Package 'pomp'

February 26, 2015

Type Package

Title Statistical Inference for Partially Observed Markov Processes

Version 0.62-1

Date 2015-02-26

URL http://pomp.r-forge.r-project.org

Description Inference methods for partially observed Markov processes (AKA stochastic dynamical systems, state-space models).

Depends R(>= 3.0.0), methods, subplex, nloptr

Imports stats, graphics, mvtnorm, deSolve, coda

License GPL(>= 2)

LazyData true

MailingList
Subscribe to pomp-announce@r-forge.r-project.org for announcements by going to http://lists.r-forge.r-project.org/mailman/listinfo/pomp-announce.

Collate aaa.R authors.R generics.R eulermultinom.R
csnippet.R pomp-fun.R plugins.R builder.R
parmat.R logmeanexp.R slice-design.R
profile-design.R sobol.R bsplines.R sannbox.R
pomp-class.R load.R pomp.R pomp-methods.R
rmeasure-pomp.R rprocess-pomp.R init-state-pomp.R
dmeasure-pomp.R dprocess-pomp.R skeleton-pomp.R
dprior-pomp.R rprior-pomp.R
simulate-pomp.R trajectory-pomp.R plot-pomp.R
pfilter.R pfilter-methods.R minim.R traj-match.R
bsmc.R bsmc2.R
mif.R mif-methods.R
proposals.R pmcmc.R pmcmc-methods.R
nlf-funcs.R nlf-guts.R nlf-objfun.R nlf.R
probe.R probe-match.R basic-probes.R spect.R spect-match.R
abc.R abc-methods.R
example.R

R topics documented:

pomp-package *Inference for partially observed Markov processes*

Description

The **pomp** package provides facilities for inference on time series data using partially-observed Markov process (POMP) models. These models are also known as state-space models or nonlinear stochastic dynamical systems. One can use **pomp** to fit nonlinear, non-Gaussian dynamic models to time-series data. The package is both a set of tools for data analysis and a platform upon which statistical inference methods for POMP models can be implemented.

Data analysis using pomp

The first step in using **pomp** is to encode one's model(s) and data in objects of class pomp. One does this via a call to pomp, which involves specifying the unobserved state process and the measurement process of the model. Details on this are given in the documentation for the pomp constructor function. Examples are given in the tutorials on the package website, in the demos (demo(package=pomp)), and via the pompExample function.

pomp version 0.62-1 provides algorithms for

1. simulation of stochastic dynamical systems; see simulate
2. particle filtering (AKA sequential Monte Carlo or sequential importance sampling); see pfilter
3. the iterated filtering method of Ionides et al. (2006, 2011, 2015); see mif
4. the nonlinear forecasting algorithm of Kendall et al. (2005); see nlf
5. the particle MCMC approach of Andrieu et al. (2010); see pmcmc
6. the probe-matching method of Kendall et al. (1999, 2005); see probe.match
7. a spectral probe-matching method (Reuman et al. 2006, 2008); see spect.match
8. synthetic likelihood a la Wood (2010); see probe
9. approximate Bayesian computation (Toni et al. 2009); see abc
10. the approximate Bayesian sequential Monte Carlo scheme of Liu & West (2001); see bsmc
11. simple trajectory matching; see traj.match.

The package also provides various tools for plotting and extracting information on models and data.

Developing inference tools on the pomp platform

pomp provides a very general interface to the components of POMP models. All the inference algorithms in **pomp** interact with the models and data via this interface. One goal of the **pomp** project has been to facilitate the development of new algorithms in an environment where they can be tested and compared on a growing body of models and datasets.

The low-level interface relevant to developers is documented here.

Comments, bug reports, feature requests

Contributions are welcome, as are comments, feature requests, and bug reports. See the package website http://pomp.r-forge.r-project.org for more information, access to the package mailing list, links to the authors' websites, references to the literature, and up-to-date versions of the package source and documentation.

Documentation

A number of tutorials, demonstrating the construction of pomp objects and the application of various inference algorithms, are available on the package homepage: http://pomp.r-forge.r-project.org. Several examples of the construction of pomp objects are provided with the package and are documented in the help pages: to view a full list of these, execute `pompExample()`. In addition, there are a number of demos, which can be viewed by executing `demo(package="pomp")`.

History

Much of the groundwork for **pomp** was laid by a working group of the National Center for Ecological Analysis and Synthesis (NCEAS), "Inference for Mechanistic Models".

License

pomp is provided under the GNU Public License (GPL).

Author(s)

Aaron A. King <kingaa at umich dot edu>

References

See the package website, http://pomp.r-forge.r-project.org, for the references.

See Also

pomp, pomp low-level interface, pfilter, simulate, mif, nlf, probe, traj.match, bsmc2, pmcmc

---

Approximate Bayesian computation

Estimation by approximate Bayesian computation (ABC)

Description

The approximate Bayesian computation (ABC) algorithm for estimating the parameters of a partially-observed Markov process.

Usage

```
## S4 method for signature pomp
abc(object, Nabc = 1, start, proposal,
    pars, rw.sd, probes, scale, epsilon,
    verbose = getOption("verbose"), ...)
## S4 method for signature probed.pomp
abc(object, probes,
    verbose = getOption("verbose"), ...)
## S4 method for signature abc
abc(object, Nabc, start, proposal,
    probes, scale, epsilon,
    verbose = getOption("verbose"), ...)
## S4 method for signature abc
continue(object, Nabc = 1, ...)
## S4 method for signature abc
conv.rec(object, pars, ...)
## S4 method for signature abcList
conv.rec(object, ...)
## S4 method for signature abc
plot(x, y, pars, scatter = FALSE, ...)
## S4 method for signature abcList
plot(x, y, ...)
```

Arguments

`object`	An object of class `pomp`.
`Nabc`	The number of ABC iterations to perform.
`start`	named numeric vector; the starting guess of the parameters.
`proposal`	optional function that draws from the proposal distribution. Currently, the proposal distribution must be symmetric for proper inference: it is the user's responsibility to ensure that it is. Several functions that construct appropriate proposal function are provided: see MCMC proposal functions for more information.
`rw.sd`	Deprecated. Will be removed in a future release. Specifying `rw.sd` is equivalent to setting `proposal=mvn.diag.rw(rw.sd)`.
`probes`	List of probes (AKA summary statistics). See `probe` for details.
`scale`	named numeric vector of scales.
`epsilon`	ABC tolerance.
`verbose`	logical; if TRUE, print progress reports.
`pars`	Names of parameters.
`scatter`	optional logical; If `TRUE`, draw scatterplots. If `FALSE`, draw traceplots.
`x`	`abc` object.
`y`	Ignored.
`...`	Additional arguments. These are currently ignored.

Running ABC

abc returns an object of class abc. One or more abc objects can be joined to form an abcList object.

Re-running ABC iterations

To re-run a sequence of ABC iterations, one can use the abc method on a abc object. By default, the same parameters used for the original ABC run are re-used (except for tol, max.fail, and verbose, the defaults of which are shown above). If one does specify additional arguments, these will override the defaults.

Continuing ABC iterations

One can continue a series of ABC iterations from where one left off using the continue method. A call to abc to perform Nabc=m iterations followed by a call to continue to perform Nabc=n iterations will produce precisely the same effect as a single call to abc to perform Nabc=m+n iterations. By default, all the algorithmic parameters are the same as used in the original call to abc. Additional arguments will override the defaults.

Methods

Methods that can be used to manipulate, display, or extract information from an abc object:

conv.rec conv.rec(object, pars) returns the columns of the convergence-record matrix corresponding to the names in pars. By default, all rows are returned.

c Concatenates abc objects into an abcList.

plot Diagnostic plots.

Author(s)

Edward L. Ionides <ionides at umich dot edu>, Aaron A. King <kingaa at umich dot edu>

References

T. Toni and M. P. H. Stumpf, Simulation-based model selection for dynamical systems in systems and population biology, Bioinformatics 26:104–110, 2010.

T. Toni, D. Welch, N. Strelkowa, A. Ipsen, and M. P. H. Stumpf, Approximate Bayesian computation scheme for parameter inference and model selection in dynamical systems Journal of the Royal Society, Interface 6:187–202, 2009.

See Also

pomp, probe, and the tutorials on the package website.

`B-splines` *B-spline bases*

Description

These functions generate B-spline basis functions. `bspline.basis` gives a basis of spline functions. `periodic.bspline.basis` gives a basis of periodic spline functions.

Usage

```
bspline.basis(x, nbasis, degree = 3, names = NULL)
periodic.bspline.basis(x, nbasis, degree = 3, period = 1, names = NULL)
```

Arguments

`x`	Vector at which the spline functions are to be evaluated.
`nbasis`	The number of basis functions to return.
`degree`	Degree of requested B-splines.
`period`	The period of the requested periodic B-splines.
`names`	optional; the names to be given to the basis functions. These will be the column-names of the matrix returned. If the names are specified as a format string (e.g., "basis%d"), `sprintf` will be used to generate the names from the column number. If a single non-format string is specified, the names will be generated by `paste`-ing `name` to the column number. One can also specify each column name explicitly by giving a length-`nbasis` string vector. By default, no column-names are given.

Details

Direct access to the underlying C routines is available. See the header file "pomp.h" for details.

Value

`bspline.basis`	Returns a matrix with `length(x)` rows and `nbasis` columns. Each column contains the values one of the spline basis functions.
`periodic.bspline.basis`	Returns a matrix with `length(x)` rows and `nbasis` columns. The basis functions returned are periodic with period `period`.

Author(s)

Aaron A. King <kingaa at umich dot edu>

Examples

```
x <- seq(0,2,by=0.01)
y <- bspline.basis(x,degree=3,nbasis=9,names="basis")
matplot(x,y,type=l,ylim=c(0,1.1))
lines(x,apply(y,1,sum),lwd=2)

x <- seq(-1,2,by=0.01)
y <- periodic.bspline.basis(x,nbasis=5,names="spline%d")
matplot(x,y,type=l)
```

`basic.probes` *Some probes for partially-observed Markov processes*

Description

Several simple and configurable probes are provided in the package. These can be used directly and as examples for building custom probes.

Usage

```
probe.mean(var, trim = 0, transform = identity, na.rm = TRUE)
probe.median(var, na.rm = TRUE)
probe.var(var, transform = identity, na.rm = TRUE)
probe.sd(var, transform = identity, na.rm = TRUE)
probe.marginal(var, ref, order = 3, diff = 1, transform = identity)
probe.nlar(var, lags, powers, transform = identity)
probe.acf(var, lags, type = c("covariance", "correlation"),
          transform = identity)
probe.ccf(vars, lags, type = c("covariance", "correlation"),
          transform = identity)
probe.period(var, kernel.width, transform = identity)
probe.quantile(var, prob, transform = identity)
```

Arguments

`var, vars`	character; the name(s) of the observed variable(s).
`trim`	the fraction of observations to be trimmed (see `mean`).
`transform`	transformation to be applied to the data before the probe is computed.
`na.rm`	if `TRUE`, remove all NA observations prior to computing the probe.
`kernel.width`	width of modified Daniell smoothing kernel to be used in power-spectrum computation: see `kernel`.
`prob`	a single probability; the quantile to compute: see `quantile`.
`lags`	In `probe.ccf`, a vector of lags between time series. Positive lags correspond to `x` advanced relative to `y`; negative lags, to the reverse. In `probe.nlar`, a vector of lags present in the nonlinear autoregressive model that will be fit to the actual and simulated data. See Details, below, for a precise description.

powers	the powers of each term (corresponding to lags) in the the nonlinear autoregressive model that will be fit to the actual and simulated data. See Details, below, for a precise description.
type	Compute autocorrelation or autocovariance?
ref	empirical reference distribution. Simulated data will be regressed against the values of ref, sorted and, optionally, differenced. The resulting regression coefficients capture information about the shape of the marginal distribution. A good choice for ref is the data itself.
order	order of polynomial regression.
diff	order of differencing to perform.
...	Additional arguments to be passed through to the probe computation.

Details

Each of these functions is relatively simple. See the source code for a complete understanding of what each does.

probe.mean, probe.median, probe.var, probe.sd return functions that compute the mean, median, variance, and standard deviation of variable var, respectively.

probe.period returns a function that estimates the period of the Fourier component of the var series with largest power.

probe.marginal returns a function that regresses the marginal distribution of variable var against the reference distribution ref. If diff>0, the data and the reference distribution are first differenced diff times and centered. Polynomial regression of order order is used. This probe returns order regression coefficients (the intercept is zero).

probe.nlar returns a function that fit a nonlinear (polynomial) autoregressive model to the univariate series (variable var). Specifically, a model of the form $y_t = \sum \beta_k y_{t-\tau_k}^{p_k} + \epsilon_t$ will be fit, where τ_k are the lags and p_k are the powers. The data are first centered. This function returns the regression coefficients, β_k.

probe.acf returns a function that, if type=="covariance", computes the autocovariance of variable var at lags lags; if type=="correlation", computes the autocorrelation of variable var at lags lags.

probe.ccf returns a function that, if type=="covariance", computes the cross covariance of the two variables named in vars at lags lags; if type=="correlation", computes the cross correlation.

probe.quantile returns a function that estimates the prob-th quantile of variable var.

Value

A call to any one of these functions returns a probe function, suitable for use in probe or probe.match. That is, the function returned by each of these takes a data array (such as comes from a call to obs) as input and returns a single numerical value.

Author(s)

Daniel C. Reuman (d.reuman at imperial dot ac dot uk)

Aaron A. King (kingaa at umich dot edu)

References

B. E. Kendall, C. J. Briggs, W. M. Murdoch, P. Turchin, S. P. Ellner, E. McCauley, R. M. Nisbet, S. N. Wood Why do populations cycle? A synthesis of statistical and mechanistic modeling approaches, Ecology, 80:1789–1805, 1999.

S. N. Wood Statistical inference for noisy nonlinear ecological dynamic systems, Nature, 466: 1102–1104, 2010.

See Also

pomp

`blowflies` *Model for Nicholson's blowflies.*

Description

`blowflies1` and `blowflies2` are `pomp` objects encoding stochastic delay-difference models.

Details

The data are from "population I", a control culture in one of A. J. Nicholson's experiments with the Australian sheep-blowfly *Lucilia cuprina*. The experiment is described on pp. 163–4 of Nicholson (1957). Unlimited quantities of larval food were provided; the adult food supply (ground liver) was constant at 0.4g per day. The data were taken from the table provided by Brillinger et al. (1980).

The models are discrete delay equations:

$$R(t+1) \sim \text{Poisson}(PN(t-\tau)\exp\left(-N(t-\tau)/N_0\right)e(t+1)\Delta t)$$

$$S(t+1) \sim \text{binomial}(N(t), \exp\left(-\delta\epsilon(t+1)\Delta t\right))$$

$$N(t) = R(t) + S(t)$$

where $e(t)$ and $\epsilon(t)$ are Gamma-distributed i.i.d. random variables with mean 1 and variances $\sigma_p^2/\Delta t$, $\sigma_d^2/\Delta t$, respectively. `blowflies1` has a timestep (Δt) of 1 day, and `blowflies2` has a timestep of 2 days. The process model in `blowflies1` thus corresponds exactly to that studied by Wood (2010). The measurement model in both cases is taken to be

$$y(t) \sim \text{negbin}(N(t), 1/\sigma_y^2)$$

, i.e., the observations are assumed to be negative-binomially distributed with mean $N(t)$ and variance $N(t) + (\sigma_y N(t))^2$.

Do

```
file.show(system.file("examples","blowflies.R",package="pomp"))
```

to view the code that constructs these pomp objects.

References

A. J. Nicholson (1957) The self-adjustment of populations to change. Cold Spring Harbor Symposia on Quantitative Biology, **22**, 153–173.

