Package 'pomp'

January 7, 2012

Type Package

Title Statistical inference for partially observed Markov processes

Version 0.40-1

Date 2012-01-06

Author Aaron A. King, Edward L. Ionides, Carles Breto, Steve Ellner, Bruce Kendall, Helen Wearing, Matthew J. Ferrari, Michael Lavine, Daniel C. Reuman

Maintainer Aaron A. King <kingaa@umich.edu>

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

Description Inference methods for partially-observed Markov processes

Depends R(>= 2.10.1), stats, methods, graphics, mvtnorm, subplex, deSolve

License GPL(>= 2)

LazyLoad true

LazyData false

Collate aaa.R version.R eulermultinom.R plugins.R
slice-design.R profile-design.R sobol.R bsplines.R sannbox.R
pomp-fun.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 simulate-pomp.R trajectory-pomp.R plot-pomp.R pfilter.R pfilter-methods.R traj-match.R bsmc.R
mif-class.R particles-mif.R mif.R mif-methods.R compare-mif.R
pmcmc.R pmcmc-methods.R compare-pmcmc.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

R topics documented:

`pomp-package` *Partially-observed Markov processes*

Description

The **pomp** package provides facilities for inference on time series data using partially-observed Markov processes (AKA state-space models or nonlinear stochastic dynamical systems). One can use **pomp** to fit nonlinear, non-Gaussian dynamic models to time-series data. The first step in using **pomp** is to encode one's model and data in an object of class `pomp`. One does this via a call to `pomp`, which involves specifying the process and measurement components of the model in one or more of a variety of ways. Details on this are given in the documentation for the `pomp` function and examples are given in the 'intro_to_pomp' vignette.

Currently, **pomp** provides algorithms for (i) simulation of stochastic dynamical systems (see `simulate`), (ii) particle filtering (AKA sequential Monte Carlo or sequential importance sampling), see `pfilter`),

(iii) the iterated filtering method of Ionides et al. (2006), see mif), (iv) the nonlinear forecasting algorithm of Kendall et al. (2005), see nlf), (v) the particle MCMC approach of Andrieu et al. (2010), see pmcmc, (vi) basic trajectory matching, see traj.match, (vi) the probe-matching method of Wood (2010) and Kendall et al. (1999), see probe.match, (vii) a spectral probe-matching method (Reuman et al., 2006), see spect.match. See the package website http://pomp.r-forge.r-project.org for these references. The package also provides various tools for plotting and extracting information on models and data as well as an API for algorithm development. Future support for additional algorithms in envisioned, and implementations of the Bayesian sequential Monte Carlo approach of Liu \& West. Much of the work in **pomp** has been done under the auspices of a working group of the National Center for Ecological Analysis and Synthesis (NCEAS), "Inference for Mechanistic Models".

The package is provided under the GNU Public License (GPL). Contributions are welcome, as are comments, suggestions for improvements, 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, and references to the literature.

Classes

pomp makes extensive use of S4 classes. The basic class, pomp, encodes a partially-observed Markov process together with a uni- or multi-variate data set and (possibly) parameters.

Vignettes

The vignette 'Introduction to pomp' illustrates the facilities of the package using familiar stochastic processes. Run `vignette("intro_to_pomp")` or look at the HTML documentation to view the vignette. Methods for accelerating your codes are discussed in the 'Advanced topics in pomp' vignette; run `vignette("advanced_topics_in_pomp")` to view it.

Author(s)

Aaron A. King <kingaa at umich dot edu>

See Also

pomp, pfilter, simulate, trajectory, mif, nlf, probe.match, traj.match, bsmc, pmcmc

---

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-class, pomp-methods

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

Description

`blowfly1` and `blowfly2` are `pomp` objects encoding stochastic delay-difference models.

Usage

```
data(blowflies)
```

See Also

`pomp-class` and the vignettes

Examples

```
data(blowflies)
plot(blowflies1)
plot(blowflies2)
```

`bsmc` *Liu and West Bayesian Particle Filter*

Description

Generates draws from the posterior distribution for the parameters using the Liu and West algorithm. `bsmc` gives draws from the posterior.

