Package 'bvpSolve'

September 22, 2015

Version 1.3.2

Title Solvers for Boundary Value Problems of Differential Equations

Author Karline Soetaert <karline.soetaert@nioz.nl>,
Jeff Cash <j.cash@imperial.ac.uk>,
Francesca Mazzia <mazzia@dm.uniba.it>

Maintainer Karline Soetaert <karline.soetaert@nioz.nl>

Depends R (>= 2.01), deSolve

Imports rootSolve, stats, graphics, grDevices

Description Functions that solve boundary value problems ('BVP') of systems of ordinary differential equations ('ODE') and differential algebraic equations ('DAE').
The functions provide an interface to the FORTRAN functions 'twpbvpC', 'colnew/colsys', and an R-implementation of the shooting method.

License GPL (>= 2)

LazyData yes

Repository CRAN

Repository/R-Forge/Project bvpsolve

Repository/R-Forge/Revision 237

Repository/R-Forge/DateTimeStamp 2015-09-11 14:28:19

Date/Publication 2015-09-22 13:00:34

NeedsCompilation yes

R topics documented:

bvpSolve-package *Solvers for Boundary Value Problems (BVP) of Ordinary Differential Equations*

Description

Functions that solve boundary value problems of a system of ordinary differential equations (ODE)

The functions provide an interface to (1) the FORTRAN code twpbvpC written by J.R. Cash, F. Mazzia and M.H. Wright,

(2) to the FORTRAN codes colnew and colsys by respectively Bader and Ascher and Ascher, Christiansen and Russell,

and (3) also implement a shooting method.

Details

Package:	bvpSolve
Type:	Package
License:	GNU Public License 2 or above

The system of ODE's can be written as an R function, or in compiled code (FORTRAN, C), similar as the initial value problems that are solved by integration routines from package deSolve.

A large number of examples have been implemented to show the functionalities of the package.

- All test problems from the website of J.R. Cash http://www.ma.ic.ac.uk/~jcash/BVP_software/PROBLEMS.PDF are implemented in package vignette "bvpTests"
- Other test problems, in R code are in the packages doc/example subdirectory.
- Test problems implemented in compiled code can be found in the packages doc/dynload subdirectory.
- Still more examples, both in R and compiled code are in the package vignette "bvpSolve".

Author(s)

Karline Soetaert (Maintainer)

Jeff Cash

Francesca Mazzia

References

Francesca Mazzia, Jeff R. Cash, Karline Soetaert (2014). Solving boundary value problems in the open source software R: Package bvpSolve. Opuscula mathematica, 34(2), 387–403. URL http://dx.doi.org/10.7494/OpMath.2014.34.2.387

J.R. Cash and M.H. Wright, (1991) A deferred correction method for nonlinear two-point boundary value problems: implementation and numerical evaluation, SIAM J. Sci. Stat. Comput. 12, 971-989.

Cash, J. R. and F. Mazzia, (2005) A new mesh selection algorithm, based on conditioning, for two-point boundary value codes. J. Comput. Appl. Math. 184 no. 2, 362–381.

G. Bader and U. Ascher, (1987) a new basis implementation for a mixed order boundary value ode solver, siam j. scient. stat. comput. 8, 487-483.

U. Ascher, J. Christiansen and R. D. Russell, (1981) collocation software for boundary-value odes, acm trans. math software 7, 209-222.

See Also

bvptwp, a deferred correction method based on mono- implicit Runge-Kutta formulas and adaptive mesh refinement, based on conditioning to solve two-point boundary value problems (Cash and Mazzia, 2005).

bvpcol, a collocation method to solve multi-point boundary value problems of ordinary differential equations. (Ascher et al., 1981).

bvpshoot, a shooting method, using solvers from packages deSolve and rootSolve.

diagnostics.bvpSolve, for a description of diagnostic messages.

plot.bvpSolve, for a description of plotting the output of the BVP solvers.

Examples

```
## Not run:
## show examples (see respective help pages for details)
example(bvptwp)
example(bvpshoot)
example(bvpcol)

## open the directory with R- examples
browseURL(paste(system.file(package = "bvpSolve"), "doc/examples", sep = ""))

## open the directory with examples in compiled code
browseURL(paste(system.file(package = "bvpSolve"), "doc/dynload", sep = ""))

## show package vignette with how to use bvpSolve
## + source code of the vignette
vignette("bvpSolve")
edit(vignette("bvpSolve"))

## package vignette with the test problems from J.Cash
## + source code of the vignette
vignette("bvpTests")
edit(vignette("bvpTests"))


## show directory with source code of the vignettes
browseURL(paste(system.file(package = "bvpSolve"), "/doc", sep = ""))
```

```
## End(Not run)
```

---

bvpcol *Solves multipoint boundary value problems of ordinary differential equations or differential algebraic equations, using a collocation method.*

Description

Solves Boundary Value Problems For Ordinary Differential Equations (ODE) or semi-explicit Differential-Algebraic Equations (DAE) with index at most 2.

It is possible to solve stiff ODE systems, by using an automatic continuation strategy

This is an implementation of the fortran codes colsys.f, colnew.f and coldae.f written by respectively U. Ascher, J. christiansen and R.D. Russell (colsys), U. Ascher and G. Bader (colnew) and U. Ascher and C. Spiteri.

The continuation strategy is an implementation of the fortran code colmod written by J.R. Cash, M.H. Wright and F. Mazzia.

Usage

```
bvpcol (yini = NULL, x, func, yend = NULL, parms = NULL,
        order = NULL, ynames = NULL, xguess = NULL, yguess = NULL,
        jacfunc = NULL, bound = NULL, jacbound = NULL,
        leftbc = NULL, posbound = NULL, islin = FALSE, nmax = 1000,
        ncomp = NULL, atol = 1e-8, colp = NULL, bspline = FALSE,
        fullOut = TRUE, dllname = NULL, initfunc = dllname,
        rpar = NULL, ipar = NULL, nout = 0, outnames = NULL,
        forcings = NULL, initforc = NULL, fcontrol = NULL,
        verbose = FALSE, epsini = NULL, eps = epsini, dae = NULL, ...)
```

Arguments

yini	either a vector with the initial (state) variable values for the ODE system, or NULL. If yini is a vector, use NA for an initial value which is not specified. If yini has a names attribute, the names will be available within func and used to label the output matrix. If yini = NULL, then the boundary conditions must be specified via function bound; if not NULL then yend should also be not NULL.
x	sequence of the independent variable for which output is wanted; the first value of x must be the initial value (at which yini is defined), the final value the end condition (at which yend is defined).
func	either an R-function that computes the values of the derivatives in the ODE system (the model definition) at point x, or a character string giving the name of a compiled function in a dynamically loaded shared library.

If func is an R-function, it must be defined as: func = function(x, y, parms,...). x is the current point of the independent variable in the integration, y is the current estimate of the (state) variables in the ODE system. If the initial values yini has a names attribute, the names will be available inside func. parms is a vector or list of parameters; ... (optional) are any other arguments passed to the function.

The return value of func should be a list, whose first element is a vector containing the values of the equations. In case where the equations are first-order, this will be the derivatives of y with respect to x. After this can come global values that are required at each point in x.

If the problem is a DAE, then the algebraic equations should be the last.

If func is a string, then dllname must give the name of the shared library (without extension) which must be loaded before bvptwp is called. See package vignette "bvpSolve" for more details.

yend — either a vector with the final (state) variable values for the ODE system, or NULL;

if yend is a vector, use NA for a final value which is not specified.

If yend has a names attribute, and yini does not, the names will be available within the functions and used to label the output matrix.

If yend = NULL, then the boundary conditions must be specified via function bound; if not NULL then yini should also be not NULL.

parms — vector or a list with parameters passed to func, jacfunc, bound and jacbound (if present).

If eps is given a value then it should be the **first** element in parms.

epsini — the initial value of the continuation parameter. If NULL and eps is given a value, then epsini takes the default starting value of 0.5. For many singular perturbation type problems, the choice of 0.1 < eps < 1 represents a (fairly) easy problem. The user should attempt to specify an initial problem that is not 'too' challenging. epsini must be initialised strictly less than 1 and greater than 0.

eps — the desired value of precision for which the user would like to solve the problem. eps must be less than or equal to epsini. If this is given a value, it must be the first value in parms.

ynames — The names of the variables; used to label the output, and avaliable within the functions.

If ynames is NULL, names can also be passed via yini, yend or yguess.

xguess — Initial grid x, a vector. If xguess is given, so should yguess be.

Supplying xguess and yguess, based on results from a previous (simpler) BVP-ODE can be used for model continuation, see example 2 of bvptwp.

yguess — First guess values of y, corresponding to initial grid xguess; a matrix with number of rows equal to the number of variables, and whose number of columns equals the length of xguess.

if the rows of yguess have a names attribute, the names will be available within the functions and used to label the output matrix.

It is also allowed to pass the output of a previous run for continuation. This will use the information that is stored in the attributes istate and rstate. It

will only work when for the previous run, fullOut was set equal to TRUE (the default). In this case, xguess need not be provided.

See example 3b.

jacfunc — jacobian (optional) - either an R-function that evaluates the jacobian of func at point x, or a string with the name of a function or subroutine in dllname that computes the Jacobian (see vignette "bvpSolve" for more about this option).

If jacfunc is an R-function, it must be defined as: jacfunc = function(x, y, parms,...). It should return the partial derivatives of func with respect to y, i.e. df(i,j) = dfi/dyj. See last example.

If jacfunc is NULL, then a numerical approximation using differences is used. This is the default.

bound — boundary function (optional) - only if yini and yend are not available. Either an R function that evaluates the i-th boundary element at point x, or a string with the name of a function or subroutine in dllname that computes the boundaries (see vignette "bvpSolve" for more about this option).