Y. Xia and H. Tong (2011) Feature Matching in Time Series Modeling. *Statistical Science* **26**, 21–46.

E. L. Ionides (2011) Discussion of "Feature Matching in Time Series Modeling" by Y. Xia and H. Tong. *Statistical Science* **26**, 49–52.

S. N. Wood (2010) Statistical inference for noisy nonlinear ecological dynamic systems. *Nature* **466**, 1102–1104.

W. S. C. Gurney, S. P. Blythe, and R. M. Nisbet (1980) Nicholson's blowflies revisited. *Nature* **287**, 17–21.

D. R. Brillinger, J. Guckenheimer, P. Guttorp and G. Oster (1980) Empirical modelling of population time series: The case of age and density dependent rates. in G. Oster (ed.), Some Questions in Mathematical Biology, vol. 13, pp. 65–90. American Mathematical Society, Providence.

See Also

pomp

Examples

```
pompExample(blowflies)
plot(blowflies1)
plot(blowflies2)
```

---

bsmc2 *Liu and West Bayesian Particle Filter*

Description

Modified versions of the Liu and West (2001) algorithm.

Usage

```
## S4 method for signature pomp
bsmc(object, params, Np, est, smooth = 0.1,
     ntries = 1, tol = 1e-17, lower = -Inf, upper = Inf, seed = NULL,
     verbose = getOption("verbose"), max.fail = 0,
     transform = FALSE, ...)
## S4 method for signature pomp
bsmc2(object, params, Np, est, smooth = 0.1,
     tol = 1e-17, seed = NULL,
     verbose = getOption("verbose"), max.fail = 0,
     transform = FALSE, ...)
```

Arguments

object	An object of class pomp or inheriting class pomp.
params, Np	Specifications for the prior distribution of particles. See details below.
est	Names of the rows of params that are to be estimated. No updates will be made to the other parameters. If est is not specified, all parameters for which there is variation in params will be estimated.
smooth	Kernel density smoothing parameters. The compensating shrinkage factor will be sqrt(1-smooth^2). Thus, smooth=0 means that no noise will be added to parameters. Generally, the value of smooth should be chosen close to 0 (i.e., shrink~0.1).
ntries	Number of draws from rprocess per particle used to estimate the expected value of the state process at time t+1 given the state and parameters at time t.
tol	Particles with log likelihood below tol are considered to be "lost". A filtering failure occurs when, at some time point, all particles are lost. When all particles are lost, the conditional log likelihood at that time point is set to be log(tol).
lower, upper	optional; lower and upper bounds on the priors. This is useful in case there are box constraints satisfied by the priors. The posterior is guaranteed to lie within these bounds.
seed	optional; an object specifying if and how the random number generator should be initialized ('seeded'). If seed is an integer, it is passed to set.seed prior to any simulation and is returned as the "seed" element of the return list. By default, the state of the random number generator is not changed and the value of .Random.seed on the call is stored in the "seed" element of the return list.
verbose	logical; if TRUE, print diagnostic messages.
max.fail	The maximum number of filtering failures allowed. If the number of filtering failures exceeds this number, execution will terminate with an error.
transform	logical; if TRUE, the algorithm operates on the transformed scale.
...	currently ignored.

Details

There are two ways to specify the prior distribution of particles. If params is unspecified or is a named vector, Np draws are made from the prior distribution, as specified by rprior. Alternatively, params can be specified as an npars x Np matrix (with rownames).

bsmc uses version of the original algorithm that includes a plug-and-play auxiliary particle filter. bsmc2 discards this auxiliary particle filter and appears to give superior performance for the same amount of effort.

Value

An object of class "bsmcd.pomp". The "params" slot of this object will hold the parameter posterior medians. The slots of this class include:

post	A matrix containing draws from the approximate posterior distribution.

prior	A matrix containing draws from the prior distribution (identical to params on call).
eff.sample.size	A vector containing the effective number of particles at each time point.
smooth	The smoothing parameter used (see above).
seed	The state of the random number generator at the time bsmc was called. If the argument seed was specified, this is a copy; if not, this is the internal state of the random number generator at the time of call.
nfail	The number of filtering failures encountered.
cond.log.evidence	A vector containing the conditional log evidence scores at each time point.
log.evidence	The estimated log evidence.
weights	The resampling weights for each particle.

Author(s)

Michael Lavine (lavine at math dot umass dot edu), Matthew Ferrari (mferrari at psu dot edu), Aaron A. King Edward L. Ionides

References

Liu, J. and M. West. Combining Parameter and State Estimation in Simulation-Based Filtering. In A. Doucet, N. de Freitas, and N. J. Gordon, editors, Sequential Monte Carlo Methods in Practice, pages 197-224. Springer, New York, 2001.

See Also

pomp-class

Examples

```
## See the "Introducton to pomp" document for examples.
```

Csnippet *C code snippets*

Description

For including snippets of C code in pomp objects.

Usage

```
Csnippet(text);
```

Arguments

text	character; a snippt of C code.

Value

An object of class `Csnippet`.

Author(s)

Aaron A. King <kingaa at umich dot edu>

`dacca` *Model of cholera transmission for historic Bengal.*

Description

`dacca` is a `pomp` object containing census and cholera mortality data from the Dacca district of the former British province of Bengal over the years 1891 to 1940 together with a stochastic differential equation transmission model. The model is that of King et al. (2008). The parameters are the MLE for the SIRS model with seasonal reservoir.

Data are provided courtesy of Dr. Menno J. Bouma, London School of Tropical Medicine and Hygiene.

Details

`dacca` is a `pomp` object containing the model, data, and MLE parameters. Parameters that naturally range over the positive reals are log-transformed; parameters that range over the unit interval are logit-transformed; parameters that are naturally unbounded or take integer values are not transformed.

References

King, A. A., Ionides, E. L., Pascual, M., and Bouma, M. J. Inapparent infections and cholera dynamics. Nature 454:877-880 (2008)

See Also

`euler.sir`, `pomp`

Examples

```
pompExample(dacca)
plot(dacca)
#MLEs on the natural scale
coef(dacca)
#MLEs on the transformed scale
coef(dacca,transform=TRUE)
plot(simulate(dacca))
# now change eps and simulate again
coef(dacca,"eps") <- 1
plot(simulate(dacca))
```

design *Design matrices for pomp calculations*

Description

These functions are useful for generating designs for the exploration of parameter space. `sobolDesign` generate a Latin hypercube design using the Sobol' low-discrepancy sequence. `profileDesign` generates a data-frame where each row can be used as the starting point for a profile likelihood calculation. `sliceDesign` generates points along slices through a specified point.

Usage

```
sobolDesign(lower, upper, nseq)
profileDesign(..., lower, upper, nprof,
              stringsAsFactors = default.stringsAsFactors())
sliceDesign(center, ...)
```

Arguments

`lower, upper`	named numeric vectors giving the lower and upper bounds of the ranges, respectively.
`...`	In `profileDesign`, additional arguments specify the parameters over which to profile and the values of these parameters. In `sliceDesign`, additional numeric vector arguments specify the locations of points along the slices.
`nseq`	Total number of points requested.
`nprof`	The number of points per profile point.
`stringsAsFactors`	should character vectors be converted to factors?
`center`	`center` is a named numeric vector specifying the point through which the slice(s) is (are) to be taken.

Value

`sobolDesign`

`profileDesign` returns a data frame with `nprof` points per profile point. The other parameters in `vars` are sampled using `sobol`.

Author(s)

Aaron A. King <kingaa at umich dot edu>

References

W. H. Press, S. A. Teukolsky, W. T. Vetterling, \& B. P. Flannery, Numerical Recipes in C, Cambridge University Press, 1992

Examples

```
## Sobol low-discrepancy design
plot(sobolDesign(lower=c(a=0,b=100),upper=c(b=200,a=1),100))

## A one-parameter profile design:
x <- profileDesign(p=1:10,lower=c(a=0,b=0),upper=c(a=1,b=5),nprof=20)
dim(x)
plot(x)

## A two-parameter profile design:
x <- profileDesign(p=1:10,q=3:5,lower=c(a=0,b=0),upper=c(b=5,a=1),nprof=20)
dim(x)
plot(x)

## A single 11-point slice through the point c(A=3,B=8,C=0) along the B direction.
x <- sliceDesign(center=c(A=3,B=8,C=0),B=seq(0,10,by=1))
dim(x)
plot(x)

## Two slices through the same point along the A and C directions.
x <- sliceDesign(c(A=3,B=8,C=0),A=seq(0,5,by=1),C=seq(0,5,length=11))
dim(x)
plot(x)
```

`eulermultinom` *Euler-multinomial death process*

Description

Density and random-deviate generation for the Euler-multinomial death process with parameters `size`, `rate`, and `dt`.

Usage

```
reulermultinom(n = 1, size, rate, dt)
deulermultinom(x, size, rate, dt, log = FALSE)
rgammawn(n = 1, sigma, dt)
```

Arguments

`n`	integer; number of random variates to generate.
`size`	scalar integer; number of individuals at risk.
`rate`	numeric vector of hazard rates.
`sigma`	numeric scalar; intensity of the Gamma white noise process.
`dt`	numeric scalar; duration of Euler step.
`x`	matrix or vector containing number of individuals that have succumbed to each death process.
`log`	logical; if TRUE, return logarithm(s) of probabilities.

Details

If N individuals face constant hazards of death in k ways at rates $r_1, r_2, \ldots, r_k$, then in an interval of duration Δt, the number of individuals remaining alive and dying in each way is multinomially distributed:

$$(N - \sum_{i=1}^{k} \Delta n_i, \Delta n_1, \ldots, \Delta n_k) \sim \text{multinomial}(N; p_0, p_1, \ldots, p_k),$$

where Δn_i is the number of individuals dying in way i over the interval, the probability of remaining alive is $p_0 = \exp(-\sum_i r_i \Delta t)$, and the probability of dying in way j is

$$p_j = \frac{r_j}{\sum_i r_i}(1 - \exp(-\sum_i r_i \Delta t)).$$

In this case, we can say that

$$(\Delta n_1, \ldots, \Delta n_k) \sim \text{eulermultinom}(N, r, \Delta t),$$

where $r = (r_1, \ldots, r_k)$. Draw m random samples from this distribution by doing

`dn <- reulermultinom(n=m,size=N,rate=r,dt=dt)`,

where `r` is the vector of rates. Evaluate the probability that $x = (x_1, \ldots, x_k)$ are the numbers of individuals who have died in each of the k ways over the interval $\Delta t =$`dt`, by doing

`deulermultinom(x=x,size=N,rate=r,dt=dt)`.

Bret\'o & Ionides discuss how an infinitesimally overdispersed death process can be constructed by compounding a binomial process with a Gamma white noise process. The Euler approximation of the resulting process can be obtained as follows. Let the increments of the equidispersed process be given by

`reulermultinom(size=N,rate=r,dt=dt)`.

In this expression, replace the rate r with $r\Delta W/\Delta t$, where $\Delta W \tilde{} Gamma(dt/\sigma^2, \sigma^2)$ is the increment of an integrated Gamma white noise process with intensity σ. That is, ΔW has mean Δt and variance $\sigma^2 \Delta t$. The resulting process is overdispersed and converges (as Δt goes to zero) to a well-defined process. The following lines of R code accomplish this:

```
dW <- rgammawn(sigma=sigma,dt=dt)
dn <- reulermultinom(size=N,rate=r,dt=dW)
```

or

```
dn <- reulermultinom(size=N,rate=r*dW/dt,dt=dt).
```

He et al. use such overdispersed death processes in modeling measles.

For all of the functions described here, direct access to the underlying C routines is available: see the header file "pomp.h", included with the package.

Value

`reulermultinom` Returns a `length(rate)` by `n` matrix. Each column is a different random draw. Each row contains the numbers of individuals succumbed to the corresponding process.

`deulermultinom` Returns a vector (of length equal to the number of columns of `x`) containing the probabilities of observing each column of `x` given the specified parameters (`size`, `rate`, `dt`).

`rgammawn` Returns a vector of length `n` containing random increments of the integrated Gamma white noise process with intensity `sigma`.

Author(s)

Aaron A. King <kingaa at umich dot edu>

References

C. Bret\'o & E. L. Ionides, Compound Markov counting processes and their applications to modeling infinitesimally over-dispersed systems. Stoch. Proc. Appl., 121:2571–2591, 2011.

D. He, E. L. Ionides, & A. A. King, Plug-and-play inference for disease dynamics: measles in large and small populations as a case study. J. R. Soc. Interface, 7:271–283, 2010.

Examples

```
print(dn <- reulermultinom(5,size=100,rate=c(a=1,b=2,c=3),dt=0.1))
deulermultinom(x=dn,size=100,rate=c(1,2,3),dt=0.1)
## an Euler-multinomial with overdispersed transitions:
dt <- 0.1
dW <- rgammawn(sigma=0.1,dt=dt)
print(dn <- reulermultinom(5,size=100,rate=c(a=1,b=2,c=3),dt=dW))
```

---

gompertz *Gompertz model with log-normal observations.*

Description

`gompertz` is a `pomp` object encoding a stochastic Gompertz population model with log-normal measurement error.

Details

The state process is $X_{t+1} = K^{1-S} X_t^S \epsilon_t$, where $S = e^{-r}$ and the ϵ_t are i.i.d. lognormal random deviates with variance σ^2. The observed variables Y_t are distributed as $\mathrm{lognormal}(\log X_t, \tau)$. Parameters include the per-capita growth rate r, the carrying capacity K, the process noise s.d. σ, the measurement error s.d. τ, and the initial condition X_0. The `pomp` object includes parameter transformations that log-transform the parameters for estimation purposes.

See Also

pomp, ricker, and the tutorials at http://pomp.r-forge.r-project.org.

Examples

```
pompExample(gompertz)
plot(gompertz)
coef(gompertz)
coef(gompertz,transform=TRUE)
```

Iterated filtering *Maximum likelihood by iterated filtering*

Description

Iterated filtering algorithms for estimating the parameters of a partially-observed Markov process.

Usage

```
## S4 method for signature pomp
mif(object, Nmif = 1, start, ivps = character(0),
    particles, rw.sd, Np, ic.lag, var.factor,
    cooling.type, cooling.fraction, cooling.factor,
    method = c("mif","unweighted","fp","mif2"),
    tol = 1e-17, max.fail = Inf,
    verbose = getOption("verbose"), transform = FALSE, ...)
## S4 method for signature pfilterd.pomp
mif(object, Nmif = 1, Np, tol, ...)
## S4 method for signature mif
mif(object, Nmif, start, ivps,
    particles, rw.sd, Np, ic.lag, var.factor,
    cooling.type, cooling.fraction,
    method, tol, transform, ...)
## S4 method for signature mif
continue(object, Nmif = 1, ...)
## S4 method for signature mif
logLik(object, ...)
## S4 method for signature mif
conv.rec(object, pars, transform = FALSE, ...)
## S4 method for signature mifList
conv.rec(object, ...)
```

Arguments

object	An object of class pomp.
Nmif	The number of filtering iterations to perform.
start	named numerical vector; the starting guess of the parameters.

`ivps` optional character vector naming the initial-value parameters (IVPs) to be estimated. Every parameter named in `ivps` must have a positive random-walk standard deviation specified in `rw.sd`. If there are no non-IVP parameters with positive `rw.sd`, i.e., only IVPs are to be estimated, see below ""Using `mif` to estimate initial-value parameters only"".

`particles` Function of prototype `particles(Np,center,sd,...)` which sets up the starting particle matrix by drawing a sample of size `Np` from the starting particle distribution centered at `center` and of width `sd`. If `particles` is not supplied by the user, the default behavior is to draw the particles from a multivariate normal distribution with mean `center` and standard deviation `sd`.