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, ...)
```

Arguments

object	An object of class pomp or inheriting class pomp.
params	A npars x Np matrix (with rownames) containing the parameters corresponding to the initial state values in xstart. Np is the number of particles, i.e., each row should contain Np draws from the prior distribution for that parameter. It is permissible to supply params as a named numeric vector, i.e., without a dim attribute. In this case, all particles will inherit the same parameter values, which is equivalent to a degenerate prior.
Np	If params is specified as a named vector, Np specifies the number of particles to use. If params is specified as a matrix, Np should not be specified; it is taken to be the number of columns of params.
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.
...	currently ignored.

Value

A list with the following elements:

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.

cond.loglik	A vector containing the conditional log likelihoods 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.
loglik	The estimated log-likelihood.
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

See Also

pomp-class

Examples

```
## See the vignettes for examples.
```

---

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.

Usage

```
data(dacca)
```

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

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

---

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)
```

Arguments

n	integer; number of random variates to generate.
size	scalar integer; number of individuals at risk.
rate	numeric vector of hazard rates.
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

`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)`.

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`).

Author(s)

Aaron A. King <kingaa at umich dot edu>

Examples

```
print(x <- reulermultinom(5,size=100,rate=c(a=1,b=2,c=3),dt=0.1))
deulermultinom(x,size=100,rate=c(1,2,3),dt=0.1)
```

gompertz *Gompertz model with log-normal observations.*

Description

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

Usage

```
data(gompertz)
```

Details

The state process is $X_{t+1} = K^{(1-S)} X_t^S \varepsilon_t$, 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. $sigma$, the measurement error s.d. tau, and the initial condition X_0. The model is parameterized internally by the logarithms of r, K, σ, and τ; the initial condition is parameterized directly. The pomp object includes parameter transformations to and from this internal parameterization.

See Also

pomp-class and the introductory vignette vignette("intro_to_pomp").

Examples

```
data(gompertz)
plot(gompertz)
coef(gompertz)
coef(gompertz,transform=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.

See Also

pomp-class and the vignettes

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'
    )
```

---

mif *The MIF algorithm*

Description

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

Usage

```
mif(object, ...)
## S4 method for signature 'pomp'
mif(object, Nmif = 1, start, pars, ivps = character(0),
    particles, rw.sd, Np, ic.lag, var.factor, cooling.factor,
    weighted = TRUE, tol = 1e-17, max.fail = 0,
    verbose = getOption("verbose"), ...)
## S4 method for signature 'pfilterd.pomp'
```

```
mif(object, Nmif = 1, start, pars, ivps = character(0),
    particles, rw.sd, Np, ic.lag, var.factor, cooling.factor,
    weighted = TRUE, tol, max.fail = 0,
    verbose = getOption("verbose"), ...)
## S4 method for signature 'mif'
mif(object, Nmif, start, pars, ivps,
    particles, rw.sd, Np, ic.lag, var.factor, cooling.factor,
    weighted = TRUE, tol, max.fail = 0,
    verbose = getOption("verbose"), ...)
## S4 method for signature 'mif'
continue(object, Nmif = 1, start, pars, ivps,
    particles, rw.sd, Np, ic.lag, var.factor, cooling.factor,
    weighted = TRUE, tol, max.fail = 0,
    verbose = getOption("verbose"), ...)
```

Arguments

object	An object of class pomp.
Nmif	The number of MIF iterations to perform.
start	named numerical vector; the starting guess of the parameters.
pars	optional character vector naming the ordinary parameters to be estimated. Every parameter named in pars must have a positive random-walk standard deviation specified in rw.sd. Leaving pars unspecified is equivalent to setting it equal to the names of all parameters with a positive value of rw.sd that are not ivps.
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 pars is empty, 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.

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.factor — a positive number not greater than 1; the exponential cooling factor, alpha.

weighted — logical; if TRUE, the MIF update (a weighted average) is used. If FALSE, the MIF update is not used; instead, an unweighed average of the filtering means is used for the update.

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. Currently, these are ignored.

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).

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-class).

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.

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

See Also

mif-methods, pomp, pomp-class, pfilter. See the "intro_to_pomp" vignette for examples.

---

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

Description

Methods of the "mif" class.

Usage

```
## S4 method for signature 'mif'
logLik(object, ...)
## S4 method for signature 'mif'
conv.rec(object, pars, transform = FALSE, ...)
## S4 method for signature 'mif'
plot(x, y = NULL, ...)
compare.mif(z)
```

Arguments

object	The mif object.
pars	Names of parameters.
x	The mif object.
y	Ignored.
z	A mif object or list of mif objects.
transform	optional logical; should the parameter transformations be applied? See coef for details.
...	Further arguments (either ignored or passed to underlying functions).

