Package 'paropt'

February 22, 2024

Type Package

Title Parameter Optimizing of ODE-Systems

Version 0.3.3

Date 2024-02-20

Author Krämer Konrad [aut, cre],
Krämer Johannes [aut],
Heyer Arnd [ths],
University of Stuttgart [uvp],
Institute of Biomaterials and Biomolecular Systems at the University of Stuttgart [his]
| file AUTHORS

Maintainer Krämer Konrad <Konrad_kraemer@yahoo.de>

BugReports https://github.com/Konrad1991/paropt

Description Enable optimization of parameters of ordinary differential equations.
Therefore, using 'SUNDIALS' to solve the ODE-System (see Hindmarsh, Alan C., Peter N. Brown, Keith E. Grant, Steven L. Lee, Radu Serban, Dan E. Shumaker, and Carol S. Woodward. (2005) <doi:10.1145/1089014.1089020>).
Furthermore, for optimization the particle swarm algorithm is used (see: Akman, Devin, Olcay Akman, and Elsa Schaefer. (2018) <doi:10.1155/2018/9160793> and Sengupta, Saptarshi, Sanchita Basak, and Richard Peters. (2018) <doi:10.3390/make1010010>).

License GPL-3

Imports Rcpp (>= 1.0.4), ast2ast, methods, dfdr, RcppThread, rlang

LinkingTo Rcpp, RcppArmadillo, RcppThread, ast2ast

Suggests knitr, rmarkdown, tinytest, deSolve

VignetteBuilder knitr

RoxygenNote 7.2.3

Encoding UTF-8

NeedsCompilation yes

Repository CRAN

Date/Publication 2024-02-22 13:00:02 UTC

R topics documented:

`optimize` *Optimize parameters of ode-systems*

Description

Optimize parameters used in an ode equation in order to match values defined in the state-data.frame

Usage

```
optimize(
  ode,
  lb,
  ub,
  npop,
  ngen,
  reltol,
  abstol,
  error,
  states,
  solvertype,
  own_error_fct,
  own_spline_fct,
  own_jac_fct,
  number_threads,
  verbose
)
```

Arguments

`ode`	the ode-system for which the parameter should be optimized.
`lb`	a data.frame containing the lower bounds for the parameters.
`ub`	a data.frame containing the upper bounds for the parameters.
`npop`	a number defining the number of particles used by the Particle Swarm Optimizer. The default value is 40.
`ngen`	a number defining the number of generations the Particle Swarm Optimizer (PSO) should run. The default value is 10000
`reltol`	a number defining the relative tolerance used by the ode-solver. The default value is 1e-06
`abstol`	a vector containing the absolute tolerance(s) for each state used by the ode-solver. The default value is 1e-08

error	a number defining a sufficient small error. When the PSO reach this value optimization is stopped. The default value is 0.0001
states	a data.frame containing the predetermined course of the states. The data.frame is used to extract the initial values of the states. Furthermore, the ode-solver returns *in silico* values of the states at the timepoints which has to be defined in the first column
solvertype	a string defining the type of solver which should be used "bdf" or "adams" are the possible values. The default value is "bdf". "bdf" is an abbreviation for Backward Differentiation Formulas. "adams" is an abbreviation for the Adams-Moulton algorithm
own_error_fct	An optional function to calculate the error between *in silico* value and the specified value in the data.frame states. The default error calculation is specified in the section notes.
own_spline_fct	An optional function to interpolate the values for variable parameters. The default function is a CatmullRome spline interpolation.
own_jac_fct	An optional function which returns the jacobian function. Furthermore it is possible to calculate the jacobian using the R package dfdr. If this is desired "dfdr" has to be passed as argument. If nothing is passed the jacobian matrix is numerically calculated.
number_threads	An optional numeric value defining the number of threads which should be used. If nothing is passed the maximum number of cores is used. The default value is NULL
verbose	A logical value defining whether the output during compilation should be shown or not. The default value is FALSE

Details

The ode system:

The ode system is an R function which accepts four arguments and returns one.