`rw.sd` numeric vector with names; the intensity of the random walk to be applied to parameters. The random walk is only applied to parameters named in `pars` (i.e., not to those named in `ivps`). The algorithm requires that the random walk be nontrivial, so each element in `rw.sd[pars]` must be positive. `rw.sd` is also used to scale the initial-value parameters (via the `particles` function). Therefore, each element of `rw.sd[ivps]` must be positive. The following must be satisfied: `names(rw.sd)` must be a subset of `names(start)`, `rw.sd` must be non-negative (zeros are simply ignored), the name of every positive element of `rw.sd` must be in either `pars` or `ivps`.

`Np` the number of particles to use in filtering. This may be specified as a single positive integer, in which case the same number of particles will be used at each timestep. Alternatively, if one wishes the number of particles to vary across timestep, one may specify `Np` either as a vector of positive integers (of length `length(time(object,t0=TRUE))`) or as a function taking a positive integer argument. In the latter case, `Np(k)` must be a single positive integer, representing the number of particles to be used at the `k`-th timestep: `Np(0)` is the number of particles to use going from `timezero(object)` to `time(object)[1]`, `Np(1)`, from `timezero(object)` to `time(object)[1]`, and so on, while when `T=length(time(object,t0=TRUE))`, `Np(T)` is the number of particles to sample at the end of the time-series.

`ic.lag` a positive integer; the timepoint for fixed-lag smoothing of initial-value parameters. The `mif` update for initial-value parameters consists of replacing them by their filtering mean at time `times[ic.lag]`, where `times=time(object)`. It makes no sense to set `ic.lag>length(times)`; if it is so set, `ic.lag` is set to `length(times)` with a warning.

For `method="mif2"`, the default is `ic.lag=length(times)`.

`var.factor` a positive number; the scaling coefficient relating the width of the starting particle distribution to `rw.sd`. In particular, the width of the distribution of particles at the start of the first `mif` iteration will be `random.walk.sd*var.factor`.

`cooling.type, cooling.fraction, cooling.factor`

specifications for the cooling schedule, i.e., the manner in which the intensity of the parameter perturbations is reduced with successive filtering iterations. `cooling.type` specifies the nature of the cooling schedule.

When `cooling.type="geometric"`, on the n-th `mif` iteration, the relative perturbation intensity is `cooling.fraction^(n/50)`.

When `cooling.type="hyperbolic"`, on the n-th `mif` iteration, the relative perturbation intensity is `(s+1)/(s+n)`, where `(s+1)/(s+50)=cooling.fraction`.

`cooling.fraction` is the relative magnitude of the parameter perturbations after 50 `mif` iterations. `cooling.factor` is now deprecated: to achieve the old behavior, use `cooling.type="geometric"` and

`cooling.fraction=(cooling.factor)^50.`

`method` — `method` sets the update rule used in the algorithm. `method="mif"` uses the iterated filtering update rule (Ionides 2006, 2011); `method="unweighted"` updates the parameter to the unweighted average of the filtering means of the parameters at each time; `method="fp"` updates the parameter to the filtering mean at the end of the time series. `method="mif2"` implements an incomplete version of the iterated Bayes map method of Ionides (2015). The latter method is, by every indication, both more efficient and more stable.

`tol, max.fail` — See the description under `pfilter`.

`verbose` — logical; if TRUE, print progress reports.

`transform` — logical; if `TRUE`, optimization is performed on the transformed scale, as defined by the user-supplied parameter transformations (see `pomp`).

`...` — additional arguments that override the defaults.

`pars` — names of parameters.

Re-running `mif` Iterations

To re-run a sequence of `mif` iterations, one can use the `mif` method on a `mif` object. By default, the same parameters used for the original `mif` run are re-used (except for `weighted`, `tol`, `max.fail`, and `verbose`, the defaults of which are shown above). If one does specify additional arguments, these will override the defaults.

Continuing `mif` Iterations

One can resume a series of `mif` iterations from where one left off using the `continue` method. A call to `mif` to perform `Nmif=m` iterations followed by a call to `continue` to perform `Nmif=n` iterations will produce precisely the same effect as a single call to `mif` to perform `Nmif=m+n` iterations. By default, all the algorithmic parameters are the same as used in the original call to `mif`. Additional arguments will override the defaults.

Using `mif` to estimate initial-value parameters only

One can use `mif`'s fixed-lag smoothing to estimate only initial value parameters (IVPs). In this case, `pars` is left empty and the IVPs to be estimated are named in `ivps`. If `theta` is the current parameter vector, then at each `mif` iteration, `Np` particles are drawn from a distribution centered at `theta` and with width proportional to `var.factor*rw.sd`, a particle filtering operation is performed, and `theta` is replaced by the filtering mean at `time(object)[ic.lag]`. Note the implication that, when `mif` is used in this way on a time series any longer than `ic.lag`, unnecessary work is done. If the time series in `object` is longer than `ic.lag`, consider replacing `object` with `window(object,end=ic.lag)`.

Methods

Methods that can be used to manipulate, display, or extract information from a `mif` object:

conv.rec `conv.rec(object, pars = NULL)` returns the columns of the convergence-record matrix corresponding to the names in `pars`. By default, all rows are returned.

logLik Returns the value in the `loglik` slot. NB: this is *not* the same as the likelihood of the model at the MLE!

c Concatenates `mif` objects into an `mifList`.

plot Plots a series of diagnostic plots when applied to a `mif` or `mifList` object.

Details

If `particles` is not specified, the default behavior is to draw the particles from a multivariate normal distribution. **It is the user's responsibility to ensure that, if the optional** `particles` **argument is given, that the** `particles` **function satisfies the following conditions:**

`particles` has at least the following arguments: `Np`, `center`, `sd`, and `...`. `Np` may be assumed to be a positive integer; `center` and `sd` will be named vectors of the same length. Additional arguments may be specified; these will be filled with the elements of the `userdata` slot of the underlying `pomp` object (see `pomp`).

`particles` returns a `length(center)` x `Np` matrix with rownames matching the names of `center` and `sd`. Each column represents a distinct particle.

The center of the particle distribution returned by `particles` should be `center`. The width of the particle distribution should vary monotonically with `sd`. In particular, when `sd=0`, the `particles` should return matrices with `Np` identical columns, each given by the parameters specified in `center`.

Author(s)

Aaron A. King <kingaa at umich dot edu>

References

E. L. Ionides, C. Bret\'o, & A. A. King, Inference for nonlinear dynamical systems, Proc. Natl. Acad. Sci. U.S.A., 103:18438–18443, 2006.

E. L. Ionides, A. Bhadra, Y. Atchad\'e, & A. A. King, Iterated filtering, Annals of Statistics, 39:1776–1802, 2011.

E. L. Ionides, D. Nguyen, Y. Atchad\'e, S. Stoev, and A. A. King. Inference for dynamic and latent variable models via iterated, perturbed Bayes maps. Proc. Natl. Acad. Sci. U.S.A., 112:719–724, 2015.

A. A. King, E. L. Ionides, M. Pascual, and M. J. Bouma, Inapparent infections and cholera dynamics, Nature, 454:877–880, 2008.

See Also

`pomp`, `pfilter`

logmeanexp *The log-mean-exp trick*

Description

logmeanexp computes the log-mean-exp of a set of numbers.

Usage

```
logmeanexp(x, se = FALSE)
```

Arguments

x	numeric
se	logical; give approximate standard error?

Value

log(mean(exp(x))) computed so as to avoid over- or underflow. If se = FALSE, the approximate standard error is returned as well.

Author(s)

Aaron A. King <kingaa at umich dot edu>

Examples

```
## generate a bifurcation diagram for the Ricker map
pompExample(ricker)
ll <- replicate(n=5,logLik(pfilter(ricker,Np=1000)))
## an estimate of the log likelihood:
logmeanexp(ll)
logmeanexp(ll,se=TRUE)
```

LondonYorke *Historical childhood disease incidence data*

Description

LondonYorke is a data-frame containing the monthly number of reported cases of chickenpox, measles, and mumps from two American cities (Baltimore and New York) in the mid-20th century (1928–1972).

Usage

```
data(LondonYorke)
```

References

W. P. London and J. A. Yorke, Recurrent Outbreaks of Measles, Chickenpox and Mumps: I. Seasonal Variation in Contact Rates, American Journal of Epidemiology, 98:453–468, 1973.

Examples

```
data(LondonYorke)

plot(cases~time,data=LondonYorke,subset=disease=="measles",type=n,main="measles",bty=l)
lines(cases~time,data=LondonYorke,subset=disease=="measles"&town=="Baltimore",col="red")
lines(cases~time,data=LondonYorke,subset=disease=="measles"&town=="New York",col="blue")
legend("topright",legend=c("Baltimore","New York"),lty=1,col=c("red","blue"),bty=n)

plot(
     cases~time,
     data=LondonYorke,
     subset=disease=="chickenpox"&town=="New York",
     type=l,col="blue",main="chickenpox, New York",
     bty=l
    )

plot(
     cases~time,
     data=LondonYorke,
     subset=disease=="mumps"&town=="New York",
     type=l,col="blue",main="mumps, New York",
     bty=l
    )
```

Low-level-interface *pomp low-level interface*

Description

A pomp object implements a partially observed Markov process (POMP) model. Basic operations on this model (with shorthand terms) include:

1. simulation of the state process given parameters (rprocess)
2. evaluation of the likelihood of a given state trajectory given parameters (dprocess)
3. simulation of the observation process given the states and parameters (rmeasure)
4. evaluation of the likelihood of a set of observations given the states and parameters (dmeasure)
5. simulation from the prior probability distribution (rprior)
6. evaluation of the prior probability density (dprior)
7. simulation from the distribution of initial states, given parameters (init.state)
8. evaluation of the deterministic skeleton at a point in state space, given parameters (skeleton)
9. computation of a trajetory of the deterministic skeleton given parameters (trajectory)

pomp provides S4 methods that implement each of these basic operations. These operations can be combined to implement computations and statistical inference methods that depend only on a model's POMP structure. For convenience, parameter transformations may also be enclosed in a pomp object.

This page documents these elements.

Usage

```
## S4 method for signature pomp
rprocess(object, xstart, times, params, offset = 0, ...)
## S4 method for signature pomp
dprocess(object, x, times, params, log = FALSE, ...)
## S4 method for signature pomp
rmeasure(object, x, times, params, ...)
## S4 method for signature pomp
dmeasure(object, y, x, times, params, log = FALSE, ...)
## S4 method for signature pomp
dprior(object, params, log = FALSE, ...)
## S4 method for signature pomp
rprior(object, params, ...)
## S4 method for signature pomp
init.state(object, params, t0, ...)
## S4 method for signature pomp
skeleton(object, x, t, params, ...)
## S4 method for signature pomp
trajectory(object, params, times, t0, as.data.frame = FALSE, ...)
## S4 method for signature pomp
pompLoad(object, ...)
## S4 method for signature pomp
pompUnload(object, ...)
```

Arguments

object	an object of class pomp.
xstart	an nvar x nrep matrix containing the starting state of the system. Columns of xstart correspond to states; rows to components of the state vector. One independent simulation will be performed for each column. Note that in this case, params must also have nrep columns.
x	a rank-3 array containing states of the unobserved process. The dimensions of x are nvars x nrep x ntimes, where nvars is the number of state variables, nrep is the number of replicates, and ntimes is the length of times.
y	a matrix containing observations. The dimensions of y are nobs x ntimes, where nobs is the number of observables and ntimes is the length of times.
times, t	a numeric vector (length ntimes) containing times. These must be in non-decreasing order.
params	a npar x nrep matrix of parameters. Each column is an independent parameter set and is paired with the corresponding column of x or xstart. In the case of init.state, params is a named vector of parameters.

offset	integer; the first offset times in times will not be returned.
t0	the initial time at which initial states are requested.
log	if TRUE, log probabilities are returned.
as.data.frame	logical; if TRUE, return the result as a data-frame.
...	In trajectory, additional arguments are passed to the ODE integrator (if the skeleton is a vectorfield) and ignored if it is a map. See ode for a description of the additional arguments accepted. In all other cases, additional arguments are ignored.

rprocess

rprocess simulates the process-model portion of partially-observed Markov process.

When rprocess is called, the first entry of times is taken to be the initial time (i.e., that corresponding to xstart). Subsequent times are the additional times at which the state of the simulated processes are required.

rprocess returns a rank-3 array with rownames. Suppose x is the array returned. Then

```
dim(x)=c(nvars,nrep,ntimes-offset),
```

where nvars is the number of state variables (=nrow(xstart)), nrep is the number of independent realizations simulated (=ncol(xstart)), and ntimes is the length of the vector times. x[,j,k] is the value of the state process in the j-th realization at time times[k+offset]. The rownames of x must correspond to those of xstart.

dprocess

dprocess evaluates the probability density of a sequence of consecutive state transitions.

dprocess returns a matrix of dimensions nrep x ntimes-1. If d is the returned matrix, d[j,k] is the likelihood of the transition from state x[,j,k-1] at time times[k-1] to state x[,j,k] at time times[k].

rmeasure

rmeasure simulate the measurement model given states and parameters.

rmeasure returns a rank-3 array of dimensions nobs x nrep x ntimes, where nobs is the number of observed variables.

dmeasure

dmeasure evaluates the probability density of observations given states.

dmeasure returns a matrix of dimensions nreps x ntimes. If d is the returned matrix, d[j,k] is the likelihood of the observation y[,k] at time times[k] given the state x[,j,k].

dprior, rprior

dprior evaluates the prior probability density and rprior simulates from the prior.

init.state

`init.state` returns an `nvar` x `nrep` matrix of state-process initial conditions when given an `npar` x `nrep` matrix of parameters, `params`, and an initial time `t0`. By default, `t0` is the initial time defined when the `pomp` object ws constructed.

skeleton

The method `skeleton` evaluates the deterministic skeleton at a point or points in state space, given parameters. In the case of a discrete-time system, the skeleton is a map. In the case of a continuous-time system, the skeleton is a vectorfield. NB: `skeleton` just evaluates the deterministic skeleton; it does not iterate or integrate.

`skeleton` returns an array of dimensions `nvar` x `nrep` x `ntimes`. If `f` is the returned matrix, `f[i,j,k]` is the i-th component of the deterministic skeleton at time `times[k]` given the state `x[,j,k]` and parameters `params[,j]`.

trajectory

`trajectory` computes a trajectory of the deterministic skeleton of a Markov process. In the case of a discrete-time system, the deterministic skeleton is a map and a trajectory is obtained by iterating the map. In the case of a continuous-time system, the deterministic skeleton is a vector-field; `trajectory` uses the numerical solvers in **deSolve** to integrate the vectorfield.

`trajectory` returns an array of dimensions `nvar` x `nrep` x `ntimes`. If `x` is the returned matrix, `x[i,j,k]` is the i-th component of the state vector at time `times[k]` given parameters `params[,j]`.

When the skeleton is a vectorfield, `trajectory` integrates it using ode. When the skeleton is a map, `trajectory` iterates it. By default, time is advanced 1 unit per iteration. The user can change this behavior by specifying the desired timestep using the argument `skelmap.delta.t` in the construction of the `pomp` object.

Parameter transformations

User-defined parameter transformations enclosed in the `pomp` object can be accessed via `partrans`.

pompLoad, pompUnload

`pompLoad` and `pompUnload` cause compiled codes associated with `object` to be dynamically linked or unlinked, respectively. When `Csnippets` are used in the construction of a `pomp` object, the resulting shared-object library is dynamically loaded (linked) before each use, and unloaded afterward. These functions are provided because in some instances, greater control may be desired. These functions have no effect on shared-object libraries linked by the user.