Methods

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.

mif Re-runs the MIF iterations. See the documentation for mif.

compare.mif Given a mif object or a list of mif objects, compare.mif produces a set of diagnostic plots.

plot Plots a series of diagnostic plots. When x is a mif object, plot(x) is equivalent to compare.mif(list(x)).

predvarplot predvarplot(object, pars = NULL, mean = FALSE, ...) produces a plot of the scaled prediction variances for each parameter. This can be used to diagnose a good value of the mif parameters var.factor and ic.lag. If used in this way, one should run mif with Nmif=1 first. Additional arguments in ... will be passed to the actual plotting function.

print Prints a summary of the mif object.

show Displays the mif object.

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.

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

See Also

mif, pomp, pomp-class, pfilter

---

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

```
nlf(object, start, est, lags, period = NA, tensor = FALSE,
    nconverge=1000, nasymp=1000, seed = 1066,
    transform = function (x) x,
    nrbf = 4, method = "subplex", skip.se = FALSE,
    verbose = FALSE, gr = NULL,
    bootstrap=FALSE, bootsamp = NULL,
    lql.frac = 0.1, se.par.frac = 0.1, eval.only = FALSE, ...)
```

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	optional function. If specified, forecasting is performed using data and model simulations transformed by this function. By default, transform is the identity function. The main purpose of transform is to achieve approximately multivariate normal forecasting errors. If data are univariate, transform should take a scalar and return a scalar. If data are multivariate, transform 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.
gr	optional; passed to optim if optim is used.
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 is functionally a wrapper for nlf.objfun, which does the statistical heavy lifting and should be consulted for details.

Value

A list corresponding to the output from the optimizer, except that the full parameter vector is returned (not just the ones fitted), the log quasilikelihood (LQL) (*not* -LQL) is reported, xstart is included, and asymptotic Wald standard errors based on M-estimator theory are returned for each fitted parameter.

Author(s)

Stephen P. Ellner <spe2 at cornell dot edu> and Bruce E. Kendall <kendall at bren dot ucsb 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. Available online at http://www2.bren.ucsb.edu/~kendall/pubs/1999Ecology.pdf

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. Available online at http://repositories.cdlib.org/postprints/818/

---

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

Description

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

Usage

```
data(ou2)
```

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 $y_i(t) \sim \text{normal}(x_i(t), \tau)$. The functions rprocess, dprocess, rmeasure, dmeasure, and skeleton are implemented using compiled C code for computational speed: see the source code for details. This object is demonstrated in the vignette "Advanced topics in pomp".

See Also

pomp and the vignettes

Examples

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

pfilter *Particle filter*

Description

Run a plain vanilla particle filter. Resampling is performed at each observation.

Usage

```
## S4 method for signature 'pomp'
pfilter(object, params, Np, tol = 1e-17,
    max.fail = 0, 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,
    max.fail = 0, pred.mean = FALSE, pred.var = FALSE,
    filter.mean = FALSE, save.states = FALSE,
    save.params = FALSE, seed = NULL,
    verbose = getOption("verbose"), ...)
```

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 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).

`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.

`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.

`...` Additional arguments unused at present.

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. In addition, `logLik` returns the log likelihood. 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. That is `coef(pfilter(obj,params=theta))==theta` if `theta` is a named vector of parameters.

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-class

Examples

```
## See the vignettes for examples.
```

---

pfilter-methods *Methods of the "pfilterd.pomp" class*

Description

Methods of the "pfilterd.pomp" class.

Usage

```
## S4 method for signature 'pfilterd.pomp'
logLik(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 `pfilterd.pomp` or inheriting class `pfilterd.pomp`.
`pars`	Names of parameters.
`...`	Additional arguments unused at present.

Author(s)

Aaron A. King <kingaa at umich dot edu>

See Also

pfilter, pomp-class

---

plugins *Plug-ins for dynamical models based on stochastic Euler algorithms*

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 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 be of the form 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. 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 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. 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 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.

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

Examples

```
## examples showing how to use these functions
## are provided in the vignette "intro_to_pomp"
```

```
## Not run:
vignette("intro_to_pomp")

