Package 'adaptivetau'

November 28, 2023

Type Package

Title Tau-Leaping Stochastic Simulation

Version 2.3

Date 2023-11-28

Author Philip Johnson

Maintainer Philip Johnson <plfj@umd.edu>

Depends R (>= 2.10.1)

Enhances compiler

Description Implements adaptive tau leaping to approximate the trajectory of a continuous-time stochastic process as described by Cao et al. (2007) The Journal of Chemical Physics <doi:10.1063/1.2745299> (aka. the Gillespie stochastic simulation algorithm). This package is based upon work supported by NSF DBI-0906041 and NIH K99-GM104158 to Philip Johnson and NIH R01-AI049334 to Rustom Antia.

License GPL (>= 3)

NeedsCompilation yes

Repository CRAN

Date/Publication 2023-11-28 17:40:11 UTC

R topics documented:

`ssa.adaptivetau` *Approximate stochastic simulation algorithm*

Description

Implements adaptive tau-leaping approximation for simulating the trajectory of a continuous-time Markov process (see reference below).

Usage

```
ssa.adaptivetau(init.values, transitions, rateFunc, params, tf,
                jacobianFunc = NULL, maxTauFunc = NULL,
                deterministic = NULL, halting = NULL,
                relratechange = rep(1, length(init.values)),
                tl.params = NULL, reportTransitions = FALSE)
```

Arguments

`init.values` Vector of initial values for all variables. This should be a simple one-dimensional vector of real numbers. *IMPORTANT:* variables must never take negative values.

`transitions` One of two possible data types:

- A list with length equal to the number of transitions. Each element of the list should be a vector specifying a transition (i.e., which state(s) change and by how much). Each entry in the vector needs a name (specifying which state variable to change, either by name or index) and a value (specifying the amount by which this variable will change).
- A two-dimensional matrix of integers specifying how each state variable (rows) should be changed for a given transition (columns). Generally this will be a sparse matrix of primarily 1s and -1s, which can make this structure inefficient.

See the example below for details as well as `ssa.maketrans` or the vignette accompanying this package.

`rateFunc` R function that returns instantaneous transition rates for each transition in the form a real-valued one-dimensional vector with length equal to the number of transitions. The order of these rates must match the order in which transitions were specified in the `transitions` parameter above. This function must accept the following arguments:

- vector of current values for all state variables (in order used in the `init.values` argument above)
- parameters as supplied in argument to ssa.adaptivetau
- single real number giving the current time (all simulations start at t=0)

`params` any R variable to be passed as-is to rateFunc, presumably specifying useful parameters.

`tf` final time of simulation. Due to the approximate nature of the tau-leaping algorithm, the simulation may overshoot this time somewhat.

`jacobianFunc` R function that returns Jacobian of transition rates. If supplied, enables the use of the implicit tau-leaping algorithm (if appropriate; used in stiff systems). In contrast to typical definition of Jacobian (but consistent with the transition matrix parameter above), columns represent transitions (functions) and rows represent state variables (different partial derivative variables). jacobianFunc takes the same parameters as rateFunc. **NOTE:** the implicit tau-leaping algorithm will reduce the observed variance for fast-changing state variables, although the mean will be correct.

`maxTauFunc` R function that returns single real number giving the maximum tau leap allowable from the current state. Only executed if the adaptive tau algorithm wants to leap in a step greater than tl.params$maxtau (should only need to use if your rate functions have a dramatically discontinuous first derivative). Takes same parameters as rateFunc.

`deterministic` Specify transitions to be treated as deterministic. If not NULL, either a TRUE/FALSE vector of length equal to the number of transitions -or- a integer vector of transition numbers to be treated as deterministic (i.e. applying `which` to the former). Deterministic transitions will be applied every timestep using the **expected** degree of change. Note this will almost certainly result in non-integer values for any affected state variables. Also, note that this is still an approximation – not a true numerical ODE solution – and at least one transition must be stochastic for this to work at all.

`halting` Specify transitions which, when executed, will cause the simulation to halt. If not NULL, either a TRUE/FALSE vector of length equal to the number of transitions -or- a integer vector of transition numbers to be treated as halting (i.e. applying `which` to the former). If halting transitions are specified, then the return value is different (since we want to know which transition halted us).

`relratechange` one-dimensional vector of length equal to the number of state variables providing an upperbound to the ratio of amount that any transition rate will change given a corresponding change in the state variable. In other words, if variable i doubles, can we be assured that no transition will **more** than double (ratio of 1)? If not, then you need to set this variable to be greater than 1.

`tl.params` List of various parameters to the tau-leaping algorithm itself (best explained by reading original reference):

epsilon default 0.05; increasing will make bigger leaps resulting in potentially more error

delta default 0.05; how close two symmetric transition rates must be before we classify them as in partial-equilibrium. Only applies to the implicit tau routine.

maxtau default Inf; maximum leap allowed. Should only need to specify if rate functions are drastically non-smooth.

extraChecks default TRUE; whether to perform a battery of validity checks on the state variables and rates after every call to "rateFunc." Slows down implementation, but unless you are SURE you have designed your transition matrix and rate function correctly, probably best to keep these checks.

