Package 'sundialr'

October 18, 2026

Type Package

Title An Interface to 'SUNDIALS' Ordinary Differential Equation (ODE) Solvers

Version 0.2.1

Date 2026-10-17

Maintainer Satyaprakash Nayak <satyaprakash.nayak@gmail.com>

URL https://github.com/sn248/sundialr, http://sn248.github.io/sundialr/

BugReports https://github.com/sn248/sundialr/issues

Description Provides a way to call the functions in 'SUNDIALS' C ODE solving library (<https://computing.llnl.gov/projects/sundials>). Currently the serial version of ODE solver, 'CVODE', sensitivity calculator 'CVODES' and differential algebraic solver 'IDA' from the 'SUNDIALS' library are implemented. The package requires ODE to be written as an 'R' or 'Rcpp' function and does not require the 'SUNDIALS' library to be installed on the local machine.

License BSD_3_clause + file LICENSE

Copyright file COPYRIGHTS

Imports Rcpp (>= 1.0.12)

LinkingTo Rcpp, RcppArmadillo

Suggests knitr, rmarkdown, testthat

SystemRequirements cmake

Encoding UTF-8

NeedsCompilation yes

VignetteBuilder knitr

Config/roxygen2/version 8.1.1

Author Satyaprakash Nayak [aut, cre, cph] (ORCID: <https://orcid.org/0000-0001-7225-1317>), Lawrence Livermore National Security [cph], Southern Methodist University [cph], University of Maryland Baltimore County [cph]

Repository CRAN

Date/Publication 2026-10-18 04:00:02 UTC

Contents

cvode 2
cvodes 4
cvsolve 7
ida 8

Index **12**

---

cvode *cvode*

Description

CVODE solver to solve stiff ODEs

Usage

```
cvode(
  time_vector,
  IC,
  input_function,
  Parameters,
  reltolerance = 1e-04,
  abstolerance = 1e-04,
  jacobian = NULL
)
```

Arguments

`time_vector`	time vector
`IC`	Initial Conditions
`input_function`	Right Hand Side function of ODEs
`Parameters`	Parameters input to ODEs
`reltolerance`	Relative Tolerance (a scalar, default value = 1e-04)
`abstolerance`	Absolute Tolerance (a scalar or vector with length equal to ydot (dy/dx), default = 1e-04)
`jacobian`	(Optional) Jacobian of the RHS with signature `function(t, y, p)` returning an n-by-n matrix where entry [i,j] is d(ydot_i)/d(y_j). Default is NULL and SUNDIALS uses internal finite-difference approximation.

Value

A Matrix. First column is the time-vector, the other columns are values of y in order they are provided.

Examples

```
# Example of solving a set of ODEs with cvode function
# ODEs described by an R function
ODE_R <- function(t, y, p){

  # vector containing the right hand side gradients
  ydot = vector(mode = "numeric", length = length(y))

  # R indices start from 1
  ydot[1] = -p[1]*y[1] + p[2]*y[2]*y[3]
  ydot[2] = p[1]*y[1] - p[2]*y[2]*y[3] - p[3]*y[2]*y[2]
  ydot[3] = p[3]*y[2]*y[2]

  # ydot[1] = -0.04 * y[1] + 10000 * y[2] * y[3]
  # ydot[3] = 30000000 * y[2] * y[2]
  # ydot[2] = -ydot[1] - ydot[3]

  ydot

}

# R code to genrate time vector, IC and solve the equations
time_vec <- c(0.0, 0.4, 4.0, 40.0, 4E2, 4E3, 4E4, 4E5, 4E6, 4E7, 4E8, 4E9, 4E10)
IC <- c(1,0,0)
params <- c(0.04, 10000, 30000000)
reltol <- 1e-04
abstol <- c(1e-8,1e-14,1e-6)

## Solving the ODEs using cvode function
df1 <- cvode(time_vec, IC, ODE_R , params, reltol, abstol)           ## using R

## Solving with a manual Jacobian  J[i,j] = d(ydot_i)/d(y_j)
JAC_R <- function(t, y, p) {
  matrix(c(
    -p[1],         p[1],                        0,
     p[2]*y[3],   -p[2]*y[3] - 2*p[3]*y[2],   2*p[3]*y[2],
     p[2]*y[2],   -p[2]*y[2],                  0
  ), nrow = 3, ncol = 3)
}
df3 <- cvode(time_vec, IC, ODE_R, params, reltol, abstol, jacobian = JAC_R)

## ODEs can also be described using Rcpp. Compiling the C++ version invokes the
## toolchain, which takes far longer than the solve itself, so it is not run here.
## Not run:
Rcpp::sourceCpp(code = '

                #include <Rcpp.h>
```

```
                using namespace Rcpp;

                // ODE functions defined using Rcpp
                // [[Rcpp::export]]
                NumericVector ODE_Rcpp (double t, NumericVector y, NumericVector p){

                // Initialize ydot filled with zeros
                NumericVector ydot(y.length());

                ydot[0] = -p[0]*y[0] + p[1]*y[1]*y[2];
                ydot[1] = p[0]*y[0] - p[1]*y[1]*y[2] - p[2]*y[1]*y[1];
                ydot[2] = p[2]*y[1]*y[1];

                return ydot;

                }')

df2 <- cvode(time_vec, IC, ODE_Rcpp , params, reltol, abstol)        ## using Rcpp

## Check that both solutions are identical
# identical(df1, df2)

## End(Not run)
```