## End(Not run)
```

---

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

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.
`pars`	optional character vector naming the ordinary parameters to be estimated. Every parameter named in `pars` must have a positive random-walk standard deviation specified in `rw.sd`. Leaving `pars` unspecified is equivalent to setting it equal to the names of all parameters with a positive value of `rw.sd`.
`dprior`	Function of prototype `dprior(params,hyperparams,...,log)` that evaluates the prior density. This defaults to an improper uniform prior.

rw.sd	numeric vector with names; used to parameterize a Gaussian random walk MCMC proposal. The random walk is only applied to parameters named in pars. The algorithm requires that the random walk be nontrivial, so each element in rw.sd[pars] 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 pars.
Np	a positive integer; the number of particles to use in each filtering operation.
hyperparams	optional list; parameters to be passed to dprior.
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. This class inherits from class pfilterd.pomp and contains the following additional slots:

pars, Nmcmc, dprior, hyperparams These slots hold the values of the corresponding arguments of the call to pmcmc.

random.walk.sd a named numeric vector containing the random-walk variances used to parameterize a Gaussian random walk MCMC proposal.

log.prior a numeric value containing the log of the prior density evaluated at the parameter vector in the params slot.

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 (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. See the "intro_to_pomp" vignette for an example [CURRENTLY, ONLY DEMONSTRATING THE MIF ALGORITHM, WHICH IS ALGORITHMICALLY VERY SIMILAR TO PMCMC SINCE THEY BOTH DEPEND CRITICALLY ON A PARTICLE FILTERING STEP].

---

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 'pmcmc'
filter.mean(object, pars, ...)
## S4 method for signature 'pmcmc'
plot(x, y = NULL, ...)
## S4 method for signature 'pmcmc'
dprior(object, params, log = FALSE, ...)
compare.pmcmc(z)
```

Arguments

object, x	The pmcmc object.
pars	Names of parameters.
y	Ignored.
z	A pmcmc object or list of pmcmc objects.
params	Named vector of parameters.
log	if TRUE, log probabilities are returned.
...	Further arguments (either ignored or passed to underlying functions).

Methods

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.

dprior `dprior(object,params,log)` evaluates the prior density at `params` with values of the hyperparameters given by `object@hyperparams`.

pmcmc Re-runs the PMCMC iterations. See the documentation for `pmcmc`.

compare.pmcmc Given a `pmcmc` object or a list of `pmcmc` objects, `compare.pmcmc` produces a set of diagnostic plots.

plot Plots a series of diagnostic plots. When `x` is a `pmcmc` object, `plot(x)` is equivalent to `compare.pmcmc(list(x))`.

filter.mean `filter.mean(object, pars = NULL)` returns the rows of the filtering-mean matrix corresponding to the names in `pars`. By default, all rows are returned.

print Prints a summary of the `pmcmc` object.

show Displays the `pmcmc` object.

pfilter See `pfilter`.

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.*

Description

Create a new `pomp` object to hold a partially-observed Markov process model together with a uni- or multi-variate time series.

Usage

```
## S4 method for signature 'data.frame'
pomp(data, times, t0, ..., rprocess, dprocess, rmeasure, dmeasure,
       measurement.model,
     skeleton = NULL, skeleton.type = c("map","vectorfield"), skelmap.delta.t = 1,
       initializer, covar, tcovar,
       obsnames, statenames, paramnames, covarnames,
       PACKAGE, parameter.transform, parameter.inv.transform)
## S4 method for signature 'numeric'
pomp(data, times, t0, ..., rprocess, dprocess, rmeasure, dmeasure,
       measurement.model,
     skeleton = NULL, skeleton.type = c("map","vectorfield"), skelmap.delta.t = 1,
       initializer, covar, tcovar,
       obsnames, statenames, paramnames, covarnames,
       PACKAGE, parameter.transform, parameter.inv.transform)
## S4 method for signature 'matrix'
pomp(data, times, t0, ..., rprocess, dprocess, rmeasure, dmeasure,
       measurement.model,
     skeleton = NULL, skeleton.type = c("map","vectorfield"), skelmap.delta.t = 1,
       initializer, covar, tcovar,
       obsnames, statenames, paramnames, covarnames,
       PACKAGE, parameter.transform, parameter.inv.transform)
## S4 method for signature 'pomp'
pomp(data, times, t0, ..., rprocess, dprocess, rmeasure, dmeasure,
       measurement.model,
     skeleton = NULL, skeleton.type = c("map","vectorfield"), skelmap.delta.t = 1,
       initializer, covar, tcovar,
       obsnames, statenames, paramnames, covarnames,
       PACKAGE, parameter.transform, parameter.inv.transform)
```