1. the first argument is **t** which defines the (time-) point of then independent variable at which the ode-system is evaluated.
2. the second argument is a vector called **y** which defines the current states at timepoint **t**
3. the third argument is a vector called **ydot** which should be filled with the derivative (left hand side) of the ode-system. It has already the correct length! **This vector has to be returned.**
4. the last argument is called **parameter** and is a vector containing the current parameter-set which is tested by the optimization algorithm.
   If the parameters can change over time. The already interpolated value is passed to the ode-system.

```
# theoretical Example: The parameter 'a' can change over time whereas 'b' is constant over time.
  parameter_set <- data.frame(
          time = c(0, 10, 20, 30, 40),
          a = c(1, 2, 3, 4, 5),
```

```
          b = c(1, NA, NA, NA, NA, NA))
  t <- 5
# Interpolation would result in 1.5 for parameter 'a'
  parameter <- c(1.5, 1) # 'a', 'b'
  y <- 1
  ydot <- vector(length(1))
  ode(t, y, ydot, parameter)
```

- The function returns *ydot*. It is only necessary to fill the vector **ydot**. Check the package *ast2ast* for more details how this works.
- The R function is translated to a C++ function using the package *ast2ast*, see also ast2ast on CRAN and `ast2ast::translate()`. Therefore, if you are starting the simulation for the first time the function has to be compiled. This can require a bit of time.

The boundaries:

The lower and upper boundaries are defined as data.frames that contain 'time' as the first column. The subsequent columns contain the information of the parameter.

```
# Here some examples
# all parameters are constant over the entire integration_time
  example1 <- data.frame(
                      time = 0,
                      a = 0,
                      b = 0.1,
                      c = 0.2,
                      d = 0.2)
# The parameter a, b, and c are constant whereas the parameter d can change over time
  example2 <- data.frame(
                time = c(0, 5, 10, 15),
                a = c(0, NA, NA, NA),
                b = c(0.1, NA, NA, NA),
                c = c(0.2, NA, NA, NA),
                d = c(0, 0.1, 0.2, 1))
# The parameter a, b are constant
# whereas parameter c and d can change over time.
# However, d is not known for all points of c
  example3 <- data.frame(
                  time = c(0, 5, 10, 15, 20, 25),
                  a = c(0, NA, NA, NA, NA, NA),
                  b = c(0.1, NA, NA, NA, NA, NA),
                  c = c(0.2, 0.2, 0, 0, 0, 0),
                  d = c(0, 0.1, 0.2, 1, NA, NA))
```

The states data.frame:

The states are defined as a data.frame that contains the 'time' as the first column. The subsequent columns are the individual states.

```
# Here some examples
# Only the initial values are defined.
  example1 <- data.frame(
                    time = seq(0, 100, 0.5),
                    prey = c(10, rep(NA, 200)),
                    predator = c(10, rep(NA, 200)))
# All values are defined at each timepoint
  example2 <- data.frame(
                time =  seq(0, 100, 0.5),
                prey =  c(10, runif(200)),
                predator =  c(10, runif(200)) )
# Only the values for prey are known and are used during optimization
  example3 <- data.frame(
                time =  seq(0, 100, 0.5),
                prey =  c(10, runif(200)),
                predator =  c(10, rep(NA, 200)) )
```

solvertype:

For solving the ode system the SUNDIALS Software is used check the Sundials homepage for more informations. The solver-type which is used during optimization: "bdf", "adams". bdf is an abbreviation for Backward Differentiation Formulas and adams means Adams-Moulton. All solvers are used in the NORMAL-Step method in a for-loop using the time-points defined in the first column of the 'states' data.frame. The bdf solver use the SUNLinSol_Dense as linear solver.

own_error_fct:

The error function calculates the error at one of the possible time-points. Moreover, the function expects three numerical scalars as arguments. The first one is the number of data-points at which the error is calculated. The second argument describes the *in silico* value at one specific time-point. The third argument is the input of the user at the specific time-point which should be matched.
Here is one example shown using the sum of squares as an alternative error function.

```
error_fct <- function(num_points, insilico, measured) {
  ret = (insilico - measured)^2
  return(ret/num_points)
}
```

own_spline_fct:

The spline function is called, directly before the ode-system is evaluated. However, the function is only called for non-constant parameters. See example Nr.2 and Nr.3 parameter *d* as described above. The results of the spline function is then stored in the vector **parameter** which is passed to the ode-system. The function expects three arguments:

- The time-point at which the function is evaluated
- a vector containing the time-points for which parameters are defined
- a vector containing the parameters at the respective time-points

The function has to return a scalar value.See the example above for a linear interpolation:

```
linear_interpolation <- function(t, time_vec, par_vec) {
  left = 0
  left_time = 0
  right = 0
  right_time = 0
  for(i in 1:length(time_vec)) {
    if(t == time_vec[i]) {
      return(par_vec[i])
    }
    if(t < time_vec[i]) {
      left = par_vec[i - 1]
      right = par_vec[i]
      left_time = time_vec[i - 1]
      right_time = time_vec[i]
      break
    }
  }
  timespan = right_time - left_time
  m = (right - left) / timespan
  ret = left + m*(t - left_time)
  return(ret)
}
```

Mentionable, is that it hasn't to be a interpolation function. See the example above:

```
fct <- function(t, time_vec, par_vec) {
  ret = 0
  for(i in par_vec) {
    ret = ret + i
  }
  return(ret)
}
```

own_jac_fct:

The jacobian function expects 5 arguments.

1. the first argument is **t** which defines the (time-) point of then independent variable at which the ode-system is evaluated.
2. the second argument is a vector called **y** which defines the current states at timepoint **t**
3. the third argument is a vector called **ydot** which should be filled with the derivative (left hand side) of the ode-system. It has already the correct length! Please do not return the vector.
4. the foruth argument is a matrix called **J** which should be filled with the respective derivatives of **ydot**. The matrix has already the correct dimensions. **This matrix has to be returned.**
5. the last argument is called **parameter** and is a vector containing the current parameter-set which is tested by the optimization algorithm.
   If the parameters can change over time. The already interpolated value is passed to the ode-system.

Value

A list is returned which contains three elements. The first one is the error of the best particle. Subsequently, a data.frame with the best parameters is included in the list. The last element are the *in silico* states returned from the ode-solver using the parameter-set at index 2.

Note

- The error between the defined states and the *in silico* states is the absolute difference normalised using the true state.
- The optimization algorithms runs in parallel. Therefore, the ode-system should not contain any printing terms or random number generators.
- a particle swarm algorithm is used for optimization.