---

cvodes *cvodes*

Description

CVODES solver to solve ODEs and calculate sensitivities

Usage

```
cvodes(
  time_vector,
  IC,
  input_function,
  Parameters,
  reltolerance = 1e-04,
  abstolerance = 1e-04,
  SensType = "STG",
  ErrCon = "F",
  jacobian = NULL,
  sensitivity = NULL
)
```

Arguments

time_vector	time vector

IC	Initial Conditions
input_function	Right Hand Side function of ODEs
Parameters	Parameters input to ODEs
reltolerance	Relative Tolerance (a scalar, default value = 1e-04)
abstolerance	Absolute Tolerance (a scalar or vector with length equal to ydot, default = 1e-04)
SensType	Sensitivity Type - allowed values are "STG" (for Staggered, default) or "SIM" (for Simultaneous)
ErrCon	Error Control - allowed values are TRUE or FALSE (default)
jacobian	(Optional) Jacobian of the RHS with signature `function(t, y, p)`. Default is NULL
sensitivity	(Optional) Sensitivity right-hand side with signature `function(t, y, ydot, iS, yS, p)` returning the derivative `d(yS_iS)/dt = J %*% yS_iS + df/dp_iS` as a numeric vector of `length(y)`, where `iS` is the 1-based parameter index. Default is NULL, in which case the sensitivity equations are approximated by finite differences of the RHS

Value

A Matrix. First column is the time-vector, the next y * p columns are sensitivities of y1 w.r.t all parameters, then y2 w.r.t all parameters etc. y is the state vector, p is the parameter vector

Examples

```
# Example of solving a set sensitivity equations for ODEs with cvodes function
# ODEs described by an R function
ODE_R <- function(t, y, p){

  # vector containing the right hand side gradients
  ydot = vector(mode = "numeric", length = length(y))

  # R indices start from 1
  ydot[1] = -p[1]*y[1] + p[2]*y[2]*y[3]
  ydot[2] = p[1]*y[1] - p[2]*y[2]*y[3] - p[3]*y[2]*y[2]
  ydot[3] = p[3]*y[2]*y[2]

  # ydot[1] = -0.04 * y[1] + 10000 * y[2] * y[3]
  # ydot[3] = 30000000 * y[2] * y[2]
  # ydot[2] = -ydot[1] - ydot[3]

  ydot

}

# R code to genrate time vector, IC and solve the equations
time_vec <- c(0.0, 0.4, 4.0, 40.0, 4E2, 4E3, 4E4, 4E5, 4E6, 4E7, 4E8, 4E9, 4E10)
IC <- c(1,0,0)
params <- c(0.04, 10000, 30000000)
reltol <- 1e-04
abstol <- c(1e-8,1e-14,1e-6)
```

```
## Solving the ODEs and Sensitivities using cvodes function
df1 <- cvodes(time_vec, IC, ODE_R , params, reltol, abstol,"STG",FALSE)         ## using R

## Solving with a manual Jacobian  J[i,j] = d(ydot_i)/d(y_j)
JAC_R <- function(t, y, p) {
  matrix(c(
    -p[1],         p[1],                          0,
     p[2]*y[3],   -p[2]*y[3] - 2*p[3]*y[2],   2*p[3]*y[2],
     p[2]*y[2],   -p[2]*y[2],                  0
  ), nrow = 3, ncol = 3)
}
df3 <- cvodes(time_vec, IC, ODE_R, params, reltol, abstol, "STG", FALSE, jacobian = JAC_R)

## Solving with a manual sensitivity right-hand side. CVODES calls it once per
## parameter with the 1-based index iS, and it returns
##   d(yS_iS)/dt = J %*% yS_iS + df/dp_iS  (a vector of length(y))
## This avoids the finite-difference approximation used when sensitivity = NULL.
SENS_R <- function(t, y, ydot, iS, yS, p) {
  J <- matrix(c(
    -p[1],         p[1],                          0,
     p[2]*y[3],   -p[2]*y[3] - 2*p[3]*y[2],   2*p[3]*y[2],
     p[2]*y[2],   -p[2]*y[2],                  0
  ), nrow = 3, ncol = 3)
  dfdp <- switch(iS,
                 c(-y[1],      y[1],       0),   # d f / d p1
                 c( y[2]*y[3], -y[2]*y[3], 0),   # d f / d p2
                 c( 0,        -y[2]^2,     y[2]^2))  # d f / d p3
  as.numeric(J %*% yS + dfdp)
}
df4 <- cvodes(time_vec, IC, ODE_R, params, reltol, abstol, "STG", FALSE, sensitivity = SENS_R)

## ODEs can also be described using Rcpp. Compiling the C++ version invokes the
## toolchain, which takes far longer than the solve itself, so it is not run here.
## Not run:
Rcpp::sourceCpp(code = '

                #include <Rcpp.h>
                using namespace Rcpp;

                // ODE functions defined using Rcpp
                // [[Rcpp::export]]
                NumericVector ODE_Rcpp (double t, NumericVector y, NumericVector p){

                // Initialize ydot filled with zeros
                NumericVector ydot(y.length());

                ydot[0] = -p[0]*y[0] + p[1]*y[1]*y[2];
                ydot[1] = p[0]*y[0] - p[1]*y[1]*y[2] - p[2]*y[1]*y[1];
                ydot[2] = p[2]*y[1]*y[1];

                return ydot;
```

```
                }')

df2 <- cvodes(time_vec, IC, ODE_Rcpp , params, reltol, abstol,"STG",FALSE)      ## using Rcpp

## Check that both solutions are identical
# identical(df1, df2)

## End(Not run)
```

