# Package 'r2sundials'

October 14, 2022

**Type** Package

**Title** Wrapper for 'SUNDIALS' Solving ODE and Sensitivity Problem

**Version** 5.0.0-10

**Date** 2021-05-17

**Maintainer** Serguei Sokol <sokol@insa-toulouse.fr>

**Description** Wrapper for widely used 'SUNDIALS' software (SUite of Nonlinear and DIfferential/ALgebraic Equation Solvers) and more precisely to its 'CVODES' solver. It is aiming to solve ordinary differential equations (ODE) and optionally pending forward sensitivity problem. The wrapper is made 'R' friendly by allowing to pass custom parameters to user's callback functions. Such functions can be both written in 'R' and in 'C++' ('RcppArmadillo' flavor). In case of 'C++', performance is greatly improved so this option is highly advisable when performance matters. If provided, Jacobian matrix can be calculated either in dense or sparse format. In the latter case 'rmumps' package is used to solve corresponding linear systems. Root finding and pending event management are optional and can be specified as 'R' or 'C++' functions too. This makes them a very flexible tool for controlling the ODE system during the time course simulation. 'SUNDIALS' library was published in Hindmarsh et al. (2005) <doi:10.1145/1089014.1089020>.

**BugReports** https://github.com/sgsokol/r2sundials/issues

**Depends** R (>= 3.0.2), rmumps (>= 5.2.1-6)

**License** GPL (>= 2)

**Imports** Rcpp (>= 1.0.0)

**LinkingTo** Rcpp, RcppArmadillo, rmumps (>= 5.2.1-6)

**Suggests** RcppXPtrUtils, slam, RUnit, deSolve, RcppArmadillo

**RoxygenNote** 7.1.1

**Encoding** UTF-8

**NeedsCompilation** yes

**Biarch** FALSE

**Author** Serguei Sokol [cre, aut],
Carol S. Woodward [ctb],
Daniel R. Reynolds [ctb],
Alan C. Hindmarsh [ctb],

David J. Gardner [ctb],
Cody J. Balos [ctb],
Radu Serban [ctb],
Scott D. Cohen [ctb],
Peter N. Brown [ctb],
George Byrne [ctb],
Allan G. Taylor [ctb],
Steven L. Lee [ctb],
Keith E. Grant [ctb],
Aaron Collier [ctb],
Lawrence E. Banks [ctb],
Steve Smith [ctb],
Cosmin Petra [ctb],
Slaven Peles [ctb],
John Loffeld [ctb],
Dan Shumaker [ctb],
Ulrike Yang [ctb],
James Almgren-Bell [ctb],
Shelby L. Lockhart [ctb],
Hilari C. Tiedeman [ctb],
Ting Yan [ctb],
Jean M. Sexton [ctb],
Chris White [ctb],
Lawrence Livermore National Security [cph],
Southern Methodist University [cph],
INRAE [cph]

**Repository** CRAN

**Date/Publication** 2021-05-17 23:20:02 UTC

# R topics documented:



---

`r2sundials-package` *Wrapper for 'SUNDIALS' Solving ODE and Sensitivity Problem*

---

## Description

Wrapper for widely used 'SUNDIALS' software (SUite of Nonlinear and DIfferential/ALgebraic Equation Solvers) and more precisely to its 'CVODES' solver. It is aiming to solve ordinary differential equations (ODE) and optionally pending forward sensitivity problem. The wrapper is made 'R' friendly by allowing to pass custom parameters to user's callback functions. Such functions

can be both written in 'R' and in 'C++' ('RcppArmadillo' flavor). In case of 'C++', performance is greatly improved so this option is highly advisable when performance matters. If provided, Jacobian matrix can be calculated either in dense or sparse format. In the latter case 'rmumps' package is used to solve corresponding linear systems. Root finding and pending event management are optional and can be specified as 'R' or 'C++' functions too. This makes them a very flexible tool for controlling the ODE system during the time course simulation. 'SUNDIALS' library was published in Hindmarsh et al. (2005) <doi:10.1145/1089014.1089020>. Interface for commonly used SUNDIALS library for solving ODE system. It is made R friendly by allowing to pass custom parameters to user's callback functions. They can be both written in R and in C++ (Rcpp flavor). In case of C++, performance is greatly improved so this option is highly advisable when performance matters. Jacobian matrix can be specified as either dense or sparse matrix. In the latter case **rmumps** package is used to solve corresponding linear systems. Root finding and corresponding event management are optional and can be too specified as R or C++ functions which makes them a very flexible tool for controlling the ODE system during the time course simulation.