If bound is an R-function, it should be defined as: bound = function(i, y, parms, ...). It should return the i-th boundary condition. See last example.

jacbound — jacobian of the boundary function (optional) - only if bound is defined. Either an R function that evaluates the gradient of the i-th boundary element with respect to the state variables, at point x, or a string with the name of a function or subroutine in dllname that computes the boundary jacobian (see vignette "bvpSolve" for more about this option).

If jacbound is an R-function, it should be defined as: jacbound = function(i, y, parms, ...). It should return the gradient of the i-th boundary condition. See examples.

If jacbound is NULL, then a numerical approximation using differences is used. This is the default.

leftbc — only if yini and yend are not available and posbound is not specified: the number of left boundary conditions.

posbound — only used if bound is given: a vector with the position (in the mesh) of the boundary conditions - its values should be sorted - and it should be within the range of x; (posbound corresponds to fortran input "Zeta" in the colnew/colsys FORTRAN codes.) See last example. Note that two-point boundary value problems can also be specified via leftbc (which is simpler).

islin — set to TRUE if the problem is linear - this will speed up the simulation.

nmax — maximal number of subintervals during the calculation.

order — the order of each derivative in func. The default is that all derivatives are 1-st order, in which case order can be set = NULL.

For higher-order derivatives, specifying the order can improve computational efficiency, but this interface is more complex.

If order is not NULL, the number of equations in func must equal the length of order; the summed values of order must equal the number of variables (ncomp). The jacobian as specified in jacfunc must have number of rows = number of equations and number of columns = number of variables. bound and jacbound remain defined in the number of variables. See example 3 and 3b.

ncomp — used if the model is specified by compiled code, the number of components (or equations). See package vignette "bvpSolve".
Also to be used if the boundary conditions are specified by bound, and there is no yguess

atol — error tolerance, a scalar.

colp — number of collocation points per subinterval.

bspline — if FALSE, then code colnew is used the default, if TRUE, then fortran code colsys is used. Code colnew incorporates a new basis representation, while colsys uses b-splines.

fullOut — if set to TRUE, then the collocation output required e.g. for continuation will be returned in attributes rwork and iwork. Use attributes(out)$rwork, attributes(out)$rwork to see their contents

dllname — a string giving the name of the shared library (without extension) that contains all the compiled function or subroutine definitions referred to in func, jacfunc, bound and jacbound. Note that ALL these subroutines must be defined in the shared library; it is not allowed to merge R-functions with compiled functions.
See package vignette "bvpSolve" or deSolve's package vignette "compiledCode".

initfunc — if not NULL, the name of the initialisation function (which initialises values of parameters), as provided in 'dllname'. See package vignette "bvpSolve".

rpar — only when 'dllname' is specified: a vector with double precision values passed to the dll-functions whose names are specified by func and jacfunc.

ipar — only when 'dllname' is specified: a vector with integer values passed to the dll-functions whose names are specified by func and jacfunc.

nout — only used if dllname is specified and the model is defined in compiled code: the number of output variables calculated in the compiled function func, present in the shared library. Note: it is not automatically checked whether this is indeed the number of output variables calculated in the dll - you have to perform this check in the code. See deSolve's package vignette "compiledCode".

outnames — only used if function is specified in compiled code and nout > 0: the names of output variables calculated in the compiled function. These names will be used to label the output matrix. The length of outnames should be = nout.

forcings — only used if 'dllname' is specified: a list with the forcing function data sets, each present as a two-columned matrix, with (time,value); interpolation outside the interval [min(times), max(times)] is done by taking the value at the closest data extreme. This feature is included for consistency with the initial value problem solvers from package deSolve.
See package vignette "compiledCode" from package deSolve.

initforc — if not NULL, the name of the forcing function initialisation function, as provided in 'dllname'. It MUST be present if forcings has been given a value.
See package vignette "compiledCode" from package deSolve.

fcontrol — A list of control parameters for the forcing functions.
See package vignette "compiledCode" from package deSolve.

verbose — when TRUE, writes messages (warnings) to the screen.

dae	if the problem is a DAE, should be a list containing the index of the problem and the number of algebraic equations nalg. See example 5
...	additional arguments passed to the model functions.

Details

If eps does not have a value and dae = NULL, then the method is based on an implementation of the Collocation methods called "colnew" and "colsys" to solve multi-point boundary value problems of ordinary differential equations.

The ODEs and boundary conditions are made available through the user-provided routines, func and vectors yini and yend or (optionally) bound. bvpcol can also solve multipoint boundary value problems (see one but last example).

The corresponding partial derivatives are optionally available through the user-provided routines, jacfunc and jacbound. Default is that they are automatically generated by R, using numerical differences.

The user-requested tolerance is provided through atol.

If the function terminates because the maximum number of subintervals was exceeded, then it is recommended that 'the program be run again with a larger value for this maximum.'

If eps does have a value, then the method is based on an implementation of the Collocation methods called "colmod". The type of problems which this is designed to solve typically involve a small positive parameter $0 < eps \ll 1$. As eps becomes progressively smaller, the problem normally becomes increasingly difficult to approximate numerically (for example, due to the appearance of narrow transition layers in the profile of the analytic solution).

The idea of continuation is to solve a chain of problems in which the parameter eps decreases monotonically towards some desired value. That is, a sequence of problems is attempted to be solved:

epsini > eps1 > eps2 > eps3 > ..... > eps > 0

where epsini is a user provided starting value and eps is a user desired final value for the parameter.

If dae is not NULL, then it is assumed that a DAE has to be solved. In that case, dae should contain give the index of the DAE and the number of algebraic equations (nalg).

(this part comes from the comments in the code coldae). With respect to the dae, it should be noted that the code does not explicitly check the index of the problem, so if the index is > 2 then the code will not work well. The number of boundary conditions required is independent of the index. it is the user's responsibility to ensure that these conditions are consistent with the constraints. The conditions at the left end point must include a subset equivalent to specifying the index-2 constraints there. For an index-2 problem in hessenberg form, the projected collocation method of Ascher and Petzold [2] is used.

Value

A matrix of class bvpSolve, with up to as many rows as elements in x and as many columns as elements in yini plus the number of "global" values returned in the second element of the return from func, plus an additional column (the first) for the x-value.

There will be one row for each element in x unless the solver returns with an unrecoverable error.

If ynames is given, or yini, yend has a names attribute, or yguess has named rows, the names will be used to label the columns of the output value.

The output will also have attributes istate and rstate which contain the collocation output required e.g. for continuation of a problem, unless fullOutput is FALSE

Note

colnew.f (Bader and Ascher, 1987), is a modification of the code colsys.f (Ascher, Christiansen and Russell, 1981), which incorporates a new basis representation replacing b-splines, and improvements for the linear and nonlinear algebraic equation solvers. To toggle on/off colsys, set bspline = TRUE/FALSE

colmod is a revised version of the package colnew by Bader and Ascher (1987), which in turn is a modification of the package colsys by Ascher, Christiansen and Russell (1981). Colmod has been adapted to allow an automatic continuation strategy to be used (Cash et al., 1995).

The mesh selection algorithm used in colmod differs from that used in colnew

Author(s)

Karline Soetaert <karline.soetaert@nioz.nl>

References

U. Ascher, J. Christiansen and R. D. Russell, (1981) collocation software for boundary-value odes, acm trans. math software 7, 209-222.

G. Bader and U. Ascher, (1987) a new basis implementation for a mixed order boundary value ode solver, siam j. scient. stat. comput. 8, 487-483.

U. Ascher, J. Christiansen and R.D. Russell, (1979) a collocation solver for mixed order systems of boundary value problems, math. comp. 33, 659-679.

U. Ascher, J. Christiansen and R.D. Russell, (1979) colsys - a collocation code for boundary value problems, lecture notes comp.sc. 76, springer verlag, B. Childs et. al. (eds.), 164-185.

J. R. Cash, G. Moore and R. W. Wright, (1995) an automatic continuation strategy for the solution of singularly perturbed linear two-point boundary value problems, j. comp. phys. 122, 266-279.

U. Ascher and R. Spiteri, 1994. collocation software for boundary value differential-algebraic equations, siam j. scient. stat. comput. 15, 938-952.

U. Ascher and L. Petzold, 1991. projected implicit runge-kutta methods for differential- algebraic equations, siam j. num. anal. 28 (1991), 1097-1120.

See Also

bvpshoot for the shooting method

bvptwp for a MIRK formula

diagnostics.bvpSolve, for a description of diagnostic messages

approx.bvpSolve, for approximating solution in new values

plot.bvpSolve, for a description of plotting the output of the BVP solvers.

