Package ‘pomp’

March 1, 2017

Type Package
Title Statistical Inference for Partially Observed Markov Processes
Version 1.11
Date 2017-03-01
URL http://kingaa.github.io/pomp
Description Tools for working with partially observed Markov process (POMP) models (also known as stochastic dynamical systems, hidden Markov models, and nonlinear, non-Gaussian, state-space models). The package provides facilities for implementing POMP models, simulating them, and fitting them to time series data by a variety of frequentist and Bayesian methods. It is also a versatile platform for implementation of inference methods for general POMP models.
Depends R(>= 3.1.2), methods
Imports stats, graphics, digest, mvtnorm, deSolve, coda, subplex, nloptr
Suggests magrittr, plyr, reshape2, ggplot2, knitr
SystemRequirements For Windows users, Rtools (see http://cran.r-project.org/bin/windows/Rtools/).
License GPL (>= 2)
LazyData true
Contact kingaa at umich dot edu
BugReports http://github.com/kingaa/pomp/issues

simulate_pomp.R trajectory_pomp.R plot_pomp.R pfilter.R

**NeedsCompilation:** yes

**Author:**  
Aaron A. King [aut, cre],  
Edward L. Ionides [aut],  
Carles Breto [aut],  
Stephen P. Ellner [ctb],  
Matthew J. Ferrari [ctb],  
Bruce E. Kendall [ctb],  
Michael Lavine [ctb],  
Dao Nguyen [ctb],  
Daniel C. Reuman [ctb],  
Helen Wearing [ctb],  
Simon N. Wood [ctb],  
Sebastian Funk [ctb],  
Steven G. Johnson [ctb]

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

**Repository:**  
CRAN

**Date/Publication:**  
2017-03-01 19:05:53

---

**R topics documented:**

* pomp-package ................................................................. 3  
  Approximate Bayesian computation ........................................ 5  
  B-splines ................................................................. 7  
  Bayesian sequential Monte Carlo ...................................... 8  
  blowflies ............................................................... 10  
  Childhood disease incidence data .................................... 11  
  dacca ................................................................. 12  
  design ................................................................. 13  
  Ensemble Kalman filters ............................................... 15  
  eulermultinom .......................................................... 17  
  Example pomp models .................................................. 19  
  gompertz ............................................................... 20  
  Iterated filtering ..................................................... 21  
  Iterated filtering 2 ................................................... 24  
  logmeanexp ............................................................ 28  
  Low-level-interface .................................................... 29  
  MCMC proposal distributions ........................................... 33  
  Nonlinear forecasting ................................................ 34  
  ou2 ................................................................. 36  
  parmat ............................................................... 37  
  Particle filter ......................................................... 37  
  Particle Markov Chain Monte Carlo .................................. 41  
  pomp constructor ....................................................... 43  
  pomp methods .......................................................... 55
Description

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

Data analysis using pomp

The first step in using pomp is to encode one’s model(s) and data in objects of class pomp. One does this via a call to pomp constructor function. Details on this are given in the documentation (pomp).

pomp version 1.11 provides algorithms for

1. simulation of stochastic dynamical systems; see simulate
2. particle filtering (AKA sequential Monte Carlo or sequential importance sampling); see pfilter
3. the iterated filtering methods of Ionides et al. (2006, 2011, 2015); see mif2
4. the nonlinear forecasting algorithm of Kendall et al. (2005); see nlf
5. the particle MCMC approach of Andrieu et al. (2010); see pmcmc
6. the probe-matching method of Kendall et al. (1999, 2005); see probe.match
7. a spectral probe-matching method (Reuman et al. 2006, 2008); see spect.match
8. synthetic likelihood a la Wood (2010); see probe
9. approximate Bayesian computation (Toni et al. 2009); see abc
10. the approximate Bayesian sequential Monte Carlo scheme of Liu & West (2001); see bsmc2
11. ensemble and ensemble adjusted Kalman filters; see enkf
12. simple trajectory matching; see traj.match.

The package also provides various tools for plotting and extracting information on models and data.
Developing inference tools on the pomp platform

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

Comments, bug reports, feature requests

Contributions are welcome, as are comments, feature requests, and bug reports. Please submit these via the issues page. See the package website for more information, access to the package new RSS feed, links to the authors’ websites, references to the literature, and up-to-date versions of the package source and documentation. Help requests are welcome, but please read the FAQ before sending requests.

We are very interested in improving the documentation and the package error and warning messages. If you find a portion of the documentation impenetrable, please let us know, preferably with suggestions for improvement. If you find an error message that is uninformative or misleading, please be sure to let us know. The best way to do so is via the package issues page. Please do read the FAQ before reporting an issue.

Documentation

A number of tutorials, demonstrating the construction of pomp objects and the application of various inference algorithms, are available on the package webpage. Examples are given in the tutorials on the package website, in the demos, and via the pompExample function. See a list of the demos via
demo(package="pomp")

and a list of the included examples via

pompExample()

History

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

License

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

Author(s)

Aaron A. King

References


See the package website, https://kingaa.github.io/pomp, for more references.
Approximate Bayesian computation

Estimation by approximate Bayesian computation (ABC)

Description

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

Usage

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

Arguments

- **object**: An object of class `pomp`.
- **Nabc**: The number of ABC iterations to perform.
- **start**: named numeric vector; the starting guess of the parameters.
- **proposal**: optional function that draws from the proposal distribution. Currently, the proposal distribution must be symmetric for proper inference: it is the user’s responsibility to ensure that it is. Several functions that construct appropriate proposal function are provided: see [MCMC proposal functions](#) for more information.
Approximate Bayesian computation

- **probes**  List of probes (AKA summary statistics). See `probe` for details.
- **scale**  named numeric vector of scales.
- **epsilon**  ABC tolerance.
- **verbose**  logical; if TRUE, print progress reports.
- **pars**  Names of parameters.
- **scatter**  optional logical; If TRUE, draw scatterplots. If FALSE, draw traceplots.
- **x**  abc object.
- **y**  Ignored.
- **...**  Additional arguments. These are currently ignored.