## Details

The DESCRIPTION file:

| | |
|---|---|
| Package: | r2sundials |
| Type: | Package |
| Title: | Wrapper for 'SUNDIALS' Solving ODE and Sensitivity Problem |
| Version: | 5.0.0-10 |
| Date: | 2021-05-17 |
| Authors@R: | c( person("Serguei", "Sokol", role=c("cre", "aut"), email="sokol@insa-toulouse.fr"), person("Carol S.", |
| Maintainer: | Serguei Sokol <sokol@insa-toulouse.fr> |
| Description: | Wrapper for widely used 'SUNDIALS' software (SUite of Nonlinear and DIfferential/ALgebraic Equat |
| BugReports: | https://github.com/sgsokol/r2sundials/issues |
| Depends: | R (>= 3.0.2), rmumps (>= 5.2.1-6) |
| License: | GPL (>=2) |
| Imports: | Rcpp (>= 1.0.0) |
| LinkingTo: | Rcpp, RcppArmadillo, rmumps (>= 5.2.1-6) |
| Suggests: | RcppXPtrUtils, slam, RUnit, deSolve, RcppArmadillo |
| RoxygenNote: | 7.1.1 |
| Encoding: | UTF-8 |
| NeedsCompilation: | yes |
| Biarch: | FALSE |
| Author: | Serguei Sokol [cre, aut], Carol S. Woodward [ctb], Daniel R. Reynolds [ctb], Alan C. Hindmarsh [ctb], |

Index of help topics:

```
get_cnst                Get Predefined Constant
r2cvodes                Solving ODE System and Sensitivity Equations
r2sundials-package      Wrapper for 'SUNDIALS' Solving ODE and
                        Sensitivity Problem
```

The main function of the package is r2cvodes() which wraps and converts its arguments in data structures and parameters convenient for calling `cvodes()` from SUNDIALS library.

When using r2sundials::r2cvodes(), some callback functions have to be written by the user either in R or in C++ (RcppArmadillo flavor). One of them is mandatory and defines the rhs of the ODE system to solve. Others are optional and can be used to

- calculate Jacobian matrix in sparse or dense format;
- calculate root vector for tracking particular events and
- handle them in a way defined by the user himself;
- calculate sensitivity rhs if sensitivity to some parameters is required and user does not want to rely on internal procedures for estimating such rhs.

Note that if 'SUNDIALS' 'CVODES' is compiled with parameter SUNDIALS_INDEX_SIZE set to 32, some memory copying is spared if a C++ function calculating sparse Jacobian is provided by user.

The version number of this package if made of CVODES original version, e.g. 5.0.0 followed by one digit for R wrapper release. This can give 5.0.0-1.

## Author(s)

NA Maintainer: Serguei Sokol <sokol@insa-toulouse.fr>

## References

Original SUNDIALS CVODES user documentation (search for cvs_guide.pdf)

## See Also

deSolve

## Examples

```
# a very simple ODE for exponential transition from 0 to 1: y'=1-y, y(0)=0
frhs=function(t, y, param, psens) 1-y
ti=seq(0, 5, length.out=101)
y0=0
res_exp=r2sundials::r2cvodes(y0, ti, frhs)
plot(ti, res_exp[1,], t="l", xlab="Time", ylab="Y")
```

---

# get_cnst *Get Predefined Constant*

---

## Description

This function returns numerical value of predefined named constants. Constants defined for this package are:

- CV_ADAMS
- CV_BDF

- CV_SIMULTANEOUS
- CV_STAGGERED
- CV_STAGGERED1
- CV_SUCCESS
- R2SUNDIALS_EVENT_HOLD
- R2SUNDIALS_EVENT_IGNORE
- R2SUNDIALS_EVENT_STOP

All these constants are exported from the package and available for use in callback functions written in R.

## Usage

```
get_cnst(s)
```

## Arguments

| | |
|---|---|
| s | character scalar, name of the constant whose value is to get. |

## Value

integer scalar, the value of the constant named by s

## Examples

```
get_cnst("CV_SUCCESS")
# [1] 0
```

---

r2cvodes *Solving ODE System and Sensitivity Equations*

---

## Description

r2cvodes sets up necessary structures and calls cvodes() from SUNDIALS library to solve user defined ODE system $y' = f(t, y, p)$, $y(t0) = y0$, where $p$ is a constant parameter vector. If requested, corresponding forward sensitivity equations $s'[i] = df/dys[i] + df/dp[i]$, $s[i](t0) = dy0(p)/dp[i]$ (here $s[i](t) = dy(t)/dp[i]$) can be solved simultaneously with the original ODE system. Root finding and proceeding can be defined as well.