Examples

```
## ================================================================================
## Example 1: simple standard problem
## solve the BVP ODE:
## d2y/dt^2=-3py/(p+t^2)^2
## y(t= -0.1)=-0.1/sqrt(p+0.01)
## y(t=  0.1)= 0.1/sqrt(p+0.01)
## where p = 1e-5
##
## analytical solution y(t) = t/sqrt(p + t^2).
##
## The problem is rewritten as a system of 2 ODEs:
## dy=y2
## dy2=-3p*y/(p+t^2)^2
## ================================================================================

#--------------------------------
# Derivative function
#--------------------------------
fun <- function(t, y, pars) {
  dy1 <- y[2]
  dy2 <- - 3 * p * y[1] / (p+t*t)^2
  return(list(c(dy1,
                dy2))) }

# parameter value
p    <- 1e-5

# initial and final condition; second conditions unknown
init <- c(-0.1 / sqrt(p+0.01), NA)
end  <- c( 0.1 / sqrt(p+0.01), NA)

# Solve bvp
sol  <- bvpcol(yini = init, yend = end,
               x = seq(-0.1, 0.1, by = 0.001), func = fun)
plot(sol, which = 1)

# add analytical solution
curve(x/sqrt(p+x*x), add = TRUE, type = "p")

diagnostics(sol)

zoom <- approx(sol, xout = seq(-0.005, 0.005, by  = 0.0001))
plot(zoom, which = 1, main = "zoom in on [-0.0005,0.0005]")


## ================================================================================
## Example 1b:
## Same problem, now solved as a second-order equation
## and with different value of "p".
## ================================================================================
```

```
fun2 <- function(t, y, pars)
{ dy <- - 3 * p * y[1] / (p+t*t)^2
  list(dy)
}

p <- 1e-4
sol2  <- bvpcol(yini = init, yend = end, order = 2,
                x = seq(-0.1, 0.1, by = 0.001), func = fun2)

# plot both runs at once:
plot(sol, sol2, which = 1)

## =============================================================================
## Example 1c: simple
## solve d2y/dx2 + 1/x*dy/dx + (1-1/(4x^2)y = sqrt(x)*cos(x),
## on the interval [1,6] and with boundary conditions:
## y(1)=1, y(6)=-0.5
##
## Write as set of 2 odes
## dy/dx = y2
## dy2/dx  = - 1/x*dy/dx - (1-1/(4x^2)y + sqrt(x)*cos(x)
## =============================================================================

f2 <- function(x, y, parms)
{
 dy  <- y[2]
 dy2 <- -1/x * y[2]- (1-1/(4*x^2))*y[1] + sqrt(x)*cos(x)
 list(c(dy, dy2))
}

x    <- seq(1, 6, 0.1)
sol  <- bvpcol(yini = c(1, NA), yend = c(-0.5, NA), bspline = TRUE,
               x = x, func = f2)
plot(sol, which = 1)

# add the analytic solution
curve(0.0588713*cos(x)/sqrt(x) + 1/4*sqrt(x)*cos(x)+0.740071*sin(x)/sqrt(x)+
      1/4*x^(3/2)*sin(x), add = TRUE, type = "l")


## =============================================================================
## Example 2. Uses continuation
## Test problem 24
## =============================================================================

Prob24<- function(t, y, ks) {     #eps is called ks here
  A <- 1+t*t
  AA <- 2*t
  ga <- 1.4
  list(c(y[2],(((1+ga)/2 -ks*AA)*y[1]*y[2]-y[2]/y[1]-
               (AA/A)*(1-(ga-1)*y[1]^2/2))/(ks*A*y[1])))
}
```

```
ini <- c(0.9129, NA)
end <- c(0.375, NA)
xguess <- c(0, 1)
yguess <- matrix(nrow = 2, ncol = 2, 0.9 )

# bvpcol works with eps NOT too small, and good initial condition ...
sol <- bvpcol(yini = ini, yend = end, x = seq(0, 1, by = 0.01),
              xguess = xguess, yguess = yguess,
              parms = 0.1, func = Prob24, verbose = FALSE)

# when continuation is used: does not need a good initial condition
sol2 <- bvpcol(yini = ini, yend = end, x = seq(0, 1, by = 0.01),
                  parms = 0.05, func = Prob24,
                  eps = 0.05)

#zoom <- approx(sol2, xout = seq(0.01, 0.02, by  = 0.0001))
#plot(zoom, which = 1, main = "zoom in on [0.01, 0.02]")

sol3 <- bvpcol(yini = ini, yend = end, x = seq(0, 1, by = 0.01),
                  parms = 0.01, func = Prob24 , eps = 0.01)

sol4 <- bvpcol(yini = ini, yend = end, x = seq(0, 1, by = 0.01),
                  parms = 0.001, func = Prob24, eps = 0.001)

# This takes a long time
## Not run:
print(system.time(
sol5 <- bvpcol(yini = ini, yend = end, x = seq(0, 1, by = 0.01),
                  parms = 1e-4, func = Prob24, eps = 1e-4)
))

## End(Not run)

plot(sol, sol2, sol3, sol4, which = 1, main = "test problem 24",
     lwd = 2)

legend("topright", col = 1:4, lty = 1:4, lwd = 2,
       legend = c("0.1", "0.05", "0.01", "0.001"), title = "eps")

## ================================================================================
## Example 3  - solved with specification of boundary, and jacobians
## d4y/dx4 =R(dy/dx*d2y/dx2 -y*dy3/dx3)
## y(0)=y'(0)=0
## y(1)=1, y'(1)=0
##
## dy/dx  = y2
## dy2/dx = y3    (=d2y/dx2)
## dy3/dx = y4    (=d3y/dx3)
## dy4/dx = R*(y2*y3 -y*y4)
## ================================================================================

# derivative function: 4 first-order derivatives
f1st<- function(x, y, S) {
```

```
  list(c(y[2],
         y[3],
         y[4],
         1/S*(y[2]*y[3] - y[1]*y[4]) ))
}

# jacobian of derivative function
df1st <- function(x, y, S) {
 matrix(nrow = 4, ncol = 4, byrow = TRUE, data = c(
             0,         1,      0,       0,
             0,         0,      1,       0,
             0,         0,      0,       1,
             -1*y[4]/S, y[3]/S, y[2]/S, -y[1]/S))
}

# boundary
g2 <- function(i, y, S)  {
  if (i == 1) return (y[1])
  if (i == 2) return (y[2])
  if (i == 3) return (y[1] - 1)
  if (i == 4) return (y[2])
}

# jacobian of boundary
dg2 <- function(i, y, S)  {
  if (i == 1) return(c(1, 0, 0, 0))
  if (i == 2) return(c(0, 1, 0, 0))
  if (i == 3) return(c(1, 0, 0, 0))
  if (i == 4) return(c(0, 1, 0, 0))
}

# we use posbound to specify the position of boundary conditions
# we can also use leftbc = 2 rather than posbound = c(0,0,1,1)
S    <- 1/100
sol  <- bvpcol(x = seq(0, 1, by = 0.01),
          ynames = c("y", "dy", "d2y", "d3y"),
          posbound = c(0, 0, 1, 1), func = f1st, parms = S, eps = S,
          bound = g2, jacfunc = df1st, jacbound = dg2)

plot(sol)

## ================================================================================
## Example 3b - solved with specification of boundary, and jacobians
## and as a higher-order derivative
## d4y/dx4 =R(dy/dx*d2y/dx2 -y*dy3/dx3)
## y(0)=y'(0)=0
## y(1)=1, y'(1)=0
## ================================================================================

# derivative function: one fourth-order derivative
f4th <- function(x, y, S) {
  list(1/S * (y[2]*y[3] - y[1]*y[4]))
}
```

```
# jacobian of derivative function
df4th <- function(x, y, S)  {
  matrix(nrow = 1, ncol = 4, byrow = TRUE, data = c(
             -1*y[4]/S, y[3]/S, y[2]/S, -y[1]/S))
}

# boundary function - same as previous example

# jacobian of boundary - same as previous

# order = 4 specifies the equation to be 4th order
# solve with bspline false
S    <- 1/100
sol  <- bvpcol (x = seq(0, 1, by = 0.01),
          ynames = c("y", "dy", "d2y", "d3y"),
          posbound = c(0, 0, 1, 1), func = f4th, order = 4,
          parms = S, eps = S, bound = g2, jacfunc = df4th,
          jacbound = dg2 )

plot(sol)

# Use (manual) continuation to find solution of a more difficult example
# Previous solution collocation from sol passed ("guess = sol")

sol2  <- bvpcol(x = seq(0, 1, by = 0.01),
          ynames = c("y", "dy", "d2y", "d3y"),
          posbound = c(0, 0, 1, 1), func = f4th,
          parms = 1e-6, order = 4, eps = 1e-6,
          bound = g2, jacfunc = df4th, jacbound = dg2 )

# plot both at same time
plot(sol, sol2, lwd = 2)

legend("bottomright", leg = c(100, 10000), title = "R = ",
         col = 1:2, lty = 1:2, lwd = 2)


## =============================================================================
## Example 4  - a multipoint bvp
## dy1 = (y2 - 1)/2
## dy2 = (y1*y2 - x)/mu
## over interval [0,1]
## y1(1) = 0; y2(0.5) = 1
## =============================================================================

multip <- function (x, y, p) {
  list(c((y[2] - 1)/2,
         (y[1]*y[2] - x)/mu))
}

bound <- function (i, y, p) {
  if (i == 1) y[2] -1    # at x=0.5: y2=1
```

```
  else y[1]              # at x=  1: y1=0
}

mu  <- 0.1
sol <- bvpcol(func = multip, bound = bound,
              x = seq(0, 1, 0.01), posbound = c(0.5, 1))

plot(sol)

# check boundary value
sol[sol[,1] == 0.5,]


## =============================================================================
## Example 5 - a bvp DAE
## =============================================================================

bvpdae <- function(t, x, ks, ...) {
  p1  <- p2 <- sin(t)
  dp1 <- dp2 <- cos(t)

  dx1 <- (ks + x[2] - p2)*x[4] + dp1
  dx2 <- dp2
  dx3 <- x[4]
  res <- (x[1] - p1)*(x[4] - exp(t))

  list(c(dx1, dx2, dx3, res), res = res)
}

boundfun <- function(i,  x, par, ...) {
  if (i == 1) return(x[1] - sin(0))
  if (i == 2) return(x[3] - 1)
  if (i == 3) return(x[2] - sin(1))
  if (i == 4) return((x[1] - sin(1))*(x[4] - exp(1)))  # Not used here..
}

x <- seq(0, 1, by = 0.01)
mass <- diag(nrow = 4)  ; mass[4, 4] <- 0

# solved using boundfun
out <- bvpcol (func = bvpdae, bound = boundfun, x = x,
               parms = 1e-4, ncomp = 4, leftbc = 2,
               dae = list(index = 2,  nalg = 1))

# solved using yini, yend
out1 <- bvpcol (func = bvpdae, x = x, parms = 1e-4,
                yini = c(sin(0), NA, 1, NA),
                yend = c(NA, sin(1), NA, NA),
                dae = list(index = 2,  nalg = 1))

# the analytic solution
ana <- cbind(x, "1" = sin(x), "2" = sin(x), "3" = 1, "4" = 0, res = 0)
plot(out, out1, obs = ana)
```

bvpshoot — *Solver for two-point boundary value problems of ordinary differential equations, using the single shooting method*

Description

Solves a boundary value problem of a system of ordinary differential equations using the single shooting method. This combines the integration routines from package deSolve with root-finding methods from package rootSolve.