See Also

`solve()`, `ast2ast::translate()`

Examples

```
# Optimize (all parameters are constant)
ode <- function(t, y, ydot, parameter) {
  a_db = at(parameter, 1)
  b_db = at(parameter, 2)
  c_db = at(parameter, 3)
  d_db = at(parameter, 4)
  predator_db = at(y,1)
  prey_db = at(y, 2)
  ydot[1] = predator_db*prey_db*c_db - predator_db*d_db
  ydot[2] = prey_db*a_db - prey_db*predator_db*b_db
  return(ydot)
}
path <- system.file("examples", package = "paropt")
states <- read.table(paste(path,"/states_LV.txt", sep = ""), header = TRUE)
lb <- data.frame(time = 0, a = 0.8, b = 0.3, c = 0.09, d = 0.09)
ub <- data.frame(time = 0, a = 1.3, b = 0.7, c = 0.4, d = 0.7)
set.seed(1)
res <- paropt::optimize(ode,
                        lb = lb, ub = ub,
                        reltol = 1e-06, abstol = c(1e-08, 1e-08),
                        error = 0.0001,
                        npop = 40, ngen = 100, # 1000 would be better
                        states = states)


# Optimize (parameter a,b and c are constant. d is variable!)
r <- function(a) {
  c(a, rep(NA, 3))
}

lb <- data.frame(time = c(0, 20, 60, 80),
```

```
                a = r(0.8), b = r(0.3), c = r(0.09), d = 0.1)
ub <- data.frame(time = c(0, 20, 60, 80),
                a = r(1.3), b = r(0.7), c = r(0.4), d = 0.6)
set.seed(1)
res <- paropt::optimize(ode,
                        lb = lb, ub = ub,
                        reltol = 1e-06, abstol = c(1e-08, 1e-08),
                        error = 0.0001,
                        npop = 40, ngen = 100, # 1000 would be better
                        states = states)


# Optimization with own error, spline and jacobian function
ode <- function(t, y, ydot, parameter) {
  a_db = at(parameter, 1)
  b_db = at(parameter, 2)
  c_db = at(parameter, 3)
  d_db = at(parameter, 4)
  predator_db = at(y,1)
  prey_db = at(y, 2)
  ydot[1] = predator_db*prey_db*c_db - predator_db*d_db
  ydot[2] = prey_db*a_db - prey_db*predator_db*b_db
  return(ydot)
}

jac <- function(t, y, ydot, J, parameter) {
  a_db = at(parameter, 1)
  b_db = at(parameter, 2)
  c_db = at(parameter, 3)
  d_db = at(parameter, 4)
  predator_db = at(y,1)
  prey_db = at(y, 2)

  J[1, 1] = prey_db*c_db - d_db
  J[2, 1] = - prey_db*b_db
  J[1, 2] = predator_db*c_db
  J[2, 2] = a_db - predator_db*b_db

  return(J)
}

error_fct <- function(c, a, b) {
  ret = (a - b)^2
  return(ret)
}

spline_fct <- function(t, time_vec, par_vec) {
  ret = 0
  for(i in par_vec) {
    ret = ret + i
  }
  return(ret)
}
```

```
path <- system.file("examples", package = "paropt")
states <- read.table(paste(path,"/states_LV.txt", sep = ""), header = TRUE)
lb <- data.frame(time = 0, a = 0.8, b = 0.3, c = 0.09, d = 0.09)
ub <- data.frame(time = 0, a = 1.3, b = 0.7, c = 0.4, d = 0.7)
set.seed(1)
res <- paropt::optimize(ode,
                        lb = lb, ub = ub,
                        reltol = 1e-06, abstol = c(1e-08, 1e-08),
                        error = 0.0001,
                        npop = 40, ngen = 100, # 1000 would be better
                        states = states,
                        verbose = TRUE,
                        own_error_fct = error_fct,
                        own_spline_fct = spline_fct,
                        own_jac_fct = jac)
```

`solve` *Solves an ode-system*

Description

Solves an ode equation and calculate an error based on the difference on a user-defined state-data.frame.

Usage

```
solve(
  ode,
  parameter,
  reltol,
  abstol,
  states,
  solvertype,
  own_error_fct,
  own_spline_fct,
  own_jac_fct,
  verbose
)
```

Arguments

`ode`	the ode-system for which the parameter should be optimized.
`parameter`	a data.frame containing the parameters.

reltol	a number defining the relative tolerance used by the ode-solver. The default value is 1e-06
abstol	a vector containing the absolute tolerance(s) for each state used by the ode-solver. The default value is 1e-08
states	a data.frame containing the predetermined course of the states. The data.frame is used to extract the initial values of the states. Furthermore, the ode-solver returns *in silico* values of the states at the timepoints which has to be defined in the first column
solvertype	a string defining the type of solver which should be used "bdf" or "adams" are the possible values. The default value is "bdf". "bdf" is an abbreviation for Backward Differentiation Formulas. "adams" is an abbreviation for the Adams-Moulton algorithm
own_error_fct	An optional function to calculate the error between *in silico* value and the specified value in the data.frame states. The default error calculation is specified in the section notes.
own_spline_fct	An optional function to interpolate the values for variable parameters. The default function is a CatmullRome spline interpolation.
own_jac_fct	An optional function which returns the jacobian function. Furthermore it is possible to calculate the jacobian using the R package dfdr. If this is desired "dfdr" has to be passed as argument. If nothing is passed the jacobian matrix is numerically calculated.
verbose	A logical value defining whether the output during compilation should be shown or not. The default value is FALSE

Details

The ode system:

The ode system is an R function which accepts four arguments and returns one.