## Usage

```
r2cvodes(
  yv,
  times,
  frhs,
  param = NULL,
  tstop = as.numeric(c()),
  abstol = 1e-08,
  reltol = 1e-08,
  integrator = as.integer(c()),
  maxord = 0L,
  maxsteps = 0L,
  hin = 0,
  hmax = 0,
  hmin = 0,
  constraints = as.numeric(c()),
  fjac = NULL,
  nz = 0L,
  rmumps_perm = as.integer(c()),
  nroot = 0L,
  froot = NULL,
  fevent = NULL,
  Ns = 0L,
  psens = as.numeric(c()),
  sens_init = as.numeric(c()),
  psens_bar = as.numeric(c()),
  psens_list = as.integer(c()),
  fsens = NULL,
  fsens1 = NULL,
  sens_method = as.integer(c()),
  errconS = TRUE
)
```

## Arguments

| | |
|---|---|
| yv | const numeric vector, initial values of state vector ($y0$). Its length defines the number of equations in the ODE system and is refered hereafter as 'Neq'. |
| times | const numeric vector, time point values at which the solution is stored |
| frhs | R function or XPtr pointer to Rcpp function, defines rhs $f(t, y, p)$, i.e. returns (or fills in place) first derivative vector of length Neq (cf. details for argument list) |
| param | any R object (default NULL), parameter object (can be a vector, list, environment, ...) passed through to user defined functions namely to frhs. It can be useful to get the code more readable as named parameters are authorized. (Starting from this one, the following parameters are optional) |
| tstop | const numeric scalar (default as.numeric(c())) if non empty, defines a time point beyond which the calculation is not made (e.g. out of definition domain). |

If a vector is given, only the first value is used.

`abstol` const numeric scalar (default `1.e-8`), absolute tolerance in ODE solving (used for values which are close to 0)

`reltol` const double (default `1.e-8`), relative tolerance in ODE solving (used for values which are not so close to 0)

`integrator` integer scalar (default `as.integer(c())`), defines which integration scheme should be used by cvodes: implicit (`CV_BDF`) or explicit (`CV_ADAMS`). Default empty vector is equivalent to `CV_BDF`. Constants `CV_BDF` and `CV_ADAMS` (as other mentioned constants) are defined by the package and are available for user.

`maxord` const integer scalar (default `0`), defines maximal order of time scheme. Default 0 values is equivalent to 5 for `integrator=CV_BDF` or to 12 for `integrator=CV_ADAMS`

`maxsteps` const integer scalar (default `0`), maximum of internal steps before reaching next time point from 'times'. Default 0 is equivalent to 500.

`hin` const numeric scalar (default `0`), value of the initial step size to be attempted. Default 0.0 corresponds to `cvodes`' default value which is internally estimated at $t0$.

`hmax` const numeric scalar (default `0`), maximum absolute value of the time step size (>= 0.0). The default 0 value is equivalent to Inf.

`hmin` const numeric scalar (default `0`), minimum absolute step size (>= 0.0)

`constraints` const numeric vector (default `as.numeric(c())`), if non empty, defines constraint flags.
If constraints[i] is
0 then no constraint is imposed on $y[i]$;
1.0 then $y[i]$ will be constrained to be $y[i]$ >= 0.0;
-1.0 then $y[i]$ will be constrained to be $y[i]$ =< 0.0.;
2.0 then $y[i]$ will be constrained to be $y[i]$ > 0.0.;
-2.0 then $y[i]$ will be constrained to be $y[i]$ < 0.0.
If a time step respecting hmin and avoiding constraint violation cannot be found, an error is triggered.

`fjac` R function of XPtr pointer to Rcpp function (default `NULL`), users supplied function that calculates Jacobian matrix df/dy which can be dense or sparse (cf. details for parameter list). If `fjac` is not supplied, a SUNDIALS internal approximation to Jacobian is used when needed.

`nz` const integer scalar (default `0`), number of non zero elements in Jacobian. If > 0, this parameter triggers a sparse Jacobian usage. In this case, the previous parameter `fjac`, must be a function (or a pointer to a function) with appropriate parameter list (cf. details) filling a sparse matrix. It is advised to have `nz` accounting not only for non zeros in the Jacobian itself but also for diagonal terms even if they are zeros. In this case, a supplementary memory allocation can be avoided. If a sparse Jacobian is used, corresponding sparse linear system is solved with the help of **rmumps** package.

`rmumps_perm` integer scalar (default `as.integer(c())`), defines permutation method that will be used by **rmumps** during symbolic analysis before solving sparse linear systems. Default value is equivalent to `RMUMPS_PERM_AUTO`. Possible values (defined in **rmumps** package) are `RMUMPS_PERM_AMF`, `RMUMPS_PERM_AMD`, `RMUMPS_PERM_AUTO`,

RMUMPS_PERM_QAMD, RMUMPS_PERM_PORD, RMUMPS_PERM_METIS, RMUMPS_PERM_SCOTCH. An appropriate choice of permutation type can reduce memory requirements as well as calculation speed for sparse system solving. If a vector is supplied, only the first value is read. If nz=0, this parameter is not used.