Preferentially bvptwp or bvpcol should be used rather than bvpshoot, as they give more precise output.

Usage

```
bvpshoot(yini = NULL, x, func, yend = NULL, parms = NULL,
         order = NULL, guess = NULL,
         jacfunc = NULL, bound = NULL, jacbound = NULL,
         leftbc = NULL, posbound = NULL, ncomp = NULL,
         atol = 1e-8, rtol = 1e-8, extra = NULL,
         maxiter = 100, positive = FALSE, method = "lsoda",...)
```

Arguments

yini	either a *vector* with the initial (state) variable values for the ODE system, or a *function* that calculates the initial condition, or NULL. If yini is a function, it should be defined as: yini <- function(y, parms,...); where y are the initial values, and parms the parameters. if yini is a vector then use NA for an initial value which is not available. If yini has a names attribute, the names will be available within the functions and used to label the output matrix. if yini = NULL then bound should be specified; if not NULL then yend should also be not NULL
x	sequence of the independent variable for which output is wanted; the first value of x must be the initial value (at which yini is defined), the final value the end condition (at which yend is defined).
func	an R-function that computes the values of the derivatives in the ODE system (the model definition) at x. func must be defined as: func = function(x, y, parms, ...). x is the current point of the independent variable in the integration, y is the current estimate of the (state) variables in the ODE system. If the initial values yini or yend has a names attribute, the names will be available inside func. parms is a vector or list of parameters; ... (optional) are any other arguments passed to the function.

The return value of func should be a list, whose first element is a vector containing the derivatives of y with respect to x, and whose next elements are global values that are required at each point in x.

Note that it is not possible to use bvpshoot with functions defined in compiled code. Use bvptwp instead.

yend — either a vector with the final (state) variable values for the ODE system, a *function* that calculates the final condition or NULL;

if yend is a vector use NA for a final value which is not available.

If yend is a function, it should be defined as: `yend <- function (y, yini, parms, ...)`; where y are the final values, yini the initial values and parms the parameters.

If yend has a names attribute, and yini does not, the names will be available within the functions and used to label the output matrix.

if `yend = NULL` then bound should be specified; if not NULL then yini should also be not NULL.

parms — vector or a list with parameters passed to func, jacfunc, bound and jacbound (if present).

order — the order of each derivative in func. The default is that all derivatives are 1-st order, in which case order can be set = NULL.

If order is not NULL, the number of equations in func must equal the length of order; the summed values of order must equal the number of variables (ncomp). The jacobian as specified in jacfunc must have number of rows = number of equations and number of columns = number of variables. bound and jacbound remain defined in the number of variables. See examples.

guess — guess for the value(s) of the unknown *initial* conditions;

if initial and final conditions are specified by yini and yend, then guess should contain one value for each NA in yini. The length of guess should thus equal the number of *unknown initial conditions* (=NAs in yini). If guess is not provided, a value = 0 is assumed for each NA in yini and a warning printed.

If initial and final conditions are specified by the boundary function bound, then guess should contain the initial guess for *all initial conditions*, i.e. its length should equal the number of state variables in the ODE system; if in this case guess has a names attribute, the names will be available within the functions and used to label the output matrix. If guess is not provided, then ncomp should specify the total number of variables, a value = 0 will be assumed for the initial conditions and a warning printed.

jacfunc — jacobian (optional) - an R-function that evaluates the jacobian of func at point x.

`jacfunc = function(x, y, parms,...)`. It should return the partial derivatives of func with respect to y, i.e. df(i,j) = dfi/dyj.

If jacfunc is NULL, then a numerical approximation using differences is used. This is the default.

jacfunc is passed to the initial value problem solver.

bound — boundary function (optional) - only if yini and yend are not available. An R function that evaluates the i-th boundary element at point x.

bound should be defined as: `bound = function(i, y, parms, ...)`. It should return the i-th boundary condition.

if not `NULL`, bound defines the root to be solved by the root solving algorithm.

`jacbound` jacobian of the boundary function (optional) - only if bound is defined. An R function that evaluates the gradient of the i-th boundary element with respect to the state variables, at point `x`.

`jacbound` should be defined as: `jacbound = function(i, y, parms, ...)`. It should return the gradient of the i-th boundary condition. See last example.

`jacbound` is passed to the root solver.

`leftbc` only if `yini` and `yend` are not available: the number of left boundary conditions.

`posbound` only used if bound is given: a vector with the position (in the mesh) of the boundary conditions - only points that are in `x` are allowed. Note that, if the boundary conditions are at the ends of the integration interval, it is simpler to use `leftbc`.

`ncomp` only used if the boundaries are specified via the boundary function bound and `guess` is not specified. The number of components.

`atol` absolute error tolerance, either a scalar or a vector, one value for each unknown element - passed to function multiroot - see help of this function.

`rtol` relative error tolerance, either a scalar or a vector, one value for each unknown element - passed to function multiroot - see help of this function.

`extra` if too many boundary conditions are given, then it is assumed that an extra parameter has to be estimated.

`extra` should contain the initial guess of this extra parameter.

`maxiter` the maximal number of iterations allowed in the root solver.

`positive` set to `TRUE` if dependent variables (y) have to be positive numbers.

`method` the integration method used, one of ("lsoda", "lsode", "lsodes", "vode", "euler", "rk4", "ode23" or "ode45").

`...` additional arguments passed to the integrator and (possibly) the model functions.

Details

This is a simple implementation of the shooting method to solve boundary value problems of ordinary differential equations.

A boundary value problem does not have all initial values of the state variable specified. Rather some conditions are specified at the end of the integration interval.

The shooting method, is a root-solving method. There are two strategies:

yini and yend specified If initial and end conditions are specified with `yini` and `yend` then the (unspecified) initial conditions are the unknown values to be solved for; the function value whose root has to be found are the deviations from the specified conditions at the end of the integration interval.

Thus, starting with an initial guess of the initial conditions (as provided in `guess`), the ODE model is solved as an initial value problem, and after termination, the discrepancy of the modeled final conditions with the known final condition is assessed (the cost function). The root of this cost function is to be found.

bound specified If initial and end conditions are specified with bound, then the unknowns are all initial conditions; the function whose root is to be found is given by bound.

Starting from a guess of the initial values, one of the integrators from package deSolve (as specified with method) is used to solve the resulting initial value problem.

Function multiroot from package rootSolve is used to retrieve the root.

For this method to work, the model should be even determined, i.e. the number of equations should equal the number of unknowns.

bvpshoot distinguishes two cases:

1. the total number of specified boundary conditions (on both the start and end of the integration interval) equals the number of boundary value problem equations (or the number of dependent variables y).

2. The number of boundary conditions specified *exceeds* the number of equations. In this case, extra parameters have to be solved for to make the model even determined.

See example nr 4.

Value

A matrix with up to as many rows as elements in x and as many columns as the number of state variables in the ODE system plus the number of "global" values returned in the next elements of the return from func, plus an additional column (the first) for the x-value.

There will be one row for each element in x unless the solver returns with an unrecoverable error.

If yini has a names attribute, it will be used to label the columns of the output value. If yini is not named, the solver will try to find the names in yend. If the boundaries are specified by bound then the names from guess will be used.

The output will have the attribute roots, which returns the value(s) of the root(s) solved for (root), the function value (f.root), and the number of iterations (iter) required to find the root.

Note

When order is not NULL, then it should contain the order of all *equations* in func. If the order of some equations > 1, then there will be less equations than variables. The number of equations should be equal to the length of order, while the number of variables will be the sum of order.

For instance, if order = c(1,2,3,4), then the first equation will be of order 1, the second one of order 2, ...and the last of order 4.

There will be 1+2+3+4 = 10 variables. For instance, if the derivative function defines (A', B", C"', D"") respectively, then the variable vector will contain values for A, B, B', C, C', C", D, D', D", D"'; in that order. This is also the order in which the initial and end conditions of all variables must be specified.

Do not specify the jacobian if the maximal order>1.

Author(s)

Karline Soetaert <karline.soetaert@nioz.nl>

See Also

bvptwp for the MIRK method

lsoda, lsode, lsodes, vode,

rk, rkMethod for details about the integration method

multiroot, the root-solving method used

diagnostics.bvpSolve, for a description of diagnostic messages

plot.bvpSolve, for a description of plotting the output of the BVP solvers.