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, or a `pomp` object.. 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 of the column of observation times. Note that, in this case, `data` is a data-frame, it will be internally coerced to an array with storage-mode 'double'. Note that the `times` must be numeric and strictly increasing.
`t0`	The zero-time. This must be no later than the time of the first observation, `times[1]`. The stochastic dynamical system is initialized at time `t0`.
`rprocess`	optional function; a function of prototype `rprocess(xstart,times,params,...)` that simulates from the unobserved process. The easiest way to specify `rprocess`

is to use one of the plugins provided as part of the **pomp** package. See below for 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 easiest way to specify dprocess is to use one of the plugins provided as part of the **pomp** package. It is not typically necessary (or even feasible) to define dprocess. See below for details.

rmeasure — optional; the measurement model simulator. This can be specified in one of three 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 "examples/pomp.h". In the above cases, if the measurement model depends on covariates, the optional argument covars will be filled with interpolated values at each call. (3) using the formula-based measurement.model facility (see below).

dmeasure — optional; the measurement model probability density function. This can be specified in one of three 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 "examples/pomp.h". In the above cases, if the measurement model depends on covariates, the optional argument covars will be filled with interpolated values at each call. (3) using the formula-based measurement.model facility (see below). 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 and used to generate rmeasure and dmeasure functions. If measurement.model is given it overrides any specification of rmeasure or dmeasure. See below for an example. **NB:** it will typically be possible to acclerate measurement model computations by writing dmeasure and/or rmeasure functions directly.

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". 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 two 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, or (2) as the name of a native (compiled) routine with prototype "pomp_skeleton" as defined in the header file "pomp.h". If the deterministic skeleton depends on covariates, the optional argument covars will be filled with interpolated values of the covariates at the time t.

With a discrete-time skeleton, 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.

`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 (i.e., if it is unspecified when `pomp` is called), the initializer assumes any parameters in `params` the names of which end in "`.0`" are initial values. These are simply copied over as initial conditions when `init.state` is called (see `init.state-pomp`). The names of the state variables are the same as the corresponding initial value parameters, but with the "`.0`" dropped.

`covar, tcovar` An optional table of covariates: `covar` is the table (with one column per variable) and `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, i.e., whenever `rprocess`, `dprocess`, `rmeasure`, `dmeasure`, or `init.state` is evaluated. The resulting interpolated values are passed to the corresponding functions as a numeric vector named `covars`.

`obsnames, statenames, paramnames, covarnames`
Optional character vectors specifying the names of observables, state variables, parameters, or covariates, respectively. These are only used in the event that one or more of the basic functions (`rprocess`, `dprocess`, `rmeasure`, `dmeasure`, `skeleton`) are defined using native routines. In that case, these name vectors are matched against the corresponding names and the indices of the names are passed to the native routines. Using this facility allows one to write one or more of `rprocess`, `dprocess`, `rmeasure`, `dmeasure`, `skeleton` in native code in a way that does not depend on the order of states, parameters, and covariates at run time. See the "advanced_topic_in_pomp" vignette for more on this topic and examples.

`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 from the user's scale to the scale that `rprocess`, `dprocess`, `rmeasure`, `dmeasure`, `skeleton`, and `initializer` will use internally. `parameter.inv.transform` should be the inverse of `parameter.transform`. Note that it is the user's responsibility to make sure this holds. If `obj` is the constructed `pomp` object, and `coef(obj)` is non-empty, a simple check of this is `x <- coef(obj,transform=TRUE); obj1 <- obj; coef(obj1,transform=TRUE) <- x; identical(coef(obj),coef(obj1))` and `identical(coef(obj1,transform=TRUE),x)`. By default, both functions are the identity transformation. See the "introduction_to_pomp" vignette for an example.

`...` Any additional arguments given to `pomp` will be stored in the `pomp` object and passed as arguments to each of the functions `rprocess`, `dprocess`, `rmeasure`, `dmeasure`, and `initializer` whenever they are evaluated.

Details

It is not typically necessary (or desirable, or even feasible) to define all of the functions

rprocess, dprocess, rmeasure, dmeasure, and skeleton in any given problem. Each algorithm makes use of a different subset of these functions. In general, the specification of process-model codes rprocess and/or dprocess can be somewhat nontrivial: for this reason, plugins have been developed to streamline this process for the user. 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 or onestep.sim plugin for rprocess and onestep.dens plugin for dprocess are available. For exact simulation of certain continuous-time Markov chains, an implementation of Gillespie's algorithm is available (see gillespie.sim). To use the plugins, consult the help documentation (?plugins) and the vignettes.