Author(s)

Aaron A. King <kingaa at umich dot edu>

See Also

pomp, pomp methods

Examples

```
pompExample(ricker)

p <- parmat(c(r=42,phi=10,sigma=0.3,N.0=7,e.0=0),10)
t <- c(1:10,20,30)
t0 <- 0
x0 <- init.state(ricker,params=p,t0=t0)
x <- rprocess(ricker,xstart=x0,times=c(t0,t),params=p,offset=1)
y <- rmeasure(ricker,params=p,x=x,times=t)
ll <- dmeasure(ricker,y=y[,3,,drop=FALSE],x=x,times=t,params=p,log=TRUE)
apply(ll,1,sum)
f <- skeleton(ricker,x=x,t=t,params=p)
z <- trajectory(ricker,params=p,times=t,t0=t0)

## short arguments are recycled:
p <- c(r=42,phi=10,sigma=0.3,N.0=7,e.0=0)
t <- c(1:10,20,30)
t0 <- 0
x0 <- init.state(ricker,params=p,t0=t0)
x <- rprocess(ricker,xstart=x0,times=c(t0,t),params=p,offset=1)
y <- rmeasure(ricker,params=p,x=x,times=t)
ll <- dmeasure(ricker,y=y,x=x,times=t,params=p,log=TRUE)
f <- skeleton(ricker,x=x,t=t,params=p)
z <- trajectory(ricker,params=p,times=t,t0=t0)
```

`MCMC proposal distributions`

MCMC proposal distributions

Description

Functions to construct proposal distributions for use with MCMC methods.

Usage

```
mvn.diag.rw(rw.sd)
mvn.rw(rw.var)
```

Arguments

`rw.sd`	named numeric vector; random-walk SDs for a multivariate normal random-walk proposal with diagonal variance-covariance matrix.
`rw.var`	square numeric matrix with row- and column-names. Specifies the variance-covariance matrix for a multivariate normal random-walk proposal distribution.

Value

Functions taking a single argument. Given a parameter vector, each returns a single draw from the corresponding proposal distribution.

Author(s)

Aaron A. King <kingaa at umich dot edu>

See Also

pmcmc, abc

---

nlf *Fit Model to Data Using Nonlinear Forecasting (NLF)*

Description

Calls an optimizer to maximize the nonlinear forecasting (NLF) goodness of fit, by simulating data from a model, fitting a nonlinear autoregressive model to the simulated time series (which may be multivariate) and using the fitted model to predict some or all variables in the data time series. NLF is an 'indirect inference' method using a quasi-likelihood as the objective function.

Usage

```
## S4 method for signature pomp
nlf(object, start, est, lags, period = NA, tensor = FALSE,
     nconverge=1000, nasymp=1000, seed = 1066,
     transform.data, nrbf = 4,
   method = c("subplex", "Nelder-Mead", "BFGS", "CG", "L-BFGS-B", "SANN", "Brent"),
     skip.se = FALSE, verbose = getOption("verbose"),
     bootstrap=FALSE, bootsamp = NULL,
     lql.frac = 0.1, se.par.frac = 0.1, eval.only = FALSE,
     transform.params, transform = FALSE, ...)
## S4 method for signature nlfd.pomp
nlf(object, start, est, lags, period, tensor,
     nconverge, nasymp, seed, transform.data, nrbf, method,
     lql.frac, se.par.frac, transform, ...)
```

Arguments

object	A pomp object, with the data and model to fit to it.
start	Named numeric vector with guessed parameters.
est	Vector containing the names or indices of parameters to be estimated.
lags	A vector specifying the lags to use when constructing the nonlinear autoregressive prediction model. The first lag is the prediction interval.
period	numeric; period=NA means the model is nonseasonal. period>0 is the period of seasonal forcing in 'real time'.
tensor	logical; if FALSE, the fitted model is a generalized additive model with time mod period as one of the predictors, i.e., a gam with time-varying intercept. If TRUE, the fitted model is a gam with lagged state variables as predictors and time-periodic coefficients, constructed using tensor products of basis functions of state variables with basis functions of time.

nconverge	Number of convergence timesteps to be discarded from the model simulation.
nasymp	Number of asymptotic timesteps to be recorded from the model simulation.
seed	Integer specifying the random number seed to use. When fitting, it is usually best to always run the simulations with the same sequence of random numbers, which is accomplished by setting seed to an integer. If you want a truly random simulation, set seed=NULL.
transform.params	deprecated. Will be removed in a future version. Use transform instead.
transform	logical; if TRUE, parameters are optimized on the transformed scale.
transform.data	optional function. If specified, forecasting is performed using data and model simulations transformed by this function. By default, transform.data is the identity function, i.e., no transformation is performed. The main purpose of transform.data is to achieve approximately multivariate normal forecasting errors. If data are univariate, transform.data should take a scalar and return a scalar. If data are multivariate, transform.data should assume a vector input and return a vector of the same length.
nrbf	A scalar specifying the number of radial basis functions to be used at each lag.
method	Optimization method. Choices are subplex and any of the methods used by optim.
skip.se	Logical; if TRUE, skip the computation of standard errors.
verbose	Logical; if TRUE, the negative log quasilikelihood and parameter values are printed at each iteration of the optimizer.
bootstrap	Logical; if TRUE the indices in bootsamp will determine which of the conditional likelihood values be used in computing the quasi-loglikelihood.
bootsamp	Vector of integers; used to have the quasi-loglikelihood evaluated using a bootstrap re-sampling of the data set.
lql.frac	target fractional change in log quasi-likelihood for quadratic standard error estimate
se.par.frac	initial parameter-change fraction for quadratic standard error estimate
eval.only	logical; if TRUE, no optimization is attempted and the quasi-loglikelihood value is evaluated at the start parameters.
...	Arguments that will be passed to optim or subplex in the control list.

Details

This runs an optimizer to maximize nlf.objfun.

Value

An object of class nlfd.pomp. logLik applied to such an object returns the log quasi likelihood. The $ method allows extraction of arbitrary slots from the nlfd.pomp object.

Author(s)

Stephen P. Ellner <spe2 at cornell dot edu>, Bruce E. Kendall <kendall at bren dot ucsb dot edu>, Aaron A. King <kingaa at umich dot edu>

References

The following papers describe and motivate the NLF approach to model fitting:

Ellner, S. P., Bailey, B. A., Bobashev, G. V., Gallant, A. R., Grenfell, B. T. and Nychka D. W. (1998) Noise and nonlinearity in measles epidemics: combining mechanistic and statistical approaches to population modeling. *American Naturalist* **151**, 425–440.

Kendall, B. E., Briggs, C. J., Murdoch, W. W., Turchin, P., Ellner, S. P., McCauley, E., Nisbet, R. M. and Wood S. N. (1999) Why do populations cycle? A synthesis of statistical and mechanistic modeling approaches. *Ecology* **80**, 1789–1805.

Kendall, B. E., Ellner, S. P., McCauley, E., Wood, S. N., Briggs, C. J., Murdoch, W. W. and Turchin, P. (2005) Population cycles in the pine looper moth (*Bupalus piniarius*): dynamical tests of mechanistic hypotheses. *Ecological Monographs* **75**, 259–276.

---

ou2 *Two-dimensional discrete-time Ornstein-Uhlenbeck process*

Description

ou2 is a pomp object encoding a bivariate discrete-time Ornstein-Uhlenbeck process.

Details

If the state process is $X(t) = (x_1(t), x_2(t))$, then

$$X(t+1) = \alpha X(t) + \sigma \epsilon(t),$$

where α and σ are 2x2 matrices, σ is lower-triangular, and $\epsilon(t)$ is standard bivariate normal. The observation process is $Y(t) = (y_1(t), y_2(t))$, where $y_i(t) \sim \mathrm{normal}(x_i(t), \tau)$. The functions `rprocess`, `dprocess`, `rmeasure`, `dmeasure`, and `skeleton` are implemented using compiled C code for computational speed: see the source code for details.

See Also

pomp

Examples

```
pompExample(ou2)
plot(ou2)
coef(ou2)
x <- simulate(ou2)
plot(x)
pf <- pfilter(ou2,Np=1000)
logLik(pf)
```

parmat *Create a matrix of parameters*

Description

parmat is a utility that makes a vector of parameters suitable for use in **pomp** functions.

Usage

```
parmat(params, nrep = 1)
```

Arguments

params	named numeric vector or matrix of parameters.
nrep	number of replicates (columns) desired.

Value

parmat returns a matrix consisting of nrep copies of params.

Author(s)

Aaron A. King <kingaa at umich dot edu>

Examples

```
## generate a bifurcation diagram for the Ricker map
pompExample(ricker)
p <- parmat(coef(ricker),nrep=500)
p["r",] <- exp(seq(from=1.5,to=4,length=500))
x <- trajectory(ricker,times=seq(from=1000,to=2000,by=1),params=p)
matplot(p["r",],x["N",,],pch=.,col=black,xlab="log(r)",ylab="N",log=x)
```

Particle filter *Particle filter*

Description

A plain vanilla sequential Monte Carlo (particle filter) algorithm. Resampling is performed at each observation.

Usage

```
## S4 method for signature pomp
pfilter(object, params, Np, tol = 1e-17,
    max.fail = Inf, pred.mean = FALSE, pred.var = FALSE,
    filter.mean = FALSE, save.states = FALSE,
    save.params = FALSE, seed = NULL,
    verbose = getOption("verbose"), ...)
## S4 method for signature pfilterd.pomp
pfilter(object, params, Np, tol, ...)
## S4 method for signature pfilterd.pomp
logLik(object, ...)
## S4 method for signature pfilterd.pomp
cond.logLik(object, ...)
## S4 method for signature pfilterd.pomp
eff.sample.size(object, ...)
## S4 method for signature pfilterd.pomp
pred.mean(object, pars, ...)
## S4 method for signature pfilterd.pomp
pred.var(object, pars, ...)
## S4 method for signature pfilterd.pomp
filter.mean(object, pars, ...)
```

Arguments

`object`	An object of class `pomp` or inheriting class `pomp`.
`params`	A `npars` x `Np` numeric matrix containing the parameters corresponding to the initial state values in `xstart`. This must have a 'rownames' attribute. If it desired that all particles should share the same parameter values, one one may supply `params` as a named numeric vector.
`Np`	the number of particles to use. This may be specified as a single positive integer, in which case the same number of particles will be used at each timestep. Alternatively, if one wishes the number of particles to vary across timesteps, one may specify `Np` either as a vector of positive integers of length `length(time(object,t0=TRUE))` or as a function taking a positive integer argument. In the latter case, `Np(k)` must be a single positive integer, representing the number of particles to be used at the `k`-th timestep: `Np(0)` is the number of particles to use going from `timezero(object)` to `time(object)[1]`, `Np(1)`, from `timezero(object)` to `time(object)[1]`, and so on, while when `T=length(time(object,t0=TRUE))`, `Np(T)` is the number of particles to sample at the end of the time-series. When `object` is of class `mif`, this is by default the same number of particles used in the `mif` iterations.
`tol`	positive numeric scalar; particles with likelihood less than `tol` are considered to be "lost". A filtering failure occurs when, at some time point, all particles are lost. When all particles are lost, the conditional likelihood at that time point is set to `tol`.

`max.fail` integer; the maximum number of filtering failures allowed. If the number of filtering failures exceeds this number, execution will terminate with an error. By default, `max.fail` is set to infinity, so no error can be triggered.

`pred.mean` logical; if `TRUE`, the prediction means are calculated for the state variables and parameters.

`pred.var` logical; if `TRUE`, the prediction variances are calculated for the state variables and parameters.

`filter.mean` logical; if `TRUE`, the filtering means are calculated for the state variables and parameters.

`save.states, save.params` logical. If `save.states=TRUE`, the state-vector for each particle at each time is saved in the `saved.states` slot of the returned `pfilterd.pomp` object. If `save.params=TRUE`, the parameter-vector for each particle at each time is saved in the `saved.params` slot of the returned `pfilterd.pomp` object.

`seed` optional; an object specifying if and how the random number generator should be initialized ('seeded'). If `seed` is an integer, it is passed to `set.seed` prior to any simulation and is returned as the "seed" element of the return list. By default, the state of the random number generator is not changed and the value of `.Random.seed` on the call is stored in the "seed" element of the return list.

`verbose` logical; if `TRUE`, progress information is reported as `pfilter` works.

`pars` Names of parameters.

`...` additional arguments that override the defaults.

Value

An object of class `pfilterd.pomp`. This class inherits from class `pomp` and contains the following additional slots:

pred.mean, pred.var, filter.mean matrices of prediction means, variances, and filter means, respectively. In each of these, the rows correspond to states and parameters (if appropriate), in that order, the columns to successive observations in the time series contained in `object`.

eff.sample.size numeric vector containing the effective number of particles at each time point.

cond.loglik numeric vector containing the conditional log likelihoods at each time point.

saved.states If `pfilter` was called with `save.states=TRUE`, this is the list of state-vectors at each time point, for each particle. It is a length-`ntimes` list of `nvars`-by-`Np` arrays. In particular, `saved.states[[t]][,i]` can be considered a sample from $f[X_t|y_{1:t}]$.

saved.params If `pfilter` was called with `save.params=TRUE`, this is the list of parameter-vectors at each time point, for each particle. It is a length-`ntimes` list of `npars`-by-`Np` arrays. In particular, `saved.params[[t]][,i]` is the parameter portion of the i-th particle at time t.

seed the state of the random number generator at the time `pfilter` was called. If the argument `seed` was specified, this is a copy; if not, this is the internal state of the random number generator at the time of call.

Np, tol, nfail the number of particles used, failure tolerance, and number of filtering failures, respectively.

loglik the estimated log-likelihood.

These can be accessed using the `$` operator as if the returned object were a list. Note that if the argument `params` is a named vector, then these parameters are included in the `params` slot of the returned `pfilterd.pomp` object.

Methods

logLik Extracts the estimated log likelihood.

cond.logLik Extracts the estimated conditional log likelihood

$$\ell_t(\theta) = \mathrm{Prob}[y_t|y_1,\ldots,y_{t-1}]$$

eff.sample.size Extracts the (time-dependent) estimated effective sample size.

pred.mean, pred.var, filter.mean Extract the mean and variance of the approximate prediction distribution and the mean of the filtering distribution, respectively.

Author(s)

Aaron A. King <kingaa at umich dot edu>

References

M. S. Arulampalam, S. Maskell, N. Gordon, & T. Clapp. A Tutorial on Particle Filters for Online Nonlinear, Non-Gaussian Bayesian Tracking. IEEE Trans. Sig. Proc. 50:174–188, 2002.

See Also

pomp, mif, pmcmc, bsmc2, and the tutorials on the package website.

Examples

```
pompExample(gompertz)
pf <- pfilter(gompertz,Np=1000) ## use 1000 particles
plot(pf)
logLik(pf)
cond.logLik(pf) ## conditional log-likelihoods
eff.sample.size(pf)             ## effective sample size
logLik(pfilter(pf))       ## run it again with 1000 particles
## run it again with 2000 particles
pf <- pfilter(pf,Np=2000,filter.mean=TRUE)
fm <- filter.mean(pf)     ## extract the filtering means
```

plugins *Plug-ins for state-process models*

Description

Plug-in facilities for implementing discrete-time Markov processes and continuous-time Markov processes using the Euler algorithm. These can be used in the `rprocess` and `dprocess` slots of `pomp`.