nroot — const integer scalar (default 0), defines number of roots that user wishes to track.

froot — R function of XPtr pointer to Rcpp function (default NULL), user defined function calculating root vector of length nroot. When at least one component of this vector crosses 0, a root finding event is triggered and corresponding root handling function is called (cf. fevent). This defines a very flexible root tracking mechanism. User can define a root based both on time values and state vector (y).

fevent — R function of XPtr pointer to Rcpp function (default NULL), defines function for root proceeding. When a root is encountered, this function is called. Its return value defines what to do with this root. User's reaction (i.e. the return value of fevent) can be one of three types:
R2SUNDIALS_EVENT_IGNORE - do nothing (can be helpful if, for example, 0 is crossed not in pertinent sens);
R2SUNDIALS_EVENT_HOLD - the time point at which the root happened as well as the corresponding rootsfound vector are added to the root matrix delivered with output attributes (cf. details). This time point is also added to the ODE solution which can lead to a new time point, originally absent in 'times' vector;
R2SUNDIALS_EVENT_STOP - stops the ODE solving. If it happens before reaching the last value in 'times', the ODE solution can have less time points than initially defined in 'times' parameter. fevent is allowed to modify the vector state y in an arbitrary way. It can be helpful for modeling some controlling events for example adding some compound to chemical mix or modifying speed vector if an obstacle is hitted. If such modification takes place, the ODE system is restarted to handle in appropriate way induced discontinuities.

Ns — const integer scalar (default 0), number of parameters for which forward sensitivity system has to be solved.

psens — numeric vector (default as.numeric(c())), if not empty, defines a vector of parameters for which (or for some components of which) forward sensitivity system has to be solved. If its length > Ns, then vector psens_list of length Ns must define which are the Ns components of psens with respect to which sensitivities are to be computed. Note that parameters used in sensitivity calculations must be put in psens vector (and not somewhere in 'param' object) only if user wish to rely on SUNDIALS internal procedure for estimation of sensitivity rhs. If user provides its own function for sensitivity rhs, he is free to use either of 'param' or psens for passing and accessing sensitivity parameters.

sens_init — numeric matrix (default as.numeric(c())), matrix with Neq rows and Ns columns defining initial condition for sensititivy system. Default value is equivalent to Neq x Ns zero matrix which means that $y0$ is not dependent on $p$.

psens_bar — numeric vector (default as.numeric(c())), a vector of Ns positive scaling factors. The values provided in this vector, describe orders of magnitude for psens components. Having such estimations can help to increase the accuracy for sensitivity vector calculations or for the right-hand side calculation for the sensitiv-

ity systems based on the internal difference-quotient function. If not defined, it is equivalent to set all psens_bar components to 1

psens_list — const integer vector (default as.integer(c())), if length(psens) > Ns, this index vector can be used to indicate which are components of psens that are concerned with sensitivity calculations. Default value is valid only when length(psens)==Ns, i.e. psens_list is not used. Values in psens_list are 1-based (as regular integer indexes in R)

fsens — R function of XPtr pointer to Rcpp function (default NULL), user defined sensitivity rhs for all psens at once.

fsens1 — R function of XPtr pointer to Rcpp function (default NULL), user defined sensitivity rhs for is-th psens component (is is passed as input parameter to fsens1, cf. details). Only one of fsens or fsens1 can be supplied. If none of them is supplied, SUNDIALS' internal difference-quotient function is used. In this case, sensitivity parameters must be passed only through psens argument.

sens_method — integer scalar (default as.integer(c())), constant defining the method used for solving sensitivity system. Allowed values are CV_SIMULTANEOUS or CV_STAGGERED. Default value is equivalent to CV_SIMULTANEOUS.

- CV_SIMULTANEOUS means that the state and sensitivity variables are corrected at the same time.
- CV_STAGGERED means that the correction step for the sensitivity variables takes place at the same time for all sensitivity equations, but only after the correction of the state variables has converged and the state variables have passed the local error test

errconS — constant logical scalar (default TRUE), specifies whether sensitivity variables are to be included (TRUE) or not (FALSE) in the error control mechanism

## Details

The package **r2sundials** was designed to avoid as much as possible memory reallocation in callback functions (frhs and others). C++ variants of these functions are fully compliant with this design principle. While R counterparts are not (as per R design). Here, we define callback function interfaces that user has to abide to. Pointers to C++ variants to be passed to r2cvodes() can be obtained with the help of **RcppXPtrUtils**. See examples for illustrations of such use.
Right hand side function frhs provided by user calculates derivative vector $y'$. This function can be defined as classical R function or a Rcpp/RcppArmadillo function. In the first case, it must have the following list of input arguments
frhs(t, y, param, psens)
and return a derivative vector of length Neq. Here t is time point (numeric scalar), y current state vector (numeric vector of length Neq), param and psens are passed through from r2cvodes() arguments.
In the C++ case, it is defined as
int (*frhs)(double t, const vec &y, vec &ydot, RObject &param, NumericVector &psens)
and return an integer status flag, e.g. CV_SUCCESS. For other possible status flags see the original SUNDIALS documentation (search for cvs_guide.pdf). The derivatives are stored in-place in ydot vector. See examples section for a usage sample.
fjac is a function calculating Jacobian matrix. Its definition varies depending on 1) kind of used Jacobian: dense or sparse and 2) on programming language used: R or C++ (i.e. Rcpp/RcppArmadillo).