It is anticipated that, in specific cases, it will be possible to obtain increased computational efficiency by writing custom versions of rprocess and/or dprocess. See the "Advanced Topics in pomp" vignette for a discussion of this. If such custom versions are desired, the following describes how each of these functions should be written in this case.

rprocess In general, the specification of rprocess can be somewhat nontrivial: for this reason, plugins have been developed to streamline this process for the user. 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 or onestep.sim plugin is available. For exact simulation of certain continuous-time Markov chains, an implementation of Gillespie's algorithm is available (see gillespie.sim). To use the plugins, consult the help documentation (?plugins) and the vignettes.

If the plugins are not used rprocess must have 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 a rank-2 array (matrix) with rows corresponding to state variables and columns corresponding to independent realizations of the process. params will similarly be a rank-2 array with rows corresponding to parameters and columns corresponding to independent realizations. The columns of params correspond to those of xstart; in particular, they will agree in number. Both xstart and params will have rownames, which are available for use by the user.

rprocess must return a rank-3 array with rownames. Suppose x is the array returned. Then dim(x)=c(nvars,nreps,ntimes), where nvars (=nrow(xstart)) is the number of state variables, nreps (=ncol(xstart)) is the number of independent realizations simulated, 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]. In particular, x[,,1] must be identical to xstart. The rownames of x must correspond to those of xstart.

At present, the following methods make use of rprocess:

- simulate
- pfilter
- mif
- nlf
- probe
- probe.match

dprocess In general, the specification of dprocess can be somewhat nontrivial: for this reason, plugins have been developed to streamline this process for the user. 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 onestep.dens plugin for dprocess is available. To use the plugins, consult the help documentation (?plugins) and the vignettes.

If the plugins are not used, dprocess must have at least the following arguments: x, times, params, log, and .... It may 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 dprocess, x may be assumed to be an nvars x nreps x ntimes array, where these terms have the same meanings as above. params will be a rank-2 array 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 will have rownames, available for use by the user.

dprocess must return a rank-2 array (matrix). Suppose d is the array returned. Then dim(d)=c(nreps,ntimes-1). 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 is returned.

In writing this function, you may assume that the transitions are consecutive. It should be quite clear that, but for this assumption, it would be quite difficult in general to write the transition probabilities. In fact, from one perspective, the algorithms in **pomp** are designed to overcome just this difficulty.

At present, no methods in pomp make use of dprocess.

The measurement-model, deterministic skeleton, and initializer components are easily specified without the use of plugins. The following is a guide to writing these components.

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 may be assumed to be a named numeric vector of length nvars, (which has the same meanings as above). t is a scalar quantity, the time at which the measurement is made. params may be assumed to be a named numeric vector of length npars.

rmeasure must return a named numeric vector. If y is the returned vector, then length(y)=nobs, where nobs is the number of observable variables.

At present, the following methods make use of rmeasure:

- simulate
- nlf
- probe
- probe.match

dmeasure if provided, must take at least the arguments y, x, t, params, log, and .... y may be assumed to 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, a named numeric vector containing parameters; and t, a scalar, the corresponding observation time. It may take additional arguments which will be filled with user-specified data as above. dmeasure must return a single numeric value, the pdf of y given x at time t. If log=TRUE, then the log of the pdf is returned.

At present, the following methods make use of dmeasure:

- pfilter
- mif

`skeleton` If `skeleton` is an R function, it 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. The optional argument `covars` is a numeric vector containing the values of the covariates at the time `t`. `covars` will have one value for each column of the covariate table specified when the `pomp` object was created. `covars` is constructed from the covariate table (see `covar`, below) by interpolation. `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`. The return value is interpreted as the 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`.

If `skeleton` is the name of a native routine, this routine must be of prototype "pomp_skeleton" as defined in the header "pomp.h" (see the "include" directory in the installed package directory).

At present, the following methods make use of `skeleton`:

- `trajectory`
- `traj.match`

`initializer` if provided, must have at least the arguments `params`, `t0`, and `....` `params` is 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.

Value

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.

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 (see Details), 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 model and if `skeleton` is specified, the user is responsible for verifying that it corresponds to the true deterministic skeleton of the model. Each **pomp**-package algorithm uses some subset of the five basic components (`rprocess`, `dprocess`, `rmeasure`, `dmeasure`, `skeleton`). If an algorithm requires a component that was not given in the construction of the `pomp` object, an error is generated.

