Package ‘DOBAD’

December 7, 2017

Type Package

Title Analysis of Discretely Observed Linear
Birth-and-Death(-and-Immigration) Markov Chains

Version 1.0.6

Date 2017-12-7

Author Charles Doss, Vladimir Minin, Marc Suchard

Maintainer Charles Doss <cdoss@umn.edu>

Description Provides Frequentist (EM) and Bayesian (MCMC) Methods for Inference of Birth-Death-Immigration Markov Chains.

License GPL (>= 2)

Depends numDeriv

Imports methods,lattice,Matrix

Suggests gtools,MCMCpack,functional

LazyLoad yes

NeedsCompilation no

Repository CRAN

Date/Publication 2017-12-07 18:32:19 UTC

R topics documented:

add.generator *Generating functions for birth-death processes with immigration*

Description

A set of generating functions for sufficient statistics for partially observed birth-death process with immigration. The sufficient statistcs are the number of births and immigrations, the mean number of deaths, and the time average of the number of particles.

Usage

```
add.generator(r,s,t,lambda,mu,nu,X0)
rem.generator(r,s,t,lambda,mu,nu,X0)
timeave.laplace(r,s,t,lambda,mu,nu,X0)
hold.generator(w,s,t,lambda,mu,nu,X0)
process.generator(s,time,lambda,mu,nu,X0)
addrem.generator(u, v, s, t, X0, lambda, mu, nu)
remhold.generator( v, w, s, t, X0, lambda, mu, nu)
addhold.generator( u, w, s, t, X0, lambda, mu, nu)
addremhold.generator( u, v, w, s, t, X0, lambda, mu, nu)
```

Arguments

`r,u,v,w`	dummy variable attaining values between 0 and 1. We use r for the single-argument generators and u,v,w for births,deaths, and holdtime for the multi-variable generators syntax, generally.
`s`	dummary variable attaining values between 0 and 1
`t,time`	length of the time interval
`lambda`	per particle birth rate
`mu`	per particle death rate
`nu`	immigration rate
`X0`	starting state, a non-negative integer

Details

Birth-death process is denoted by X_t

Sufficient statistics are defined as

N_t^+ = number of additions (births and immigrations)

N_t^- = number of deaths

R_t = time average of the number of particles,

$$\int_0^t X_y dy$$

Function add.generator calculates

$$H_i^+(r,s,t) = \sum_{n=0}^{\infty}\sum_{j=0}^{\infty} Pr(N_t^+ = n, X_t = j|X_o = i)r^n s^j$$

Function rem.generator calculates

$$H_i^-(r,s,t) = \sum_{n=0}^{\infty}\sum_{j=0}^{\infty} Pr(N_t^- = n, X_t = j|X_o = i)r^n s^j$$

Function timeave.laplace calculates

$$H_i^*(r,s,t) = \sum_{j=0}^{\infty}\int_0^{\infty} e^{-rx} dPr(R_t \leq x, X_t = j|X_o = i)s^j$$

Function processor.generator calculates

$$G_i(s,t) = \sum_{j=0}^{\infty} Pr(X_t = j|X_o = i)r^n s^j$$

Function addrem.generator calculates

$$H_i(u,v,s,t) = \sum_{j=0}^{\infty}\sum_{n_1=0}^{\infty}\sum_{n_2=0}^{\infty} Pr(X_t = j, N_t^+ = n_1, N_t^- = n_2|X_o = i)u^{n_1} v^{n_2} s^j$$

Function addhold.generator calculates

$$H_i(u,,w,s,t) = \sum_{j=0}^{\infty}\sum_{n1\geq 0} u_1^n \int_0^{\infty} e^{-rx} dPr(R_t \leq x, N_t^+ = n_1, X_t = j|X_o = i)s^j$$

Function remhold.generator is the same as addhold.generator but with N- instead of N+.

Value

Numeric value of the corresponding generating function.

Author(s)

Marc A. Suchard, Charles Doss

See Also

add.joint.mean.many

`add.joint.mean.many` *Mean counts and particle time averages for birth-death processes with immigration*

Description

A set of functions for calculating the joint and conditional mean sufficient statistics for partially observed birth-death process with immigration. The sufficient statistcs are the number of births and immigrations, the mean number of deaths, and the time average of the number of particles. The conditional expectations of these quantities are calculated for a finite time interval, conditional on the number of particles at the beginning and the end of the interval.

Usage

```
add.joint.mean.many(t,lambda,mu,nu=0,X0=1,delta=0.001,n=1024)
rem.joint.mean.many(t,lambda,mu,nu=0,X0=1,delta=0.001,n=1024)
timeave.joint.mean.many(t,lambda,mu,nu=0,X0=1,delta=0.001,n=1024)
add.cond.mean.many(t,lambda,mu,nu=0,X0=1,delta=0.001,n=1024,
 prec.tol=1e-12, prec.fail.stop=TRUE)
rem.cond.mean.many(t,lambda,mu,nu=0,X0=1,delta=0.001,n=1024,
prec.tol=1e-12, prec.fail.stop=TRUE)
timeave.cond.mean.many(t,lambda,mu,nu=0,X0=1,delta=0.001,n=1024,
prec.tol=1e-12, prec.fail.stop=TRUE)
add.joint.mean.one(t,lambda,mu,nu=0,X0=1,Xt,delta=0.001,n=1024,r=4)
rem.joint.mean.one(t,lambda,mu,nu=0,X0=1,Xt,delta=0.001,n=1024,r=4)
timeave.joint.mean.one(t,lambda,mu,nu=0,X0=1,Xt,delta=0.001,n=1024,r=4)
add.cond.mean.one(t,lambda,mu,nu=0,X0=1,Xt,trans.prob=NULL,
joint.mean=NULL,delta=1e-04,n=1024, r=4,
prec.tol=1e-12, prec.fail.stop=TRUE)
rem.cond.mean.one(t,lambda,mu,nu=0,X0=1,Xt,trans.prob=NULL,
joint.mean=NULL,delta=1e-04,n=1024, r=4,
prec.tol=1e-12, prec.fail.stop=TRUE)
timeave.cond.mean.one(t,lambda,mu,nu=0,X0=1,Xt,trans.prob=NULL,
joint.mean=NULL, delta=1e-04,n=1024,r=4,
prec.tol=1e-12, prec.fail.stop=TRUE)
hold.cond.mean.one(t,lambda,mu,nu=0,X0=1,Xt, trans.prob=NULL,joint.mean=NULL,
delta=1e-04,n=1024,r=4,prec.tol=1e-12, prec.fail.stop=TRUE)
add.joint.meanSq.one(t, lambda, mu, nu = 0, X0 = 1, Xt, joint.mean=NULL, delta = 0.001,
 n=1024,r=4)
add.cond.meanSq.one(t, lambda, mu, nu = 0, X0 = 1, Xt, trans.prob=NULL,
joint.mean=NULL, delta = 0.001, n
= 1024,r=4, prec.tol=1e-12, prec.fail.stop=TRUE)
addrem.joint.mean.one(t, lambda, mu, nu = 0, X0 = 1, Xt, delta = 0.001,
n = 1024,r=4)
addrem.cond.mean.one(t, lambda, mu, nu = 0, X0 = 1, Xt, trans.prob=NULL,
delta = 0.001,n = 1024, r=4,prec.tol=1e-12, prec.fail.stop=TRUE)
addhold.joint.mean.one(t, lambda, mu, nu = 0, X0 = 1, Xt, delta = 0.001,
```

```
n = 1024,r=4)
addhold.cond.mean.one(t, lambda, mu, nu = 0, X0 = 1, Xt,
trans.prob=NULL, delta = 0.001, n = 1024, r=4, prec.tol=1e-12, prec.fail.stop=TRUE)
remhold.joint.mean.one(t, lambda, mu, nu = 0, X0 = 1, Xt, delta = 1e-04,
n = 1024,r=4)
remhold.cond.mean.one(t, lambda, mu, nu = 0, X0 = 1, Xt,
trans.prob=NULL, delta = 1e-04,
n = 1024,r=4, prec.tol=1e-12, prec.fail.stop=TRUE)
add.joint.meanSq.one(t, lambda, mu, nu = 0, X0 = 1, Xt, joint.mean=NULL,
delta = 0.001, n = 1024,r=4)
add.cond.meanSq.one(t, lambda, mu, nu = 0, X0 = 1, Xt,
trans.prob=NULL,joint.mean=NULL, delta = 0.001,
n= 1024, r=4, prec.tol=1e-12, prec.fail.stop=TRUE )
rem.joint.meanSq.one(t, lambda, mu, nu = 0, X0 = 1, Xt,
joint.mean=NULL, delta = 0.001, n = 1024,r=4)
rem.cond.meanSq.one(t, lambda, mu, nu = 0, X0 = 1, Xt, trans.prob=NULL,
joint.mean=NULL, delta = 0.001,n = 1024,r=4, prec.tol=1e-12, prec.fail.stop=TRUE)
hold.joint.meanSq.one(t, lambda, mu, nu = 0, X0 = 1, Xt, r=4,  n = 1024,
delta = 0.0001)
hold.cond.meanSq.one(t, lambda, mu, nu = 0, X0 = 1, Xt, trans.prob=NULL,
n= 1024,delta = 0.0001, r=4, prec.tol=1e-12, prec.fail.stop=TRUE)
all.cond.mean.PO(data,lambda,mu,nu=0,delta=0.001,n=1024, r=4,
prec.tol=1e-12, prec.fail.stop=TRUE)
all.cond.mean2.PO(data,lambda,mu,nu=0,delta=0.001,n=1024,r=4,
prec.tol=1e-12,  prec.fail.stop=TRUE)
```