- For dense Jacobian calculated in R, the arguments are:
  `fjac(t, y, ydot, param, psens)`
  and the expected return value is `Neq` x `Neq` dense Jacobian matrix df/dy.
- For dense Jacobian calculated in C++ the definition is following:
  `int (*fjac)(double t, const vec &y, const vec &ydot, mat &J, RObject &param, NumericVector &psens, vec &tmp1, vec &tmp2, vec &tmp3)`
  It must return a status flag. The resulting Jacobian is stored in-place in the the matrix `J`. Auxiliary vectors `tmp1` to `tmp3` are of length `Neq` and are made available for intermediate storage thus avoiding memory reallocation at each call to `fjac()`.
- For sparse Jacobian calculated in R, the arguments are:
  `fjac(t, yv, ydotv, param, psens)`
  The return value is a list with fields `i` (row indices), `p` (column pointers) and `v` (matrix values) defining the content of sparse Jacobian in CSC (condensed sparse column) format. The values stored in i and p vectors are supposed to be 1-based, as it is common in R language.
- For sparse Jacobian calculated in C++ the definition is following:
  `int (*fjac)(double t, const vec &y, const vec &ydot, uvec &i, uvec &p, vec &v, int n, int nz, RObject &param, NumericVector &psens, vec &tmp1, vec &tmp2, vec &tmp3)`
  here `n=Neq`, `nz` is passed through from r2cvodes() arguments. The resulting sparse Jacobian is stored in-place in vectors `i`, `p`, `v` corresponding to the CSC (Compressed Sparse Column) format. Their respective dimensions are `nz`, `n+1` and `nz`. The values stored in `i` and `p` must be 0 based as per usage in C++. The return value is a status flag.

`froot` calculates a root vector, i.e. a vector whose components are tracked for 0 crossing during the time course in ODE solving. If written in R, its call follows the following pattern:
`froot(t, y, param, psens)`
and it must return a numeric vector of length 'nroot'. If written in C++, it is defined as
`int (*froot)(double t, const vec &y, vec &vroot, RObject &param, NumericVector &psens)`
The tracked values are stored in-place in `vroot`. The returned value is a status flag.
`fevent` handles the event of root finding. If written in R, the calling pattern is
`fevent(t, yvec, Ns, ySm, rootsfound, param, psens)`
and the return value is a list with named component "ynew", "flag" and "sens_init". The last item is required only when `Ns > 0`. Current value of sensitivity matrix $dy/dp$ can be found in parameter `ySm`. Integer vector 'rootsfound' of length 'nroot' provides information on `vroot` components that triggered the root event. If `rootsfound[i] != 0`, it means that `vroot[i]` is a root otherwise it is not. Moreover, the sign of `rootsfound[i]` is meaningful. If `rootsfound[i] > 0` then `vroot[i]` is increasing at 0 crossing. Respectively, if `rootsfound[i] < 0` then `vroot[i]` is decreasing. The vector 'ynew' in the output list can define a new state vector after event handling (for example, an abrupt change in velocity direction and/or magnitude after an obstacle hit). The field 'flag' in the output list is authorized to take only three values: `R2SUNDIALS_EVENT_IGNORE`, `R2SUNDIALS_EVENT_HOLD` and `R2SUNDIALS_EVENT_STOP` described here-before. The matrix `sens_init` is used for a possible restarting of sensitivity calculation. It must contain a derivative matrix $dynew/dp$ of size `Neq` x `Ns`. If this field is absent (NULL) then a zero matrix is assumed.
If written in C++, this function is defined as
`int (*fevent)(double t, const vec &y, vec &ynew, int Ns, std::vector<vec> &ySv, const ivec &rootsfound, RObject &param, NumericVector &psens)`
The new state vector can be stored in-place in `ynew` and the status flag indicating what to do with this event is the return value. If sensitivity calculation is going on (i.e. `Ns > 0`), the current value of sensitivity vectors can be found in `ySv` and their new values can be stored in-place in the same