Usage

```
onestep.sim(step.fun, PACKAGE)
euler.sim(step.fun, delta.t, PACKAGE)
discrete.time.sim(step.fun, delta.t = 1, PACKAGE)
gillespie.sim(rate.fun, v, d, PACKAGE)
onestep.dens(dens.fun, PACKAGE)
```

Arguments

`step.fun`	This can be either an R function, a `Csnippet`, or the name of a compiled, dynamically loaded native function containing the model simulator. It should be written to take a single Euler step from a single point in state space. If it is an R function, it should have prototype `step.fun(x,t,params,delta.t,...)`. Here, `x` is a named numeric vector containing the value of the state process at time `t`, `params` is a named numeric vector containing parameters, and `delta.t` is the length of the Euler time-step. For examples on the use of `Csnippet` to write fast simulators easily, see the tutorials on the package website. If `step.fun` is the name of a native function, it must be of type "pomp_onestep_sim" as defined in the header "pomp.h", which is included with the **pomp** package. For details on how to write such codes, see Details.
`rate.fun`	This can be either an R function, a `Csnippet`, or the name of a compiled, dynamically loaded native function that computes the transition rates. If it is an R function, it should be of the form `rate.fun(j,x,t,params,...)`. Here, `j` is the number of the event, `x` is a named numeric vector containing the value of the state process at time `t` and `params` is a named numeric vector containing parameters. For examples on the use of `Csnippet` to write fast simulators easily, see tutorials on the package website. If `rate.fun` is a native function, it must be of type "pomp_ssa_rate_fn" as defined in the header "pomp.h", which is included with the package. For details on how to write such codes, see Details.
`v, d`	Matrices that specify the continuous-time Markov process in terms of its elementary events. Each should have dimensions `nvar` x `nevent`, where `nvar` is the number of state variables and `nevent` is the number of elementary events. `v`

describes the changes that occur in each elementary event: it will usually comprise the values 1, -1, and 0 according to whether a state variable is incremented, decremented, or unchanged in an elementary event. `d` is a binary matrix that describes the dependencies of elementary event rates on state variables: `d[i,j]` will have value 1 if event rate `j` must be updated as a result of a change in state variable `i` and 0 otherwise

`dens.fun` This can be either an R function, a `Csnippet`, or a compiled, dynamically loaded native function containing the model transition log probability density function.

If it is an R function, it should be of the form `dens.fun(x1,x2,t1,t2,params,...)`. Here, `x1` and `x2` are named numeric vectors containing the values of the state process at times `t1` and `t2`, `params` is a named numeric vector containing parameters.

If `dens.fun` is the name of a native function, it should be of type "pomp_onestep_pdf" as defined in the header "pomp.h", which is included with the **pomp** package. This function should return the log likelihood of a transition from `x1` at time `t1` to `x2` at time `t2`, assuming that no intervening transitions have occurred. For details on how to write such codes, see Details.

`delta.t` Size of Euler time-steps.

`PACKAGE` an optional argument that specifies to which dynamically loaded library we restrict the search for the native routines. If this is "base", we search in the R executable itself. This argument is ignored if `step.fun`, `rate.fn`, or `dens.fun` is provided as an R function or a `Csnippet`.

Details

`onestep.sim` is the appropriate choice when it is possible to simulate the change in state from one time to another, regardless of how large the interval between them is. To use `onestep.sim`, you must write a function `step.fun` that will advance the state process from one arbitrary time to another. `euler.sim` is appropriate when one cannot do this but can compute the change in state via a sequence of smaller steps. This is desirable, for example, if one is simulating a continuous time process but is willing to approximate it using an Euler approach. `discrete.time.sim` is appropriate when the process evolves in discrete time. In this case, by default, the intervals between observations are integers.

To use `euler.sim` or `discrete.time.sim`, you must write a function `step.fun` that will take a single Euler step, of size at most `delta.t`. `euler.sim` and `discrete.time.sim` will create simulators that take as many steps as needed to get from one time to another. See below for information on how `euler.sim` chooses the actual step size it uses.

`gillespie.sim` allows exact simulation of a continuous-time, discrete-state Markov process using Gillespie's algorithm. This is an "event-driven" approach: correspondingly, to use `gillespie.sim`, you must write a function `rate.fun` that computes the rates of each elementary event and specify two matrices (`d`,`v`) that describe, respectively, the dependencies of each rate and the consequences of each event.

`onestep.dens` will generate a suitable `dprocess` function when one can compute the likelihood of a given state transition simply by knowing the states at two times under the assumption that the state has not changed between the times. This is typically possible, for instance, when the `rprocess` function is implemented using `onestep.sim`, `euler.sim`, or `discrete.time.sim`. [NB:

currently, there are no high-level algorithms in **pomp** that use dprocess. This function is provided for completeness only, and with an eye toward future development.]

If step.fun is written as an R function, it must have at least the arguments x, t, params, delta.t, and .... On a call to this function, x will be a named vector of state variables, t a scalar time, and params a named vector of parameters. The length of the Euler step will be delta.t. If the argument covars is included and a covariate table has been included in the pomp object, then on a call to this function, covars will be filled with the values, at time t, of the covariates. This is accomplished via interpolation of the covariate table. Additional arguments may be given: these will be filled by the correspondingly-named elements in the userdata slot of the pomp object (see pomp). If step.fun is written in a native language, it must be a function of type "pomp_onestep_sim" as specified in the header "pomp.h" included with the package (see the directory "include" in the installed package directory).

If rate.fun is written as an R function, it must have at least the arguments j, x, t, params, and .... Here, j is the an integer that indicates which specific elementary event we desire the rate of. x is a named vector containing the value of the state process at time t, and params is a named vector containing parameters. If the argument covars is included and a covariate table has been included in the pomp object, then on a call to this function, covars will be filled with the values, at time t, of the covariates. This is accomplished via interpolation of the covariate table. If rate.fun is a native function, it must be of type "pomp_ssa_rate_fn" as defined in the header "pomp.h", which is included with the package.

In writing dens.fun, you must assume that no state transitions have occurred between t1 and t2. If dens.fun is written as an R function, it must have at least the arguments x1, x2, t1, t2, params, and .... On a call to this function, x1 and x2 will be named vectors of state variables at times t1 and t2, respectively. The named vector params contains the parameters. If the argument covars is included and a covariate table has been included in the pomp object, then on a call to this function, covars will be filled with the values, at time t1, of the covariates. If the argument covars is included and a covariate table has been included in the pomp object, then on a call to this function, covars will be filled with the values, at time t1, of the covariates. This is accomplished via interpolation of the covariate table. As above, any additional arguments will be filled by the correspondingly-named elements in the userdata slot of the pomp object (see pomp). If dens.fun is written in a native language, it must be a function of type "pomp_onestep_pdf" as defined in the header "pomp.h" included with the package (see the directory "include" in the installed package directory).

Value

onestep.sim, euler.sim, discrete.time.sim, and gillespie.sim each return functions suitable for use as the argument rprocess argument in pomp.

onestep.dens returns a function suitable for use as the argument dprocess in pomp.

Author(s)

Aaron A. King <kingaa at umich dot edu>

See Also

eulermultinom, pomp

pmcmc *The PMCMC algorithm*

Description

The Particle MCMC algorithm for estimating the parameters of a partially-observed Markov process.

Usage

```
## S4 method for signature pomp
pmcmc(object, Nmcmc = 1, start, proposal, pars, rw.sd, Np,
    tol = 1e-17, max.fail = 0, verbose = getOption("verbose"), ...)
## S4 method for signature pfilterd.pomp
pmcmc(object, Nmcmc = 1, Np, tol, ...)
## S4 method for signature pmcmc
pmcmc(object, Nmcmc, start, proposal, Np, tol,
    max.fail = 0, verbose = getOption("verbose"), ...)
## S4 method for signature pmcmc
continue(object, Nmcmc = 1, ...)
```

Arguments

`object`	An object of class pomp.
`Nmcmc`	The number of PMCMC iterations to perform.
`start`	named numeric vector; the starting guess of the parameters.
`proposal`	optional function that draws from the proposal distribution. Currently, the proposal distribution must be symmetric for proper inference: it is the user's responsibility to ensure that it is. Several functions that construct appropriate proposal function are provided: see MCMC proposal functions for more information.
`pars, rw.sd`	Deprecated. Will be removed in a future release.
`Np`	a positive integer; the number of particles to use in each filtering operation.
`tol`	numeric scalar; particles with log likelihood below `tol` are considered to be "lost". A filtering failure occurs when, at some time point, all particles are lost.
`max.fail`	integer; maximum number of filtering failures permitted. If the number of failures exceeds this number, execution will terminate with an error.
`verbose`	logical; if TRUE, print progress reports.
`...`	Additional arguments. These are currently ignored.

Value

An object of class pmcmc.

Re-running PMCMC Iterations

To re-run a sequence of PMCMC iterations, one can use the pmcmc method on a pmcmc object. By default, the same parameters used for the original PMCMC run are re-used (except for tol, max.fail, and verbose, the defaults of which are shown above). If one does specify additional arguments, these will override the defaults.

Continuing PMCMC Iterations

One can continue a series of PMCMC iterations from where one left off using the continue method. A call to pmcmc to perform Nmcmc=m iterations followed by a call to continue to perform Nmcmc=n iterations will produce precisely the same effect as a single call to pmcmc to perform Nmcmc=m+n iterations. By default, all the algorithmic parameters are the same as used in the original call to pmcmc. Additional arguments will override the defaults.

Details

pmcmc implements an MCMC algorithm in which the true likelihood of the data is replaced by an unbiased estimate computed by a particle filter. This gives an asymptotically correct Bayesian procedure for parameter estimation (Andrieu and Roberts, 2009). An extension to give a correct Bayesian posterior distribution of unobserved state variables (as in Andrieu et al, 2010) has not yet been implemented.

Author(s)

Edward L. Ionides <ionides at umich dot edu>, Aaron A. King <kingaa at umich dot edu>

References

C. Andrieu, A. Doucet and R. Holenstein, Particle Markov chain Monte Carlo methods, J. R. Stat. Soc. B, to appear, 2010.

C. Andrieu and G.O. Roberts, The pseudo-marginal approach for efficient computation, Ann. Stat. 37:697-725, 2009.

See Also

pmcmc-class, pmcmc-methods, pomp, pomp-class, pfilter.

---

pmcmc-methods *Methods of the "pmcmc" class*

Description

Methods of the "pmcmc" class.

Usage

```
## S4 method for signature pmcmc
logLik(object, ...)
## S4 method for signature pmcmc
conv.rec(object, pars, ...)
## S4 method for signature pmcmcList
conv.rec(object, ...)
## S4 method for signature pmcmc
plot(x, y, ...)
## S4 method for signature pmcmcList
plot(x, y, ...)
## S4 method for signature pmcmc
c(x, ..., recursive = FALSE)
## S4 method for signature pmcmcList
c(x, ..., recursive = FALSE)
```

Arguments

`object, x`	The `pmcmc` or `pmcmcList` object.
`pars`	Names of parameters.
`y, recursive`	Ignored.
`...`	Further arguments (either ignored or passed to underlying functions).

Methods

conv.rec `conv.rec(object, pars)` returns the columns of the convergence-record matrix corresponding to the names in `pars` as an object of class `mcmc` or `mcmc.list`.

plot Diagnostic plots.

logLik Returns the value in the `loglik` slot.

c Concatenates `pmcmc` objects into a `pmcmcList`.

Author(s)

Edward L. Ionides <ionides at umich dot edu>, Aaron A. King <kingaa at umich dot edu>

References

C. Andrieu, A. Doucet and R. Holenstein, Particle Markov chain Monte Carlo methods, J. Roy. Stat. Soc B, to appear, 2010.

C. Andrieu and G.O. Roberts, The pseudo-marginal approach for efficient computation, Ann Stat 37:697-725, 2009.

See Also

`pmcmc`, `pomp`, `pomp-class`, `pfilter`

pomp *Partially-observed Markov process object constructor.*

Description

This function constructs a pomp object, encoding a partially-observed Markov process model together with a uni- or multivariate time series. One implements the model by specifying its *components*, each of which can be written as R functions or, for much greater computational efficiency, using C code. The preferred way to specify most components (as detailed below) is through the use of Csnippets, snippets of C that are compiled and linked into a running R session.

Usage

```
## S4 method for signature data.frame
pomp(data, times, t0, ..., rprocess, dprocess, rmeasure, dmeasure,
       measurement.model,
       skeleton, skeleton.type = c("map","vectorfield"), skelmap.delta.t = 1,
       initializer, rprior, dprior, params, covar, tcovar,
       obsnames, statenames, paramnames, covarnames, zeronames,
       PACKAGE, parameter.transform, parameter.inv.transform, globals)
## S4 method for signature numeric
pomp(data, times, t0, ..., rprocess, dprocess, rmeasure, dmeasure,
       measurement.model,
       skeleton, skeleton.type = c("map","vectorfield"), skelmap.delta.t = 1,
       initializer, rprior, dprior, params, covar, tcovar,
       obsnames, statenames, paramnames, covarnames, zeronames,
       PACKAGE, parameter.transform, parameter.inv.transform, globals)
## S4 method for signature matrix
pomp(data, times, t0, ..., rprocess, dprocess, rmeasure, dmeasure,
       measurement.model,
       skeleton, skeleton.type = c("map","vectorfield"), skelmap.delta.t = 1,
       initializer, rprior, dprior, params, covar, tcovar,
       obsnames, statenames, paramnames, covarnames, zeronames,
       PACKAGE, parameter.transform, parameter.inv.transform, globals)
## S4 method for signature pomp
pomp(data, times, t0, ..., rprocess, dprocess, rmeasure, dmeasure,
       measurement.model, skeleton, skeleton.type, skelmap.delta.t,
       initializer, rprior, dprior, params, covar, tcovar,
       obsnames, statenames, paramnames, covarnames, zeronames,
       PACKAGE, parameter.transform, parameter.inv.transform, globals)
```

Arguments

data, times	The time series data and times at which observations are made. data can be specified as a vector, a matrix, a data-frame. Alternatively, a pomp object can be supplied in the data argument. If data is a numeric vector, times must be a numeric vector of the same length.

If data is a matrix, it should have dimensions nobs x ntimes, where nobs is the number of observed variables and ntimes is the number of times at which observations were made (i.e., each column is a distinct observation of the nobs variables). In this case, times must be given as a numeric vector (of length ntimes).

If data is a data-frame, times must name the column of observation times. Note that, in this case, data will be internally coerced to an array with storage-mode 'double'.

times must be numeric and strictly increasing.

t0 — The zero-time, at which the stochastic dynamical system is to be initialized. This must be no later than the time of the first observation, i.e., t0 <= times[1].

rprocess — optional function; a function of prototype

```
rprocess(xstart,times,params,...)
```

that simulates from the unobserved process. The form of this function is given below. **pomp** provides a number of plugins that construct appropriate rprocess arguments corresponding to several common stochastic simulation algorithms. See below for more details.

dprocess — optional function; a function of prototype

```
dprocess(x,times,params,log,...)
```

that evaluates the likelihood of a sequence of consecutive state transitions. The form of this function is given below. It is not typically necessary (or even feasible) to define dprocess. This functionality is provided to support future algorithm development.

rmeasure — optional; the measurement model simulator. This can be specified in one of four ways: (1) as a function of prototype

```
rmeasure(x,t,params,...)
```

that makes a draw from the observation process given states x, time t, and parameters params. (2) as the name of a native (compiled) routine with prototype pomp_measure_model_simulator as defined in the header file 'pomp.h'. (To view the header file, execute

```
file.show(system.file("include/pomp.h",package="pomp"))
```

in an R session.) (3) using the formula-based measurement.model facility (see below). (4) as a snippet of C code (via Csnippet) that draws from the observation process as above. The last is typically the preferred option, as it results in much faster code execution.

dmeasure — optional; the measurement model probability density function. This can be specified in one of four ways: (1) as a function of prototype

```
dmeasure(y,x,t,params,log,...)
```

that computes the p.d.f. of y given x, t, and params. (2) as the name of a native (compiled) routine with prototype pomp_measure_model_density as defined in the header file 'pomp.h'. (To view the header file, execute

```
file.show(system.file("include/pomp.h",package="pomp"))
```

in an R session.) (3) using the formula-based `measurement.model` facility (see below). (4) as a snippet of C code (via `Csnippet`) that computes the p.d.f. as above. The last is typically the preferred option, as it results in much faster code execution. As might be expected, if `log=TRUE`, this function should return the log likelihood.