Arguments

`t`	length of the time interval
`lambda`	per particle birth rate
`mu`	per particle death rate
`nu`	immigration rate
`X0`	starting state, a non-negative integer
`Xt`	ending state, a non-negative integer
`data`	CTMC_PO_1 or an analogous list. List isn't always accepted (in all.cond.mean functions it isn't). all.cond.means both accept CTMC_PO_1 or CTMC_PO_many.
`trans.prob`	Either NULL or a precomputed transition probability for a process with the parameters passed in. This saves the repeated computation of the same transition probability for multiple conditional expectations over the same interval. If NULL, the probability will just be computed in the function.
`joint.mean`	This is a parameter in some of the computations for some squared means. It is either NULL or the corresponding (first-order, unsquared) mean. If NULL the probability will just be computed in the function. (It is needed to convert a factorial mean to a squared mean.) Note that this is ALWAYS an unsquared mean, regardless of whether the function is a *.meanSq.* or a *.mean.* function. In the latter case, if a non-NULL value is passed, the called function doesn't do much besides divide.

delta	increment length used in numerical differentiation
n	number of coefficients to pull off via FFT, in *.one functions this number determines the number of intervals in the Rieman sum approximation of the integral
prec.tol	"Precision tolerance"; to compute conditional means, first the joint means are computed and then they are normalized by transition probabilities. The precision parameters govern the conditions under which the function will quit if these values are very small. If the joint-mean is smaller than prec.tol then the value of prec.fail.stop decides whether to stop or continue.
prec.fail.stop	If true, then when joint-mean values are smaller than prec.tol the program stops; if false then it continues, usually printing a warning.
r	See numDeriv package; this is 'r' argument for grad/genD/hessian which determines how many richardson-method iterations are done.

Details

Birth-death process is denoted by X_t

Sufficient statistics are defined as

N_t^+ = number of additions (births and immigrations)

N_t^- = number of deaths

R_t = time average of the number of particles, $\int_0^t X_y dy$

Function add.joint.mean.many returns a vector of length n, where the j-th element of the vector is equal to

$$E(N_t^+ 1_{X_t=j} | X_0 = X0)$$

Function rem.joint.mean.many returns a vector of length n, where the j-th element of the vector is equal to

$$E(N_t^- 1_{X_t=j} | X_0 = X0)$$

Function timeave.joint.mean.many returns a vector of length n, where the j-th element of the vector is equal to

$$E(R_t 1_{X_t=j} | X_0 = X0)$$

Function add.cond.mean.many returns a vector of length n, where the j-th element of the vector is equal to

$$E(N_t^+ | X_0 = X0, X_t = j)$$

Function rem.cond.mean.many returns a vector of length n, where the j-th element of the vector is equal to

$$E(N_t^- | X_0 = X0, X_t = j)$$

Function timeave.cond.mean.many returns a vector of length n, where the j-th element of the vector is equal to

$$E(R_t | X_0 = X0, X_t = j)$$

Function add.joint.mean.one returns $E(N_t^+ 1_{X_t=Xt} | X_0 = X0)$

Function rem.joint.mean.one returns $E(N_t^- 1_{X_t=Xt} | X_0 = X0)$

Function timeave.joint.mean.one returns $E(R_t 1_{X_t=Xt}|X_0 = X0)$

Function add.cond.mean.one returns $E(N_t^|X_0 = X0, X_t = Xt)$

Function rem.cond.mean.one returns $E(N_t^-|X_0 = X0, X_t = Xt)$

Function timeave.cond.mean.one returns $E(R_t|X_0 = X0, X_t = Xt)$

Function add.joint.meanSq.one returns $E((N_t^-)^2, X_t = Xt|X_0 = X0)$

Function add.cond.meanSq.one returns $E((N_t^-)^2|X_0 = X0, X_t = Xt)$

Function addrem.joint.mean.one returns $E((N_t^- N_t^-), X_t = Xt|X_0 = X0)$

Function addrem.cond.mean.one returns $E((N_t^- N_t^-)|X_0 = X0, X_t = Xt)$

all.cond.mean.PO and all.cond.mean2.PO compute the first and second order means respectively for a partially observed process (with possibly more than one observation point). So they amalgamate the above functions and also apply them to multiple observations. The outcomes are labeled appropriately.

Note that all.cond.mean.PO are not methods, they can accept either CTMC_PO_many or CTMC_PO_1 (via their use of CTMCPO2indepIntervals function).

"Hold" and "timeave" are the same.

The .many functions are less safe about differentiation right now. This should be changed in the future.

Author(s)

Marc A. Suchard, Vladimir N. Minin, Charles Doss

See Also

add.generator

Examples

```
library(DOBAD)
my.lambda = 2
my.mu = 3
my.nu =1
my.time = 0.5
my.start = 10
my.end = 2
my.n = 1024

#Calculate the mean number of additions (births and immigrations)
#conditional on "my.start" particles at time 0 and "my.end" particles at time "my.time"
add.cond.mean.one(t=my.time,lambda=my.lambda,mu=my.mu,nu=my.nu,X0=my.start,Xt=my.end)

#Calculate a vector mean number of deaths joint with "my.end" particles at
# time "my.time" and conditional on "my.start" particles at time 0
DOBAD:::rem.joint.mean.one(t=my.time,lambda=my.lambda,mu=my.mu,nu=my.nu,X0=my.start,Xt=my.end)

#Calculate a vector mean particle time averages conditional on
# "my.start" particles at time 0 and 1 to "my.n" particles at time "my.time"
```

```
# WARNING: conditional expectations for large values of |X_0-X_t| may be
# unreliable
timeave.cond.mean.many(t=my.time,lambda=my.lambda,mu=my.mu,nu=my.nu,X0=my.start,n=my.n)[1:20]
```

`add.uncond.mean.one` *ENplus, ENminus, Eholdtime, unconditional on ending state.*

Description

ENplus, ENminus, Eholdtime, unconditional on ending state. i.e. sum over j of Eij(Nplus), etc. Expected number of total jumps up/down/holdtime over the given interval, conditional on starting state.

Usage

```
add.uncond.mean.one(t, X0, lambda, mu, nu, delta = 0.001, r = 4)
rem.uncond.mean.one(t, X0, lambda, mu, nu, delta = 0.001, r = 4)
hold.uncond.mean.one(t, X0, lambda, mu, nu, delta = 0.001, r = 4)
```

Arguments

`t`	time
`X0`	starting state
`lambda`	birth rate
`mu`	death rate
`nu`	immigration rate
`delta`	paramter for derivative.
`r`	parameter for derivative.

Details

Uses generating functions.

Value

Each return a numeric.

Author(s)

Charles Doss

ARsim — *Accept-Reject Simulation*

Description

Generic Code for acceptance-rejection sampling.

Usage

```
ARsim(margSimFn, acceptFn, N, keepTestInfo = FALSE)
```

Arguments

`margSimFn`	This is a simulation function. It should take no arguments (or have default values that will be used). It should output one simulation; we will refer to it as being "type X".
`acceptFn`	Should take "type X" as argument and output True or False. It should actually output a list (T/F, extraInformation). ExtraInformation is generally whatever is being used to do the accept/reject part; it is technically only required if "keepTestInfo" is passed a True.
`N`	How many simulations total to run (regardless of eventual acceptance ratio); There is not currently a parameter for choosing to stop after a given number of acceptances.
`keepTestInfo`	True or False; if True then the result will be two lists, the second of which has the second output from acceptFn; generally the data used to decide whether to accept or reject the simulations.

Details

Does accept-reject simultion; margSimFn is run N times; acceptFn decides which to keep and which to remove;

Value

Returns a list with one (if keepTestInfo==FALSE) or two (if keepTestInfo==TRUE) components. The first is $acceptSims, and the second is $testVals. The component acceptSims are the simulated values that were accepted. To do further analysis, testVals is the corresponding list of information used to evaluate.

In future: Will have option to pass all simulations as output, and to accept simulations (but presumably with a different acceptFn) to allow for more reuse.

As an implementation note: want to do "replicate" inside this function so as to regulate the types of output.

Author(s)

Charles Doss

BD.EMInference.prodExpecs

Functions That Calculate Product Expectations Needed for Inference via EM Algorithm

Description

In order to calculate the information matrix for partial data, several conditional expectations of products of sufficient statistics are needed.

Usage

```
getBDsummaryExpecs(sims, fnc=function(x){x})
getBDsummaryProdExpecs(sims, getsd=FALSE)
```

Arguments

`sims`	A list of Birth-Death CTMCs.
`fnc`	A one argument function. It should be a function from Reals to Reals, capable of accepting a vector as its argument
`getsd`	Also return estimate of standard deviations of the prods

Details

Assume we have a linear-birth-death process X_t with birth parameter λ, death parameter μ, and immigration parameter $\beta\lambda$ (for some known, real β). We observe the process at a finite set of times over a time interval [0,T].

In order to calculate the information matrix for partial data, several conditional expectations of products of sufficient statistics are needed. We have a method for simulation conditional on the data, sim.condBD, which we use to estimate these.

Generally for getting the information matrix after running the EM algorithm, sim.condBD is called to simulate with the given parameters (estimates, usually), and the output sims are passed. It is often important that the same set of sims are used to get all the results if the goal is to create an information matrix.