**Running ABC**

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

**Re-running ABC iterations**

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

**Continuing ABC iterations**

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

**Methods**

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

- `conv.rec(object, pars)` returns the columns of the convergence-record matrix corresponding to the names in `pars`. By default, all rows are returned.
- `c` Concatenates `abc` objects into an `abcList`.
- `plot` Diagnostic plots.
- `covmat(object, start, thin, expand)` computes the empirical covariance matrix of the ABC samples beginning with iteration `start` and thinning by factor `thin`. It expands this by a factor `expand*2/n`, where `n` is the number of parameters estimated. The intention is that the resulting matrix is a suitable input to the proposal function `mvn.rw`.

**Author(s)**

Edward L. Ionides, Aaron A. King
B-splines

References


See Also

pomp, probe, MCMC proposal distributions, and the tutorials on the package website.

B-splines

B-spline bases

Description

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

Usage

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

Arguments

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

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.
Bayesian sequential Monte Carlo

C API

Access to the underlying C routines is available: see the header file `pomp.h` for definition and documentation of the C API. At an R prompt, execute

```r
gcm Nhshow Nhsystem Nhfile [BincludeOpomp NhBLpackage\]BpompBHII
```
to view this file.

Author(s)

Aaron A. King

Examples

```r
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')
```

Bayesian sequential Monte Carlo

The Liu and West Bayesian particle filter

Description


Usage

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

Arguments

object: An object of class pomp or inheriting class pomp.

params, Np: Specifications for the prior distribution of particles. See details below.
est: Names of the rows of params that are to be estimated. No updates will be made
to the other parameters. If est is not specified, all parameters for which there is
variation in params will be estimated.

smooth: Kernel density smoothing parameters. The compensating shrinkage factor will
be sqrt(1-smooth^2). Thus, smooth=0 means that no noise will be added to
parameters. Generally, the value of smooth should be chosen close to 0 (i.e.,
shrink~0.1).

ntries: Number of draws from rprocess per particle used to estimate the expected
value of the state process at time t+1 given the state and parameters at time t.
tol: Particles with log likelihood below tol are considered to be “lost”. A filtering
failure occurs when, at some time point, all particles are lost. When all particles
are lost, the conditional log likelihood at that time point is set to be log(tol).

lower, upper: optional; lower and upper bounds on the priors. This is useful in case there are
box constraints satisfied by the priors. The posterior is guaranteed to lie within
these bounds.

verbose: logical; if TRUE, print diagnostic messages.

max.fail: The maximum number of filtering failures allowed. If the number of filtering
failures exceeds this number, execution will terminate with an error.

transform: logical; if TRUE, the algorithm operates on the transformed scale.

Details

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

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

Value

An object of class “bsmc.pomp”. The “params” slot of this object will hold the parameter posterior
medians. The slots of this class include:

tpost: A matrix containing draws from the approximate posterior distribution.
nprior: A matrix containing draws from the prior distribution (identical to params on
call).
neff.sample.size: A vector containing the effective number of particles at each time point.
nsmooth: The smoothing parameter used (see above).
**blowflies**

Model for Nicholson’s *blowflies*.

**Description**

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

**Details**

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

The models are discrete delay equations:

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

\[
S(t+1) \sim \text{binomial}(N(t), \exp(-\delta\epsilon(t+1)\Delta t))
\]

\[
N(t) = R(t) + S(t)
\]

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

\[
y(t) \sim \text{negbin}(N(t), 1/\sigma_y^2)
\]
Childhood disease incidence data

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

Do

pompExample(blowflies, show=TRUE)

to view the code that constructs these pomp objects.

References


See Also

pomp

Examples

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

Childhood disease incidence data

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).

ewmeas and ewcitmeas are data frames containing weekly reported cases of measles in England and Wales. ewmeas records the total measles reports for the whole country, 1948–1966. One questionable data point has been replaced with an NA. ewcitmeas records the incidence in seven English
cities 1948–1987. These data were kindly provided by Ben Bolker, who writes: “Most of these data have been manually entered from published records by various people, and are prone to errors at several levels. All data are provided as is; use at your own risk.”

Usage

LondonYorke
ewmeas
ewcitmeas

References


Examples

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'  
)

plot(reports~time, data=ewmeas, type='l')

plot(reports~date, data=ewcitmeas, subset=city=="Liverpool", type='l')

dacca

Model of cholera transmission for historic Bengal.
**Description**

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

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

**Details**

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

**References**


**See Also**

euler.sir, pomp

**Examples**

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

design

*Design matrices for pomp calculations*

**Description**

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

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

Arguments

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

Details

The Sobol’ sequence generation is performed using codes from the `NLopt` library by S. Johnson.