`measurement.model`

optional; a formula or list of formulae, specifying the measurement model. These formulae are parsed internally to generate `rmeasure` and `dmeasure` functions. If `measurement.model` is given it overrides any specification of `rmeasure` or `dmeasure`. **NB:** This is a convenience function, primarily designed to facilitate exploration; it will typically be possible to acclerate measurement model computations by writing `dmeasure` and/or `rmeasure` using `Csnippet`s.

`skeleton, skeleton.type, skelmap.delta.t`

The function `skeleton` specifies the deterministic skeleton of the unobserved Markov process. If we are dealing with a discrete-time Markov process, its deterministic skeleton is a map: indicate this by specifying `skeleton.type="map"`. In this case, the default assumption is that time advances 1 unit per iteration of the map; to change this, set `skelmap.delta.t` to the appropriate time-step. If we are dealing with a continuous-time Markov process, its deterministic skeleton is a vectorfield: indicate this by specifying `skeleton.type="vectorfield"`.

The skeleton function can be specified in one of three ways: (1) as an R function of prototype

```
skeleton(x,t,params,...)
```

that evaluates the deterministic skeleton at state `x` and time `t` given the parameters `params`, (2) as the name of a native (compiled) routine with prototype `pomp_skeleton` as defined in the header file 'pomp.h'. (To view the header file, execute

```
file.show(system.file("include/pomp.h",package="pomp"))
```

in an R session.) (3) as a snippet of C code (via `Csnippet`) that performs this evaluation. The latter is typically the preferred option, for reasons of computational efficiency.

`initializer` optional function of prototype

```
initializer(params,t0,...)
```

that yields initial conditions for the state process when given a vector, `params`, of parameters.

By default, any parameters in `params`, the names of which end in "`.0`", are assumed to be initial values of states. To initialize the unobserved state process, these are simply copied over as initial conditions. The names of the resulting state variables are obtained by dropping the "`.0`" suffix.

`rprior` optional; function drawing a sample from a prior distribution on parameters. This can be specified in one of three ways: (1) as an R function of prototype

```
rprior(params,...)
```

that makes a draw from the prior distribution given `params`, (2) as the name of a native (compiled) routine with prototype "pomp_rprior" as defined in the header file 'pomp.h', or (To view the header file, execute

```
file.show(system.file("include/pomp.h",package="pomp"))
```

in an R session.) (3) as a snippet of C code (via `Csnippet`). As above, the latter is typically preferable.

`dprior` optional; function evaluating the prior distribution. This can be specified in one of three ways: (1) as an R function of prototype

```
dprior(params,log=FALSE,...)
```

that evaluates the prior probability density, (2) as the name of a native (compiled) routine with prototype "pomp_dprior" as defined in the header file 'pomp.h', or (To view the header file, execute

```
file.show(system.file("include/pomp.h",package="pomp"))
```

in an R session.) (3) as a snippet of C code (via `Csnippet`). As above, the latter is typically preferable.

`params` optional named numeric vector of parameters. This will be coerced internally to storage mode `double`.

`covar, tcovar` An optional matrix or data frame of covariates: `covar` is the table of covariates (one column per variable); `tcovar` the corresponding times (one entry per row of `covar`).

`covar` can be specified as either a matrix or a data frame. In either case the columns are taken to be distinct covariates. If `covar` is a data frame, `tcovar` can be either the name or the index of the time variable.

If a covariate table is supplied, then the value of each of the covariates is interpolated as needed. The resulting interpolated values are passed to the corresponding functions as a numeric vector named `covars`; see below.

`obsnames, statenames, paramnames, covarnames`

Optional character vectors specifying the names of observables, state variables, parameters, and covariates, respectively. These are only used in the event that one or more of the basic functions (`rprocess`, `dprocess`, `rmeasure`, `dmeasure`, `skeleton`, `rprior`, `dprior`) are defined using `Csnippet` or native routines.

`zeronames` optional character vector specifying the names of accumulator variables (see below).

`PACKAGE` An optional string giving the name of the dynamically loaded library in which any native routines are to be found.

`parameter.transform, parameter.inv.transform`

Optional functions specifying parameter transformations. These functions must have arguments `params` and `....` `parameter.transform` should transform parameters to the scale that `rprocess`, `dprocess`, `rmeasure`, `dmeasure`, `skeleton`, and `initializer` will use internally. `parameter.inv.transform` should be the inverse of `parameter.transform`. The parameter transformations can be defined (as above) using either R functions, native routines, or `Csnippet`s.

Note that it is the user's responsibility to make sure that these transformations are mutually inverse. If `obj` is the constructed `pomp` object, and `coef(obj)` is non-empty, a simple check of this property is

```
x <- coef(obj,transform=TRUE)
obj1 <- obj
coef(obj1,transform=TRUE) <- x
identical(coef(obj),coef(obj1))
identical(coef(obj1,transform=TRUE),x).
```

By default, both functions are the identity transformation. See the demos (`demo(package="pomp")`), `pompExample`, and the tutorials on the package website for examples.

`globals` optional character; C code that will be included in the source for (and therefore hard-coded into) the shared-object library created when the call to `pomp` uses `Csnippets`. If no `Csnippets` are used, `globals` has no effect.

`...` Any additional arguments given to `pomp` will be stored in the `pomp` object and passed as arguments to each of the basic functions whenever they are evaluated.

Value

`pomp` returns an object of class `pomp`. If `data` is an object of class `pomp`, then by default the returned `pomp` object is identical to `data`. If additional arguments are given, these override the defaults.

Important note

It is not typically necessary (or even feasible) to define all of the components `rprocess`, `dprocess`, `rmeasure`, `dmeasure`, and `skeleton` in any given problem. Each algorithm makes use of only a subset of these components. Any algorithm requiring a component that has not been defined will return an informative error.

The state process model

Specification of process-model codes `rprocess` and/or `dprocess` in most cases is facilitated by **pomp**'s so-called `plugins`, which have been developed to handle common use-cases. Currently, if one's process model evolves in discrete time or one is willing to make such an approximation (e.g., via an Euler approximation), then the `euler.sim`, `discrete.time.sim`, and `onestep.sim` plugins for `rprocess` and `onestep.dens` plugin for `dprocess` are available. In addition, for exact simulation of certain continuous-time Markov chains, an implementation of Gillespie's algorithm is available (see `gillespie.sim`). To learn more about the use of plugins, consult the help documentation (`plugins`) and the tutorials on the package website. Several of the demos and examples make use of these as well.

In specific cases, it may be possible to obtain increased computational efficiency by writing custom versions of `rprocess` and/or `dprocess` instead of using the plugins. If such custom versions are desired, the following describes how these functions should be written.

`rprocess` If the plugins are not used `rprocess` must be an R function with at least the following arguments: `xstart`, `times`, `params`, and `....` It can also take additional arguments. It is guaranteed that these will be filled with the corresponding elements the user has included as additional arguments in the construction of the `pomp` object.

In calls to `rprocess`, `xstart` can be assumed to be an `nvar` x `nrep` matrix; its rows correspond to components of the state vector and columns correspond to independent realizations of the process. `params` will similarly be an `npar` x `nrep` matrix with rows corresponding to

parameters and columns corresponding to independent realizations. Note that the columns of `params` correspond to those of `xstart`; in particular, they will agree in number. Both `xstart` and `params` are guaranteed to have rownames.

`rprocess` must return a rank-3 array with rownames. Suppose `x` is the array returned. Then `dim(x)=c(nvars,nrep,ntimes)`, where `ntimes` is the length of the vector `times`. `x[,j,k]` is the value of the state process in the `j`-th realization at time `times[k]`. In particular, `x[,,1]` must be identical to `xstart`. The rownames of `x` must correspond to those of `xstart`.

`dprocess` If the plugins are not used, `dprocess` must have at least the following arguments: `x`, `times`, `params`, `log`, and `...`. It may take additional arguments: again, these will be filled with the corresponding elements the user defines when the `pomp` object is constructed.

In calls to `dprocess`, `x` may be assumed to be an `nvars` x `nrep` x `ntimes` array, where these terms have the same meanings as above. `params` will be a matrix with rows corresponding to individual parameters and columns corresponding to independent realizations. The columns of `params` correspond to those of `x`; in particular, they will agree in number. Both `x` and `params` are guaranteed to have rownames.

`dprocess` must return a `nrep` x `ntimes-1` matrix. Suppose `d` is the array returned. `d[j,k]` is the probability density of the transition from state `x[,j,k-1]` at time `times[k-1]` to state `x[,j,k]` at time `times[k]`. If `log=TRUE`, then the log of the pdf must be returned.

In writing this function, you may assume that the transitions are consecutive. It should be clear that, but for this assumption, it will in general be impossible to write the transition probabilities explicitly. In such cases, algorithms that make no use of `dprocess`, which are said to have the "plug and play" property, are useful. Most of the algorithms in **pomp** have this property. In particular, **at present, no methods in pomp make use of** `dprocess`**.**

The observation process model

The following is a guide to writing the measurement model components as R functions. For a description on how to write these components using `Csnippets`, see the tutorials on the package website.

`rmeasure` if provided, must take at least the arguments `x`, `t`, `params`, and `...`. It may take additional arguments, which will be filled with user-specified data as above. `x` will be a named numeric vector of length `nvars` (which has the same meaning as above). `t` will be a scalar quantity, the time at which the measurement is made. `params` will be a named numeric vector of length `npars`. The `rmeasure` function may take additional arguments which will be filled with user-specified data as above.

`rmeasure` must return a named numeric vector of length `nobs`, the number of observable variables.

`dmeasure` if provided, must take at least the arguments `y`, `x`, `t`, `params`, `log`, and `...`. `y` will be a named numeric vector of length `nobs` containing (actual or simulated) values of the observed variables; `x` will be a named numeric vector of length `nvar` containing state variables; `params` will be a named numeric vector containing parameters; and `t` will be a scalar, the corresponding observation time. The `dmeasure` function may take additional arguments which will be filled with user-specified data as above.

`dmeasure` must return a single numeric value, the probability density of `y` given `x` at time `t`. If `log=TRUE`, then `dmeasure` should return the log of the probability density.

The deterministic skeleton

The following describes how to specify the deterministic skeleton as an R function. For a description on how to write this component using Csnippets, see the tutorials on the package website.

If skeleton if provided, must have at least the arguments x, t, params, and .... x is a numeric vector containing the coordinates of a point in state space at which evaluation of the skeleton is desired. t is a numeric value giving the time at which evaluation of the skeleton is desired. Of course, these will be irrelevant in the case of an autonomous skeleton. params is a numeric vector holding the parameters. skeleton may take additional arguments, which will be filled, as above, with user-specified data.

skeleton must return a numeric vector of the same length as x, which contains the value vectorfield (if the dynamical system is continuous) or the value of the map (if the dynamical system is discrete), at the point x at time t.

The state-process initializer

if provided, must have at least the arguments params, t0, and .... params will be a named numeric vector of parameters. t0 will be the time at which initial conditions are desired. initializer must return a named numeric vector of initial states.

Covariates

If the pomp object contains covariates (via the covar argument; see above), then whenever any of the R functions described above are called, they will each be supplied with an additional argument covars. This will be a named numeric vector containing the (interpolated) values of the covariates at the time t. In particular, covars will have one value for each column of the covariate table.

Accumulator variables

In formulating models, one often wishes to define a state variable that will accumulate some quantity over the interval between successive observations. **pomp** provides a facility to make such features more convenient. Specifically, variables named in the pomp's zeronames argument will be set to zero immediately following each observation. See euler.sir and the tutorials on the package website for examples.

Warning

Some error checking is done by pomp, but complete error checking is impossible. If the user-specified functions do not conform to the above specifications, then the results may be invalid. In particular, if both rmeasure and dmeasure are specified, the user should verify that these two functions correspond to the same probability distribution. If skeleton is specified, the user is responsible for verifying that it corresponds to a deterministic skeleton of the model. Each **pomp**-package algorithm uses some subset of the five basic functions (rprocess, dprocess, rmeasure, dmeasure, skeleton). If an algorithm requires a component that has not been specified, an informative error will be generated.

Author(s)

Aaron A. King <kingaa at umich dot edu>

See Also

pomp methods, pomp low-level interface, process model plugins

Examples

```
## Not run:
pompExample()
pomp.home <- system.file("examples",package="pomp")
pomp.examples <- list.files(pomp.home)
file.show(
          file.path(pomp.home,pomp.examples),
          header=paste("======",pomp.examples,"=======")
         )

## End(Not run)
```

---

pomp simulation *Running simulations of a partially-observed Markov process*

Description

`simulate` can be used to generate simulated data sets and/or to simulate the state process.

Usage

```
## S4 method for signature pomp
simulate(object, nsim = 1, seed = NULL, params,
         states = FALSE, obs = FALSE, times, t0,
         as.data.frame = FALSE, include.data = FALSE, ...)
```

Arguments

`object`	An object of class pomp.
`nsim`	The number of simulations to perform. Note that the number of replicates will be `nsim` times `ncol(xstart)`.
`seed`	optional; if set, the pseudorandom number generator (RNG) will be initialized with `seed`. the random seed to use. The RNG will be restored to its original state afterward.
`params`	either a named numeric vector or a numeric matrix with rownames. The parameters to use in simulating the model. If `params` is not given, then the contents of the `params` slot of `object` will be used, if they exist.
`states`	Do we want the state trajectories?
`obs`	Do we want data-frames of the simulated observations?
`times, t0`	`times` specifies the times at which simulated observations will be made. `t0` specifies the start time (the time at which the initial conditions hold). The default for `times` is is `times=time(object,t0=FALSE)` and `t0=timezero(object)`, respectively.

`as.data.frame, include.data`
: logical; if `as.data.frame=TRUE`, the results are returned as a data-frame. A factor variable, 'sim', distinguishes one simulation from another. If, in addition, `include.data=TRUE`, the original data are included as an additional 'simulation'. If `as.data.frame=FALSE`, `include.data` is ignored.

`...`
: further arguments that are currently ignored.

Details

Simulation of the state process and of the measurement process are each accomplished by a single call to the user-supplied `rprocess` and `rmeasure` functions, respectively. This makes it possible for the user to write highly optimized code for these potentially expensive computations.

Value

If `states=FALSE` and `obs=FALSE` (the default), a list of `nsim` pomp objects is returned. Each has a simulated data set, together with the parameters used (in slot `params`) and the state trajectories also (in slot `states`). If `times` is specified, then the simulated observations will be at times `times`.

If `nsim=1`, then a single pomp object is returned (and not a singleton list).

If `states=TRUE` and `obs=FALSE`, simulated state trajectories are returned as a rank-3 array with dimensions `nvar` x `(ncol(params)*nsim)` x `ntimes`. Here, `nvar` is the number of state variables and `ntimes` the length of the argument `times`. The measurement process is not simulated in this case.

If `states=FALSE` and `obs=TRUE`, simulated observations are returned as a rank-3 array with dimensions `nobs` x `(ncol(params)*nsim)` x `ntimes`. Here, `nobs` is the number of observables.

If both `states=TRUE` and `obs=TRUE`, then a named list is returned. It contains the state trajectories and simulated observations as above.