Value

getBDsummaryExpecs simply returns (an estimate of) E(fnc(Nt+)), E(fnc(Nt-)), and E(fnc(Rt)), where Nt+, Nt-, and Rt are the numbre of jumps up, the number of jumps down, and the total holding time, respectively. They are returned in that order, also with labels "Nplus", "Nminus", and "Holdtime".

getBDsummaryProdExpecs returns E(Nt+ * Nt-), E(Nt+ * Rt), and E(Nt- * Rt), in that order, also with the labels "NplusNminus", "NplusHoldtime", "NminusHoldtime".

Returns another row of with corresponding standard deviations if getsd=TRUE.

Author(s)

Charles Doss

See Also

add.joint.mean.many, sim.condBD

BD.MCMC.SC *MCMC on Linear Birth Death Process*

Description

Bayesian parameter estimation via Gibbs sampler MCMC on Linear Birth Death process, (_S_pecial _C_ase of constrained immigration) in which the data is the state at discrete time points.

Usage

```
BD.MCMC.SC(Lguess, Mguess, beta.immig, alpha.L, beta.L, alpha.M, beta.M,
    data, burnIn = 100, N = 1000, n.fft = 1024,
verbose=1, verbFile=NULL, simMethod=-1,...)
```

Arguments

`Lguess`	Starting point for λ
`Mguess`	Starting point for μ
`beta.immig`	Immigration rate = beta.immig $*\lambda$.
`alpha.L`	Shape parameter for prior for λ
`beta.L`	Rate parameter for prior for λ
`alpha.M`	Shape parameter for prior for μ
`beta.M`	Rate parameter for prior for μ
`data`	Partially observed chain. Has components $times and $states where dat$states[i] is the state observed at time dat$times[i].
`N`	Number of iterations to run the MCMC for.
`burnIn`	Number of initial parameter estimates to throw out. (So need burnIn « N.) Choose burnIn==0 throws nothing away.
`n.fft`	Number of terms to use in the fast fourier transform or the riemann integration when using the generating functions to compute probabilities or joint expectations for the birth-death process. See the add.joint.mean.many, etc, functions.
`verbose`	Chooses level of printing. Increasing from 0, which is no printing.
`verbFile`	Character signifying the file to print to. If NULL just to standard output.
`simMethod`	Switch between using Accept-reject simulation and using the exact simulation method. If -1, the function attempts to determine the best one of the two for the given parameters. Value of 0 fixes it at AR, and 1 fixes it at the exact method.
`...`	Unused at this point.

Details

Assume we have a linear-birth-death process X_t with birth parameter λ, death parameter μ, and immigration parameter $\beta\lambda$ (for some known, real β). We observe the process at a finite set of times over a time interval [0,T]. This runs MCMC to do parameter estimation. The method is Gibbs sampling, by augmenting the state space to include the the fully observed chain. Then Gibbs sampling is performed using the the conditional simulation of sim.condBD and the fact that, given the fully observed chain as data, independent gamma priors are conjugate priors, with independent posteriors.

Value

Returns a $N - burnIn$x2 matrix, the nth row being the estimators/samples at the nth iteration. The first column is for lambda (birth), the second for mu (death).

Author(s)

Charles Doss

See Also

`add.joint.mean.many`

`bdARsimCondEnd` *Conditional Simulation of BD via Accept-Reject*

Description

Simulates linear birth-death processes conditional on observing the end time (or a series of discrete observations), via simple accept reject (ie marginal simulation and accepting if it has the right end state).

Usage

```
bdARsimCondEnd(Naccepted = NULL, Ntotal = NULL,Nmax=NULL,
 bd.PO = new("CTMC_PO_1", states = c(5, 7, 3),
 times = c(0, 0.4, 1)), L = 0.5, m = 0.7, nu = 0.4)
bdARsimCondEnd.1(Naccepted = NULL, Ntotal = NULL, Nmax=NULL,
T = 1.02, L = 0.3, m = 0.4, nu = 0.1, a = 8, b = 9)
```

Arguments

`Naccepted`	Number of accepted sims to have at the end. Naccepted overrides Ntotal. If you want to use Ntotal, Naccepted should be NULL. Note that the number of sims will be >= Naccepted, probably not exactly equal to Naccepted.
`Ntotal`	Number of marginal sims to do; no guarantee of how many sims you will get out, but a better guarantee of how long it will take. If it gets no sims, it returns list().

Nmax	Different than Ntotal; it works with Naccepted. The function quits when either it has Naccepted sims or when it has done Nmax attempts. If it hits the max,returns whatever has been simulated so far, possibly list() if nothing.
T	Length of time of the chain.
L	Linear Birth rate.
m	Linear death rate.
nu	Immigration rate.
bd.PO	For bdARsimCondEnd, this is a list of observations essentially; Either class "CTMC_PO_1" or the analogous list.
a	Starting state.
b	Ending state (when you have just one observation).

Details

Outputs a list of BDMC objects. If Naccepted is not NULL then the list will be at least Naccepted long.

Value

List of BDMC objects.

Author(s)

Charles Doss

Examples

```
bdARsimCondEnd.1(Naccepted=10);  #default parameters; simulates at least10.
bdARsimCondEnd.1(Ntotal=10);  #default parameters; maybe end with none.
```

BDloglikelihood.PO *Calculate log likelihood of Partially Observed BD process*

Description

Calculates the log likelihood of a "partially observed birth-death-immigration process."

Usage

```
## S3 method for class 'CTMC_PO_1'
BDloglikelihood.PO(partialDat, L, m, nu,  n.fft = 1024)
## S3 method for class 'CTMC_PO_many'
BDloglikelihood.PO(partialDat, L, m, nu,  n.fft = 1024)
## S3 method for class 'list'
BDloglikelihood.PO(partialDat, L, m, nu,  n.fft = 1024)
BDloglikelihood.PO(partialDat, L, m, nu, n.fft = 1024)
```

Arguments

`L`	lambda, birth rate.
`m`	mu, death rate.
`nu`	nu, Immigration rate.
`partialDat`	Either of class "CTMC_PO_many", or of class "CTMC_PO_1" or the latter's analog in list form, ie a list with the two components "states" and "times" for the "list" and default versions of this method.
`n.fft`	precision for riemann integration / fast fourier transform.

Details

Immigration can be arbitrary here. Calculates likelihood of the b-d-i proces when it is observed at discrete timepoints.

Value

Real number.

Author(s)

charles doss

Examples

```
library(DOBAD)
T=25;
L <- .3
mu <- .6
beta.immig <- 1.2;
initstate <- 17;

#generate process
dat <- birth.death.simulant(t=T, lambda=L, m=mu, nu=L*beta.immig, X0=initstate);
#"observe" process
delta <- 2
partialData <- getPartialData( seq(0,T,delta), dat);
#calculate the likelihood
BDloglikelihood.PO(partialDat=partialData, L=L, m=mu, nu=beta.immig*L);
```

`BDMC-class` *Class "BDMC"*

Description

Birth-Death(-Immigration) CTMCs. Changes in state must be by 1 only.

Objects from the Class

Objects can be created by calls of the form `new("BDMC", ...)`.

Slots

`states`: Object of class "numeric" ~~

`times`: Object of class "numeric" ~~

`T`: Object of class "numeric" ~~

Extends

Class "CTMC", directly.

Methods

BDsummaryStats `signature(sim = "BDMC")`: ...

getStates `signature(object = "BDMC")`: ...

getTimes `signature(object = "BDMC")`: ...

See Also

BDsummaryStats,BDMC-method, getT,BDMC-method

Examples

```
showClass("BDMC")
```

BDMC_many-class   *Class* "BDMC_many"

Description

A vector of BDMCs. Changes in the state for each element list should be by 1 only.

Objects from the Class

Objects can be created by calls of the form `new("BDMC_many", ...)` and supplying a "CTMC_many" object.

Slots

`CTMCs`: Object of class "list", a "CTMC_many" object

Extends

Class "CTMC_many", directly.

Methods

No methods defined with class "BDMC_many" in the signature.

Examples

```
showClass("BDMC_many")
```

`BDPOloglikeGradSqr.CTMC_PO_many`
Gradient-Squared of PartialData likelihood

Description

In Louis' 82 formula for the information of partially observed data, the last term is the gradient-squared of the partial data likelihood. It doesn't have to be calculated because it's 0 at the MLE, but it's coded here for debugging purposes.

Usage

```
BDPOloglikeGradSqr.CTMC_PO_many(partialDat, L, m, beta, n.fft = 1024)
```

Arguments

`partialDat`	CTMC_PO_many.
`L`	lambda at which to calculate information; usually MLE.
`m`	mu at which to calculate information; usually MLE.
`beta`	known constant defining nu via nu=beta*lambda.
`n.fft`	deprecated unused.

`BDsummaryStats` *Get summary statistics for EM Algorithm on Linear Birth-Death Process*

Description

When passed in a birth-death markov chain, this extracts the summary statistics that are needed for computing the MLE (if immigration is a fixed known constant multiple of birth).

That is, BDsummaryStats returns the counts of the total number of jumps up, the total number of jumps down, and the total holding/waiting time (

$$\sum_i d(i) * i$$

, where d(i) is time spent in state i).

BDsummaryStats.PO does something similar, but for a partially observed process.

NijBD takes a BD CTMC and calculates the number of jumps up and the number of jumps down.

waitTimes takes a CTMC and calculates the waiting time.