Value

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

Author(s)

Aaron A. King

References

- Steven G. Johnson, The `NLopt` nonlinear-optimization package, [http://ab-initio.mit.edu/nlopt](http://ab-initio.mit.edu/nlopt)
Examples

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

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

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

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

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

Ensemble Kalman filters

**Description**

The ensemble Kalman filter and ensemble adjustment Kalman filter.

**Usage**

```r
## S4 method for signature 'pomp'
enkf(object, params, Np, h, R,  
   verbose = getOption("verbose"), ...)

## S4 method for signature 'pomp'
eakf(object, params, Np, C, R,  
   verbose = getOption("verbose"), ...)

## S4 method for signature 'kalmand.pomp'
logLik(object, ...)

## S4 method for signature 'kalmand.pomp'
cond.logLik(object, ...)

## S4 method for signature 'kalmand.pomp'
pred.mean(object, pars, ...)

## S4 method for signature 'kalmand.pomp'
filter.mean(object, pars, ...)
```
Ensemble Kalman filters

Arguments

- **object**: An object of class `pomp` or inheriting class `pomp`.
- **params**: optional named numeric vector containing the parameters at which the filtering should be performed. By default, `params = coef(object)`.
- **np**: the number of particles to use.
- **verbose**: logical; if TRUE, progress information is reported.
- **h**: function returning the expected value of the observation given the state.
- **C**: matrix converting state vector into expected value of the observation.
- **R**: matrix; variance of the measurement noise.
- **pars**: Names of variables.
- **...**: additional arguments (currently ignored).

Value

An object of class `kalmand.pomp`. This class inherits from class `pomp`.

Methods

- **logLik**: Extracts the estimated log likelihood.
- **cond.logLik**: Extracts the estimated conditional log likelihood
  \[ \ell_t(\theta) = \text{Prob}[y_t|y_1, \ldots, y_{t-1}], \]
  where \(y_t\) are the data, at time \(t\).
- **pred.mean**: Extract the mean of the approximate prediction distribution. This prediction distribution is that of
  \[ X_t|y_1, \ldots, y_{t-1}, \]
  where \(X_t, y_t\) are the state vector and data, respectively, at time \(t\).
- **filter.mean**: Extract the mean of the filtering distribution, which is that of
  \[ X_t|y_1, \ldots, y_t, \]
  where \(X_t, y_t\) are the state vector and data, respectively, at time \(t\).

Author(s)

Aaron A. King

References

See Also

*pomp*, *pfilter*, and the tutorials on the package website.

---

**eulermultinom**  
*The Euler-multinomial distributions and Gamma white-noise processes*

**Description**

This page documents both the Euler-multinomial family of distributions and the package’s simulator of Gamma white-noise processes.

**Usage**

```r
reulermultinom(n = 1, size, rate, dt)

deulermultinom(x, size, rate, dt, log = FALSE)

rgammawn(n = 1, sigma, dt)
```

**Arguments**

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

**Details**

If \( N \) individuals face constant hazards of death in \( k \) ways at rates \( r_1, r_2, \ldots, r_k \), then in an interval of duration \( \Delta t \), 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 \( \Delta n_i \) 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 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
where $\mathbf{r}$ is the vector of rates. Evaluate the probability that $\mathbf{x} = (x_1, \ldots, x_k)$ are the numbers of individuals who have died in each of the $k$ ways over the interval $\Delta t = dt$, by doing

\[(\text{deulermultinom}(x=x, \text{size}=N, \text{rate}=r, dt=dt)).\]

Breto & Ionides (2011) discuss how an infinitesimally overdispersed death process can be constructed by compounding a binomial process with a Gamma white noise process. The Euler approximation of the resulting process can be obtained as follows. Let the increments of the equidispersed process be given by

\[(\text{reulermultinom}(\text{size}=N, \text{rate}=r, dt=dt)).\]

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

\begin{verbatim}
dW <- rgammawn(sigma=sigma, dt=dt)
dn <- reulermultinom(size=N, rate=r, dt=dW)
\end{verbatim}

or

\begin{verbatim}
dn <- reulermultinom(size=N, rate=r*dW/dt, dt=dt).
\end{verbatim}

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

For all of the functions described here, access to the underlying C routines is available: see below.

**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 ($\text{size}$, $\text{rate}$, $dt$).

- **rgammawn**: Returns a vector of length $n$ containing random increments of the integrated Gamma white noise process with intensity $\sigma$.

**C API**

An interface for C codes using these functions is provided by the package. At an R prompt, execute

\begin{verbatim}
file.show(system.file("include/pomp.h", package="pomp"))
\end{verbatim}

to view the `pomp.h` header file that defines and explains the API.
Example pomp models

Author(s)
Aaron A. King

References

Examples
print(dn <- reulermultinom(S, size=100, rate=c(a=1, b=2, c=3), dt=0.1))
dreulermultinom(x=dn, size=100, rate=c(1,2,3), dt=0.1)
## an Euler-multinomial with overdispersed transitions:
dt <- 0.1
dW <- rgamma(100, sigma=0.1, dt=dt)
print(dn <- reulermultinom(S, size=100, rate=c(a=1, b=2, c=3), dt=dW))

Example pomp models  Examples of the construction of POMP models

Description
pompExample loads pre-built example pomp objects.