Author(s)

Aaron A. King <kingaa at umich dot edu>

See Also

pomp

Examples

```
pompExample(ou2)
x <- simulate(ou2,seed=3495485,nsim=10)
x <- simulate(ou2,seed=3495485,nsim=10,states=TRUE,obs=TRUE)
x <- simulate(ou2,seed=3495485,nsim=10,obs=TRUE,
              as.data.frame=TRUE,include.data=TRUE)
```

pomp-methods *Methods of the "pomp" class*

Description

Methods of the pomp class.

Usage

```
## S3 method for class pomp
as.data.frame(x, row.names, optional, ...)
## S4 method for signature pomp
coef(object, pars, transform = FALSE, ...)
## S4 replacement method for signature pomp
coef(object, pars, transform = FALSE, ...) <- value
## S4 method for signature pomp
obs(object, vars, ...)
## S4 method for signature pomp
partrans(object, params, dir = c("forward","inverse"), ...)
## S4 method for signature pomp
plot(x, y, variables, panel = lines,
      nc = NULL, yax.flip = FALSE,
      mar = c(0, 5.1, 0, if (yax.flip) 5.1 else 2.1),
      oma = c(6, 0, 5, 0), axes = TRUE, ...)
## S4 method for signature pomp
print(x, ...)
## S4 method for signature pomp
show(object)
## S4 method for signature pomp
states(object, vars, ...)
## S4 method for signature pomp
time(x, t0 = FALSE, ...)
## S4 replacement method for signature pomp
time(object, t0 = FALSE, ...) <- value
## S4 method for signature pomp
timezero(object, ...)
## S4 replacement method for signature pomp
timezero(object, ...) <- value
## S4 method for signature pomp
window(x, start, end, ...)
## S4 method for signature pomp
as(object, class)
## S4 method for signature pomp,data.frame
coerce(from, to = "data.frame", strict = TRUE)
```

Arguments

object, x	The pomp object.

pars	optional character; names of parameters to be retrieved or set.
vars	optional character; names of observed variables to be retrieved.
transform	optional logical; should the parameter transformations be applied?
value	numeric; values to be assigned.
params	a vector or matrix of parameters to be transformed.
dir	direction of the transformation. dir="forward" applies the transformation from the "natural" scale to the "internal" scale. This is the transformation specified by the parameter.transform argument to pomp; it is stored in the 'par.trans' slot of object. dir="inverse" applies the inverse transformation (stored in the 'par.untrans' slot).
t0	logical; if TRUE on a call to time, the zero time is prepended to the time vector; if TRUE on a call to time<-, the first element in value is taken to be the initial time.
start, end	start and end times of the window.
class	character; name of the class to which object should be coerced.
from, to	the classes between which coercion should be performed.
strict	ignored.
y	ignored.
variables	optional character; names of variables to plot.
panel	a function of prototype panel(x, col, bg, pch, type, ...) which gives the action to be carried out in each panel of the display.
nc	the number of columns to use. Defaults to 1 for up to 4 series, otherwise to 2.
yax.flip	logical; if TRUE, the y-axis (ticks and numbering) should flip from side 2 (left) to 4 (right) from series to series.
mar, oma	the 'par' settings for 'mar' and 'oma' to use. Modify with care!
axes	logical; indicates if x- and y- axes should be drawn.
row.names, optional	ignored.
...	Further arguments (either ignored or passed to underlying functions).

Details

coef coef(object) returns the contents of the params slot of object. coef(object,pars) returns only those parameters named in pars.

coef(object,transform=TRUE)

returns

parameter.inv.transform(coef(object))

, where parameter.inv.transform is the user parameter inverse transformation function specified when object was created. Likewise,

coef(object,pars,transform=TRUE)

returns

```
parameter.inv.transform(coef(object))[pars]
```

.

coef<- Assigns values to the params slot of the pomp object. `coef(object) <- value` has the effect of replacing the parameters of `object` with `value`. If `coef(object)` exists, then `coef(object,pars) <- value` replaces those parameters of `object` named in `pars` with the elements of `value`; the names of `value` are ignored. If some of the names in `pars` do not already name parameters in `coef(object)`, then they are concatenated. If `coef(object)` does not exist, then `coef(object,pars) <- value` assigns `value` to the parameters of `object`; in this case, the names of `object` will be `pars` and the names of `value` will be ignored. `coef(object,transform=TRUE) <- value` assigns `parameter.transform(value)` to the `params` slot of `object`. Here, `parameter.transform` is the parameter transformation function specified when `object` was created. `coef(object,pars,transform=TRUE) <- value` first, discards any names the `value` may have, sets `names(value) <- pars`, and then replaces the elements of `object`'s `params` slot `parameter.transform(value)`. In this case, if some of the names in `pars` do not already name parameters in `coef(object,transform=TRUE)`, then they are concatenated.

obs These functions are synonymous. `obs(object)` returns the array of observations. `obs(object,vars)` gives just the observations of variables named in `vars`. `vars` may specify the variables by position or by name.

states `states(object)` returns the array of states. `states(object,vars)` gives just the state variables named in `vars`. `vars` may specify the variables by position or by name.

time `time(object)` returns the vector of observation times. `time(object,t0=TRUE)` returns the vector of observation times with the zero-time `t0` prepended.

time<- `time(object) <- value` replaces the observation times slot (`times`) of `object` with `value`. `time(object,t0=TRUE) <- value` has the same effect, but the first element in `value` is taken to be the initial time. The second and subsequent elements of `value` are taken to be the observation times. Those data and states (if they exist) corresponding to the new times are retained.

timezero, timezero<- `timezero(object)` returns the zero-time `t0`. `timezero(object) <- value` sets the zero-time to `value`.

window `window(x,start=t1,end=t2` returns a new pomp object, identical to `x` but with only the data in the window between times `t1` and `t2` (inclusive). By default, `start` is the time of the first observation and `end` is the time of the last.

show Displays the pomp object.

plot Plots the data and state trajectories (if the latter exist). Additional arguments are passed to the low-level plotting routine.

print Prints the pomp object in a nice way.

as, coerce The `coerce` method should typically not be used directly. It is defined by `setAs` as a method to be used by `as`. A pomp object can be coerced to a data frame via

```
as(object,"data.frame").
```

The data frame contains the times, the data, and the state trajectories, if they exist.

Author(s)

Aaron A. King <kingaa at umich dot edu>

See Also

pomp, pomp low-level interface, simulate

---

pompExample *Pre-built examples of pomp objects.*

Description

pompExample loads pre-built example pomp objects.

Usage

```
pompExample(example, ..., envir = .GlobalEnv)
```

Arguments

example	example to load given as a name or literal character string. Evoked without an argument, pompExample lists all available examples.
...	additional arguments define symbols in the environment within which the example code is executed.
envir	the environment into which the objects should be loaded. If envir=NULL, then the created objects are returned in a list.

Details

Directories in the the global option pomp.examples (set using options()) are searched for files named 'example.R'. If found, this file will be sourced in a temporary environment. Additional arguments to pompExample define variables within this environment and will therefore be available when the code in 'example.R' is sourced.

Value

By default, pompExample has the side effect of creating one or more pomp objects in the global workspace. If envir=NULL, there are no side effects; rather, the pomp objects are returned as a list.

Author(s)

Aaron A. King <kingaa at umich dot edu>

See Also

blowflies, dacca, gompertz, ou2, ricker, rw2, euler.sir, gillespie.sir, bbs, verhulst

Examples

```
pompExample()
pompExample(euler.sir)
pompExample("gompertz")
pompExample(ricker,envir=NULL)
file.show(system.file("examples/bbs.R",package="pomp"))
```

---

probe *Probe a partially-observed Markov process.*

Description

probe applies one or more "probes" to time series data and model simulations and compares the results. It can be used to diagnose goodness of fit and/or as the basis for "probe-matching", a generalized method-of-moments approach to parameter estimation. probe.match calls an optimizer to adjust model parameters to do probe-matching, i.e., to minimize the discrepancy between simulated and actual data. This discrepancy is measured using the "synthetic likelihood" as defined by Wood (2010). probe.match.objfun constructs an objective function for probe-matching suitable for use in optim-like optimizers.

Usage

```
## S4 method for signature pomp
probe(object, probes, params, nsim, seed = NULL, ...)
## S4 method for signature probed.pomp
probe(object, probes, params, nsim, seed, ...)
## S4 method for signature pomp
probe.match.objfun(object, params, est, probes, nsim,
            seed = NULL, fail.value = NA, transform = FALSE, ...)
## S4 method for signature probed.pomp
probe.match.objfun(object, probes, nsim, seed, ...)
## S4 method for signature pomp
probe.match(object, start, est = character(0),
            probes, nsim, seed = NULL,
            method = c("subplex","Nelder-Mead","SANN","BFGS",
                       "sannbox","nloptr"),
            verbose = getOption("verbose"),
            fail.value = NA, transform = FALSE, ...)
## S4 method for signature probed.pomp
probe.match(object, probes, nsim, seed,
            ..., verbose = getOption("verbose"))
## S4 method for signature probe.matched.pomp
probe.match(object, est, probes,
            nsim, seed, transform, fail.value, ...,
            verbose = getOption("verbose"))
```

Arguments

object	An object of class pomp.
probes	A single probe or a list of one or more probes. A probe is simply a scalar- or vector-valued function of one argument that can be applied to the data array of a pomp. A vector-valued probe must always return a vector of the same size. A number of basic examples are provided with the package (see basic.probes).
params	optional named numeric vector of model parameters. By default, params=coef(object).
nsim	The number of model simulations to be computed.
seed	optional; if non-NULL, the random number generator will be initialized with this seed for simulations. See simulate-pomp.
start	named numeric vector; the initial guess of parameters.
est	character vector; the names of parameters to be estimated.
method	Optimization method. Choices refer to algorithms used in optim, subplex, and nloptr.
verbose	logical; print diagnostic messages?
fail.value	optional numeric scalar; if non-NA, this value is substituted for non-finite values of the objective function. It should be a large number (i.e., bigger than any legitimate values the objective function is likely to take).
transform	logical; if TRUE, optimization is performed on the transformed scale.
...	Additional arguments. In the case of probe, these are currently ignored. In the case of probe.match, these are passed to the optimizer (one of optim, subplex, nloptr, or sannbox). These are passed via the optimizer's control list (in the case of optim, subplex, and sannbox) or the opts list (in the case of nloptr).

Details

A call to probe results in the evaluation of the probe(s) in probes on the data. Additionally, nsim simulated data sets are generated (via a call to simulate) and the probe(s) are applied to each of these. The results of the probe computations on real and simulated data are stored in an object of class probed.pomp.

A call to probe.match results in an attempt to optimize the agreement between model and data, as measured by the specified probes, over the parameters named in est. The results, including coefficients of the fitted model and values of the probes for data and fitted-model simulations, are stored in an object of class probe.matched.pomp.

The objective function minimized by probe.match — in a form suitable for use with optim-like optimizers — is created by a call to probe.match.objfun. Specifically, probe.match.objfun will return a function that takes a single numeric-vector argument that is assumed to cotain the parameters named in est, in that order. This function will return the negative synthetic log likelihood for the probes specified.

Value

probe returns an object of class probed.pomp. probed.pomp is derived from the pomp class and therefore have all the slots of pomp. In addition, a probed.pomp class has the following slots:

probes list of the probes applied.

datvals, simvals values of each of the probes applied to the real and simulated data, respectively.

quantiles fraction of simulations with probe values less than the value of the probe of the data.

pvals two-sided p-values: fraction of the `simvals` that deviate more extremely from the mean of the `simvals` than does `datavals`.

synth.loglik the log synthetic likelihood (Wood 2010). This is the likelihood assuming that the probes are multivariate-normally distributed.

`probe.match` returns an object of class `probe.matched.pomp`, which is derived from class `probed.pomp`. `probe.matched.pomp` objects therefore have all the slots above plus the following:

est, transform, fail.value values of the corresponding arguments in the call to `probe.match`.

value value of the objective function at the optimum.

evals number of function and gradient evaluations by the optimizer. See `optim`.

convergence, msg Convergence code and message from the optimizer. See `optim` and `nloptr`.

`probe.match.objfun` returns a function suitable for use as an objective function in an `optim`-like optimizer.

Author(s)

Daniel C. Reuman, Aaron A. King

References

B. E. Kendall, C. J. Briggs, W. M. Murdoch, P. Turchin, S. P. Ellner, E. McCauley, R. M. Nisbet, S. N. Wood Why do populations cycle? A synthesis of statistical and mechanistic modeling approaches, Ecology, 80:1789–1805, 1999.

S. N. Wood Statistical inference for noisy nonlinear ecological dynamic systems, Nature, 466: 1102–1104, 2010.

See Also

pomp-class, pomp-methods, basic.probes, `probe.match`

Examples

```
pompExample(ou2)
good <- probe(
              ou2,
              probes=list(
                y1.mean=probe.mean(var="y1"),
                y2.mean=probe.mean(var="y2"),
                y1.sd=probe.sd(var="y1"),
                y2.sd=probe.sd(var="y2"),
                extra=function(x)max(x["y1",])
                ),
              nsim=500
              )
```

```
summary(good)
plot(good)

bad <- probe(
             ou2,
             params=c(alpha.1=0.1,alpha.4=0.2,x1.0=0,x2.0=0,
                      alpha.2=-0.5,alpha.3=0.3,
                      sigma.1=3,sigma.2=-0.5,sigma.3=2,
                      tau=1),
             probes=list(
               y1.mean=probe.mean(var="y1"),
               y2.mean=probe.mean(var="y2"),
               y1.sd=probe.sd(var="y1"),
               y2.sd=probe.sd(var="y2"),
               extra=function(x)range(x["y1",])
               ),
             nsim=500
             )
summary(bad)
plot(bad)
```

---

probed.pomp-methods *Methods of the "probed.pomp", "probe.matched.pomp", "spect.pomp", and "spect.matched.pomp" classes*

Description

Methods of the probed.pomp, probe.matched.pomp, spect.pomp, and spect.matched.pomp classes

Usage

```
## S4 method for signature probed.pomp
summary(object, ...)
## S4 method for signature probed.pomp
plot(x, y, ...)
## S4 method for signature probe.matched.pomp
summary(object, ...)
## S4 method for signature probe.matched.pomp
plot(x, y, ...)
## S4 method for signature spect.pomp
summary(object, ...)
## S4 method for signature probed.pomp
logLik(object, ...)
## S4 method for signature probed.pomp
values(object, ...)
## S4 method for signature spect.pomp
plot(x, y, max.plots.per.page = 4,
          plot.data = TRUE,
          quantiles = c(.025, .25, .5, .75, .975),
```

```
          quantile.styles = list(lwd=1, lty=1, col="gray70"),
          data.styles = list(lwd=2, lty=2, col="black"))
## S4 method for signature spect.matched.pomp
summary(object, ...)
## S4 method for signature spect.matched.pomp
plot(x, y, ...)
## S4 method for signature probed.pomp
as(object, class)
```

Arguments

`object, x`	the object to be summarized or plotted.
`y`	ignored.
`max.plots.per.page`	maximum number of plots per page
`plot.data`	plot the data spectrum?
`quantiles`	quantiles to plot
`quantile.styles`	plot style parameters for the quantiles
`data.styles`	plot style parameters for the data spectrum
`class`	character; name of the class to which `object` should be coerced.
`...`	Further arguments (either ignored or passed to underlying functions).

Methods

plot displays diagnostic plots.

summary displays summary information.

values extracts the realized values of the probes on the data and on the simulations as a data frame in long format. The variable `.id` indicates whether the probes are from the data or simulations.

logLik returns the synthetic likelihood for the probes. NB: in general, this is not the same as the likelihood.

as when a 'probed.pomp' is coerced to a 'data.frame', the first row gives the probes applied to the data; the rest of the rows give the probes evaluated on simulated data. The rownames of the result can be used to distinguish these.

