16 Two point boundary value problems

Functions:

shoot: solve the 2 point boundary value problem by shooting
shootf: solve the 2 point boundary value problem by shooting to a fitting point
solvde: solve the 2 point boundary value problem by relaxation

---

shoot

Description:

This function solves the two point boundary problem for a system of ODE's by the shooting technique.

shoot(v, delv, x1, x2, eps, h1, hmin, derivs, load, score) $\longrightarrow$ **[y, v, f, dv]**

Where:

nvar is the number of ODE's,
v is a real list representing initial values for n_2 parameters from which the initial values for the dependent variables may be computed using the function *load*,
delv is a real list being suggested increments to be used in the computation of finite differences by the algorithm,
x1 a real number being the point from which shooting is initiated,
x2 a real number being the point to which shooting is targeted,
eps a real number being the tolerance passed to the Runge-Kutta solver,
h1 a real number being the initial step passed to the Runge-Kutta solver,
hmin a real number being the minimum step size passed to the Runge-Kutta solver,
derivs is a list $[f_1, ..., f_n]$ of the names of Senac functions representing the right hand sides of the system of equations

$$y_i' = f_i(x, y_1, ..., y_n), \quad 1 \leq i \leq n$$

to be integrated,
load is a list of the names of Senac functions of n_2 parameters from which the initial values of the dependent variables at x_1 may be computed,
score is a list of the names of n_2 Senac function of $nvar + 1$ parameters used to evaluate the boundary conditions at x_2 by evaluating n_2 functions,
y is a real list of length *nvar* being the solution values for the dependent variables,
f is a real list of length n_2 being the values of the boundary functions at x_2,
dv is a real list of length n_2 being the current increments in the boundary functions at x_2.

Example:

In this example the function shoot is used to solve a two point boundary value problem. We Solve for eigenvalues for Spheroidal Harmonics. Both Prolate and Oblate case are handled simultaneously, leading to six first-order equations. Unknown to shoot, these are actually two independant sets of 3 coupled equations, one set with c^2 positive and the other with c^2 negative.

```
-> shoot: solve the 2 point boundary value problem by shooting #
-> eps : 0.000001d0;
-> m : 1; n : 2;
-> c_squared : 4d0;
-> v : [n*(n+1) - m*(m+1) + c_squared/2.0, n*(n+1) - m*(m+1) - c_squared/2.0];
-> h1 : 0.1d0; hmin : 0.0d0;
-> x1 : -0.9; x2 : 0.0;
-> dx : 0.0001d0;
-> factr : -3d0; (= -0.5*1.0*(n+m)*(1.0/m)) #
-> delv : [0.001d0*proj(v,1), 0.001d0*proj(v,1)];
-> dydx1(x,y1,y2,y3,y4,y5,y6) := y2;
-> dydx2(x,y1,y2,y3,y4,y5,y6) := (2*x*(m+1)*y2 - (y3-c_squared*x*x)*y1)/(1-x*x);
-> dydx3(x,y1,y2,y3,y4,y5,y6) := 0.0d0;
-> dydx4(x,y1,y2,y3,y4,y5,y6) := y5;
-> dydx5(x,y1,y2,y3,y4,y5,y6) := (2*x*(m+1)*y5 - (y6+c_squared*x*x)*y4)/(1-x*x);
-> dydx6(x,y1,y2,y3,y4,y5,y6) := 0.0d0;
-> derivs : [dydx1,dydx2,dydx3,dydx4,dydx5,dydx6];
-> y3(x,v1,v2) := v1;
-> y2(x,v1,v2) := -(y3(x,v1,v2)-c_squared)*factr/2.0/(m+1);
-> y1(x,v1,v2) := factr +y2(x,v1,v2)*dx;
-> y6(x,v1,v2) := v2;
-> y5(x,v1,v2) := -(y6(x,v1,v2)-c_squared)*factr/2.0/(m+1);
-> y4(x,v1,v2) := factr +y5(x,v1,v2)*dx;
-> load : [y1,y2,y3,y4,y5,y6];
-> n-m = odd, case #
-> f1(x,v1,v2) := y1(x,v1,v2);
-> f2(x,v1,v2) := y4(x,v1,v2);
-> score : [f1,f2];
-> shoot(v,delv,x1,x2,eps,h1,hmin,derivs,load,score)?
[[0.134516614, 4.179144975, 6.000000000, -0.375849789, 3.042623122,
2.006000000], [40004.000006251, 40004.000006252], [-2.999850000, -3.000150000],
[39998.000006251, 40002.000006252]]
```

shootf

Description:

This function solves the two point boundary problem for a system of ODE's by the technique of shooting to a fitting point.

shootf(v1, v2, delv1, delv2, x1, x2, xf, eps, h1, hmin, derivs, load1, load2, score)
⟶ [y, v1, v2, f, dv1, dv2]