Examples

```
## ================================================================================
## Example 1: simple standard problem
## solve the BVP ODE:
## d2y/dt^2=-3py/(p+t^2)^2
## y(t= -0.1)=-0.1/sqrt(p+0.01)
## y(t=  0.1)= 0.1/sqrt(p+0.01)
## where p = 1e-5
##
## analytical solution y(t) = t/sqrt(p + t^2).
##
## The problem is rewritten as a system of 2 ODEs:
## dy=y2
## dy2=-3p*y/(p+t^2)^2
## ================================================================================

#--------------------------------
# Derivative function
#--------------------------------
fun <- function(t, y, pars)
{ dy1 <- y[2]
  dy2 <- - 3*p*y[1] / (p+t*t)^2
  return(list(c(dy1,
                dy2))) }


# parameter value
p    <- 1e-5

# initial and final condition; second conditions unknown
init <- c(y = -0.1 / sqrt(p+0.01), dy = NA)
end  <- c(     0.1 / sqrt(p+0.01), NA)

# Solve bvp
sol  <- bvpshoot(yini = init, x = seq(-0.1, 0.1, by = 0.001),
          func = fun, yend = end, guess = 1)

plot(sol, which = "y", type = "l")

# add analytical solution
curve(x/sqrt(p+x*x), add = TRUE, type = "p")
```

```
## ================================================================================
## Example 1b: simple
## solve d2y/dx2 + 1/x*dy/dx + (1-1/(4x^2)y = sqrt(x)*cos(x),
## on the interval [1,6] and with boundary conditions:
## y(1)=1, y(6)=-0.5
##
## Write as set of 2 odes
## dy/dx = y2
## dy2/dx  = - 1/x*dy/dx - (1-1/(4x^2)y + sqrt(x)*cos(x)
## ================================================================================

f2 <- function(x, y, parms)
{
 dy  <- y[2]
 dy2 <- -1/x*y[2] - (1-1/(4*x^2))*y[1] + sqrt(x)*cos(x)
 list(c(dy, dy2))
}

x    <- seq(1, 6, 0.1)
sol  <- bvpshoot(yini = c(y = 1, dy = NA), yend = c(-0.5, NA),
          x = x, func = f2, guess = 1)

# plot both state variables
plot(sol, type = "l", lwd = 2)

# plot only y and add the analytic solution
plot(sol, which = "y")

curve(0.0588713*cos(x)/sqrt(x)+1/4*sqrt(x)*cos(x)+0.740071*sin(x)/sqrt(x)+
      1/4*x^(3/2)*sin(x), add = TRUE, type = "l")


## ================================================================================
## Example 2 - initial condition is a function of the unknown x
## tubular reactor with axial dispersion
## y''=Pe(y'+Ry^n) Pe=1,R=2,n=2
## on the interval [0,1] and with initial conditions:
## y'0=Pe(y(0)-1), y'(1)=0
##
## dy=y2
## dy2=Pe(dy-Ry^n)
## ================================================================================

Reactor<-function(x, y, parms)
{
  list(c(y[2],
         Pe * (y[2]+R*(y[1]^n))))
}

Pe <- 1
R  <- 2
n  <- 2
```

```
x  <- seq(0, 1, by = 0.01)

# 1. yini is a function here
yini <- function (x, parms) c(x, Pe*(x-1))

system.time(
  sol <- bvpshoot(func = Reactor, yend = c(y = NA, dy = 0),
          yini = yini, x = x, extra = 1)
)
plot(sol, which = "y", main = "Reactor", type = "l", lwd = 2)
attributes(sol)$roots

# 2. using boundary function rather than yini...
bound <- function(i, y, p) {
  if (i == 1) return(y[2] - Pe*(y[1]-1))
  if (i == 2) return(y[2])
}

# need to give number of left boundary conditions + guess of all initial
# conditions (+ names)
system.time(
Sol2<- bvpshoot(func = Reactor, x = x, leftbc = 1,
            bound = bound, guess = c(y = 1, dy = 1) )
)
attributes(Sol2)$roots


# 3. boundary function with jacobian of boundary function
jacbound <- function(i, y, p) {
  if (i == 1) return(c(-Pe*y[1], 1))
  if (i == 2) return(c(0, 1))
}
system.time(
Sol3<-bvpshoot(func = Reactor, x = x, leftbc = 1, bound = bound,
           jacbound = jacbound, guess = c(y = 1, dy = 1) )
)
attributes(Sol3)$roots

## =============================================================================
## Example 2b - same as 2 but written as higher-order equation
## y''=Pe(y'+Ry^n) Pe=1,R=2,n=2
## on the interval [0,1] and with initial conditions:
## y'0=Pe(y(0)-1), y'(1)=0
## =============================================================================

Reactor2<-function(x, y, parms)
  list (Pe * (y[2]+R*(y[1]^n)))

Pe <- 1
R  <- 2
n  <- 2
x  <- seq(0, 1, by = 0.01)
```

```
# 1. yini is a function here
yini <- function (x, parms) c(x, Pe*(x-1))

# need to specify that order = 2
system.time(
  sol2 <- bvpshoot(func = Reactor2, yend = c(y = NA, dy = 0), order=2,
          yini = yini, x = x, extra = 1)
)
max(abs(sol2-sol))

## =============================================================================
## Example 3 - final condition is a residual function
## The example MUSN from Ascher et al., 1995.
## Numerical Solution of Boundary Value Problems for Ordinary Differential
## Equations, SIAM, Philadelphia, PA, 1995.
##
## The problem is
##      u' =  0.5*u*(w - u)/v
##      v' = -0.5*(w - u)
##      w' = (0.9 - 1000*(w - y) - 0.5*w*(w - u))/z
##      z' =  0.5*(w - u)
##      y' = -100*(y - w)
##   on the interval [0 1] and with boundary conditions:
##      u(0) = v(0) = w(0) = 1,  z(0) = -10,  w(1) = y(1)
## =============================================================================

musn <- function(t, Y, pars)
{
  with (as.list(Y),
  {
   du <- 0.5 * u * (w-u)/v
   dv <- -0.5 * (w-u)
   dw <- (0.9 - 1000 * (w-y) - 0.5 * w * (w-u))/z
   dz <- 0.5 * (w-u)
   dy <- -100 * (y-w)
   return(list(c(du, dv, dw, dy, dz)))
  })
}

#--------------------------------
# Residuals of end conditions
#--------------------------------
res  <- function (Y, yini, pars)  with (as.list(Y), w-y)

#--------------------------------
# Initial values; y= NA= not available
#--------------------------------

init <- c(u = 1, v = 1, w = 1, y = NA, z = -10)
sol  <-bvpshoot(y = init, x = seq(0, 1, by = 0.05), func = musn,
           yend = res, guess = 1, atol = 1e-10, rtol = 0)
pairs(sol, main = "MUSN")
```

```
## ================================================================================
## Example 4 - solve also for unknown parameter
## Find the 4th eigenvalue of Mathieu's equation:
## y''+(lam-10cos2t)y=0   on the interval [0,pi]
## y(0)=1, y'(0)=0  and y'(pi)=0
## The 2nd order problem is rewritten as 2 first-order problems:
## dy=y2
## dy2= -(lam-10cos(2t))*y
## ================================================================================

mathieu<- function(t, y, lam)
{
 list(c(y[2], -(lam - 10 * cos(2*t)) * y[1]))
}

yini <- c(1, 0)   # initial condition(y1=1,dy=y2=0)
yend <- c(NA, 0)  # final condition at pi (y1=NA, dy=0)

# there is one extra parameter to be fitted: "lam"; its initial guess = 15
Sol  <- bvpshoot(yini = yini, yend = yend, x = seq(0, pi, by = 0.01),
          func = mathieu, guess = NULL, extra = 15)
plot(Sol)
attr(Sol, "roots")  # root gives the value of "lam" (17.1068)
```

---

bvptwp *Solves two-point boundary value problems of ordinary differential equations, using a mono-implicit Runge-Kutta formula*

Description

Solves a boundary value problem of a system of ordinary differential equations. This is an implementation of the fortran code twpbvpc, based on mono-implicit Runge-Kutta formulae of orders 4, 6 and 8 in a deferred correction framework and that uses conditioning in the mesh selection.

written by J.R. Cash, F. Mazzia and M.H. Wright.

Rather than MIRK, it is also possible to select a lobatto method. This then uses the code 'twpbvplc', written by Cash and Mazzia.

It is possible to solve stiff systems, by using an automatic continuation strategy. This then uses the code 'acdc'.

Usage

```
bvptwp(yini = NULL, x, func, yend = NULL, parms = NULL,
       order = NULL, ynames = NULL, xguess = NULL, yguess = NULL,
       jacfunc = NULL, bound = NULL, jacbound = NULL,
       leftbc = NULL, posbound = NULL, islin = FALSE, nmax = 1000,
       ncomp = NULL, atol = 1e-8, cond = FALSE, lobatto = FALSE,
       allpoints = TRUE, dllname = NULL, initfunc = dllname,
       rpar = NULL, ipar = NULL, nout = 0, outnames = NULL,
```

```
forcings = NULL, initforc = NULL, fcontrol = NULL,
verbose = FALSE, epsini = NULL, eps = epsini, ...)
```

Arguments

yini	either a vector with the initial (state) variable values for the ODE system, or NULL. If yini is a vector, use NA for an initial value which is not specified. If yini has a names attribute, the names will be available within func and used to label the output matrix. If yini = NULL, then the boundary conditions must be specified via function bound; if not NULL then yend should also be not NULL.
x	sequence of the independent variable for which output is wanted; the first value of x must be the initial value (at which yini is defined), the final value the end condition (at which yend is defined).
func	either an R-function that computes the values of the derivatives in the ODE system (the model definition) at point x, or a character string giving the name of a compiled function in a dynamically loaded shared library. If func is an R-function, it must be defined as: func = function(x, y, parms,...). x is the current point of the independent variable in the integration, y is the current estimate of the (state) variables in the ODE system. If the initial values yini has a names attribute, the names will be available inside func. parms is a vector or list of parameters; ... (optional) are any other arguments passed to the function. The return value of func should be a list, whose first element is a vector containing the derivatives of y with respect to x, and whose next elements are global values that are required at each point in x. If func is a string, then dllname must give the name of the shared library (without extension) which must be loaded before bvptwp is called. See package vignette "bvpSolve" for more details.
yend	either a vector with the final (state) variable values for the ODE system, or NULL; if yend is a vector, use NA for a final value which is not specified. If yend has a names attribute, and yini does not, the names will be available within the functions and used to label the output matrix. If yend = NULL, then the boundary conditions must be specified via function bound; if not NULL then yini should also be not NULL.
parms	vector or a list with parameters passed to func, jacfunc, bound and jacbound (if present). If eps is given a value then it should be the **first** element in parms.
epsini	the initial value of the continuation parameter. If NULL and eps is given a value, then epsini takes the default starting value of 0.5. For many singular perturbation type problems, the choice of $0.1 < \text{eps} < 1$ represents a (fairly) easy problem. The user should attempt to specify an initial problem that is not 'too' challenging. epsini must be initialised strictly less than 1 and greater than 0.

eps — the desired value of precision for which the user would like to solve the problem. eps must be less than or equal to epsini. If this is given a value, it must be the first value in parms.

ynames — The names of the variables; used to label the output, and avaliable within the functions.