Usage
pompExample(example, ..., show = FALSE, envir = .GlobalEnv)

Arguments
example  example to load given as a name or literal character string. Evoked without an argument, pompExample lists all available examples.
...  additional arguments define symbols in the environment within which the example code is executed.
show  logical; if TRUE, display, but do not execute, the example R code.
envir  the environment into which the objects should be loaded. If envir=NULL, then the created objects are returned in a list.

Details
Directories listed in the global option pomp.examples (which can be changed using options()) are searched for file named ‘<example>.R’. If found, this file will be sourced in a temporary environment. Additional arguments to pompExample define variables within this environment and will therefore be available when the code in ‘<example>.R’ is sourced.

The codes that construct these pomp objects can be found in the ‘examples’ directory in the installed package. Do system.file("examples",package="pomp") to find this directory.
Value

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

Author(s)

Aaron A. King

See Also

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

Examples

pompExample()
pompExample(euler.sir)
pompExample("gompertz")

pompExample(ricker, envir=NULL)
## Not run:
pompExample(bbs, show=TRUE)

## End(Not run)

---
gompertz Gompertz model with log-normal observations.

Description

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

Details

The state process is $X_{t+1} = K^{1-S} X_t^S \epsilon_t$, where $S = e^{-r}$ and the $\epsilon_t$ are i.i.d. lognormal random deviates with variance $\sigma^2$. The observed variables $Y_t$ are distributed as 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 pomp object includes parameter transformations that log-transform the parameters for estimation purposes.

See Also


Examples

pompExample(gompertz)
plot(gompertz)
coef(gompertz)
coef(gompertz, transform=TRUE)
Iterated filtering

Maximum likelihood by iterated filtering

Description

Iterated filtering algorithms for estimating the parameters of a partially-observed Markov process. Running \texttt{mif} causes the iterated filtering algorithm to run for a specified number of iterations. At each iteration, the particle filter is performed on a perturbed version of the model. Specifically, parameters to be estimated are subjected to random perturbations at each observation. This extra variability effectively smooths the likelihood surface and combats particle depletion by introducing diversity into the population of particles. At the iterations progress, the magnitude of the perturbations is diminished according to a user-specified cooling schedule. For most purposes, \texttt{mif} has been superseded by \texttt{mif2}.

Usage

```r
## S4 method for signature 'pomp'
mif(object, Nmif = 1, start, ivps = character(0),
    rw.sd, Np, ic.lag, var.factor = 1,
    cooling.type, cooling.fraction.50,
    method = c("mif","unweighted","fp","mif2"),
    tol = 1e-17, max.fail = Inf,
    verbose = getOption("verbose"), transform = FALSE, ...)
## S4 method for signature 'pfilterd.pomp'
mif(object, Nmif = 1, Np, tol, ...)
## S4 method for signature 'mif'
mif(object, Nmif, start, ivps,
    rw.sd, Np, ic.lag, var.factor,
    cooling.type, cooling.fraction.50,
    method, tol, transform, ...)
## S4 method for signature 'mif'
continue(object, Nmif = 1, ...)
## S4 method for signature 'mif'
conv.rec(object, pars, transform = FALSE, ...)
## S4 method for signature 'mifList'
conv.rec(object, ...)
```

Arguments

- \texttt{object} An object of class \texttt{pomp}.
- \texttt{Nmif} The number of filtering iterations to perform.
- \texttt{start} named numerical vector; the starting guess of the parameters.
- \texttt{ivps} optional character vector naming the initial-value parameters (IVPs) to be estimated. Every parameter named in \texttt{ivps} must have a positive random-walk standard deviation specified in \texttt{rw.sd}. If there are no regular parameters with positive \texttt{rw.sd}, i.e., only IVPs are to be estimated, see below “Using \texttt{mif} to estimate initial-value parameters only”.

Iterated filtering

- **rw.sd** numeric vector with names; the intensity of the random walk to be applied to parameters. `names(rw.sd)` must be a subset of `names(start)`. The random walk is not dynamically added to the initial-value parameters (named in `ivps`). The algorithm requires that the random walk be nontrivial, so that `rw.sd` be positive for at least one element.

- **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 set so, `ic.lag` is set to `length(times)` with a warning.

- **var.factor** optional positive scalar; 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`. By default, `var.factor=1`.

- **cooling.type, cooling.fraction.50** specifications for the cooling schedule, i.e., the manner in which the intensity of the parameter perturbations is reduced with successive filtering iterations. `cooling.type` specifies the nature of the cooling schedule.

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

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

- **method** method sets the update rule used in the algorithm. `method="mif"` uses the iterated filtering update rule (Ionides 2006, 2011); `method="unweighted"` updates the parameter to the unweighted average of the filtering means of the parameters at each time; `method="fp"` updates the parameter to the filtering mean at the end of the time series.

- **tol, max.fail** See the description under `pfilter`.

- **verbose** logical; if TRUE, print progress reports.

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

- **...** additional arguments that override the defaults.

- **pars** names of parameters.
Iterated filtering

Value

Upon successful completion, mif returns an object of class mif. The latter inherits from the pfilterd.pomp and pomp classes.

Regular parameters vs initial-value parameters

Initial-value parameters (IVPs) differ from regular parameters in that the majority of the information about these parameters is restricted to the early part of the time series. That is, increasing the length of the time series provides progressively less additional information about IVPs than it does about regular parameters. In mif, while regular parameters are perturbed at the initial time and after every observation, IVPs are perturbed only at the initial time.