parameter. Note that if ynew is different from the vale of y when the root was found, the ODE is restarted from this time point to handle correctly the discontinuity. As a result, there will be two columns corresponding to the same time point: one with the state vector at root finding and one with ynew values. The same is true for sensitivity output, if it is part of the problem.
fsens calculates rhs for sensitivity system. If written in R, it must be defined as
fsens(Ns, t, y, ydot, ySm, param, psens)
and return a dataframe in which i-th column correspond to $s'[i]$ sensitivity derivative vector. Among other parameters, it receives ySm which is a Neq x Ns matrix having the current values of sensitivity vector (i-th vector is in i-th column).
If written in C++, it has to be defined as
int (*fsens)(int Ns, double t, const vec &y, const vec &ydot, const std::vector<vec> &ySv, std::vector<vec> &ySdotv, RObject &param, NumericVector &psens, const vec &tmp1v, const vec &tmp2v)
Note a slight difference in the input parameters compared with the R counterpart. Here ySv plays the role of ySm and is not a matrix but a vector of Armadillo vectors. To access m-th component of $s[i]$, one can simply do ySv[i][m] and the whole $s[i]$ is selected as ySv[i]. Such data structure was retained to keep as low as possible new memory reallocation. The resulting sensitivity derivatives are to be stored in-place in ySdotv according to the same data organization scheme as in ySv. This function returns a status flag.
fsens1 does the same as fsens but provides derivatives of sensitivity vectors on one-by-one basis. This second form is provided for user's convenience as in some cases the code can become more readable if it calculates only one vector s'[i] at a time. If written in R, this function has to be defined as
fsens1(Ns, t, y, iS, ydot, yS, param, psens)
here iS is the index of calculated vector $s'[iS]$ and yS contains the current value of $s[iS]$. If written in C++, this functions has to be defined as
int (*fsens1)(int Ns, double t, const vec &yv, const vec &ydotv, int iS, const vec &ySv, vec &ySdotv, RObject &param, NumericVector &psens, const vec &tmp1v, const vec &tmp2v)
The result, i.e. $s'[iS]$ is to be stored in-place in ySdotv vector. This function returns a status flag.

## Value

numeric matrix, ODE solution where each column corresponds to a state vector at a given time point. The columns (their number is refered to as Nt) are named by time points while the rows heritates the names from yv. If no names are found in yv, the rows are simply named 'V1', 'V2' and so on. After a normal execution and without root handling, column number is equal to the length of times. However, if root handling is used, it can add or remove some time points from times. So the user must not assume that column number of output matrix is equal to length(times). Instead, actual number of time points for which the solution was calculated can be retrieved from an attribute named "times". Moreover, several attributes are defined in the returned matrix. We have mentioned "times", the others are:

**stats** Some statistics about various events that could happen during cvodes run like the number of rhs calls, Jacobian calls, number of internal time steps, failure number and so on. Any component <name> in stats vector corresponds to SUNDIALS function pattern CVodeGet<name>, e.g. "NumRhsEvals" was obtained with CVodeGetNumRhsEvals() call. For detailed meaning of each statistics, user is invited to refer to SUNDIALS documentation (search for cvs_guide.pdf);

**roots** matrix with row number nroot+1 and column number equal to number of roots found by the cvodes() and retained by the user. Each column is a composite vector made of time point

and rootsfound vector described here-before.