---

cvsolve *cvsolve*

Description

CVSOLVE solver to solve stiff ODEs with discontinuities

Usage

```
cvsolve(
  time_vector,
  IC,
  input_function,
  Parameters,
  Events = NULL,
  reltolerance = 1e-04,
  abstolerance = 1e-04,
  jacobian = NULL
)
```

Arguments

time_vector	time vector
IC	Initial Conditions
input_function	Right Hand Side function of ODEs
Parameters	Parameters input to ODEs
Events	Discontinuities in the solution (a DataFrame, default value is NULL). Three columns, names ignored: the 1-based index of the state, the time of the discontinuity, and the value to add to that state at that time. The value is always added to the current value of the state, including at the initial time, so the initial conditions in IC are the starting point and an event at t = 0 adds to them.
reltolerance	Relative Tolerance (a scalar, default value = 1e-04)
abstolerance	Absolute Tolerance (a scalar or vector with length equal to ydot, default = 1e-04)
jacobian	(Optional) Jacobian of the RHS with signature function(t, y, p) returning an n-by-n matrix where entry [i,j] is d(ydot_i)/d(y_j). Default is NULL and SUNDIALS uses internal finite-difference approximation.

Value

A Matrix. First column is the time-vector, the other columns are values of y in order they are provided.

Examples

```
# Example of solving a set of ODEs with multiple discontinuities using cvsolve
# A simple One dimensional equation, y = -0.1 * y
# ODEs described by an R function
ODE_R <- function(t, y, p){

  # vector containing the right hand side gradients
  ydot = vector(mode = "numeric", length = length(y))

  # R indices start from 1
  ydot[1] = -p[1]*y[1]

  ydot

}

# R code to generate time vector, IC and solve the equations
TSAMP <- seq(from = 0, to = 100, by = 0.1)      # sampling time points
IC <- c(1)
params <- c(0.1)

# A dataset describing the dosing at times at which additions to y[1] are to be done
# Names of the columns don't matter, but they MUST be in the order of state index,
# times and Values at discontinuity.
TDOSE <- data.frame(ID = 1, TIMES = c(0, 10, 20, 30, 40, 50), VAL = 100)
df1 <- cvsolve(TSAMP, c(1), ODE_R, params)         # solving without any discontinuity
df2 <- cvsolve(TSAMP, c(1), ODE_R, params, TDOSE, 0.001, 0.001, NULL)  # solving with discontinuity

## Solving with a manual Jacobian  J[1,1] = d(ydot[1])/d(y[1]) = -p[1]
JAC_R <- function(t, y, p) matrix(-p[1], nrow = 1, ncol = 1)
df3 <- cvsolve(TSAMP, IC, ODE_R, params, jacobian = JAC_R)
df4 <- cvsolve(TSAMP, IC, ODE_R, params, TDOSE, jacobian = JAC_R)
```