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 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, the IVPs to be estimated are named in ivps and no positive entries in rw.sd correspond to any parameters not named in ivps. If theta is the current parameter vector, then at each mif iteration, Np particles are drawn from a normal distribution centered at theta and with width proportional to var.factor*rw.sd, a particle filtering operation is performed, and theta is replaced by the filtering mean at time(object)[ic.lag]. Note the implication that, when mif is used in this way on a time series any longer than ic.lag, unnecessary work is done. If the time series in object is longer than ic.lag, consider replacing object with window(object,end=ic.lag).

Methods

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

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

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

c Concatenates mif objects into a mifList.

plot Plots a series of diagnostic plots when applied to a mif or mifList object.
Author(s)

Aaron A. King

References


See Also

pomp, pfilter, mif2

Description

An improved iterated filtering algorithm for estimating the parameters of a partially-observed Markov process. Running mif2 causes the algorithm to perform a specified number of particle-filter iterations. At each iteration, the particle filter is performed on a perturbed version of the model, in which the parameters to be estimated are subjected to random perturbations at each observation. This extra variability effectively smooths the likelihood surface and combats particle depletion by introducing diversity into particle population. As the iterations progress, the magnitude of the perturbations is diminished according to a user-specified cooling schedule. The algorithm is presented and justified in Ionides et al. (2015).

Usage

```r
## S4 method for signature 'pomp'
mif2(object, Nmif = 1, start, Np, rw.sd, transform = FALSE,
     cooling.type = c("hyperbolic", "geometric"), cooling.fraction.50,
     tol = 1e-17, max.fail = Inf, verbose = getOption("verbose"), ...)
## S4 method for signature 'pfilterd.pomp'
mif2(object, Nmif = 1, Np, tol, ...)
## S4 method for signature 'mif2d.pomp'
mif2(object, Nmif, start, Np, rw.sd, transform,
     cooling.type, cooling.fraction.50, tol, ...)
## S4 method for signature 'mif2d.pomp'
continue(object, Nmif = 1, ...)
```
Arguments

object: An object of class `pomp`.

`Nmif`: The number of filtering iterations to perform.

`start`: named numerical vector; the starting guess of the parameters. By default, `start = coef(object)`.

`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))`) or as a function taking a positive integer argument. In the latter case, `Np(n)` must be a single positive integer, representing the number of particles to be used at the `n`-th timestep: `Np(1)` is the number of particles to use going from `timezero(object)` to `time(object)[1]`, `Np(2)`, from `time(object)[1]` to `time(object)[2]`, and so on. **Note that this behavior differs from that of `mif`!**

`rw.sd`: specification of the magnitude of the random-walk perturbations that will be applied to some or all model parameters. Parameters that are to be estimated should have positive perturbations specified here. The specification is given using the `rw.sd` function, which creates a list of unevaluated expressions. The latter are evaluated in a context where the model time variable is defined (as `time`). The expression `ivp(s)` can be used in this context as shorthand for `ifelse(time==time[1],s,0)`.

Likewise, `ivp(s,lag)` is equivalent to `ifelse(time==time[lag],s,0)`. See below for some examples. The perturbations that are applied are normally distributed with the specified s.d. If `transform = TRUE`, then they are applied on the estimation scale.

`transform`: logical; if `TRUE`, optimization is performed on the estimation scale, as defined by the user-supplied parameter transformations (see `pomp`). This can be used, for example, to enforce positivity or interval constraints on model parameters. See the tutorials on the package website for examples.

`cooling.type`, `cooling.fraction`: specifications for the cooling schedule, i.e., the manner in which the intensity of the parameter perturbations is reduced with successive filtering iterations. `cooling.type` specifies the nature of the cooling schedule. See below (under “Specifying the perturbations”) for more detail.

`tol`, `max.fail`: passed to the particle filter. See the descriptions under `pfilter`.
verbose logical; if TRUE, print progress reports.

... additional arguments that override the defaults.

pars names of parameters.

Value

Upon successful completion, mif2 returns an object of class mif2d.pomp. This class inherits from the pfilterd.pomp and pomp classes.

Specifying the perturbations: the rw.sd function

This function simply returns a list containing its arguments as unevaluated expressions. These are then evaluated in a context containing the model time variable. This allows for easy specification of the structure of the perturbations that are to be applied. For example,

```r
rw.sd(a=0.05,
    b=ifelse(0.2, time==time[1], 0),
    c=ivp(0.2),
    d=ifelse(time==time[13], 0.2, 0),
    e=ivp(0.2, lag=13),
    f=ifelse(time<23, 0.02, 0))
```

results in perturbations of parameter a with s.d. 0.05 at every time step, while parameters b and c both get perturbations of s.d. 0.2 only before the first observation. Parameters d and e, by contrast, get perturbations of s.d. 0.2 only before the thirteenth observation. Finally, parameter f gets a random perturbation of size 0.02 before every observation falling before \( t = 23 \).

On the \( m \)-th IF2 iteration, prior to time-point \( n \), the \( d \)-th parameter is given a random increment normally distributed with mean 0 and standard deviation \( c_{m,n} \sigma_{d,n} \), where \( c \) is the cooling schedule and \( \sigma \) is specified using rw.sd, as described above. Let \( N \) be the length of the time series and \( \alpha = \text{cooling.fraction}.50 \). Then, when \( \text{cooling.type} = \text{"geometric"} \), we have

\[
c_{m,n} = \alpha^{\frac{n-1+(m-1)N}{50N}}.
\]

When \( \text{cooling.type} = \text{"hyperbolic"} \), we have

\[
c_{m,n} = \frac{s + 1}{s + n + (m - 1)N},
\]

where \( s \) satisfies

\[
\frac{s + 1}{s + 50N} = \alpha.
\]

Thus, in either case, the perturbations at the end of 50 IF2 iterations are a fraction \( \alpha \) smaller than they are at first.