If ynames is NULL, names can also be passed via yini, yend or yguess.

xguess — Initial grid x, a vector. bvptwp requires the length of xguess to be at least equal to the length of x. If this is not the case, then xguess and yguess will be interpolated to x and a warning printed. If xguess is given, so should yguess be.

Supplying xguess and yguess, based on results from a previous (simpler) BVP-ODE can be used for model continuation, see example 2.

yguess — First guess values of y, corresponding to initial grid xguess; a matrix with number of rows equal to the number of equations, and whose number of columns equals the length of xguess.

if the rows of yguess have a names attribute, the names will be available within the functions and used to label the output matrix.

jacfunc — jacobian (optional) - either an R-function that evaluates the jacobian of func at point x, or a string with the name of a function or subroutine in dllname that computes the Jacobian (see vignette "bvpSolve" for more about this option).

If jacfunc is an R-function, it must be defined as: `jacfunc = function(x, y, parms,...)`. It should return the partial derivatives of func with respect to y, i.e. df(i,j) = dfi/dyj. See last example.

If jacfunc is NULL, then a numerical approximation using differences is used. This is the default.

bound — boundary function (optional) - only if yini and yend are not available. Either an R function that evaluates the i-th boundary element at point x, or a string with the name of a function or subroutine in dllname that computes the boundaries (see vignette "bvpSolve" for more about this option).

If bound is an R-function, it should be defined as: `bound = function(i, y, parms, ...)`. It should return the i-th boundary condition. See last example.

jacbound — jacobian of the boundary function (optional) - only if bound is defined. Either an R function that evaluates the gradient of the i-th boundary element with respect to the state variables, at point x, or a string with the name of a function or subroutine in dllname that computes the boundary jacobian (see vignette "bvpSolve" for more about this option).

If jacbound is an R-function, it should be defined as: `jacbound = function(i, y, parms, ...)`. It should return the gradient of the i-th boundary condition. See last example.

If jacbound is NULL, then a numerical approximation using differences is used. This is the default.

leftbc — only if yini and yend are not available and posbound is not specified: the number of left boundary conditions.

posbound — only used if bound is given: a vector with the position (in the mesh) of the boundary conditions - only the boundary points are allowed. Note that it is simpler to use leftbc.

islin	set to TRUE if the problem is linear - this will speed up the simulation.
nmax	maximal number of subintervals during the calculation.
order	the order of each derivative in func. The default is that all derivatives are 1-st order, in which case order can be set = NULL. If order is not NULL, the number of equations in func must equal the length of order; the summed values of order must equal the number of variables (ncomp). The jacobian as specified in jacfunc must have number of rows = number of equations and number of columns = number of variables. bound and jacbound remain defined in the number of variables. See example 4 and 4b.
ncomp	used if the model is specified by compiled code, the number of components. See package vignette "bvpSolve". Also to be used if the boundary conditions are specified by bound, and there is no yguess
atol	error tolerance, a scalar.
cond	if TRUE, uses conditioning in the mesh selection
lobatto	if TRUE, selects a lobatto method.
allpoints	sometimes the solver estimates the solution in a number of extra points, and by default the solutions at these extra points will also be returned. By setting allpoints equal to FALSE, only output corresponding to the elements in x will be returned.
dllname	a string giving the name of the shared library (without extension) that contains all the compiled function or subroutine definitions referred to in func, jacfunc, bound and jacbound. Note that ALL these subroutines must be defined in the shared library; it is not allowed to merge R-functions with compiled functions. See package vignette "bvpSolve".
initfunc	if not NULL, the name of the initialisation function (which initialises values of parameters), as provided in 'dllname'. See package vignette "bvpSolve".
rpar	only when 'dllname' is specified: a vector with double precision values passed to the dll-functions whose names are specified by func and jacfunc.
ipar	only when 'dllname' is specified: a vector with integer values passed to the dll-functions whose names are specified by func and jacfunc.
nout	only used if dllname is specified and the model is defined in compiled code: the number of output variables calculated in the compiled function func, present in the shared library. Note: it is not automatically checked whether this is indeed the number of output variables calculated in the dll - you have to perform this check in the code. See deSolve's package vignette "compiledCode".
outnames	only used if function is specified in compiled code and nout > 0: the names of output variables calculated in the compiled function. These names will be used to label the output matrix. The length of outnames should be = nout.
forcings	only used if 'dllname' is specified: a list with the forcing function data sets, each present as a two-columned matrix, with (time,value); interpolation outside the interval [min(times), max(times)] is done by taking the value at the closest data extreme. This feature is included for consistency with the initial value problem solvers from package deSolve. See package vignette "compiledCode" from package deSolve.

initforc	if not NULL, the name of the forcing function initialisation function, as provided in 'dllname'. It MUST be present if forcings has been given a value. See package vignette "compiledCode" from package deSolve.
fcontrol	A list of control parameters for the forcing functions. See package vignette "compiledCode" from package deSolve.
verbose	if TRUE then more verbose output will be generated as "warnings".
...	additional arguments passed to the model functions.

Details

This is an implementation of the method *twpbvpC*, written by Cash, Mazzia and Wright, to solve two-point boundary value problems of ordinary differential equations.

A boundary value problem does not have all initial values of the state variable specified. Rather some conditions are specified at the end of the integration interval. The number of unknown boundary conditions must be equal to the number of equations (or the number of dependent variables y).

The ODEs and boundary conditions are made available through the user-provided routines, func and vectors yini and yend or (optionally) bound.

The corresponding partial derivatives for func and bound are optionally available through the user-provided routines, jacfunc and jacbound. Default is that they are automatically generated by bvptwp, using numerical differences.

The user-requested tolerance is provided through tol. The default is $1e^{-}6$

If the function terminates because the maximum number of subintervals was exceeded, then it is recommended that 'the program be run again with a larger value for this maximum.' It may also help to start with a simple version of the model, and use its result as initial guess to solve the more complex problem (continuation strategy, see example 2, and package vignette "bvpSolve").

Models may be defined in **compiled C or Fortran** code, as well as in an **R**-function.

This is similar to the initial value problem solvers from package deSolve. See package vignette "bvpSolve" for details about writing compiled code.

The **fcontrol** argument is a list that can supply any of the following components (conform the definitions in the approx function):

method specifies the interpolation method to be used. Choices are "linear" or "constant",

rule an integer describing how interpolation is to take place outside the interval [min(times), max(times)]. If rule is 1 then an error will be triggered and the calculation will stop if times extends the interval of the forcing function data set. If it is 2, the *default*, the value at the closest data extreme is used, a warning will be printed if verbose is TRUE,

Note that the default differs from the approx default

f For method="constant" a number between 0 and 1 inclusive, indicating a compromise between left- and right-continuous step functions. If y0 and y1 are the values to the left and right of the point then the value is y0*(1-f)+y1*f so that f=0 is right-continuous and f=1 is left-continuous,

ties Handling of tied `times` values. Either a function with a single vector argument returning a single number result or the string "ordered".

Note that the default is "ordered", hence the existence of ties will NOT be investigated; in the `C` code this will mean that -if ties exist, the first value will be used; if the dataset is not ordered, then nonsense will be produced.

Alternative values for `ties` are `mean`, `min` etc

The defaults are:

```
fcontrol=list(method="linear", rule = 2,  f = 0, ties = "ordered")
```

Note that only ONE specification is allowed, even if there is more than one forcing function data set.

This -advanced- feature is explained in `deSolve`'s package vignette "compiledCode".

Value

A matrix of class `bvpSolve`, with up to as many rows as elements in `x` and as many columns as elements in `yini` or `ncomp` plus the number of "global" values returned from `func`, plus an additional column (the first) for the `x`-value.

Typically, there will be one row for each element in `x` unless the solver returns with an unrecoverable error. In certain cases, `bvptwp` will return the solution also in a number of extra points. This will occur if the number of points as in `x` was not sufficient. In order to not return these extra points, set `allpoints` equal to `FALSE`.

If `ynames` is given, or `yini`, `yend` has a names attribute, or `yguess` has named rows, the names will be used to label the columns of the output value.

Note

When `order` is not `NULL`, then it should contain the order of all *equations* in `func`. If the order of some equations > 1, then there will be less equations than variables. The number of equations should be equal to the length of `order`, while the number of variables will be the sum of order.