1. the first argument is **t** which defines the (time-) point of then independent variable at which the ode-system is evaluated.
2. the second argument is a vector called **y** which defines the current states at timepoint **t**
3. the third argument is a vector called **ydot** which should be filled with the derivative (left hand side) of the ode-system. It has already the correct length! **This vector has to be returned.**
4. the last argument is called **parameter** and is a vector containing the current parameter-set which is tested by the optimization algorithm.
   If the parameters can change over time. The already interpolated value is passed to the ode-system.

```
# theoretical Example: The parameter 'a' can change over time whereas 'b' is constant over time.
 parameter_set <- data.frame(
   time = c(0, 10, 20, 30, 40),
   a = c(1, 2, 3, 4, 5),
   b = c(1, NA, NA, NA, NA, NA))
```

```
t <- 5
# Interpolation would result in 1.5 for parameter 'a'
parameter <- c(1.5, 1) # 'a', 'b'
y <- 1
ydot <- vector(length(1))
ode(t, y, ydot, parameter)
```

- The function returns *ydot*. It is only necessary to fill the vector **ydot**. Check the package *ast2ast* for more details how this works.
- The R function is translated to a C++ function using the package *ast2ast*, see also ast2ast on CRAN and `ast2ast::translate()`. Therefore, if you are calling 'solve' for the first time the function has to be compiled. This can require a bit of time.

The parameters:

The lower and upper boundaries are defined as a data.frame that contains 'time' as the first column. The subsequent columns contain the information of the parameter.

```
# Here some examples
# all parameters are constant over the entire integration_time
example1 <- data.frame(
  time = 0,
  a = 0.4,
  b = 1.1,
  c = 0.1,
  d = 0.4)
# The parameter a, b, and c are constant whereas the parameter d can change over time
example2 <- data.frame(
  time = c(0, 5, 10, 15),
  a = c(0.4, NA, NA, NA),
  b = c(1.1, NA, NA, NA),
  c = c(0.1, NA, NA, NA),
  d = c(0.4, 0.5, 0.3, 0.4))
# The parameter a, b are constant
# whereas parameter c and d can change over time.
# However, d is not known for all points of c
example3 <- data.frame(
  time = c(0, 5, 10, 15, 20, 25),
  a = c(1.1, NA, NA, NA, NA, NA),
  b = c(0.1, NA, NA, NA, NA, NA),
  c = c(0.2, 0.2, 0, 0, 0, 0),
  d = c(0, 0.1, 0.2, 1, NA, NA))
```

The states data.frame:

The states are defined as a data.frame that contains the 'time' as the first column. The subsequent columns are the individual states.

```
# Here some examples
# Only the initial values are defined.
example1 <- data.frame(
  time = seq(0, 100, 0.5),
  prey = c(10, rep(NA, 200)),
  predator = c(10, rep(NA, 200)))
# All values are defined at each timepoint
example2 <- data.frame(
  time =  seq(0, 100, 0.5),
  prey =  c(10, runif(200)),
  predator =  c(10, runif(200)) )
# Only the values for prey are known and are used during optimization
example3 <- data.frame(
  time =  seq(0, 100, 0.5),
  prey =  c(10, runif(200)),
  predator =  c(10, rep(NA, 200)) )
```

solvertype:

For solving the ode system the SUNDIALS Software is used check the Sundials homepage for more informations. The solver-type which is used during optimization: "bdf", "adams". bdf is an abbreviation for Backward Differentiation Formulas and adams means Adams-Moulton. All solvers are used in the NORMAL-Step method in a for-loop using the time-points defined in the first column of the 'states' data.frame. The bdf solver use the SUNLinSol_Dense as linear solver.

own_error_fct:

The error function calculates the error at one of the possible time-points. Moreover, the function expects three numerical scalars as arguments. The first one is the number of data-points at which the error is calculated. The second argument describes the *in silico* value at one specific time-point. The third argument is the input of the user at the specific time-point which should be matched.
Here is one example shown using the sum of squares as an alternative error function.

```
error_fct <- function(num_points, insilico, measured) {
  ret = (insilico - measured)^2
  return(ret/num_points)
}
```

own_spline_fct:

The spline function is called, directly before the ode-system is evaluated. However, the function is only called for non-constant parameters. See example Nr.2 and Nr.3 parameter *d* as described above. The results of the spline function is then stored in the vector **parameter** which is passed to the ode-system. The function expects three arguments:

- The time-point at which the function is evaluated
- a vector containing the time-points for which parameters are defined
- a vector containing the parameters at the respective time-points