Re-running mif2 Iterations

To re-run a sequence of mif2 iterations, one can use the mif2 method on a mif2d.pomp object. By default, the same parameters used for the original mif2 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 \texttt{mif2} Iterations

One can resume a series of \texttt{mif2} iterations from where one left off using the \texttt{continue} method. A call to \texttt{mif2} to perform \textit{Nmif} \textit{m} iterations followed by a call to \texttt{continue} to perform \textit{Nmif} \textit{n} iterations will produce precisely the same effect as a single call to \texttt{mif2} to perform \textit{Nmif} \textit{m+n} iterations. By default, all the algorithmic parameters are the same as used in the original call to \texttt{mif2}. Additional arguments will override these defaults.

Methods

Methods that can be used to manipulate, display, or extract information from a \texttt{mif2d.pomp} object:

- \texttt{conv.rec} \texttt{conv.rec(object, pars = NULL)} returns the columns of the convergence-record matrix corresponding to the names in \texttt{pars}. By default, all rows are returned.
- \texttt{logLik} Returns the value in the \texttt{loglik} slot. NB: this is \textit{not} the same as the likelihood of the model at the MLE!
- \texttt{c} Concatenates \texttt{mif2d.pomp} objects into a \texttt{mif2List}.
- \texttt{plot} Plots a series of diagnostic plots when applied to a \texttt{mif2d.pomp} or \texttt{mif2List} object.

Author(s)

Aaron A. King, Edward L. Ionides, and Dao Nguyen

References


See Also

\texttt{pomp}, \texttt{pfilter}, \texttt{mif}, and the IF2 tutorial on the package website.

Examples

```r
## Not run:
pompExample(ou2)

guess1 <- guess2 <- coef(ou2)
guess1[c('x1.0','x2.0','alpha.2','alpha.3')] <- 0.5*guess1[c('x1.0','x2.0','alpha.2','alpha.3')]
guess2[c('x1.0','x2.0','alpha.2','alpha.3')] <- 1.5*guess1[c('x1.0','x2.0','alpha.2','alpha.3')]

m1 <- mif2(ou2,Nmif=100,start=guess1,Np=1000,
cooling.type="hyperbolic",cooling.fraction.50=0.05,
rw.sd=rw.sd(x1.0=ivp(0.5),x2.0=ivp(0.5),
alpha.2=0.1,alpha.3=0.1))

m2 <- mif2(ou2,Nmif=100,start=guess2,Np=1000,
cooling.type="hyperbolic",cooling.fraction.50=0.05,
rw.sd=rw.sd(x1.0=ivp(0.5),x2.0=ivp(0.5),
alpha.2=0.1,alpha.3=0.1))
```
The log-mean-exp trick

Description

`logmeanexp` computes

\[ \log \left( \frac{1}{N} \sum_{n=1}^{N} e^{x_i} \right), \]

avoiding over- and under-flow in doing so. It can optionally return an estimate of the standard error in this quantity.

Usage

`logmeanexp(x, se = FALSE)`

Arguments

- `x` numeric
- `se` logical; give approximate standard error?

Details

When `se = TRUE`, `logmeanexp` uses a jackknife estimate of the variance in `log(x)`.

Value

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

Author(s)

Aaron A. King
Examples

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

Description

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

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

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

This page documents these elements.

Usage

```r
## S4 method for signature 'pomp'
## S4 method for signature 'pomp'
## S4 method for signature 'pomp'
## S4 method for signature 'pomp'
```
Arguments

- **object**: an object of class pomp.
- **xstart**: an nvar x nrep matrix containing the starting state of the system. Columns of xstart correspond to states; rows to components of the state vector. One independent simulation will be performed for each column. Note that in this case, params must also have nrep columns.
- **x**: a rank-3 array containing states of the unobserved process. The dimensions of x are nvars x nrep x ntimes, where nvars is the number of state variables, nrep is the number of replicates, and ntimes is the length of times.
- **y**: a matrix containing observations. The dimensions of y are nobs x ntimes, where nobs is the number of observables and ntimes is the length of times.
- **times, t**: a numeric vector (length ntimes) containing times. These must be in non-decreasing order.
- **params**: a npar x nrep matrix of parameters. Each column is an independent parameter set and is paired with the corresponding column of x or xstart. In the case of init.state, params is a named vector of parameters.
- **offset**: integer; the first offset times in times will not be returned.
- **t0**: the initial time at which initial states are requested.
- **nsim**: optional integer; the number of initial states to simulate. By default, this is equal to the number of columns of params.
- **log**: if TRUE, log probabilities are returned.
- **as.data.frame**: logical; if TRUE, return the result as a data-frame.
- **...**: In trajectory, additional arguments are passed to the ODE integrator (if the skeleton is a vectorfield) and ignored if it is a map. See ode for a description of the additional arguments accepted. In all other cases, additional arguments are ignored.

rprocess

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

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

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

\[
\text{dim}(x) = c(nvars, nrep, ntimes-offset),
\]

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

**dprocess**

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

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

**rmeasure**

`rmeasure` simulate the measurement model given states and parameters.