For instance, if `order = c(1,2,3,4)`, then the first equation will be of order 1, the second one of order 2, ...and the last of order 4.

There will be 1+2+3+4 = 10 variables. For instance, if the derivative function defines (A', B", C"', D"") respectively, then the variable vector will contain values for A, B, B', C, C', C", D, D', D", D"'; in that order. This is also the order in which the initial and end conditions of all variables must be specified.

If neq are the number of equations, and ncomp the number of variables, then the Jacobian of the derivative function as specified in `jacfunc` must be of dimension (neq, ncomp).

The jacobian of the boundaries, as specified in `jacbound` should return a vector of length = ncomp

Author(s)

Karline Soetaert <karline.soetaert@nioz.nl>

Jeff Cash <j.cash@imperial.ac.uk>

Francesca Mazzia <mazzia@dm.uniba.it>

References

J.R. Cash and M.H. Wright, A deferred correction method for nonlinear two-point boundary value problems: implementation and numerical evaluation, SIAM J. Sci. Stat. Comput., 12 (1991) 971 989.

Cash, J. R. and F. Mazzia, A new mesh selection algorithm, based on conditioning, for two-point boundary value codes. J. Comput. Appl. Math. 184 (2005), no. 2, 362–381.

Cash, J. R. and F. Mazzia, in press. Hybrid Mesh Selection Algorithms Based on Conditioning for Two-Point Boundary Value Problems, Journal of Numerical Analysis, Industrial and Applied Mathematics.

See Also

bvpshoot for the shooting method

bvpcol for the collocation method

diagnostics.bvpSolve, for a description of diagnostic messages

plot.bvpSolve, for a description of plotting the output of the BVP solvers.

Examples

```
## =============================================================================
## Example 1: simple standard problem
## solve the BVP ODE:
## d2y/dt^2=-3py/(p+t^2)^2
## y(t= -0.1)=-0.1/sqrt(p+0.01)
## y(t=  0.1)= 0.1/sqrt(p+0.01)
## where p = 1e-5
##
## analytical solution y(t) = t/sqrt(p + t^2).
##
## The problem is rewritten as a system of 2 ODEs:
## dy=y2
## dy2=-3p*y/(p+t^2)^2
## =============================================================================

#-------------------------------
# Derivative function
#-------------------------------
fun <- function(t, y, pars) {
  dy1 <- y[2]
  dy2 <- - 3*p*y[1] / (p+t*t)^2
  return(list(c(dy1,
                dy2))) }


# parameter value
p    <- 1e-5

# initial and final condition; second conditions unknown
init <- c(y = -0.1 / sqrt(p+0.01), dy=NA)
```

```
end  <- c(     0.1 / sqrt(p+0.01), NA)

# Solve bvp
sol  <- as.data.frame(bvptwp(yini = init, x = seq(-0.1, 0.1, by = 0.001),
        func = fun, yend = end))
plot(sol$x, sol$y, type = "l")

# add analytical solution
curve(x/sqrt(p+x*x), add = TRUE, type = "p")


## =============================================================================
## Example 1b:
## Same problem, now solved as a second-order equation.
## =============================================================================

fun2 <- function(t, y, pars)  {
  dy <- - 3 * p * y[1] / (p+t*t)^2
  list(dy)
}
sol2  <- bvptwp(yini = init, yend = end, order = 2,
                x = seq(-0.1, 0.1, by = 0.001), func = fun2)

## =============================================================================
## Example 2: simple
## solve d2y/dx2 + 1/x*dy/dx + (1-1/(4x^2)y = sqrt(x)*cos(x),
## on the interval [1,6] and with boundary conditions:
## y(1)=1, y(6)=-0.5
##
## Write as set of 2 odes
## dy/dx = y2
## dy2/dx  = - 1/x*dy/dx - (1-1/(4x^2)y + sqrt(x)*cos(x)
## =============================================================================

f2 <- function(x, y, parms) {
  dy  <- y[2]
  dy2 <- -1/x*y[2] - (1-1/(4*x^2))*y[1] + sqrt(x)*cos(x)
  list(c(dy, dy2))
}

x    <- seq(1, 6, 0.1)
sol  <- bvptwp(yini = c(y = 1, dy = NA),
               yend = c(-0.5, NA), x = x, func = f2)
plot(sol, which = "y")

# add the analytic solution
curve(0.0588713*cos(x)/sqrt(x)+1/4*sqrt(x)*cos(x)+0.740071*sin(x)/sqrt(x)+
      1/4*x^(3/2)*sin(x), add = TRUE, type = "l")

## =============================================================================
## Example 3  - solved with automatic continuation
## d2y/dx2 = y/xi
## =============================================================================
```

```
Prob1 <- function(t, y, xi)
   list(c( y[2] , y[1]/xi ))

x <-  seq(0, 1, by = 0.01)
xi <- 0.1
twp <- bvptwp(yini = c(1, NA), yend = c(0, NA), x = x,
             islin = TRUE, func = Prob1, parms = xi, eps = xi)

xi <-0.00001
twp2 <- bvptwp(yini = c(1, NA), yend = c(0, NA), x = x,
               islin = TRUE, func = Prob1, parms = xi, eps = xi)

plot(twp, twp2, which = 1, main = "test problem 1")

# exact solution
curve(exp(-x/sqrt(xi))-exp((x-2)/sqrt(xi))/(1-exp(-2/sqrt(xi))),
      0, 1, add = TRUE, type = "p")

curve(exp(-x/sqrt(0.1))-exp((x-2)/sqrt(0.1))/(1-exp(-2/sqrt(0.1))),
      0, 1, add = TRUE, type = "p")

## =====================================================================================
## Example 4  - solved with specification of boundary, and jacobians
## d4y/dx4 =R(dy/dx*d2y/dx2 -y*dy3/dx3)
## y(0)=y'(0)=0
## y(1)=1, y'(1)=0
##
## dy/dx  = y2
## dy2/dx = y3    (=d2y/dx2)
## dy3/dx = y4    (=d3y/dx3)
## dy4/dx = R*(y2*y3 -y*y4)
## =====================================================================================

f2<- function(x, y, parms, R) {
  list(c(y[2], y[3], y[4], R*(y[2]*y[3] - y[1]*y[4]) ))
}

df2 <- function(x, y, parms, R) {
  matrix(nrow = 4, ncol = 4, byrow = TRUE, data = c(
              0,        1,     0,     0,
              0,        0,     1,     0,
              0,        0,     0,     1,
              -1*R*y[4],R*y[3],R*y[2],-R*y[1]))
}

g2 <- function(i, y, parms, R) {
  if (i == 1) return(y[1])
  if (i == 2) return(y[2])
  if (i == 3) return(y[1]-1)
  if (i == 4) return(y[2])
}
```

```
dg2 <- function(i, y, parms, R) {
  if (i == 1) return(c(1, 0, 0, 0))
  if (i == 2) return(c(0, 1, 0, 0))
  if (i == 3) return(c(1, 0, 0, 0))
  if (i == 4) return(c(0, 1, 0, 0))
}

init <- c(1, NA)
R    <- 100
sol  <- bvptwp(x = seq(0, 1, by = 0.01), leftbc = 2,
          func = f2, R = R, ncomp = 4,
          bound = g2, jacfunc = df2, jacbound = dg2)
plot(sol[,1:2])  # columns do not have names

mf <- par ("mfrow")
sol  <- bvptwp(x = seq(0, 1, by = 0.01), leftbc = 2,
          func = f2, ynames = c("y", "dy", "d2y", "d3y"), R=R,
          bound = g2, jacfunc = df2, jacbound = dg2)
plot(sol)        # here they do
par(mfrow = mf)

## =============================================================================
## Example 4b - solved with specification of boundary, and jacobians
## and as a higher-order derivative
## d4y/dx4 =R(dy/dx*d2y/dx2 -y*dy3/dx3)
## y(0)=y'(0)=0
## y(1)=1, y'(1)=0
## =============================================================================

# derivative function: one fourth-order derivative
f4th <- function(x, y, parms, R) {
  list(R * (y[2]*y[3] - y[1]*y[4]))
}

# jacobian of derivative function
df4th <- function(x, y, parms, R)  {
 df <- matrix(nrow = 1, ncol = 4, byrow = TRUE, data = c(
             -1*R*y[4], R*y[3], R*y[2], -R*y[1]))
}

# boundary function - same as previous example

# jacobian of boundary - same as previous

# order = 4 specifies the equation to be 4th order
sol2 <- bvptwp(x = seq(0, 1, by = 0.01),
          ynames = c("y", "dy", "d2y", "d3y"),
          posbound = c(0, 0, 1, 1), func = f4th, R = R, order = 4,
          bound = g2, jacfunc = df4th, jacbound = dg2)

max(abs(sol2-sol))
```

`diagnostics.bvpSolve` *Prints Diagnostic Characteristics of BVP Solvers*

Description

Prints several diagnostics of the simulation to the screen, e.g. conditioning parameters

Usage

```
## S3 method for class 'bvpSolve'
diagnostics(obj, ...)
## S3 method for class 'bvpSolve'
approx(x, xout = NULL, ...)
```

Arguments

`obj`	the output as produced by `bvptwp`, `bvpcol` or `bvpshoot`.
`x`	the output as produced by `bvpcol`
`xout`	points `x` for which new variable values should be generated.
`...`	optional arguments to the generic function.

Details

When the integration output is saved as a data.frame, then the required attributes are lost and method `diagnostics` will not work anymore.

Value

S3 method `diagnostics` prints diagnostic features of the simulation.

What exactly is printed will depend on the solution method.