**sens** sensitivity 3D array with dimensions Neq x Nt x Ns

## Examples

```
# Ex.1. Solve a scalar ODE describing exponential transition form 0 to 1
# y'=-a*(y-1), y(0)=0, a is a parameter that we arbitrary choose to be 2.
# define rhs function (here in R).
frhs_exp=function(t, y, p, psens) -p$a*(y-1)
# define parameter list
p=list(a=2)
# define time grid from 0 to 5 (arbitrary units)
ti=seq(0, 5, length.out=101)
# define initial state vector
y0=0
# we are set for a very simple r2cvodes() call
res_exp=r2sundials::r2cvodes(y0, ti, frhs_exp, param=p)
# compare the result to theoretical values: 1-exp(-a*t)
stopifnot(diff(range(1-exp(-p$a*ti) - res_exp)) < 1.e-6)

# Ex. 2. Same problem but frhs is written in C++
library(RcppXPtrUtils)
ptr_exp=cppXPtr(code='
int rhs_exp(double t, const vec &y, vec &ydot, RObject &param, NumericVector &psens) {
  NumericVector p(param);
  ydot[0] = -p["a"]*(y[0]-1);
  return(CV_SUCCESS);
}
', depends=c("RcppArmadillo","r2sundials","rmumps"),
 includes="using namespace arma;\n#include <r2sundials.h>", cacheDir="lib", verbose=FALSE)
# For ease of use in C++, we convert param to a numeric vector instead of a list.
pv=c(a=p$a)
# new call to r2cvodes() with XPtr pointer ptr_exp.
res_exp2=r2sundials::r2cvodes(y0, ti, ptr_exp, param=pv)
stopifnot(diff(range(res_exp2 - res_exp)) < 1.e-14)

# Ex.3. Bouncing ball simulation.
# A ball falls from a height y=5 m with initial vertical speed vy=0 m/s
# and horizontal speed vx=1 m/s. The forces acting on the ball are 1) the gravity
# (g=9.81 m/s^2) and 2) air resistance f=-k_r*v (k_r=0.1 N*s/m).
# When the ball hits the ground, it bounces instantly retaining k=0.9 part
# of its vertical and horizontal speed. At the bounce, the vertical speed
# changes its sign to the opposite while horizontal speed keeps the original sign.
# Simulation should stop after the 5-th bounce or at tmax=10 s which ever comes first.
# This example illustrates usage of root finding and handling.
# We give here an example of callback functions for root handling in C++.
yv=c(x=0, y=5, vx=1, vy=0) # initial state vector
pv=c(g=9.81, k_r=0.1, k=0.9, nbounce=5, tmax=10) # parameter vector
ti=seq(0, 20, length.out=201L) # time grid

# rhs
ptr_ball=cppXPtr(code='
```

```
int rhs_ball(double t, const vec &y, vec &ydot, RObject &param, NumericVector &psens) {
  NumericVector p(param);
  ydot[0] = y[2]; // dx/dt=vx
  ydot[1] = y[3]; // dy/dt=vy
  ydot[2] = -p["k_r"]*y[2]; // dvx/dt= -k_r*vx
  ydot[3] = -p["g"] - p["k_r"]*y[3]; // dvy/dt=-g -k_r*vy
  return(CV_SUCCESS);
}
', depends=c("RcppArmadillo","r2sundials","rmumps"),
 includes="using namespace arma;\n#include <r2sundials.h>", cacheDir="lib", verbose=FALSE)

# root function
ptr_ball_root=cppXPtr(code='
int root_ball(double t, const vec &y, vec &vroot, RObject &param, NumericVector &psens) {
  NumericVector p(param);
  vroot[0] = y[1]; // y==0
  vroot[1] = t-p["tmax"]; // t==p["tmax"]
  return(0);
}
', depends=c("RcppArmadillo","r2sundials","rmumps"),
 includes="using namespace arma;\n#include <r2sundials.h>", cacheDir="lib", verbose=FALSE)

# event handler function
ptr_ball_event=cppXPtr(code='
int event_ball(double t, const vec &y, vec &ynew, int Ns, std::vector<vec> &ySv,
          const ivec &rootsfound, RObject &param, NumericVector &psens) {
  NumericVector p(param);
  static int nbounce=0;
  if (rootsfound[1] != 0) // time is out
     return(R2SUNDIALS_EVENT_STOP);
  if (rootsfound[0] > 0)
    // cross 0 in ascending sens, can happen when y < 0 in limits of abstol
    return(R2SUNDIALS_EVENT_IGNORE);
  ynew=y;
  if (++nbounce < p["nbounce"]) {
    // here nbounce=1:4
    ynew[2] *= p["k"]; // horizontal speed is lowered
    ynew[3] *= -p["k"]; // vertical speed is lowered and reflected
    return(R2SUNDIALS_EVENT_HOLD);
  } else {
    // here nbounce=5
    nbounce=0; // reinit counter for possible next calls to r2cvodes
    return(R2SUNDIALS_EVENT_STOP);
  }
}
', depends=c("RcppArmadillo","r2sundials","rmumps"),
 includes="using namespace arma;\n#include <r2sundials.h>", cacheDir="lib", verbose=FALSE)

# ODE solving and plotting
res_ball <- r2sundials::r2cvodes(yv, ti, ptr_ball, param=pv, nroot=2L,