Usage

```
BDsummaryStats(sim)
BDsummaryStats.PO(dat)
NijBD(BDhist)
NijBD.CTMC_many(BDhists)
waitTimes(stateHist, timeHist, T)
```

Arguments

`sim`	A fully observed BDMC (or list with $states, $times, $T), or a BDMC_many.
`dat`	Partially observed CTMC (list with $states, $times, $T), no "BD" restrictions on the structure of the chain.
`BDhist`	States of a BDMC; can be either a vector of states (each differing from its predecessor by 1) or a BDMC in list or class form.
`BDhists`	CTMC_many object
`stateHist`	Vector of states (integers). Corresponds to timeHist.
`timeHist`	Vector of times (reals). Corresponds to stateHist, i.e. stateHist[i] is the state at and after timeHist[i].
`T`	Total time the chain was observed for.

Details

Assume we have a linear-birth-death process X_t with birth parameter λ, death parameter μ, and immigration parameter $\beta\lambda$ (for some known, real β). We observe the process at a finite set of times over a time interval [0,T].

If the process is fully observed then to calculate the MLEs, we need the number of jumps up, down, and the total holding time. BDsummaryStats takes a BD CTMC and returns these three values, in a vector, with the names "Nplus" and "Nminus" for the number of jumps up and number of jumps down, respectively, and the name "Holdtime" for the total holding time.

If the process is not fully observed, then these statistics aren't known. (The EM algorithm is essentially trying to get a best-guess of these statistics). BDsummaryStats.PO returns, rather, a very naive guess. It pretends that the process is essentially fully observed and computes the statistics from that. Note it's not the same as calling BDsummaryStats since a BD process has stipulations on its format that a partially observed BD process doesn't. The values are returned with the same naming convention as BDsummaryStats.

NijBD takes the list of states of a BD CTMC, and returns a 2x(n+1) matrix, where n is the maximum state the chain visits. NijBD(arg)[1,k] is the number of jumps down from state k-1, and NijBD(arg)[2,k] is the number of jumps up from state k-1.

waitTimes takes any fully observed CTMC and returns a numeric vector of length n+1 where the maximum state passed in is n. The ith entry is the waiting time in the i-1st state. So seq(0, to=n, by=1) %*% waitTimes gives the total holding time.

Value

See details

Author(s)

Charles Doss

See Also

BDMC-class

`birth.death.simulant` *Simulation of birth-death processes with immigration*

Description

A set of functions for simulating and summarizing birth-death simulations

Usage

```
birth.death.simulant(t,X0=1,lambda=1,mu=2,nu=1, condCounts=NULL)
```

Arguments

`t`	length of the time interval
`lambda`	per particle birth rate
`mu`	per particle death rate
`nu`	immigration rate
`X0`	starting state, a non-negative integer
`condCounts`	is either null or a numeric vector with items named "Nplus" and "Nminus" (possibly from BDsummaryStats).

Details

Birth-death process is denoted by X_t

Function birth.death.simulant returns a BDMC object.

Author(s)

Marc A. Suchard

See Also

`add.joint.mean.many`,`add.generator`

Examples

```
my.lambda = 2
my.mu = 3
my.nu =1
my.time = 0.5
my.start = 10
my.end = 2
my.n = 2000

# simulate a birth death trajectory
my.simulant=birth.death.simulant(t=my.time,X0=my.start,lambda=my.lambda,mu=my.mu,nu=my.nu)
print(my.simulant)

# summarize the simulated trajectory
BDsummaryStats(my.simulant)
```

---

bracket-methods *Methods for Function `[` in Package* **DOBAD**

Description

Methods for function `[` in package **DOBAD**

Methods

`signature(x = "CTMC_many", i = "ANY", j = "ANY", drop = "ANY")` Returns a CTMC_many object from the list of CTMCs indicated by the subscripts.

`signature(x = "CTMC_PO_many", i = "ANY", j = "ANY", drop = "ANY")` Returns a CTMC_PO_many object from the list of CTMCs indicated by the subscripts.

---

combineCTMC *Combine several CTMCs into one CTMC*

Description

Pastes together several CTMCs into one. It doesn't check that the rules of the CTMCs are held to.

Usage

```
combineCTMC(sims)
```

Arguments

`sims`	a list each of whose element is a CTMC; so sims[[i]] is a CTMC. sims[[i]] can be of class "CTMC" or a list.

Details

Note that each CTMC should include "0" as its first time. And the last state of sims[[i]] and the first state of sims[[i+1]] should "match" in that the user should check they follow the rules of whatever the generating process is for the CTMC.

Value

Returns a list (not a CTMC object!) with states, times, and T.

`CTMC-class` *Class "CTMC"*

Description

Continuous time Markov Chain class

Objects from the Class

Objects can be created by calls of the form `new("CTMC", ...)`.

Slots

`states`**:** numerics; usually integers.

`times`**:** numerics; an _increasing_ sequence.

`T`**:** final "observation" time of the chain, or time at which it is posited to exist.

Methods

getStates `signature(object = "CTMC")`: ...

getT `signature(object = "CTMC")`: ...

getTimes `signature(object = "CTMC")`: ...

Author(s)

Charles Doss

See Also

getT,CTMC-method

Examples

```
showClass("CTMC")
```

`CTMC.simulate` *Simulate from ("regular") CTMC*

Description

Only CTMCs that have finite number of states to jump directly to starting from all given starting states are allowed.

Usage

```
CTMC.simulate(rate.fn, jumpLim.fn, T.time, init.state)
```

Arguments

`rate.fn`	Rate function from N^2 -> R.
`jumpLim.fn`	Takes a state (integeR) as argument and returns an integer-pair. The 1st entry is the minimum possible state that can be jumped to from the argument as starting point, and the second is the maximum. These must be finite.
`T.time`	length of time to simulate for.
`init.state`	Starting state of sim.

Details

Simulates from a CTMC whose states are the integers. This version requires that each state can only jump to finitely many other states. This information is encapsulated in jumpLim.fn. This isn't fundamental but makes things proceed faster.

Author(s)

Charles Doss

`CTMC.simulate.piecewise` *Simulate from piecewise constant/homogeneous CTMC*

Description

Via the CTMC.simulate function.

Usage

```
CTMC.simulate.piecewise(rate.fns, jumpLim.fns, T.times, init.state)
```

Arguments

`rate.fns`	a LIST of rate functions corresponding to jumpLim.fns and T.times. Length is number of homogeneous pieces, we'll call it M.
`jumpLim.fns`	a LIST of 'jumpLim' functions of length M like the list rate.fns. See the documentation for CTMC.simulate for an explanation of what each is.
`T.times`	Of length M+1 so that there are M intervals corresponding to rate.fns.
`init.state`	A starting state for the simulated chain.

Value

An object of type CTMC.

Author(s)

Charles Doss

`CTMC2list` *Convert Between two representations of a Continuous Time Markov Chain.*

Description

Convert Between two representations of a Continuous Time Markov Chain.

Usage

```
CTMC2list(aCTMC)
```

Arguments

`aCTMC`	CTMC obj

Details

Convert between two representations.

Value

return a list.

`CTMCPO2indepIntervals` *Converts CTMC_PO (either CTMC_PO_1 or CTMC_PO_many) to independent intervals.*

Description

The markov property means that conditional on endpoints, each interval of a markov chain is independent of the others. For this reason computations are often done on intervals.

Usage

```
## S3 method for class 'CTMC_PO_1'
CTMCPO2indepIntervals(partialDat)
## S3 method for class 'CTMC_PO_many'
CTMCPO2indepIntervals(partialDat)
```

Arguments

`partialDat`	CTMC_PO_1 or CTMC_PO_many

Value

This function converts data into a nx3 matrix where the first column is the starting state, the second is the ending state and the third is the length of time the interval spanned. No distinction is made between data from "separate" units or separate intervals from the same markov chain.

`CTMC_PO_1-class` *Class "CTMC_PO_1"*

Description

Partially observed CTMC.

Objects from the Class

Objects can be created by calls of the form `new("CTMC_PO_1", ...)`. Like CTMCs butdon't have an ending time; the final observation time serves that purpose.

Slots

`states`: Object of class `"numeric"` ~~

`times`: Object of class `"numeric"` ~~

Methods

BDsummaryStats.PO `signature(dat = "CTMC_PO_1")`: ...

getStates `signature(object = "CTMC_PO_1")`: ...

getTimes `signature(object = "CTMC_PO_1")`: ...

Author(s)

Charles Doss

Examples

```
showClass("CTMC_PO_1")
```

`CTMC_PO_many-class` *Class "CTMC_PO_many"*

Description

~~ A concise (1-5 lines) description of what the class is. ~~

Objects from the Class

Objects can be created by calls of the form `new("CTMC_PO_many", ...)`. This class is a grouping of data, essentially. CTMC_PO_1 is a series of observations from a single chain; this is several of those single observations together.

Slots

`BDMCsPO`**:** Object of class `"list"` ~~

Methods

No methods defined with class "CTMC_PO_many" in the signature.

Examples

```
showClass("CTMC_PO_many")
```

`derivType` *Helper for getting means from generating functions*

Description

Choose whether to do one-sided or two-sided differentiation. The latter is more effective/less unstable but not always defined.

Usage

```
derivType(L, mu, eps = 1e-04)
```

Arguments

`L`

`mu`

`eps`

Details

Getting the means of interest from generating functions involves differentiation which is usually done numerically. The functions of interest are fully defined on one side of the point of interest but have limited (if any) definition on the other side of the point. For instance, if lambda=mu then the generator for the process N+ is not defined for r>1. If lambda and mu are close then the process is defined for r>1 but very close to 1. The function derivType takes lambda and mu and an epsilon and decides whether that epsilon is small enough to do a two sided derivative with epsilon as "h" or if a one sided derivative is needed.