Where:

v1 is a real list representing initial values for n_2 parameters from which the initial values for the dependent variables at x_1 may be computed using the function *load*1,
v2 is a real list representing initial values for n_1 parameters from which the initial values for the dependent variables at x_2 may be computed using the function *load*2,
delv1 is a real list being suggested increments to be used in the computation of finite differences by the algorithm at x_1,
delv2 is a real list being suggested increments to be used in the computation of finite differences by the algorithm at x_2,
x1 a real number being a point from which shooting is initiated,
x2 a real number being the other point to which shooting is initiated,
xf a real number being the target point for the shooting algorithm,
eps a real number being the tolerance passed to the Runge-Kutta solver,
h1 a real number being the initial step passed to the Runge-Kutta solver,
hmin a real number being the minimum step size passed to the Runge-Kutta solver,
derivs is a list $[f_1, ..., f_n]$ of the names of Senac functions representing the right hand sides of the system of equations

$$y_i' = f_i(x, y_1, ..., y_n), \quad 1 \le i \le n$$

to be integrated,
load1 is a Senac function of n_2 parameters from which the initial values of the dependent variables at x_1 may be computed,
load2 is a Senac function of n_1 parameters from which the initial values of the dependent variables at x_2 may be computed,
score is a list of *nvar* Senac functions of $nvar + 1$ parameters evaluating the *nvar* functions which should match at xf,
y is a floating point list being the solution values for the dependent variables,
f a floating point list of length *nvar* being the values of the matching functions at xf,
dv1 is a floating point list of length n_2 being the increments in the boundary functions at x_1,
dv2 is a floating point list of length n_1 being the increments in the boundary functions at x_2.

Example:

In this example of `shootf` the equations are described by `[f, g, h]`, the values at the left hand boundary by `[a1, a2, a3]`, the values at the right hand boundary by `[b1, b2, b3]`, and the matching functions by `[s1, s2, s3]`.

```
-> f(x,y1,y2,y3):=y1;
->  g(x,y1,y2,y3):=y1+y2;
->  h(x,y1,y2,y3):=y1+y2+y3;
-> a1(x,v):=1d0+v/100d0;
-> a2(x,v):=2d0+v/100d0;
->  a3(x,v):=v/100d0;
->  b1(x,v,w):=4d0;
->  b1(x,v,w):=1d0-w/200d0;
->  b2(x,v,w):=4d0;
-> b3(x,v,w):=0d0;
->  s1(x,y1,y2,y3):= y1-y2;
->  s2(x,y1,y2,y3):= y2-y3+1/100d0;
->  s3(x,y1,y2,y3):= y3-y1;
->  shootf([1],[1,2],[0.01],[0.01,0.01],0,2,1,0.0001, 0.05, 0.01, [f,g,h],
 [a1,a2,a3], [b1,b2,b3],[s1,s2,s3])?
[[0.364182223, 1.107335236, -1.653609976], [-111.276753538],
[-1007260940.321419600, 455.580393365], [-4.720628870, -4.160845492,
8.881474362], [-112.276753538], [-1007260941.321419600, 453.580393365]]
```

solvde

Description:

This function solves the two point boundary problem for a system of ODE's be the relaxation technique.

solvde(scalv, indexv, nb, y, difeq) $\longrightarrow$ y

Where:

scalv a real list of typical sizes for each dependent variable used to weight errors,
indexv an integer list of the column ordering of variables used to construct the matrix of derivatives used by the algorithm,
nb is the number of boundary conditions at the first mesh point,
y is on input a real matrix representing a guess for the value of each dependent variable at each mesh point and on output an improved estimate.
difeq is a Senac function of the form

$$difeq(k, k1, k2, jsf, is1, isf, indexv, ne, nsi, nsj, y, nyj, nyk) \rightarrow s$$

used by *solvde* to compute the matrix s.

Pre-variables:

solvde_conv [1.0d-4] is a small positive floating point number used in the convergence test,
solvde_slowc [1.0d0] is a number between 0d0 and 1d0 being the fraction of an improvement added at each step,
solvde_nmax [10] the maximum number of equations,
solvde_itmax [200] an upper bound for the number of iterations at each call.

Example:

This artificial example of the use of `solvde` uses a random matrix as the value of the function `difeq`. The non-converging relaxation is interrupted using Ĉ.

```
-> verbose:true;
-> bat("solvde");
-> scalv:[1,1,1];
-> indexv:[1,2,3];
-> nb:3;
-> y:matrix([1,1,0,0],[1,1,1,0],[1,1,1,1]);
-> difeq(k,k1,k2,jsf,is1,isf,indexv,ne,nsi,nsj,y,nyj,nyk):=
        random_matrix(0d0,1d0,nsi,nsj);
-> solvde(scalv,indexv,nb,y,difeq)?
it =   1.00     err =   63.3     fac =   1.581d-2
kmax =
#(4 4 4)
ermax =
#(-271.93861117055167d0 132.24154054483867d0 196.75741152961947d0)
it =   2.00     err =   7.89     fac =  0.127
kmax =
#(3 3 1)
ermax =
#(-16.74109613823354d0 11.756922156256929d0 13.170137327173814d0)
it =   3.00     err =   5.10     fac =  0.196
kmax =
#(3 3 4)
ermax =
#(-13.332452215465137d0 7.763334073280239d0 22.946758290893527d0)
it =   4.00     err =   100.     fac =   9.964d-3
kmax =
#(2 2 2)
ermax =
#(197.08637522186922d0 -283.4240551531272d0 127.04158143232098d0)
it =   5.00     err =   16.9     fac =   5.906d-2
kmax =
#(3 4 4)
ermax =
#(29.475431512397865d0 -27.321191339983187d0 20.617438174037694d0)
it =   6.00     err =   412.     fac =   2.426d-3
kmax =
#(4 4 4)
 ^C
```