The diagnostics of all solvers include the number of function evaluations, the number of jacobian evaluations, and the number of steps. The diagnostics of both `bvptwp` and `bvpcol` also include the the number of boundary evaluations and the number of boundary jacobian evaluations. In case the problem was solved with `bvpshoot`, the diagnostics of the initial value problem solver will also be written to screen.

Note that the number of function evaluations are *without* the extra calls performed to generate the ordinary output variables (if present).

In case the method used was `bvptwp`, will also return the *conditioning parameters*. They are: kappa, kappa1, kappa2, sigma and gamma1.

See http://www.scpe.org/index.php/scpe/article/view/626

the kappa's are based on the Inf-norm, gamma1 is based on the 1-norm, If kappa, kappa1 and gamma1 are of moderate size, the problem is well conditioned. If large, the problem is ill-conditioned. If kappa1 is large and gamma1 is small, the problem is ill-conditioned in the maximum and well conditioned in the 1-norm. This is typical for problems that involve different time scales ("stiff"

problems). If kappa1 is small and kappa, kappa2 are large the problem has not the correct dichotomy.

S3 method `approx` calculates an approximate solution vector at points inbetween the original `x`-values. If beyond the integration interval, it will not extrapolate, but just return the values at the edges. This works only when the solution was generated with `bvpcol`, and usses information in the arrays rwork and iwork, stored as attributes. The returned matrix will be of class "bvpSolve"

See Also

diagnostics.deSolve for a description of diagnostic messages of the initial value problem solver as used by `bvpshoot`

`plot.bvpSolve`, for a description of plotting the output of the BVP solvers.

Examples

```
## ================================================================================
## Diagnostic messages
## ================================================================================
f2 <- function(x, y, parms) {
 dy  <- y[2]
 dy2 <- -1/x*y[2] - (1-1/(4*x^2))*y[1] + sqrt(x)*cos(x)
 list(c(dy, dy2))
}

x    <- seq(1, 6, 0.1)
yini <- c(y = 1, dy = NA)
yend <- c(-0.5, NA)

sol   <- bvptwp(yini = yini, yend = yend, x = x, func = f2)
sol2  <- bvpcol(yini = yini, yend = yend, x = x, func = f2)
sol3  <- bvpshoot(yini = yini, yend = yend, x = x, func = f2, guess = 0)

plot(sol, which = "y")
diagnostics(sol)
diagnostics(sol2)
diagnostics(sol3)

## ================================================================================
## approx
## ================================================================================

soldetail <- approx(sol2, xout = seq(2,4,0.01))
plot(soldetail)

# beyond the interval
approx(sol2, xout = c(0,1,2))
approx(sol2, xout = c(6,100))
```

`plot.bvpSolve` *Plot and Print Methods for Output of bvp solvers*

Description

Plot the output of boundary value solver routines.

Usage

```
## S3 method for class 'bvpSolve'
plot(x, ..., which = NULL, ask = NULL,
                        obs = NULL, obspar= list())
## S3 method for class 'bvpSolve'
print(x, ...)
```

Arguments

`x`	the output of `bvpSolve`, as returned by the boundary value solvers, and to be plotted. It is allowed to pass several objects of class `bvpSolve` after `x` (unnamed) - see second example.
`which`	the name(s) or the index to the variables that should be plotted. Default = all variables, except the first column.
`ask`	logical; if `TRUE`, the user is *ask*ed before each plot, if `NULL` the user is only asked if more than one page of plots is necessary and the current graphics device is set interactive, see `par(ask=.)` and `dev.interactive`.
`obs`	a `data.frame` or `matrix` with "observed data" that will be added as `points` to the plots. `obs` can also be a `list` with multiple data.frames and/or matrices containing observed data. By default the first column of an observed data set should contain the `time`-variable. The other columns contain the observed values and they should have names that are known in `x`. If the first column of `obs` consists of factors or characters (strings), then it is assumed that the data are presented in long (database) format, where the first three columns contain (name, time, value). If `obs` is not `NULL` and `which` is `NULL`, then the variables, common to both `obs` and `x` will be plotted.
`obspar`	additional graphics arguments passed to `points`, for plotting the observed data
`...`	additional arguments. The graphical arguments are passed to `plot.default`. The dots may also contain other objects of class `bvpSolve`, as returned by the boundary value solvers, and to be plotted on the same graphs as `x` - see second example. In this case, `x` and and these other objects should be compatible, i.e. the names should be the same and they should have same number of rows. The arguments after ... must be matched exactly.

Details

`print.bvpSolve` prints the matrix without the attributes.

`plot.bvpSolve` plots multiple figures on a page.

The number of panels per page is automatically determined up to 3 x 3 (`par(mfrow = c(3, 3))`). This default can be overwritten by specifying user-defined settings for `mfrow` or `mfcol`. Set `mfrow` equal to `NULL` to avoid the plotting function to change user-defined `mfrow` or `mfcol` settings.

Other graphical parameters can be passed as well. Parameters are vectorized, either according to the number of plots (`xlab, ylab, main, sub, xlim, ylim, log, asp, ann, axes, frame.plot,panel.first,panel.last, cex.lab,cex.axis,cex.main`) or according to the number of lines within one plot (other parameters e.g. `col`, `lty`, `lwd` etc.) so it is possible to assign specific axis labels to individual plots, resp. different plotting style. Plotting parameter `ylim`, or `xlim` can also be a list to assign different axis limits to individual plots.

Similarly, the graphical parameters for observed data, as passed by `obspar` can be vectorized, according to the number of observed data sets.

See Also

`diagnostics.bvpSolve`, for a description of diagnostic messages.

Examples

```
## ================================================================================
## The example MUSN from Ascher et al., 1995.
## Numerical Solution of Boundary Value Problems for Ordinary Differential
## Equations, SIAM, Philadelphia, PA, 1995.
##
## The problem is
##      u' =  0.5*u*(w - u)/v
##      v' = -0.5*(w - u)
##      w' = (0.9 - 1000*(w - y) - 0.5*w*(w - u))/z
##      z' =  0.5*(w - u)
##      y' = -100*(y - w)
##   on the interval [0 1] and with boundary conditions:
##      u(0) = v(0) = w(0) = 1,  z(0) = -10,  w(1) = y(1)
## ================================================================================

musn <- function(t, Y, pars)  {
  with (as.list(Y),
  {
   du <- 0.5 * u * (w-u)/v
   dv <- -0.5 * (w-u)
   dw <- (0.9 - 1000 * (w-y) - 0.5 * w * (w-u))/z
   dz <- 0.5 * (w-u)
   dy <- -100 * (y-w)
   return(list(c(du, dv, dw, dz, dy)))
  })
}

#-------------------------------
```

```
# Boundaries
#-------------------------------
bound <- function(i,y,pars) {
  with (as.list(y), {
    if (i ==1) return (u-1)
    if (i ==2) return (v-1)
    if (i ==3) return (w-1)
    if (i ==4) return (z+10)
    if (i ==5) return (w-y)
 })
}

xguess <- seq(0, 1, len = 5)
yguess <- matrix(ncol = 5, (rep(c(1, 1, 1, -10, 0.91), times = 5)) )
rownames(yguess) <- c("u", "v", "w", "z", "y")

sol  <- bvpcol (bound = bound, x = seq(0, 1, by = 0.05),
          leftbc = 4, func = musn, xguess = xguess, yguess = yguess)

mf <- par("mfrow")
plot(sol)
par(mfrow = mf)

## ================================================================================
## Example 2. Example Problem 31 from Jeff Cash's website
## ================================================================================

Prob31 <- function(t, Y, pars)  {
  with (as.list(Y), {
    dy    <- sin(Tet)
    dTet  <- M
    dM    <- -Q/xi
    T     <- 1/cos (Tet) +xi*Q*tan(Tet)
    dQ    <- 1/xi*((y-1)*cos(Tet)-M*T)
    list(c( dy, dTet, dM, dQ))
  })
}

ini <- c(y = 0, Tet = NA, M = 0, Q = NA)
end <- c(y = 0, Tet = NA, M = 0, Q = NA)

# run 1
xi <-0.1
twp  <- bvptwp(yini = ini, yend = end, x = seq(0, 1, by = 0.01),
               func = Prob31, atol = 1e-10)

# run 2
xi <- 0.05
twp2 <- bvptwp(yini = ini, yend = end, x = seq(0, 1, by = 0.01),
               func = Prob31, atol = 1e-10)

# run 3
xi <- 0.01
```

```
twp3 <- bvptwp(yini = ini, yend = end, x = seq(0, 1, by = 0.01),
               func = Prob31, atol = 1e-10)

# print all outputs at once
plot(twp, twp2, twp3, xlab = "x", ylab = names(ini))


# change type, colors, ...
plot(twp, twp2, twp3, type = c("l", "b", "p"),
     main = paste ("State Variable", names(ini)),
     col = c("red", "blue", "orange"), cex = 2)

## =============================================================================
## Assume we have two 'data sets':
## =============================================================================
# data set in 'wide' format
obs1 <- cbind(time = c(0, 0.5, 1), Tet = c(0.4, 0.0, -0.4))

# data set in 'long' format
obs2 <- data.frame(name = "Tet", time = c(0, 0.5, 1), value = c(0.35, 0.0, -0.35))

plot(twp, twp2, obs = obs1, obspar = list(pch = 16, cex = 1.5))

plot(twp, twp2, obs = list(obs1, obs2),
    obspar = list(pch = 16, cex = 1.5))

plot(twp, twp2, obs = list(obs1, obs2), which = c("Tet", "Q"),
    obspar = list(pch = 16:17, cex = 1.5, col = c("red", "black"))
    )
```

Index