1) 
call load2(x2,v(nn2+1),y) Path from \( x_2 \) to \( x_f \) with best trial values \( v_2 \).
call odeint(y,nvar,x2,xf,\text{EPS},h1,hmin,nok,nbad,derivs,rkqs)
call score(xf,y,f2)
do i=1,n
   f(i)=f1(i)-f2(i)
endo
return
END
\end{verbatim}

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:

17.3 Relaxation Methods

In relaxation methods we replace ODEs by approximate \textit{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

\[
\frac{dy}{dx} = g(x,y)
\]  \hspace{1cm} (17.3.1)

with an algebraic equation relating function values at two points \( k, k-1 \):

\[
y_k - y_{k-1} = (x_k - x_{k-1}) \left[ \frac{1}{2}(x_k + x_{k-1}), \frac{1}{2}(y_k + y_{k-1}) \right] = 0
\]  \hspace{1cm} (17.3.2)