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

**dmeasure**

`dmeasure` evaluates the probability density of observations given states.

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

**dprior, rprior**

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

**init.state**

`init.state` returns an `nvar` x `nsim` matrix of state-process initial conditions when given an `npar` x `nsim` matrix of parameters, `params`, and an initial time `t0`. By default, `t0` is the initial time defined when the pomp object `ws` was constructed. If `nsim` is not specified, then `nsim=ncol(params)`.

**skeleton**

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

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

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

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

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

Parameter transformations

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

pompLoad, pompUnload

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

Author(s)

Aaron A. King

See Also

pomp, pomp methods

Examples

pompExample(ricker)

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

## short arguments are recycled:
  p <- c(r=42,phi=10,sigma=0.3,N.0=7,e.0=0)
  t <- c(1:10,20,30)
MCMC proposal distributions

MCMC proposal distributions

Description

Functions to construct proposal distributions for use with MCMC methods.

Usage

\[
\begin{align*}
\text{mvn.rw}(\text{rw.var}) \\
\text{mvn.diag.rw}(\text{rw.sd}) \\
\text{mvn.rw.adaptive}(\text{rw.sd}, \text{rw.var}, \text{scale.start} = \text{NA}, \text{scale.cooling} = 0.999, \\
\quad \text{shape.start} = \text{NA}, \text{target} = 0.234, \text{max.scaling} = 50)
\end{align*}
\]

Arguments

- **rw.var**: square numeric matrix with row- and column-names. Specifies the variance-covariance matrix for a multivariate normal random-walk proposal distribution.
- **rw.sd**: named numeric vector; random-walk SDs for a multivariate normal random-walk proposal with diagonal variance-covariance matrix.
- **scale.start**, **scale.cooling**, **shape.start**, **target**, **max.scaling**: parameters to control the proposal adaptation algorithm. Beginning with MCMC iteration **scale.start**, the scale of the proposal covariance matrix will be adjusted in an effort to match the target acceptance ratio. This initial scale adjustment is “cooled”, i.e., the adjustment diminishes as the chain moves along. The parameter **scale.cooling** specifies the cooling schedule: at \(n\) iterations after **scale.start**, the current scaling factor is multiplied with **scale.cooling**\(^n\). The maximum scaling factor allowed at any one iteration is **max.scaling**. After **shape.start** accepted proposals have accumulated, a scaled empirical covariance matrix will be used for the proposals, following Roberts and Rosenthal (2009).

Value

Each of these calls constructs a function suitable for use as the proposal argument of **pmcmc** or **abc**. Given a parameter vector, each such function returns a single draw from the corresponding proposal distribution.
Nonlinear forecasting

Author(s)
Aaron A. King, Sebastian Funk

References

See Also
pmcmc, abc

Description
nlf calls an optimizer to maximize the nonlinear forecasting (NLF) goodness of fit. The latter is computed by simulating data from a model, fitting a nonlinear autoregressive model to the simulated time series, and quantifying the ability of the resulting fitted model to predict the data time series. NLF is an ‘indirect inference’ method using a quasi-likelihood as the objective function.

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

Arguments

object
A pomp object, with the data and model to fit to it.

start
Named numeric vector with guessed parameters.

est
Vector containing the names or indices of parameters to be estimated.

lags
A vector specifying the lags to use when constructing the nonlinear autoregressive prediction model. The first lag is the prediction interval.
period numeric; period=NA means the model is nonseasonal. period>0 is the period of seasonal forcing in 'real time'.
tensor logical; if FALSE, the fitted model is a generalized additive model with time mod period as one of the predictors, i.e., a gam with time-varying intercept. If TRUE, the fitted model is a gam with lagged state variables as predictors and time-periodic coefficients, constructed using tensor products of basis functions of state variables with basis functions of time.
nconverge number of convergence timesteps to be discarded from the model simulation.
nasymp number of asymptotic timesteps to be recorded from the model simulation.
seed integer specifying the random number seed to use. When fitting, it is usually best to always run the simulations with the same sequence of random numbers, which is accomplished by setting seed to an integer. If you want a truly random simulation, set seed=NULL.
transform logical; if TRUE, parameters are optimized on the transformed scale.
transform.data optional function. If specified, forecasting is performed using data and model simulations transformed by this function. By default, transform.data is the identity function, i.e., no transformation is performed. The main purpose of transform.data is to achieve approximately multivariate normal forecasting errors. If data are univariate, transform.data should take a scalar and return a scalar. If data are multivariate, transform.data should assume a vector input and return a vector of the same length.
nrbf integer scalar; 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.
bootsamp vector of integers; used to have the quasi-loglikelihood evaluated using a bootstrap re-sampling of the data set.
lql.frac target fractional change in log quasi-likelihood for quadratic standard error estimate
se.par.frac initial parameter-change fraction for quadratic standard error estimate
eval.only logical; if TRUE, no optimization is attempted and the quasi-loglikelihood value is evaluated at the start parameters.
... Arguments that will be passed to optim or subplex in the control list.

Details
This runs an optimizer to maximize nlf.objfun.