`reportTransitions`

default FALSE; whether to include a matrix of executed transitions per time interval in the output (alongside with the states at each time point).

Details

The initial values, transition matrix & transition rates must all be designed such that variables are always non-negative. The algorithm relies on this invariant.

See reference for details but, in brief the adaptive tau-leaping algorithm dynamically switches between three methods for simulating events:

exact no approximation – executes a single transition at a time

explicit tau-leaping subdivides transitions into those that might hit cause a variable to hit 0 ("critical") and those that do not. Duration of time step picked dynamically with the goal of maximizing the step while minimized the change in the transition rates (the approximation assumes that these rates do not change). Non-critical transitions are advanced by a poisson-distributed random variable; critical transitions are handled more like the exact algorithm.

implicit tau-leaping in addition to dividing between critical and non-critical, also identifies transitions in quasi-equilbrium (reversible pairs of transition that have roughly equal forward-backward flow). Duration of time step picked on basis of non-critical, non-equilibrium transitions. This has the potential to greatly increase the timestep size for stiff systems. Similar idea to the explicit method but necesitates solving an implicit equation via Newton's method. Thus you must supply a function to calculate the Jacobian to enable this method.

All error messages that reference variables or transitions by number use 1-based numbering where the first variable (or transition) is 1.

Consider calling `enableJIT(1)` before running ssa.adaptivetau. In most circumstances, this should yield some speedup by byte-code compiling the rate function.

Value

If no halting transitions are specified, then a two-dimensional matrix with rows for every timepoint at which the rateFunc was evaluated and columns giving the value for every state variable at that time. The first column specifies the time.

If halting transitions are specified, then a list with two elements. The first ('dynamics') is the same two-dimensional matrix as without halting transitions. The second ('haltingTransition') gives the number of the transition that halted the simulation, or NA otherwise.

If the 'reportTransitions' option is used, then a list is returned with two elements (or three, if combined with halting transitions). The final element of the list is a two-dimensional matrix called 'transitions' with a row for each timepoint and columns giving the number of times each transition was executed in the between the current time and the previous time.

Note

Development of this package was supported in part by National Science Foundation award DBI-0906041 and National Institute of Health award K99-GM104158. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the NSF or NIH.

Author(s)

Philip Johnson

References

Cao Y, Gillespie DT, Petzold LR, *The Journal of Chemical Physics*, 2007

See Also

For systems with sparse transition matrices, see helper function `ssa.maketrans`. `ssa.exact` exposes a R interface to the C++ implementation of the exact, non-approximate simulation algorithm (sometimes known as "Gillespie").

Examples

```
## Simple Lotka-Volterra example

# We have three potential transitions:
transitions = list(c(prey = +1), # prey grow
                   c(prey = -2, pred = +1), # predation
                   c(pred = -1)) # predator dies

# Function to calculate transition rates, given variables and parameters
lvrates <- function(x, params, t) {
  return(c(params$preygrowth*x["prey"],     # rate of prey growing
           x["prey"]*x["pred"]*params$eat,  # rate of predation
           x["pred"]*params$preddeath))     # rate of predators dying
}

# Set the Lotka-Volterra parameters
params = list(preygrowth=10, eat=0.01, preddeath=10);

# Set the random seed (only necessary if you want to reproduce results)
set.seed(4)

# Perform the stochastic simulation!
r=ssa.adaptivetau(c(prey = 1000, pred = 500),
                  transitions, lvrates, params, tf=12)

# Plot the results
matplot(r[,"time"], r[,c("prey","pred")], type='l', xlab='Time',
        ylab='Counts (log scale)', log='y')
legend("bottomleft", legend=c("prey", "predator"), lty=1:2, col=1:2)


# However, if you are interested in very fine-scale variance, perhaps the
# default parameters smooth too much.  Try reducing the tl.param epsilon
# to see a better approximation:
s=ssa.adaptivetau(c(prey = 1000, pred = 500),
                  transitions, lvrates, params, tf=12,
                  tl.params = list(epsilon=0.01)) # reduce "epsilon"
par(mfrow=c(2,1));
matplot(r[r[,"time"]<2,"time"], r[r[,"time"]<2,c("prey","pred")],
        type='l', xlab='Time', ylab='Counts', main="Original (epsilon=default)")
matplot(s[s[,"time"]<2,"time"], s[s[,"time"]<2,c("prey","pred")],
```

```
type='l', xlab='Time', ylab='Counts', main="Fine-scale (epsilon=0.01)")
```

`ssa.exact` *Exact stochastic simulation algorithm*

Description

Simulates the trajectory of a continuous-time Markov process (aka. the Gillespie simulation algorithm).