In addition, slots of this object can be accessed via the `$` operator.

Author(s)

Daniel C. Reuman, Aaron A. King

See Also

`probe`, `probed.pomp`, `probe.matched.pomp`, `probe.match`

ricker *Ricker model with Poisson observations.*

Description

ricker is a pomp object encoding a stochastic Ricker model with Poisson measurement error.

Details

The state process is $N_{t+1} = rN_t \exp(-N_t + e_t)$, where the e_t are i.i.d. normal random deviates with zero mean and variance σ^2. The observed variables y_t are distributed as $\mathrm{Poisson}(\phi N_t)$.

See Also

pomp, gompertz, and the tutorials on the package website.

Examples

```
pompExample(ricker)
plot(ricker)
coef(ricker)
```

rw2 *Two-dimensional random-walk process*

Description

rw2 is a pomp object encoding a 2-D normal random walk.

Details

The random-walk process is fully but noisily observed.

See Also

pomp, ou2

Examples

```
pompExample(rw2)
plot(rw2)
x <- simulate(rw2,nsim=10,seed=20348585L,params=c(x1.0=0,x2.0=0,s1=1,s2=3,tau=1))
plot(x[[1]])
```

sannbox *Simulated annealing with box constraints.*

Description

sannbox is a straightforward implementation of simulated annealing with box constraints.

Usage

```
sannbox(par, fn, control = list(), ...)
```

Arguments

par	Initial values for the parameters to be optimized over.
fn	A function to be minimized, with first argument the vector of parameters over which minimization is to take place. It should return a scalar result.
control	A named list of control parameters. See 'Details'.
...	ignored.

Details

The control argument is a list that can supply any of the following components:

trace Non-negative integer. If positive, tracing information on the progress of the optimization is produced. Higher values may produce more tracing information.

fnscale An overall scaling to be applied to the value of fn during optimization. If negative, turns the problem into a maximization problem. Optimization is performed on fn(par)/fnscale.

parscale A vector of scaling values for the parameters. Optimization is performed on par/parscale and these should be comparable in the sense that a unit change in any element produces about a unit change in the scaled value.

maxit The total number of function evaluations: there is no other stopping criterion. Defaults to 10000.

temp starting temperature for the cooling schedule. Defaults to 1.

tmax number of function evaluations at each temperature. Defaults to 10.

candidate.dist function to randomly select a new candidate parameter vector. This should be a function with three arguments, the first being the current parameter vector, the second the temperature, and the third the parameter scaling. By default, candidate.dist is

```
function(par,temp,scale) rnorm(n=length(par),mean=par,sd=scale*temp)
```

.

sched cooling schedule. A function of a three arguments giving the temperature as a function of iteration number and the control parameters temp and tmax. By default, sched is

```
function(k,temp,tmax) temp/log(((k-1)%/%tmax)*tmax+exp(1))
```

.

Alternatively, one can supply a numeric vector of temperatures. This must be of length at least maxit.

Value

sannbox returns a list with components:

counts two-element integer vector. The first number gives the number of calls made to fn. The second number is provided for compatibility with optim and will always be NA.

convergence provided for compatibility with optim; will always be 0.

final.params last tried value of par.

final.value value of fn corresponding to final.params.

par best tried value of par.

value value of fn corresponding to par.

Author(s)

Daniel Reuman, Imperial College London and Aaron A. King <kingaa at umich dot edu>

See Also

traj.match, probe.match.

---

sir *SIR models.*

Description

euler.sir is a pomp object encoding a simple seasonal SIR model. Simulation is performed using an Euler multinomial approximation. gillespie.sir has the same model implemented using Gillespie's algorithm. bbs is a nonseasonal SIR model together with data from a 1978 outbreak of influenza in a British boarding school.

Details

This example is discussed extensively in the "Introduction to **pomp**" and "Advanced topics in **pomp**" vignettes, available on the package website.

The codes that construct these pomp objects can be found in the "examples" directory in the installed package. Do system.file("examples",package="pomp")) to find this directory. For the basic rprocess, dmeasure, rmeasure, and skeleton functions, these codes use compiled native routines built into the package's library. View "src/sir.c" in the package source or file.show("examples/sir.c") from an R session to view these codes.

The boarding school influenza outbreak is described in Anonymous (1978).

References

Anonymous (1978). Influenza in a boarding school. British Medical Journal 1:587

See Also

pomp and the tutorials on the package website.

Examples

```
pompExample(euler.sir)
plot(euler.sir)
plot(simulate(euler.sir,seed=20348585))
coef(euler.sir)

pompExample(gillespie.sir)
plot(gillespie.sir)
plot(simulate(gillespie.sir,seed=20348585))
coef(gillespie.sir)

pompExample(bbs)
plot(bbs)
coef(bbs)
```

---

spect *Power spectrum computation for partially-observed Markov processes.*

Description

spect estimates the power spectrum of time series data and model simulations and compares the results. It can be used to diagnose goodness of fit and/or as the basis for frequency-domain parameter estimation (spect.match).

spect.match tries to match the power spectrum of the model to that of the data. It calls an optimizer to adjust model parameters to minimize the discrepancy between simulated and actual data.

Usage

```
  ## S4 method for signature pomp
spect(object, params, vars, kernel.width, nsim, seed = NULL,
      transform = identity,
      detrend = c("none","mean","linear","quadratic"),
      ...)
  ## S4 method for signature spect.pomp
spect(object, params, vars, kernel.width, nsim, seed = NULL, transform,
      detrend, ...)
  spect.match(object, start, est = character(0),
              vars, nsim, seed = NULL,
              kernel.width, transform = identity,
              detrend = c("none","mean","linear","quadratic"),
              weights, method = c("subplex","Nelder-Mead","SANN"),
              verbose = getOption("verbose"),
              eval.only = FALSE, fail.value = NA, ...)
```

Arguments

object	An object of class pomp.
params	optional named numeric vector of model parameters. By default, params=coef(object).
vars	optional; names of observed variables for which the power spectrum will be computed. This must be a subset of rownames(obs(object)). By default, the spectrum will be computed for all observables.
kernel.width	width parameter for the smoothing kernel used for calculating the estimate of the spectrum.
nsim	number of model simulations to be computed.
seed	optional; if non-NULL, the random number generator will be initialized with this seed for simulations. See simulate-pomp.
transform	function; this transformation will be applied to the observables prior to estimation of the spectrum, and prior to any detrending.
detrend	de-trending operation to perform. Options include no detrending, and subtraction of constant, linear, and quadratic trends from the data. Detrending is applied to each data series and to each model simulation independently.
weights	optional. The mismatch between model and data is measured by a weighted average of mismatch at each frequency. By default, all frequencies are weighted equally. weights can be specified either as a vector (which must have length equal to the number of frequencies) or as a function of frequency. If the latter, weights(freq) must return a nonnegative weight for each frequency.
start	named numeric vector; the initial guess of parameters.
est	character vector; the names of parameters to be estimated.
method	Optimization method. Choices are subplex and any of the methods used by optim.
verbose	logical; print diagnostic messages?
eval.only	logical; if TRUE, no optimization is attempted. Instead, the probe-mismatch value is simply evaluated at the start parameters.
fail.value	optional scalar; if non-NA, this value is substituted for non-finite values of the objective function.
...	Additional arguments. In the case of spect, these are currently ignored. In the case of spect.match, these are passed to optim or subplex in the control list.

Details

A call to spect results in the estimation of the power spectrum for the (transformed, detrended) data and nsim model simulations. The results of these computations are stored in an object of class spect.pomp.

A call to spect.match results in an attempt to optimize the agreement between model and data spectrum over the parameters named in est. The results, including coefficients of the fitted model and power spectra of fitted model and data, are stored in an object of class spect.matched.pomp.

Value

`spect` returns an object of class `spect.pomp`, which is derived from class `pomp` and therefore has all the slots of that class. In addition, `spect.pomp` objects have the following slots:

kernel.width width parameter of the smoothing kernel used.

transform transformation function used.

freq numeric vector of the frequencies at which the power spectrum is estimated.

datspec, simspec estimated power spectra for data and simulations, respectively.

pvals one-sided p-values: fraction of the simulated spectra that differ more from the mean simulated spectrum than does the data. The metric used is L^2 distance.

detrend detrending option used.

`spect.match` returns an object of class `spect.matched.pomp`, which is derived from class `{spect.pomp}` and therefore has all the slots of that class. In addition, `spect.matched.pomp` objects have the following slots:

est, weights, fail.value values of the corresponding arguments in the call to `spect.match`.

evals number of function and gradient evaluations by the optimizer. See `optim`.

value Value of the objective function.

convergence, msg Convergence code and message from the optimizer. See `optim`.

Author(s)

Daniel C. Reuman, Cai GoGwilt, Aaron A. King

References

D.C. Reuman, R.A. Desharnais, R.F. Costantino, O. Ahmad, J.E. Cohen (2006) Power spectra reveal the influence of stochasticity on nonlinear population dynamics. *Proceedings of the National Academy of Sciences* **103**, 18860-18865.

D.C. Reuman, R.F. Costantino, R.A. Desharnais, J.E. Cohen (2008) Color of environmental noise affects the nonlinear dynamics of cycling, stage-structured populations. *Ecology Letters*, **11**, 820-830.

See Also

pomp-class, pomp-methods, `probe`, `probe.match`

Examples

```
pompExample(ou2)
good <- spect(
              ou2,
              vars=c("y1","y2"),
              kernel.width=3,
              detrend="mean",
              nsim=500
              )
```

```
summary(good)
plot(good)

ou2.bad <- ou2
coef(ou2.bad,c("x1.0","x2.0","alpha.1","alpha.4")) <- c(0,0,0.1,0.2)
bad <- spect(
             ou2.bad,
             vars=c("y1","y2"),
             kernel.width=3,
             detrend="mean",
             nsim=500
             )
summary(bad)
plot(bad)
```

`traj.match` *Trajectory matching*

Description

Facilities for matching trajectories to data. Trajectory matching is equivalent to maximum likelihood estimation under the assumption that process noise is entirely absent, i.e., that all stochasticity is measurement error.

Usage

```
  ## S4 method for signature pomp
traj.match(object, start, est = character(0),
           method = c("Nelder-Mead","subplex","SANN","BFGS",
                      "sannbox","nloptr"),
           transform = FALSE, ...)
  ## S4 method for signature traj.matched.pomp
traj.match(object, est, transform, ...)
  ## S4 method for signature pomp
traj.match.objfun(object, params, est, transform = FALSE, ...)
```

Arguments

`object`	A pomp object. If `object` has no `skeleton` slot, an error will be generated.
`start`	named numeric vector containing an initial guess for parameters. By default `start=coef(object)` if the latter exists.
`params`	optional named numeric vector of parameters. This should contain all parameters needed by the `skeleton` and `dmeasure` slots of `object`. In particular, any parameters that are to be treated as fixed should be present here. Parameter values given in `params` for parameters named in `est` will be ignored. By default, `params=coef(object)` if the latter exists.

`est`	character vector containing the names of parameters to be estimated. In the case of `traj.match.objfun`, the objective function that is constructed will assume that its argument contains the parameters in this order.
`method`	Optimization method. Choices are `subplex`, "sannbox", and any of the methods used by `optim`.
`transform`	logical; if `TRUE`, optimization is performed on the transformed scale.
`...`	Extra arguments that will be passed either to the optimizer (`optim`, `subplex`, `nloptr`, or `sannbox`, via their `control` (`optim`, `subplex`, `sannbox`) or `opts` (`nloptr`) lists) or to the ODE integrator. In `traj.match`, extra arguments will be passed to the optimizer. In `traj.match.objfun`, extra arguments are passed to `trajectory`. If extra arguments are needed by both optimizer and `trajectory`, construct an objective function first using `traj.match.objfun`, then give this objective function to the optimizer.

Details

In **pomp**, trajectory matching is the term used for maximizing the likelihood of the data under the assumption that there is no process noise. Specifically, `traj.match` calls an optimizer (`optim`, `subplex`, and `sannbox` are the currently supported options) to minimize an objective function. For any value of the model parameters, this objective function is calculated by

1. computing the deterministic trajectory of the model given the parameters. This is the trajectory returned by `trajectory`, which relies on the model's deterministic skeleton as specified in the construction of the `pomp` object `object`.
2. evaluating the negative log likelihood of the data under the measurement model given the deterministic trajectory and the model parameters. This is accomplished via the model's `dmeasure` slot. The negative log likelihood is the objective function's value.

The objective function itself — in a form suitable for use with `optim`-like optimizers — is created by a call to `traj.match.objfun`. Specifically, `traj.match.objfun` will return a function that takes a single numeric-vector argument that is assumed to cotain the parameters named in `est`, in that order.

Value

`traj.match` returns an object of class `traj.matched.pomp`. This class inherits from class `pomp` and contains the following additional slots:

transform, est the values of these arguments on the call to `traj.match`.

evals number of function and gradient evaluations by the optimizer. See `optim`.

value value of the objective function. Larger values indicate better fit (i.e., `traj.match` attempts to maximize this quantity.

convergence, msg convergence code and message from the optimizer. See `optim`.

Available methods for objects of this type include `summary` and `logLik`. The other slots of this object can be accessed via the `$` operator.

`traj.match.objfun` returns a function suitable for use as an objective function in an `optim`-like optimizer.

See Also

trajectory, pomp, optim, subplex

Examples

```
pompExample(ou2)
true.p <- c(
    alpha.1=0.9,alpha.2=0,alpha.3=-0.4,alpha.4=0.99,
    sigma.1=2,sigma.2=0.1,sigma.3=2,
    tau=1,
            x1.0=50,x2.0=-50
    )
simdata <- simulate(ou2,nsim=1,params=true.p,seed=43553)
guess.p <- true.p
res <- traj.match(
  simdata,
  start=guess.p,
  est=c(alpha.1,alpha.3,alpha.4,x1.0,x2.0,tau),
  maxit=2000,
  method="Nelder-Mead",
  reltol=1e-8
  )

summary(res)

plot(range(time(res)),range(c(obs(res),states(res))),type=n,xlab="time",ylab="x,y")
points(y1~time,data=as(res,"data.frame"),col=blue)
points(y2~time,data=as(res,"data.frame"),col=red)
lines(x1~time,data=as(res,"data.frame"),col=blue)
lines(x2~time,data=as(res,"data.frame"),col=red)

pompExample(ricker)
ofun <- traj.match.objfun(ricker,est=c("r","phi"),transform=TRUE)
optim(fn=ofun,par=c(2,0),method="BFGS")

pompExample(bbs)
## some options are passed to the ODE integrator
ofun <- traj.match.objfun(bbs,est=c("beta","gamma"),transform=TRUE,hmax=0.001,rtol=1e-6)
optim(fn=ofun,par=c(0,-1),method="Nelder-Mead",control=list(reltol=1e-10))
```

---

verhulst *Simple Verhulst-Pearl (logistic) model.*

Description

verhulst is a pomp object encoding a univariate stochastic logistic model with measurement error.

Details

The model is written as an Ito diffusion, $dn = rn(1 - n/K)dt + \sigma n dW$, where W is a Wiener process. It is implemented using the euler.sim plug-in.

See Also

pomp-class and the vignettes

Examples

```
pompExample(verhulst)
plot(verhulst)
coef(verhulst)
params <- cbind(
                c(n.0=100,K=10000,r=0.2,sigma=0.4,tau=0.1),
                c(n.0=1000,K=11000,r=0.1,sigma=0.4,tau=0.1)
                )
x <- simulate(verhulst,params=params,states=TRUE)
matplot(time(verhulst),t(x[n,,]),type=l)
y <- trajectory(verhulst,params=params)
matlines(time(verhulst),t(y[n,,]),type=l,lwd=2)
```

Index