Author(s)

Aaron A. King <kingaa at umich dot edu>

See Also

pomp-methods, plugins, `time`, `time<-`, `timezero`, `timezero<-`, `coef`, `coef<-`, `obs`, `states`, `window`, `as.data.frame.pomp`

Examples

```
## For examples, see the vignettes, the data()-loadable
## example \code{pomp} objects, and the provided example files.
## Not run:
vignette("intro_to_pomp")
vignette("advanced_topics_in_pomp")
data(package="pomp")
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-methods *Methods of the "pomp" class*

Description

Methods of the pomp class.

Usage

```
## 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'
data.array(object, vars, ...)
## 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'
show(object)
## S4 method for signature 'pomp'
```

```
as(object, class)
## S3 method for class 'pomp'
as.data.frame(x, row.names, optional, ...)
## S4 method for signature 'pomp,data.frame'
coerce(from, to = "data.frame", strict = TRUE)
## S4 method for signature 'pomp'
print(x, ...)
## 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, ...)
```

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.
`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(coe` 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, data.array These functions are synonymous. `obs(object)` returns the array of observations. `obs(object,vars)` gives just the observations of variables named `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.

rprocess simulates the process model. See rprocess-pomp.

dprocess evaluates the process model density. See dprocess-pomp.

rmeasure simulates the measurement model. See rmeasure-pomp.

dmeasure evaluates the measurement-model density. See dmeasure-pomp.

skeleton evaluates the deterministic skeleton (be it a vector field or a map). See skeleton-pomp.

init.state returns a vector of initialial conditions. See init.state-pomp.

simulate simulate can be used to simulate state and observation trajectories. See documentation under simulate-pomp.

Author(s)

Aaron A. King <kingaa at umich dot edu>

See Also

pomp, pomp-class, rprocess, dprocess, rmeasure, dmeasure, init.state, simulate

---

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).

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

```
              nsim, seed = NULL,
              method = c("subplex","Nelder-Mead","SANN","BFGS","sannbox"),
              verbose = getOption("verbose"),
              eval.only = FALSE, fail.value = NA, ...)
  ## S4 method for signature 'probe.matched.pomp'
probe.match(object, start, est,
              probes, weights,
              nsim, seed = NULL,
              method = c("subplex","Nelder-Mead","SANN","BFGS","sannbox"),
              verbose = getOption("verbose"),
              eval.only = FALSE, fail.value, ...)
```

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.
weights	optional numeric vector of relative weights. Must be of the same length as probes.
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 probe, these are currently ignored. In the case of probe.match, these are passed to optim or subplex in the control list.

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.

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

value value of the objective function.

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

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

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

Examples

```
data(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 '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.

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.

Author(s)

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

Aaron A. King (kingaa at umich dot edu)

See Also

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

`profileDesign` *Design matrices for likelihood profile calculations.*

Description

`profileDesign` generates a data-frame where each row can be used as the starting point for a profile likelihood calculation.

Usage

```
profileDesign(..., lower, upper, nprof)
```

Arguments

`...`	Specifies the parameters over which to profile.
`lower, upper`	Named numeric vectors, specifying the range over which the other parameters are to be sampled.
`nprof`	The number of starts per profile point.

Value

`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>

See Also

`sobol`

Examples

```
## 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)
```

ricker *Ricker model with Poisson observations.*

Description

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

Usage

```
data(ricker)
```

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 variance σ^2. The observed variables y_t are distributed as $\mathrm{Poisson}(\phi N_t)$.

See Also

pomp-class and the vignettes

Examples

```
data(ricker)
plot(ricker)
coef(ricker)
```

rw2 *Two-dimensional random-walk process*

Description

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

Usage

```
data(rw2)
```

Details

The random-walk process is fully but noisily observed.

See Also

pomp-class and the vignettes

Examples

```
data(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]])
```

---

simulate-pomp *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, ...)
```

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.
...	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-class

Examples

```
data(ou2)
x <- simulate(ou2,seed=3495485,nsim=10)
x <- simulate(ou2,seed=3495485,nsim=10,states=TRUE,obs=TRUE)
```

---

sir

Seasonal SIR model implemented using two stochastic simulation algorithms.

Description

euler.sir is a pomp object encoding a simple seasonal SIR model. Simulation is performed using an Euler multinomial approximation (AKA tau leap method). gillespie.sir has the same model implemented using Gillespie's algorithm.

Usage

```
data(euler.sir)
data(gillespie.sir)
```

See Also

pomp-class and the vignettes

Examples