`doublebracket-methods` *Methods for Function* `[[` *in Package* **DOBAD**

Description

Methods for function `[[` in package **DOBAD**

Methods

`signature(x = "CTMC_many", i = "ANY", j = "ANY", drop = "ANY")` Returns the indicated CTMC object.

`signature(x = "CTMC_PO_many", i = "ANY", j = "ANY", drop = "ANY")` Returns the indicated CTMC_PO_1 object.

EM.BD.SC *Expectation-Maximization on Linear Birth Death (_S_pecial _C_ase with constrained immigration)*

Description

EM Algorithm for estimating rate parameters of a linear Birth-Death process, in which the data is the state at discrete time points

Usage

```
EM.BD.SC(dat, initParamMat, tol = 1e-04, M = 30, beta.immig,  dr =
1e-07, n.fft = 1024, r=4, prec.tol=1e-12, prec.fail.stop=TRUE,
verbose=1, verbFile=NULL)
EM.BD.SC.1(dat,init.params, tol = 0.001, M = 30, beta.immig,  dr =
1e-07, n.fft = 1024, r=4, prec.tol=1e-12, prec.fail.stop=TRUE,
verbose=1, verbFile=NULL)
```

Arguments

`initParamMat`	nx2 matrix. Each row is an initial parameter setting. n is the number of times to run the full EM algorithm. On the nth time the initial "guess" of the lambda is initParamMat[n,1] and of mu it's initParamMat[n,2]. Used to automate starting at dispersed values to ensure global maximum. Frequently n is one.
`init.params`	Vector of length two, first number is the first guess for lambda, second is the guess for mu. This is like a single row from initParamMat.
`M`	Maximum number of iterations for (each) EM algorithm to run through. EM algorithm stops at Mth iteration.
`tol`	Tolerance for EM algorithm; when two iterations are within tol of each other the algorithm ends. Algorithm also ends after M iterations have been reached. (note: One can debate whether 'tol' should refer to the estimates or to the actual likelihood. here it is the estimates, though).
`beta.immig`	Immigration rate is constrained to be a multiple of the birth rate. immigrationrate = beta.immig * lambda where lambda is birth rate.
`n.fft`	Number of terms to use in the fast fourier transform or the riemann integration when using the generating functions to compute probabilities or joint expectations for the birth-death process. See the add.cond.mean.many, etc, functions.
`dat`	Partially observed chain. Either of class "CTMC_PO_many" for several independent histories, of class "CTMC_PO_1" for one history, or a list with components $times and $states where dat$states[i] is the state observed at time dat$times[i] (ie, if it is a list then it is analogous to "CTMC_PO_1").
`dr`	Parameter for numerical differentiation
`r`	Parameter for numerical differentiation; see numDeriv package documentation.

`prec.tol`	"Precision tolerance"; to compute conditional means, first the joint means are computed and then they are normalized by transition probabilities. The precision parameters govern the conditions under which the function will quit if these values are very small. If the joint-mean is smaller than prec.tol then the value of prec.fail.stop decides whether to stop or continue.
`prec.fail.stop`	If true, then when joint-mean values are smaller than prec.tol the program stops; if false then it continues, usually printing a warning.
`verbose`	Chooses level of printing. Increasing from 0, which is no printing.
`verbFile`	Character signifying the file to print to. If NULL just to standard output.

Details

Assume we have a linear-birth-death process X_t with birth parameter λ, death parameter μ, and immigration parameter $\beta\lambda$ (for some known, real β). We observe the process at a finite set of times over a time interval [0,T]. Runs EM algorithm to do maximum likelihood.

EM.BD.SC will run the algorithm on multiple starting values and return the history for the best starting value. EM.BD.SC.1 only runs the algorithm for one starting value. Otherwise they are the same.

Value

Returns a $M + 1$x2 matrix, the nth row being the estimators at the nth iteration. The first column is for lambda (birth), the second for mu (death). If tol is reached before M iterations then many of the rows will be empty, but the M+1st always contains the estimators.

Author(s)

Charles Doss

See Also

`add.cond.mean.many`

`EM.BD.SC.cov.1sv` *Expectation-Maximization on Linear Birth Death (and constrained Immigration) with Covariates*

Description

EM algorithm for maximum likelihood estimation of rate parameters of linear Birth-Death-Immigration processes in which data is the state at discrete time points, and one has covariates.

Usage

```
EM.BD.SC.cov.1sv(BDMCs.PO, ZZ.LL, ZZ.MM, coefs.LL.init, coefs.MM.init,
tol = 1e-04, M = 30, beta.immig,
dr = 1e-07, n.fft = 1024, r = 4,
prec.tol = 1e-12, prec.fail.stop = TRUE,
nlmiterlim = 100, nlmgradtol = 1e-09, nlmstepmax = 0.5, verbose = 1, verbFile = NULL)
```

Arguments

`BDMCs.PO`	Data for doing estimation. See dat argument to EM.BD.SC. Of class "CTMC_PO_many" (several independent histories) or "CTMC_PO_1" (one history).
`ZZ.LL`	Covariates (design matrix) for predicting lambda. (Could be duplicates of ZZ.MM.) The model is: log lambda = ZZ.LL %*% gamma.LL, for some coefficients gamma.LL.
`ZZ.MM`	Covariates (design matrix) for predicting mu. (Could be duplicates of ZZ.LL.) The model is: log mu = ZZ.mu %*% gamma.mu, for some coefficients gamma.mu.
`beta.immig`	Immigration rate is constrained to be a multiple of the birth rate. immigrationrate = beta.immig * lambda where lambda is birth rate.
`coefs.LL.init`	Initial linear coefficients determining lambda, the birth rate.
`coefs.MM.init`	Initial linear coefficients determining mu, the death rate.
`tol`	Tolerance for EM algorithm; when two iterations are within tol of each other the algorithm ends. Algorithm also ends after M iterations have been reached. (note: One can debate whether 'tol' should refer to the estimates or to the actual likelihood. here it is the estimates, though).
`M`	Maximum number of iterations for (each) EM algorithm to run through. EM algorithm stops at Mth iteration.
`dr`	Parameter for numerical differentiation.
`n.fft`	Number of terms to use in the fast fourier transform or the riemann integration when using the generating functions to compute probabilities or joint expectations for the birth-death process. See the add.cond.mean.many, etc, functions.
`r`	Parameter for numerical differentiation; see numDeriv package documentation.
`nlmiterlim`	For optim() call in M.step.SC.cov.
`nlmgradtol`	For optim() call in M.step.SC.cov.
`nlmstepmax`	For optim() call in M.step.SC.cov.
`prec.tol`	"Precision tolerance"; to compute conditional means, first the joint means are computed and then they are normalized by transition probabilities. The precision parameters govern the conditions under which the function will quit if these values are very small. If the joint-mean is smaller than prec.tol then the value of prec.fail.stop decides whether to stop or continue.
`prec.fail.stop`	If true, then when joint-mean values are smaller than prec.tol the program stops; if false then it continues, usually printing a warning.
`verbose`	Chooses level of printing. Increasing from 0, which is no printing.
`verbFile`	Character signifying the file to print to. If NULL just to standard output.

Details

Assume we have a linear-birth-death process X_t with birth parameter λ, death parameter μ, and immigration parameter $\beta\lambda$ (for some known, real β). We use a log linear model so that log lambda = ZZ.LL %*% gamma.LL, for some coefficients gamma.LL and similarly log lambda = ZZ.MM %*% gamma.MM for some coefficients gamma.MM.

We observe the process at a finite set of times over a time interval [0,T]. Runs EM algorithm to do maximum likelihood.

Value

Returns a list with elements coeffs.LL, an $pp.LL$x$M+1$ matrix, and coeffs.MM, an $pp.MM$x$M+1$ matrix where M is the number of EM iterations and pp.LL and pp.MM are the number of coefficients for Lambda and Mu respectively. The M+1 columns gives the final estimators.

Author(s)

Charles R. Doss.

Examples

```
library(Matrix)
library(functional)



set.seed(1234)
mm <- 30; ## num individuals. arbitrary.
pp <- 2; ## num covariates, = HALF the number parameters
ZZ <- matrix(rnorm(mm*pp, -2, .5), nrow=mm, ncol=pp); ## arbitrary ...
ZZ.l1 <- apply(ZZ, 1, Compose(sum,abs))
coefs0.LL <- rnorm(pp, 0, 1)
ZZ <- (1/ZZ.l1)*ZZ ## will need |coefs.LL_j-coefs0.MM.j|< logKK / max( ||z_i||_1)
KK <- 2
diffs0 <- (rbeta(pp, 2,2)-1/2) * log(KK) ## want |lambda-mu| within a factor of KK
coefs0.MM <- coefs0.LL+diffs0;
coefs0 <- matrix(c(coefs0.LL, coefs0.MM), nrow=pp,ncol=2)
theta0 <- exp(ZZ %*% coefs0);
initstates <- rpois(mm, 3)+1
Ts <- abs(rnorm(mm,1,1)) / (theta0[,1]*initstates)
bb <- 1.1; ##beta
arg <- cbind(Ts,theta0, bb*coefs0.LL, initstates);
colnames(arg) <- NULL
BDMCs <- apply(arg, 1,
function(aa){birth.death.simulant(aa[1],aa[5], aa[2],aa[3],aa[4])})
t.obs <- apply(cbind(rpois(mm,2)+2, Ts), 1,
 function(aa){sort(runif(aa[1], 0, aa[2]))}) ##at least 2 observs
BDMCs.PO <- apply(cbind(t.obs,BDMCs), 1,
function(aa){getPartialData(aa[[1]],aa[[2]])})
BDMCs.PO <- new("CTMC_PO_many", BDMCsPO=BDMCs.PO);
```

```
#### Run the EM: (commented for speed for CRAN checks)
##emRes1 <- EM.BD.SC.cov.1sv(BDMCs.PO,
##                           ZZ.LL=ZZ, ZZ.MM=ZZ,
##                      coefs.LL.init=coefs0.LL, ##initialize at truth (which are not MLEs)
##                           coefs.MM.init=coefs0.MM,
##                           tol=1e-4,
##                           M=2, ## for speed; increase.
##                           beta.immig=bb,
##                           dr=1e-7, n.fft=1024, r=4,
##                          prec.tol=1e-12, prec.fail.stop=TRUE,
##                           verbose=1, verbFile="BD_EM_covariates_tutorial.txt")
```

`EMutilities` *Functions related to implementing the EM algorithm on partially observed Birth-Death Chain*

Description

These are functions for the EM algorithm on a partially observed linear birth-death process where the immigration rate is a constant scalar times the birthrate. The ".SC" suffix refers to this constraint ("SC" stands for "Special Case").