---

ida *ida*

Description

IDA solver to solve stiff DAEs

Usage

```
ida(
  time_vector,
```

```
  IC,
  IRes,
  input_function,
  Parameters,
  reltolerance = 1e-04,
  abstolerance = 1e-04,
  jacobian = NULL
)
```

Arguments

time_vector	time vector
IC	Initial Value of y
IRes	Initial value of ydot
input_function	Right Hand Side function of DAEs
Parameters	Parameters input to ODEs
reltolerance	Relative Tolerance (a scalar, default value = 1e-04)
abstolerance	Absolute Tolerance (a scalar or vector with length equal to ydot, default = 1e-04)
jacobian	(Optional) Jacobian with signature `function(t, y, ydot, cj, p)` returning an n-by-n matrix of `dF/dy + cj*dF/dydot`. Default NULL.

Value

A Matrix. First column is the time-vector, the other columns are values of y in order they are provided.

Examples

```
# Example of solving a set of Differential Algebraic Equations (DAEs)
# with IDA function
# DAEs (residuals) described by an R function
DAE_R <- function(t, y, ydot, p) {
  # vector containing the residuals
  res <- vector(mode = "numeric", length = length(y))

  # R indices start from 1
  res[1] <- -0.04 * y[1] + 10000 * y[2] * y[3] - ydot[1]
  res[2] <- -res[1] - 30000000 * y[2] * y[2] - ydot[2]
  res[3] <- y[1] + y[2] + y[3] - 1.0

  res
}

# R code to genrate time vector, IC and solve the equations
time_vec <- c(0.0, 0.4, 4.0, 40.0, 4E2, 4E3, 4E4, 4E5, 4E6, 4E7, 4E8, 4E9, 4E10)
IC <- c(1, 0, 0)
IRes <- c(-0.4, 0.4, 0)
params <- c(0.04, 10000, 30000000)
reltol <- 1e-04
```

```
abstol <- c(1e-8, 1e-14, 1e-6)

## Solving the DAEs using the ida function
df1 <- ida(time_vec, IC, IRes, DAE_R, params, reltol, abstol) ## using R

## Solving with a manual Jacobian
## J[i,j] = dF_i/dy_j + cj * dF_i/dydot_j
##
## F1 = -0.04*y1 + 1e4*y2*y3 - ydot1
## F2 = 0.04*y1 - 1e4*y2*y3 - 3e7*y2^2 - ydot2
## F3 = y1 + y2 + y3 - 1  (algebraic constraint)
DAE_jac <- function(t, y, ydot, p) {
  res <- numeric(length(y))
  f1 <- -0.04 * y[1] + 10000 * y[2] * y[3]
  res[1] <- f1 - ydot[1]
  res[2] <- -f1 - 30000000 * y[2] * y[2] - ydot[2]
  res[3] <- y[1] + y[2] + y[3] - 1.0
  res
}
JAC_IDA <- function(t, y, ydot, cj, p) {
  matrix(
    c(
      -0.04 - cj,      0.04,                                  1,
      10000 * y[3],   -10000 * y[3] - 60000000 * y[2] - cj,   1,
      10000 * y[2],   -10000 * y[2],                          1
    ),
    nrow = 3, ncol = 3
  )
}
df3 <- ida(time_vec, IC, IRes, DAE_jac, params, reltol, abstol, jacobian = JAC_IDA)

## DAEs can also be described using Rcpp. Compiling the C++ version invokes the
## toolchain, which takes far longer than the solve itself, so it is not run here.
## Not run:
Rcpp::sourceCpp(
  code = '

      #include <Rcpp.h>
      using namespace Rcpp;

      // ODE functions defined using Rcpp
      // [[Rcpp::export]]
      NumericVector DAE_Rcpp (double t, NumericVector y,
                              NumericVector ydot, NumericVector p){

      // Initialize ydot filled with zeros
      NumericVector res(y.length());

      res[0] = -0.04 * y[0] + 10000 * y[1] * y[2];
      res[1] = -res[0] - 30000000 * y[1] * y[1] - ydot[1];
      res[0] = res[0] - ydot[0];
      res[2] = y[0] + y[1] + y[2] - 1.0;
```

```
      return res;

      }'
)

df2 <- ida(time_vec, IC, IRes, DAE_Rcpp, params, reltol, abstol) ## using Rcpp

## End(Not run)
```

Index

cvode, 2
cvodes, 4
cvsolve, 7

ida, 8