Usage

```
ssa.exact(init.values, transitions, rateFunc, params, tf,
          reportTransitions = FALSE)
```

Arguments

`init.values`	Vector of initial values for all variables. This should be a simple one-dimensional vector of real numbers. *IMPORTANT:* variables must never take negative values.
`transitions`	Two-dimensional matrix of integers specifying how each state variable (rows) should be changed for a given transition (columns). Generally this will be a sparse matrix of primarily 1s and -1s.
`rateFunc`	R-function that returns instantaneous transition rates for each transition in the form a real-valued one-dimensional vector with length equal to the number of transitions. The order of these rates must match the order in which transitions were specified in the `transitions` parameter above. This function must accept the following arguments: • vector of current values for all state variables (in order used in the `init.values` argument above) • parameters as supplied in argument to ssa.adaptivetau • single real number giving the current time (all simulations start at t=0)
`params`	any R variable to be passed as-is to rateFunc, presumably specifying useful parameters.
`tf`	final time of simulation. Due to the approximate nature of the tau-leaping algorithm, the simulation may overshoot this time somewhat.
`reportTransitions`	default FALSE; whether to include a matrix of executed transitions per time interval in the output (alongside with the states at each time point).

Details

This function is supplied as a bonus with the adaptivetau package, since the C++ function that underlies this (exact) stochastic simulation algorithm is used in the (approximate) adaptive tau stochastic simulation as well.

The initial values, transition matrix, and transition rates must all be designed such that variables are always non-negative. The algorithm relies on this invariant.

Consider calling `enableJIT(1)` before running ssa.exact. In most circumstances, this should yield some speedup by byte-code compiling the rate function.

Value

Two-dimensional matrix with rows for every timepoint at which the rateFunc was evaluated and columns giving the value for every state variable at that time. The first column specifies the time.

If the 'reportTransitions' option is used, then a list is returned with two elements. The first element is the 'dynamics' matrix described above. The second is a two-dimensional matrix called 'transitions' with a row for each timepoint and columns giving the number of times each transition was executed between the current time and the previous time.

Author(s)

Philip Johnson

See Also

This function is a bonus the comes along with the approximate (but faster) `ssa.adaptivetau`. For systems with sparse transition matrices, see helper function `ssa.maketrans`.

Examples

```
## Lotka-Volterra example
lvrates <- function(x, params, t) {
  with(params, {
    return(c(preygrowth*x["prey"],      ## prey growth rate
             x["prey"]*x["pred"]*eat,   ## prey death / predator growth rate
             x["pred"]*preddeath))      ## predator death rate
  })
}
params=list(preygrowth=10, eat=0.01, preddeath=10);
r=ssa.exact(c(prey = 1000, pred = 500),
            matrix(c(1,0, -2,1, 0,-1), nrow=2), lvrates, params, tf=2)
matplot(r[,"time"], r[,c("prey","pred")], type='l', xlab='Time', ylab='Counts')
legend("topleft", legend=c("prey", "predator"), lty=1:2, col=1:2)
```

`ssa.maketrans` *Make transition matrix*

Description

Helper function to (easily) construct sparse transition matrices for ssa.adapativetau.

Usage

```
ssa.maketrans(variables, ...)
```

Arguments

variables	Either the number of variables in your system OR a vector of strings specifying the names of the variables (in which case the length of this vector is the number of variables). The number of variables equals the number of rows in the returned transition matrix.
...	At least one additional argument is *required*. Each should be a matrix with an arbitrary number of columns (each represents a separate transition) and an even number of rows. Going down a particular column, entries alternate between specifying the variable that will change and the amount by which it will change. Variables be specified either by numerical index (1,2,...,numVariables) or character name (if names supplied in the parameter above). See examples below.

Details

Making large transition matrices can be a real pain, particularly if the entries in the end will be quite sparse. The concept of this function is simple but the easiest way to understand it is probably to check out the examples.

Value

two-dimensional matrix with rows for each variable and columns for each transition.

Author(s)

Philip Johnson

See Also

You probably only want to use this in preparation for calling ssa.adaptivetau.

Examples

```
## trivial Lotka-Volterra example from ssa.adaptivetau
nu = ssa.maketrans(2, #number of variables
                   rbind(1, +1),
                   rbind(1, -1, 2, +1),
                   rbind(2, -1))

## slightly more complicated SIR epidemiological model with two distinct
## susceptible and infected variables (think male and female), birth
## (into S) and death (exclusively from I and R)
nu = ssa.maketrans(c("Sm", "Sf", "Im", "If", "R"), #list of variable names
                   rbind(c("Sm","Sf"), +1),
                   rbind(c("Sm","Sf"), -1, c("Im","If"), +1),
                   rbind(c("Im","If"), -1),
                   rbind(c("Im","If"), -1, "R", +1),
                   rbind("R", -1))
```

Index