  froot=ptr_ball_root, fevent=ptr_ball_event)
plot(res_ball["x",], res_ball["y",], xlab="X [m]", ylab="Y [m]",
  t="l", main="Bouncing ball simulation")
```

```
# Ex.4. Robertson chemical reactions
# This example is often used as an illustration and a benchmark for stiff ODE.
# We will demonstrate here:
# • how to use sparse Jacobian (not really meaningful for 3x3 sytem but just to give a hint);
# • how to make sensitivity calculations with user provided rhs.
#
# Let simulate the following chemical system of 3 compounds y1, y2 and y3
#  y1' = -k1*y1 + k3*y2*y3
#  y2' =  k1*y1 - k2*y2*y2 - k3*y2*y3
#  y3' =  k2*y2*y2
# Jacobian df/dy is
#
# | -k1 |      k3*y3       |  k3*y2 |
# |-----+-------------------+--------|
# |  k1 | -2*k2*y2 - k3*y3 | -k3*y2 |
# |-----+-------------------+--------|
# |  0  |     2*k2*y2      |   0    |


yv <- c(y1=1, y2=0, y3=0) # initial values
pv <- c(k1 = 0.04, k2 = 3e7, k3 = 1e4) # parameter vector
ti=10^(seq(from = -5, to = 11, by = 0.1)) # exponential time grid

# pointer to rhs function
ptr_rob=cppXPtr(code='
int rhs_rob(double t, const vec &y, vec &ydot, RObject &param, NumericVector &psens) {
  NumericVector p(param);
  ydot[0] = -p["k1"]*y[0] + p["k3"]*y[1]*y[2];
  ydot[2] = p["k2"]*y[1]*y[1];
  ydot[1] = -ydot[0] - ydot[2];
  return(CV_SUCCESS);
}
', depends=c("RcppArmadillo","r2sundials","rmumps"),
 includes="using namespace arma;\n#include <r2sundials.h>", cacheDir="lib", verbose=FALSE)
# pointer to sparse jacobian function
ptr_rob_jacsp=cppXPtr(code='
int spjac_rob(double t, const vec &y, const vec &ydot, uvec &ir, uvec &pj, vec &v, int n, int nz,
              RObject &param, NumericVector &psens, vec &tmp1, vec &tmp2, vec &tmp3) {
  if (nz < 8)
   stop("spjac_rob: not enough room for non zeros, must have at least 8, instead got %d", nz);
  NumericVector p(param);
  int i=0;
  pj[0] = 0; // init pj
  // first column
  ir[i] = 0;
  v[i++] = -p["k1"];
  ir[i] = 1;
  v[i++] = p["k1"];
  pj[1] = i;
  // second column
  ir[i] = 0;
  v[i++] = p["k3"]*y[2];
```

```
  ir[i] = 1;
  v[i++] = -p["k3"]*y[2]-2*p["k2"]*y[1];
  ir[i] = 2;
  v[i++] = 2*p["k2"]*y[1];
  pj[2] = i;
  // third column
  ir[i] = 0;
  v[i++] = p["k3"]*y[1];
  ir[i] = 1;
  v[i++] = -p["k3"]*y[1];
  ir[i] = 2;
  v[i++] = 0; // just to have the main diagonal fully in Jacobian
  pj[3] = i;
  return(0);
}
', depends=c("RcppArmadillo","r2sundials","rmumps"),
 includes="using namespace arma;\n#include <r2sundials.h>", cacheDir="lib", verbose=FALSE)
# pointer to sensitivity rhs function
ptr_rob_sens1=cppXPtr(code='
int sens_rob1(int Ns, double t, const vec &y, const vec &ydot, int iS, const vec &yS, vec &ySdot,
              RObject &param, NumericVector &psens, vec &tmp1, vec &tmp2) {
  // calculate (df /dy)s_i(t) + (df /dp_i) for i = iS
  NumericVector p(param);
  // (df/dy)s_i(t)
  ySdot[0] = -p["k1"]*yS[0] + p["k3"]*y[2]*yS[1] + p["k3"]*y[1]*yS[2];
  ySdot[1] = p["k1"]*yS[0] - (p["k3"]*y[2]+2*p["k2"]*y[1])*yS[1] - p["k3"]*y[1]*yS[2];
  ySdot[2] = 2*p["k2"]*y[1]*yS[1];
  // + (df/dp_i)
  switch(iS) {
    case 0:
      ySdot[0] -= y[0];
      ySdot[1] += y[0];
      break;
    case 1:
      ySdot[1] -= y[1]*y[1];
      ySdot[2] += y[1]*y[1];
      break;
    case 2:
      ySdot[0] += y[1]*y[2];
      ySdot[1] -= y[1]*y[2];
  }
  return(CV_SUCCESS);
}
', depends=c("RcppArmadillo","r2sundials","rmumps"),
 includes="using namespace arma;\n#include <r2sundials.h>", cacheDir="lib", verbose=FALSE)
# Note that we don't use psens param for sensitivity calculations as we provide our own fsens1.
res_rob <- r2sundials::r2cvodes(yv, ti, ptr_rob, param=pv, nz=8, fjac=ptr_rob_jacsp, Ns=3,
                                fsens1=ptr_rob_sens1)
# plot ODE solution
layout(t(1:3)) # three sublots in a row
for (i in 1:3)
   plot(ti, res_rob[i,], log="x", t="l", xlab="Time", ylab=names(yv)[i])
# plot sensitivities
```

```
layout(matrix(1:9, nrow=3)) # 9 subplots in a square
for (j in 1:3) # run through pv
   for (i in 1:3) # run through y
      plot(ti, attr(res_rob, "sens")[i,,j], log="x", t="l", xlab="Time",
           ylab=parse(text=paste0("partialdiff*y[", i, "]/partialdiff*k[", j, "]")))
```

# Index