E.step.SC performs the "Expectation step" and M.step.SC performs the maximization step.

BDloglikelihood.PO computes the log likelihood of a partially observed birth-death process.

Usage

```
M.step.SC(EMsuffStats, T,beta.immig)
E.step.SC(theData, oldParams, beta.immig,  dr=0.001, n.fft=1024,
                        r=4, prec.tol, prec.fail.stop)
```

Arguments

`EMsuffStats`	Vector with names "Nplus", "Nminus", and "Holdtime", which are the number of jumps up, number of jumps down, and the total holding time, respectively. These often come from the E.step.SC function.
`T`	total Time the chain was observed for (ie usually the last observation time).
`beta.immig`	Immigration rate is constrained to be a multiple of the birth rate. immigrationrate = beta.immig * lambda where lambda is birth rate.
`oldParams`	Parameters with which to compute the expectation
`n.fft`	Number of terms to use in the fast fourier transform or the riemann integration when using the generating functions to compute probabilities or joint expectations for the birth-death process. See the add.joint.mean.many, etc, functions.

`theData`	Partially observed chain. Has components $times and $states where dat$states[i] is the state observed at time dat$times[i]. (No $T component needed).
`dr`	Parameter for numerical differentiation
`r`	Parameter for differentiation; see numDeriv package documentation.
`prec.tol`	"Precision tolerance"; to compute conditional means, first the joint means are computed and then they are normalized by transition probabilities. The precision parameters govern the conditions under which the function will quit if these values are very small. If the joint-mean is smaller than prec.tol then the value of prec.fail.stop decides whether to stop or continue.
`prec.fail.stop`	If true, then when joint-mean values are smaller than prec.tol the program stops; if false then it continues, usually printing a warning.

Details

Assume we have a linear-birth-death process X_t with birth parameter λ, death parameter μ, and immigration parameter $\beta\lambda$ (for some known, real β). We observe the process at a finite set of times over a time interval [0,T].

E.step.SC computes the needed expectations for the EM algorithm. These are the expectations of the sufficient statistics, conditional on the data. These expectations are computed with respect to the measure given by oldParams, i.e. the chain governed by oldParams.

M.Step.SC maximizes the partial-data likelihood given the passed in expecatations of the sufficient statistics, to get the parameter iterates for the next step of the EM algorithm. (This is easy when we are in the "Special Case" where immigration is constrained.)

BDloglikelihood.PO computes the log likelihood of the passed in birth-death process.

Value

M.step.SC returns a length 2 vector with first element lambda-hat and second element mu-hat, the respective maximizers of the likelihood.

E.step.SC returns a vector with names "Nplus", "Nminus", and "Holdtime."

BDloglikelihood.PO returns a real number, the log-likelihood of the data.

Author(s)

Charles Doss

See Also

`EM.BD.SC`

getBDinform *Helpers for Getting Information Matrix for MLE estimates on Partially Observed Linear Birth Death (_S_pecial _C_ase with constrained immigration)*

Description

Assume we have data that is the state at discrete time points of a linear birth-death process, which has immigration parameter constrained to be a known constant times the birth rate. After using EM Algorithm for estimating rate parameters of a linear Birth-Death process, these functions compute matrices related to the information matrix.

Usage

```
getBDinform.full.SC.manual(ENplus, ENminus, L, m)
getBDinform.lost.SC.manual(ENplus, ENminus, EHoldtime,
                             ENplusSq, ENminusSq, EHoldtimeSq,
                             ENplusNminus, ENplusHoldtime, ENminusHoldtime,
                             L, m, beta.immig, T)
getBDinform.PO.SC.manual(ENplus, ENminus, EHoldtime,
                             ENplusSq, ENminusSq, EHoldtimeSq,
                             ENplusNminus, ENplusHoldtime, ENminusHoldtime,
                             L, m, beta.immig, T)
```

Arguments

L	Lambda, birth rate
m	Mu, death rate
beta.immig	Immigration rate is constrained to be a multiple of the birth rate. immigrationrate = beta.immig * lambda where lambda is birth rate.
T	Amount of time process is observed for; corresponds to time window over which all the expectations are computed.
ENplus	Expectation of the N_T^+, the number of jumps up , conditional on the data.
ENminus	Expectation of N_T^-, the number of jumps down, conditional on the data.
EHoldtime	Expectation of R_T^+, the total holdtime, conditional on the data.
ENplusSq	Expectation of N_T^{+2}, the square of the number of jumps up, conditional on the data.
ENminusSq	Expectation of N_T^{-2}, the square of the number of jumps down, conditional on the data.
EHoldtimeSq	Expectation of R_T^2, the square of the total holdtime, conditional on the data.
ENplusNminus	Expectation of $N_T^+ N_T^-$, the product of the number of jumps up and the number of jumps down, conditional on the data.
ENplusHoldtime	Expectation of $N_T^+ R_T$, the product of the number of jumps up and the total holdtime, conditional on the data.

`ENminusHoldtime`
: Expectation of $N_T^- R_T$, the product of the number of jumps down and the total holdtime, conditional on the data.

Details

Assume we have a linear-birth-death process X_t with birth parameter λ, death parameter μ, and immigration parameter $\beta\lambda$ (for some known, real β). We observe the process at a finite set of times over a time interval [0,T]. Can run the EM algorithm to do maximum likelihood. These functions are used to then compute pieces related to the information matrix.

See equations 3.2 and 3.3 in the Louis paper for the notation.

getBDinform.lost.SC.manual computes $I_{X|Y}$.

getBDinform.full.SC.manual computes I_X.

getBDinform.PO.SC.manual computes I_Y (i.e. the difference between the other two functions).

They have the "manual" suffix because the user passes in the expectations. Some of them can be computed analytically by the methods in this package, but others cannot, so those are usually done by Monte Carlo (conditional on the data) simulation.

NOTE: To make sure the answers are coherent, it is important to pass in expectations that are consistent with each other. For instance, if the expectations ENplus, ENminus, and EHoldtime are computed analytically but simulations are used to estimate the rest, then the results may be nonsense, because the values passed in were not necessarily feasible expectations all from the same measure.

Value

Symmetric 2x2 matrix; First row/column corresponds to lambda, second corresponds to mu

Author(s)

Charles Doss

Source

Louis, T A. (1982). Finding the observed information matrix when using the EM algorithm. *J. Roy. Statist. Soc. Ser. B*. 44 226-233.

`getBDinform.PO` *Get Information Matrix for MLE estimates on Partially Observed Linear Birth Death (_S_pecial _C_ase with constrained immigration)*

Description

Assume we have data that is the state at discrete time points of a linear birth-death process, which has immigration parameter constrained to be a known constant times the birth rate. After using EM Algorithm for estimating rate parameters of a linear Birth-Death process, this function gives the information matrix associated.

Usage

```
getBDinform.PO.SC(partialData,Lhat,Mhat, beta.immig,delta=.001,
            n=1024,r=4, prec.tol=1e-12,prec.fail.stop=TRUE)
```

Arguments

`Lhat` MLE for lambda, the birth rate.

`Mhat` MLE for mu, the death rate.

`beta.immig` Immigration rate is constrained to be a multiple of the birth rate. immigrationrate = beta.immig * lambda where lambda is birth rate.

`partialData` Partially observed chain. CTMC_PO_1 or CTMC_PO_many

`n` n for riemann integral approximatoin.

`r, delta,prec.tol,prec.fail.stop` see help for, say, all.cond.mean.PO

Details

Assume we have a linear-birth-death process X_t with birth parameter λ, death parameter μ, and immigration parameter $\beta\lambda$ (for some known, real β). We observe the process at a finite set of times over a time interval [0,T]. After running the EM algorithm to do estimation, this function returns the information to get, for instance, asymptotic CIs.

See the Louis paper for the method.

To calculate the information matrix, the expecatations of the products of the sufficient statistics, conditional on the data, are needed. They are calculated by Monte Carlo, and N is the number of simulations to run.

Value

Symmetric 2x2 matrix; First row/column corresponds to lambda, second corresponds to mu

Author(s)

Charles Doss

Source

Louis, T A. (1982). Finding the observed information matrix when using the EM algorithm. *J. Roy. Statist. Soc. Ser. B*. 44 226-233.

`getBDjTimes` *Get Jump times of a BD process.*

Description

get times of jumps, split into jumps up and jumps down.