The function has to return a scalar value.See the example above for a linear interpolation:

```
linear_interpolation <- function(t, time_vec, par_vec) {
  left = 0
  left_time = 0
  right = 0
  right_time = 0
  for(i in 1:length(time_vec)) {
    if(t == time_vec[i]) {
      return(par_vec[i])
    }
    if(t < time_vec[i]) {
      left = par_vec[i - 1]
      right = par_vec[i]
      left_time = time_vec[i - 1]
      right_time = time_vec[i]
      break
    }
  }
  timespan = right_time - left_time
  m = (right - left) / timespan
  ret = left + m*(t - left_time)
  return(ret)
}
```

Mentionable, is that it hasn't to be a interpolation function. See the example above:

```
fct <- function(t, time_vec, par_vec) {
  ret = 0
  for(i in par_vec) {
    ret = ret + i
  }
  return(ret)
}
```

own_jac_fct:

The jacobian function expects 5 arguments.

1. the first argument is **t** which defines the (time-) point of then independent variable at which the ode-system is evaluated.
2. the second argument is a vector called **y** which defines the current states at timepoint **t**
3. the third argument is a vector called **ydot** which should be filled with the derivative (left hand side) of the ode-system. It has already the correct length! Please do not return the vector.
4. the foruth argument is a matrix called **J** which should be filled with the respective derivatives of **ydot**. The matrix has already the correct dimensions. **This matrix has to be returned.**
5. the last argument is called **parameter** and is a vector containing the current parameter-set which is tested by the optimization algorithm.
   If the parameters can change over time. The already interpolated value is passed to the ode-system.

Value

A list is returned which contains two elements. The first one is the error of the best particle. The other element is a data.frame containing the *in silico* states returned from the ode-solver using the parameter-set passed by the user..

Note

- The error between the defined states and the *in silico* states is the absolute difference normalised using the true state.

See Also

optimize(), ast2ast::translate()

Examples

```
# Solve an ode-system
ode <- function(t, y, ydot, parameter) {
  a_db = at(parameter, 1)
  b_db = at(parameter, 2)
  c_db = at(parameter, 3)
  d_db = at(parameter, 4)
  predator_db = at(y,1)
  prey_db = at(y, 2)
  ydot[1] = predator_db*prey_db*c_db - predator_db*d_db
  ydot[2] = prey_db*a_db - prey_db*predator_db*b_db
  return(ydot)
}
path <- system.file("examples", package = "paropt")
states <- read.table(paste(path,"/states_LV.txt", sep = ""), header = TRUE)
parameter <- data.frame(time = 0, a = 1.1, b = 0.4, c = 0.1, d = 0.4)
res <- paropt::solve(ode,
                     parameter = parameter,
                     reltol = 1e-06, abstol = c(1e-08, 1e-08),
                     states = states, verbose = FALSE)


 # solving with own error, spline and jacobian function

 jac <- function(t, y, ydot, J, parameter) {
  a_db = at(parameter, 1)
  b_db = at(parameter, 2)
  c_db = at(parameter, 3)
  d_db = at(parameter, 4)
  predator_db = at(y,1)
  prey_db = at(y, 2)

  J[1, 1] = prey_db*c_db - d_db
  J[2, 1] = - prey_db*b_db
  J[1, 2] = predator_db*c_db
  J[2, 2] = a_db - predator_db*b_db
```

```
  return(J)
}

error_fct <- function(c, a, b) {
  ret = (a - b)^2
  return(ret)
}

spline_fct <- function(t, time_vec, par_vec) {
  ret = 0
  for(i in par_vec) {
    ret = ret + i
  }
  return(ret)
}

path <- system.file("examples", package = "paropt")
states <- read.table(paste(path,"/states_LV.txt", sep = ""), header = TRUE)
parameter <- data.frame(time = 0, a = 1.1, b = 0.4, c = 0.1, d = 0.4)
res <- paropt::solve(ode,
                     parameter = parameter,
                     reltol = 1e-06, abstol = c(1e-08, 1e-08),
                     states = states, verbose = FALSE)

res <- paropt::solve(ode,
                        parameter = parameter,
                        reltol = 1e-06, abstol = c(1e-08, 1e-08),
                        states = states, verbose = FALSE,
                        own_error_fct = error_fct,
                        own_spline_fct = spline_fct,
                        own_jac_fct = jac)
```

Index