```
data(euler.sir)
plot(euler.sir)
x <- simulate(euler.sir,nsim=10,seed=20348585)
plot(x[[1]])

data(gillespie.sir)
plot(gillespie.sir)
x <- simulate(gillespie.sir,nsim=1,seed=20348585)
plot(x)
```

---

sliceDesign *Design matrices for likelihood slices.*

Description

sliceDesign generates a data-frame representing points taken along one or more slices through a point in a multidimensional space.

Usage

```
sliceDesign(center, ...)
```

Arguments

center	center is a named numeric vectors specifying the point through which the slice(s) is (are) to be taken.
...	Additional numeric vector arguments specify the slices.

Value

sliceDesign returns a data frame with one row per point along a slice. The column slice is a factor that tells which slice each point belongs to.

Author(s)

Aaron A. King <kingaa at umich dot edu>

See Also

profileDesign

Examples

```
## 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)
```

---

sobol *Sobol' low-discrepancy sequence*

Description

Generate a data-frame containing a Sobol' low-discrepancy sequence.

Usage

```
sobol(vars, n)
sobolDesign(lower, upper, nseq)
```

Arguments

vars	Named list of ranges of variables.
lower,upper	named numeric vectors giving the lower and upper bounds of the ranges, respectively.
n,nseq	Number of vectors requested.

Value

sobol	Returns a data frame with n 'observations' of the variables in vars.
sobolDesign	Returns a data frame with nseq 'observations' of the variables over the range specified.

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

See Also

sliceDesign, profileDesign

Examples

```
plot(sobol(vars=list(a=c(0,1),b=c(100,200)),100))
plot(sobolDesign(lower=c(a=0,b=100),upper=c(b=200,a=1),100))
```

---

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 (d.reuman at imperial dot ac dot uk)

Cai GoGwilt

Aaron A. King (kingaa at umich dot edu)

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

```
data(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

Match trajectories to data.

Usage

```
  ## S4 method for signature 'pomp'
traj.match(object, start, est,
           method = c("Nelder-Mead", "sannbox", "subplex"),
           gr = NULL, eval.only = FALSE, ...)
  ## S4 method for signature 'traj.matched.pomp'
traj.match(object, start, est,
           method = c("Nelder-Mead", "sannbox", "subplex"),
           gr = NULL, eval.only = FALSE, ...)
```

Arguments

`object`	A pomp object.
`start`	initial guess for parameters.
`est`	character vector containing the names of parameters to be estimated.
`method`	Optimization method. Choices are `subplex`, "sannbox", and any of the methods used by `optim`.
`gr`	Passed to `optim`.
`eval.only`	logical; if `TRUE`, no optimization is attempted and the log likelihood value is evaluated at the `start` parameters.
`...`	Arguments that will be passed to `optim` or `subplex` via their `control` lists.

Details

Trajectory matching is accomplished using `optim`. The `trajectory` method is used for this, which in turn uses the `skeleton` slot of the pomp object. The quantity maximized is the likelihood of the data given the trajectory, as returned by `dmeasure`.

Value

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

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.

See Also

`trajectory`, `pomp`, `optim`, `subplex`

Examples

```
data(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')
```

`trajectory` *Compute trajectories of the determinstic skeleton.*

Description

The method `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 vectorfield; `trajectory` integrates the vectorfield to obtain a trajectory.

Usage

```
## S4 method for signature 'pomp'
trajectory(object, params, times, t0, ...)
```

Arguments

`object`	an object of class `pomp`.
`params`	a rank-2 array of parameters. Each column of `params` is a distinct parameter vector.
`times, t0`	`times` is a numeric vector specifying the times at which a trajectory is desired. `t0` specifies the start time (the time at which the initial conditions hold). The default for `times` is `times=time(object,t0=FALSE)` and `t0=timezero(object)`, respectively.
`...`	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.

Details

This function makes repeated calls to the user-supplied `skeleton` of the `pomp` object. For specifications on supplying this, see `pomp`.

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.

Value

Returns an array of dimensions `nvar` x `nreps` 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]`.

Author(s)

Aaron A. King <kingaa at umich dot edu>

See Also

pomp, traj.match, ode

Examples

```
data(euler.sir)
x <- trajectory(euler.sir)
plot(time(euler.sir),x["I",1,],type='l',xlab='time',ylab='I')
lines(time(euler.sir),x["cases",1,],col='red')

coef(euler.sir,c("gamma")) <- log(12)
x <- trajectory(euler.sir)
plot(time(euler.sir),x["I",1,],type='l',xlab='time',ylab='I')
lines(time(euler.sir),x["cases",1,],col='red')
```

---

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

Description

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

Usage

```
data(verhulst)
```

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

```
data(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