Usage

```
getBDjTimes(bdMC, getTimes = TRUE)
```

Arguments

`bdMC`	A BDMC
`getTimes`	Bool. If true returns times, otherwise returns indices of times vector.

Details

List with 2 components of times/indices. First is times of jumps up, second is times of jumps down.

Value

If getTimes is TRUE:

`timesup`	times of jumps up
`timesdown`	times of jumps down

If getTImes is FALSE:

`indsup`	indices of times for jumps up
`indsdown`	indices of times for jumps down

Author(s)

charles doss

`getBDMCsPOlist-methods` *Methods for Function* `getBDMCsPOlist` *in Package* **DOBAD**

Description

Methods for function `getBDMCsPOlist` in package **DOBAD**

Methods

`signature(object = "CTMC_PO_many")` Return the list of CTMC_PO_1 objects corresponding to the CTMC_PO_many argument.

getDataSummary.CTMC_PO_many
Calculate Some Summarizing Information for the Given Data

Description

Computes some summarizing statistics for a CTMC_PO_many object and returns them, possibly also saving them to a file.

Usage

```
getDataSummary.CTMC_PO_many(dat, file = "dataSummary.rsav")
```

Arguments

`dat`	Discretely Observed BDI process.
`file`	Filename to save to.

Details

See the function definition for the variable names used. Saving and loading to/from a file seemed like the simplest approach.

Author(s)

charles doss

getInitParams
Get multiple starting parameters for EM

Description

This is an ad-hoc function that has some hardwired rules for grabbing a few starting parameter values given one passed in one. You pass in your summary data and it makes a basic guess at starting parameters and then flips them and scales them in various ways depending on what numInitParams is. It returns between 1 and 6 different values (pairs) of starting parameters. Useful to get more than one parameter for if you are automating the EM in some way. Otherwise it just gives you the smart starting guess.

Usage

```
getInitParams(numInitParams=1, summary.PO, T, beta.immig, diffScale)
```

Arguments

`numInitParams`	How many parameters you want returned; between 1 and 6. Note: the paramaters after the 1st are fairly arbitrary.
`summary.PO`	Summary data from partially observed process. "Nplus", "Nminus", and "Holdtime" should be names in that order of number of observed jumsp up, jumps down, and Holding time.
`T`	total time of chain.
`beta.immig`	Scalar multiple of lambda that gives you the immigration rate, ie immigrate = beta.immig * birthrate.
`diffScale`	Note that we don't have a solution in the case mu == lambda. So if the two are close then numerical differentiation requires smaller values essentially. So usually pass something like "100*dr" where dr is the value that's passed through the add.joint.mean.*, etc (called delta) for numeric differentiation.

`getIthJumpTime` *Get the jump times from a CTMC.*

Description

Get the time of the ith jump

Usage

```
getIthJumpTime(CTMC, i)
getIthJumpTimes(timesList, i)
getIthState(CTMC,i)
```

Arguments

`CTMC`	A CTMC.
`timesList`	List of positive numerics, each of which is the list of times from a CTMC.
`i`	Positive integer. Which jump to get the time of. Need to know the CTMC(s) jumped at least i times!

Details

Need to know the CTMC(s) jumped at least i times.

Value

getIthJumpTime returns a single positive numeric. getIthJumpTimes returns a vector of positive numerics. getIthState returns a nonnegative integer, the state.

Author(s)

Charles Doss

getMCstate — *Get the state of a CTMC at various times*

Description

Returns the state of the CTMC at each of times in Ts.

Usage

```
getMCstate(CTMC, Ts)
```

Arguments

`CTMC`	a CTMC.
`Ts`	vector of times >0.

getNewParams.SC — *Solve for new parameters in restricted model in EM algorithm.*

Description

Basically one step of the EM algorithm. Given old parameters and the data, get the new parameters.

Usage

```
getNewParams.SC(theData,oldParams, beta.immig,  dr = 0.001, r=4,n.fft =
1024, prec.tol, prec.fail.stop)
```

Arguments

`oldParams`	Parameters from previous iteration
`beta.immig`	immigrationrate = beta.immig * birthrate
`theData`	The discretely observed BDI process. Of class CTMC_PO_many, CTMC_PO_1, list.
`dr`	tuning parameter for differentiation
`r`	Parameter for differentiation; see numDeriv package documentation.
`n.fft`	
`prec.tol`	"Precision tolerance"; to compute conditional means, first the joint means are computed and then they are normalized by transition probabilities. The precision parameters govern the conditions under which the function will quit if these values are very small. If the joint-mean is smaller than prec.tol then the value of prec.fail.stop decides whether to stop or continue.
`prec.fail.stop`	If true, then when joint-mean values are smaller than prec.tol the program stops; if false then it continues, usually printing a warning.

getPartialData — *Get "partially Observed " Chain from a fully observed one.*

Description

This effectively turns "Truth" into "data," ie it is passed a fully observed chain and returns only a partially observed one.

Usage

```
getPartialData(observeTimes, CTMC)
```

Arguments

`observeTimes`	Times at which CTMC is to "be observed" ie at which "data" is to be gathered.
`CTMC`	A continuous time markov chain.

Details

Returns a CTMC_PO_1, ie discretely observed CTMC, from observing CTMC at observeTimes

Value

Returns CTMC_PO_1.

getStates — *Get list of jump states.*

Description

Object accessor.

Usage

```
getStates(object)
```

Arguments

`object`	A CTMC or generalization. has a list of jump states.

Details

Gets list of states at each associated time.

Value

numeric vector, integer valued.

Author(s)

Charles Doss

`getSubMC` *Extract a Sub Markov Chain*

Description

Create a new sub markov chain from a given one.

Usage

```
getSubMC(CTMC, T)
```

Arguments

`CTMC`	A CTMC object
`T`	Time to cut off the given CTMC to form a new one.

Details

Creates a new CTMC identical to the given CTMC from time 0 to T.

Value

a CTMC.

`getT-methods` *~~ Methods for Function getT in Package 'DOBAD' ~~*

Description

~~ Methods for function `getT` in Package 'DOBAD' ~~

Methods

`signature(object = "BDMC")` Same as for CTMC.

`signature(object = "BDMC_many")` Sum of time for each of component BDMCs.

`signature(object = "CTMC")` time the chain is observed for. Ie difference in first time we see the state and the last time.

`signature(object = "CTMC_many")` Sum of time for components

`signature(object = "CTMC_PO_1")` Difference in time first observation and last.

`signature(object = "CTMC_PO_many")` Sum of time for components

getTimes *Get list of jump times.*

Description

Object accessor. First jump time is 0.

Usage

```
getTimes(object)
```

Arguments

`object`	A CTMC. has a list of jump times.

Details

Gets list of jump times.

Value

numeric vector, positive values.

Author(s)

Charles Doss

getTs-methods *~~ Methods for Function getTs in Package 'DOBAD' ~~*

Description

Accessor for vector of total times for each individual markov chain in a many-markov-chain object.

Methods

`signature(object = "BDMC_many")` Vector of total times for each individual markov chain in the object.

`signature(object = "CTMC_many")` Same as above.

`signature(object = "CTMC_PO_many")` Same as above.

graph.CTMC *Plot CTMCs (possibly Partially Observed)*

Description

Plot in piecewise fashion the CTMCs. If it is partially observed, it just plots it as if it were fully observed; i.e., the chain is pretended to continue in the same state until we see a jump.

Usage

```
graph.CTMC(CTMC, filename = NA, height = 6, width = 4.5, xlab="time",
ylab="State", ...)
graph.CTMC.PO(CTMC, filename = NA, height = 6, width = 4.5,
type="l", ...)
```

Arguments

CTMC	Either a fully observed CTMC or a partially observed one. Partially observed ones don't have a "T" and fully observed do.
filename	filename string, or NA.
height	Passed to trellis if filename isn't NA.
width	Passed to trellis if filename isn't NA.
xlab	X label.
ylab	Y label
type	As in the plot parameter.
...	

Details

If your data is S4 class, you can use the plot method.

See Also

See also the s4 methods written for the plot function plot-methods.

list2CTMC — *Convert a list representation of a CTMC to the class version*

Description

Convert a list representation of a CTMC to the class version

Usage

```
list2CTMC(aCTMC)
```

Arguments

`aCTMC`	A CTMC represented as a list. Should have a "states", "times" vectors and a T numeric.

Value

Returns the same data but as an object of class CTMC.

Nij — *Count transitions in a fully observed CTMC.*

Description

Returns a matrix with counts of transitions

Usage

```
Nij(CTMC)
```

Arguments

`CTMC`	(Fully observed) CTMC.

Details

the (i,j) element is the number of transitions from state (i-1) to state (j-1) that were observed.

Value

numeric matrix(ncol=max(CTMC)+1, nrow=max(CTMC)+1) where max(CTMC) is the max state of the CTMC.

`Nplus` *Calculate summary statistics for BDMCs and CTMC_POs*

Description

Nplus (number jumps up), Nminus (number jumps down), and holdtime (waiting time weighted by the waiting state) are fundamental summary statistics for the Restricted Immigration BD model. These functions compute those for BDMC, or compute the observed numbers for CTMC_PO_1 or CTMC_PO_many.

Usage

```
Nplus(sim)
Nminus(sim)
Nplus.CTMC_PO_many(ctmcpomany)
 Nminus.CTMC_PO_many(ctmcpomany)
```

Arguments

`sim`	Arg for Nplus, Nminus. BDMC generally. Needs to have a getStates method.
`ctmcpomany`	A CTMC_PO_many.

Value

Returns an integer, the number of jumps up.

Author(s)

Charles Doss

Examples

```
Nplus(birth.death.simulant(1))
```

`num.deriv` *Numerical Differentiation*

Description

Numerical derivative of one-d function defined on R.

Usage

```
num.deriv(ftn, var, delta = 0.001, ...)
genDoneSided(func, x, sides, method = "Richardson",
method.args = list(eps = 1e-04,
    d = 1e-04, zero.tol = sqrt(.Machine$double.eps/7e-07), r = 4,
    v = 2), ...)
hessianOneSided(func, x, sides, method = "Richardson", method.args = list(eps = 1e-04,
     d = 1e-04, zero.tol = sqrt(.Machine$double.eps/7e-07), r = 4,
     v = 2), ...)
```

Arguments

`ftn, func`	A (differentiable) function defined on R.
`var,x`	Value(s) at which to differentiate. x can be vector.
`sides`	of length equal to x; +1 is differentiate from above -1 from below.
`method`	see numDeriv package docs
`method.args`	see numDeriv package docs
`delta`	Small number defining accuracy of numeric derivative.
`...`	

Details

See the 'numDeriv' package from whence the genD function and hessian function come. The versions here are one-sided adapatations of the originals from that package.

Value

Real number.

---

plot-methods   *Plot CTMCs*

Description

Plotting for fully observed and partially observed Continuous Time Markov Chains.

Methods

x = "CTMC", y = "missing" x is a fully observed CTMC.

x = "CTMC_PO_1", y = "missing" x is discrete observations from a CTMC.

power.coef.one — *Gets coefficients of a power series..*

Description

Reads off coefficients of a power series.

Usage

```
power.coef.one(power.series, n = 1000, k, ...)
power.coef.many(power.series, n = 1024, ...)
```

Arguments

power.series	A function from C to C. Note that its single argument must be named "s". Should be a power series.
n	Parameter for numerical riemann integration or for FFT.
k	The coefficient to get.
...	

Value

A Real number, the kth coefficient, for .one, or a vector of coefficients for .many.

process.prob.one — *Calculate transition probability for linear birth death process.*

Description

Calculate transition probability for linear birth death process.

Usage

```
process.prob.one(t, lambda, mu, nu = 0, X0 = 1, Xt,eps.t=1e-10,
eps.params=1e-10, n = -111)
process.prob.many(t, lambda, mu, nu = 0, X0 = 1, n = 1024)
```

Arguments

`t`	Time for transition.
`lambda`	Linear birth rate
`mu`	linear death rate
`nu`	immigration rate.
`X0`	starting state.
`Xt`	ending state.
`n`	Deprecated; for backwards compatibility.
`eps.t`	One precision level below which the function switches to using the generating function instead of the Orthogonal Polynomial Solution to calculate transition probability. Needed when the parameters or time are close to a boundary for which the OPS isn't defined.
`eps.params`	Another precision level like eps.t.

Details

Calculates P(X_t=Xt |X_0=X0).

`sampleJumpTime2` *Functions for Simulating Conditionally the first Jump of a chain.*

Description

Simulates the time of the first jump given that we know whether it's up or down and have observed the chain at some point.

Usage

```
sampleJumpTime2(T, a, b, up = TRUE, L, m, nu)
p.i(T, a, b, up, L, m, nu, n.fft = 1024, subdivisions = 100)
f.i(t, T, a, b, up, L, m, nu, n.fft = 1024)
```

Arguments

`T`	Time of (First) observation; i.e. time at which we know the state of the chain.
`t`	time between 0 and T at which to get the density of tau for f.i.
`a`	Starting state of the chain at time 0 (X_0=a). a>=0.
`b`	Given State of the chain at time T. (X_T=a). b>=0.
`up`	Boolean, telling whether the first jump is up (TRUE) or down (FALSE).
`L`	Linear birth rate.
`m`	Linear death rate.
`nu`	Immigration rate.
`subdivisions`	Parameter for numerical integration ("integrate" R function).
`n.fft`	Parameter for numerical riemann integration ("by hand").

Details

Let tau be the time of the first jump (after time 0) and X_t is the chain at time t.

Function sampleJumpTime2 simulates the value of the first jump of a BDMC, conditional on some data. What is given is the state of the BDMC at the beginning and end, where the end is time T, as well as whether the first jump is up or down. (To simulate the chain over the time from 0 to T, repeatedly call this function alternatively with p.i)

The Function p.i simulates whether the first jump is up or down, given the data. i.e. if up==true then this returns the probability [tau < T AND X_tau = a+1] and if up==false then it's [tau < T AND X_tau = a-1].

The function f.i returns the "density" at t of tau, ie "P([tau ==t AND X_tau = a+1] | X0=a, Xt=b)" and if up==false then it's "P([tau ==t AND X_tau = a-1] | X0=a, Xt=b)". Note that it doesn't actually integrate to 1. p.i(T) is the integral of f.i to time T. f.i(.)/p.i(T) is actually a density on [0,T]. If X_T != X_0 then we know the first jump is before time T. However, keep in mind the event of interest is that the first jump is up (down) and at time t; even if we know there will be a first jump down, that doesn't prove the first jump won't be up. In general, we have $\int_0^T f.i(up) + \int_0^T f.i(down) + P(firstjumpisaftertimeT) = 1$. That is, $\int_0^t f.i(up)(s)ds$ is the probability the first jump is before time t _and_ it is up (given that the chain starts at a and ends at b).

Value

A time (real number) between 0 and T.

Author(s)

Charles Doss

See Also

`p.i`

`sim.condBD` *Simulate birth-death process, Conditionally upon observing its state at finite, discrete time points*

Description

Functions for simulating a linear-birth-death process with birth parameter lambda, death parameter mu, and immigration parameter modelParams["n"]*lambda, conditional upon observing it at a finite discrete set of times over a finite time interval, [0,T].

Usage

```
sim.condBD(bd.PO = list(states = c(5, 7, 3),times = c(0, 0.4, 1)), N = 1,
           L = 0.5, m = 0.7, nu = 0.4, n.fft = 1024, prevSims=NULL)
## S3 method for class 'CTMC_PO_1'
sim.condBD.main(bd.PO=
      new("CTMC_PO_1", states=c(5,7,3), times=c(0,.4,1)),
      L=.5, m=.7, nu=.4,  n.fft=1024)
## S3 method for class 'list'
sim.condBD.main(bd.PO=
      list(states=c(5,7,3), times=c(0,.4,1)),
      L=.5, m=.7, nu=.4,  n.fft=1024)
## Default S3 method:
sim.condBD.main(bd.PO=
      list(states=c(5,7,3), times=c(0,.4,1)),
      L=.5, m=.7, nu=.4,  n.fft=1024)
sim.condBD.1(T=1, a=5, b=7, L=.5, m=.7, nu=.4, n.fft=1024)
```

Arguments

N	Number of simulations/replications to do
bd.PO	_P_artially _O_bserved process, i.e., the data. Needs to have "components" 'states', 'times', and 'T'. Can be CTMC_PO_1 or a list.
a	Starting state of chain
b	Ending state of chain
T	Duration of chain
L	Lambda, linear birth rate parameter
m	mu, linear death rate parameter
nu	nu, immigration rate parameter.
n.fft	Number of terms to use in the fast fourier transform when using the generating functions to compute probabilities or joint expectations for the birth-death process. See the add.joint.mean.many, etc, functions.
prevSims	A possibly-NULL list of previous simulation results which will be prepended to the current simulations.

Details

sim.condBD, given discretely observed data from a chain, simulates N birthdeath chains conditionally on the data.

sim.condBD.1 is the helper; it simulates one piece of the chain, given a starting and ending state. That process is repeated (via markov prop) to simulate across many data points. So it only takes arguments a,b,and T rather than a "CTMC_PO_1" as its data.

The method of simulating exactly is essentially that of Hobolth and Stone (2008). Briefly: we can write out the density of the time of the first jump up or of the first jump down. We can integrate it from 0 to T to compute the probability there is a jump up (down, respectively) in that time interval. Thus we can simulate whether or not there is a jump, and whether it is up or down. Then using the above mentioned density, we can simulate the time at which it occurs. For more details see Hobolth and Stone (2008).

Value

An object of class BDMC, ie a (linear) Birth-Death Markov Chain, except for sim.condBD which returns a list of objects of class BDMC.

Author(s)

Charles Doss

Source

Hobolth and Stone. (2008) Efficient Simulation from Finite-state, Continuous-Time Markov Chains with Incomplete Observations, submitted Annals of Applied Statistics.

See Also

`add.joint.mean.many`

`simplify` *Transform Lists to Vectors*

Description

Takes objects which are lists but are conceptually vectors, and transforms them into vector objects.

Usage

```
simplify(simpleList)
```

Arguments

`simpleList`	A list each of whose components is a (numeric) vector of length 1.

Details

simpleList is a list each of whose components is a (numeric) vector of length1; simplify returns a vectorized form of this list.

Value

numeric vector whose length is the number of components of simpleList.

Note

The base R unlist function probably makes this redundant.

See Also

unlist

`sub-methods` *Subscripting CTMCs*

Description

Subscripting methods for CTMCs.

Methods

`signature(x = "CTMC_many")` Gets x@CTMC[i]

`signature(x = "CTMC_PO_many")` Gets x@BDMCsPO[i]

`signature(x = "CTMC_many")` Gets x@CTMC[[i]]

`signature(x = "CTMC_PO_many")` Gets x@BDMCsPO[[i]]

Index