Value
An object of class nlfd.pomp. logLik applied to such an object returns the log quasi likelihood. The $ method allows extraction of arbitrary slots from the nlfd.pomp object.
Author(s)
Stephen P. Ellner, Bruce E. Kendall, Aaron A. King

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

ou2  
Two-dimensional discrete-time Ornstein-Uhlenbeck process

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

Details
If the state process is $X(t) = (x_1(t), x_2(t))$, then
$$X(t+1) = \alpha X(t) + \sigma \epsilon(t),$$
where $\alpha$ and $\sigma$ are 2x2 matrices, $\sigma$ 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.

See Also
pomp

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

Create a matrix of parameters

Description

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

Usage

parmat(params, nrep = 1)

Arguments

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

Value

parmat returns a matrix consisting of nrep copies of params.

Author(s)

Aaron A. King

Examples

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

Particle filter

Description

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

Usage

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

Arguments

- **object**: An object of class `pomp` or inheriting class `pomp`.
- **params**: optional named numeric vector containing the parameters at which the filtering should be performed. By default, `params = coef(object)`.
- **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))` or as a function taking a positive integer argument. In the latter case, `Np(k)` must be a single positive integer, representing the number of particles to be used at the k-th timestep: `Np(0)` is the number of particles to use going from `timezero(object)` to `time(object)[1]`, `Np(1)`, from `timezero(object)` to `time(object)[1]`, and so on, while when `T=length(time(object), t0=TRUE))`, `Np(T)` is the number of particles to sample at the end of the time-series. When `object` is of class `mif`, this is by default the same number of particles used in the `mif` iterations.
- **tol**: positive numeric scalar; particles with likelihood less than `tol` are considered to be incompatible with the data. See the section on *Filtering Failures* below for more information.
- **max.fail**: integer; the maximum number of filtering failures allowed (see below). If the number of filtering failures exceeds this number, execution will terminate with an error. By default, `max.fail` is set to infinity, so no error can be triggered.
pred.mean logical; if TRUE, the prediction means are calculated for the state variables and parameters.

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

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

filter.traj logical; if TRUE, a filtered trajectory is returned 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.

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

pars Names of parameters.

vars Names of state variables.

... additional arguments that override the defaults.

Value

An object of class pfilterd.pomp. This class inherits from class pomp. The following additional slots can be accessed via the $ operator:

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 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.

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

Methods

logLik Extracts the estimated log likelihood.

cond.logLik Extracts the estimated conditional log likelihood

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

where $y_t$ are the data, at time $t$.

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

$$\left(\sum_i w_{it}^2\right)^{-1},$$

where $w_{it}$ is the normalized weight of particle $i$ at time $t$. 
Particle filter

**pred.mean, pred.var** Extract the mean and variance of the approximate prediction distribution. This prediction distribution is that of

$$X_t | y_1, \ldots, y_{t-1},$$

where $X_t, y_t$ are the state vector and data, respectively, at time $t$.

**filter.mean** Extract the mean of the filtering distribution, which is that of

$$X_t | y_1, \ldots, y_t,$$

where $X_t, y_t$ are the state vector and data, respectively, at time $t$.

**Filtering failures**

If the degree of disagreement between model and data becomes sufficiently large, a “filtering failure” results. A filtering failure occurs when, at some time point, none of the $N_p$ particles is compatible with the data. In particular, if the conditional likelihood of a particle at any time is below the tolerance value $tol$, then that particle is considered to be uninformative and its likelihood is taken to be zero. A filtering failure occurs when this is the case for all particles. A warning is generated when this occurs unless the cumulative number of failures exceeds $max.Nfail$, in which case an error is generated.

**Author(s)**

Aaron A. King

**References**


**See Also**

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

**Examples**

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

The Particle MCMC algorithm for estimating the parameters of a partially-observed Markov process. Running `pmcmc` causes a particle random-walk Metropolis-Hastings Markov chain algorithm to run for the specified number of proposals.

Usage

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

Arguments

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

Value

An object of class `pmcmc`. 

Re-running PMCMC Iterations

To re-run a sequence of PMCMC iterations, one can use the \texttt{pmcmc} method on a \texttt{pmcmc} object. By default, the same parameters used for the original PMCMC run are re-used (except for \texttt{tol}, \texttt{max.fail}, and \texttt{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 \texttt{continue} method. A call to \texttt{pmcmc} to perform \texttt{Nmc}=m iterations followed by a call to \texttt{continue} to perform \texttt{Nmc}=n iterations will produce precisely the same effect as a single call to \texttt{pmcmc} to perform \texttt{Nmc}=m+n iterations. By default, all the algorithmic parameters are the same as used in the original call to \texttt{pmcmc}. Additional arguments will override the defaults.

Details

\texttt{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).

Note that \texttt{pmcmc} does not make use of any parameter transformations supplied by the user.

Methods

\texttt{c} Concatenates \texttt{pmcmc} objects into a \texttt{pmcmclist}.

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

\texttt{filter.traj(object, vars)} returns filter trajectories from a \texttt{pmcmc} or \texttt{pmcmclist} object.

\texttt{plot} Diagnostic plots.

\texttt{logLik} Returns the value in the \texttt{loglik} slot.

\texttt{coef} Returns the last state of the MCMC chain. As such, it’s not very useful for inference.

\texttt{covmat(object, start, thin, expand)} computes the empirical covariance matrix of the MCMC samples beginning with iteration \texttt{start} and thinning by factor \texttt{thin}. It expands this by a factor \texttt{expand^2/n}, where \texttt{n} is the number of parameters estimated. By default, \texttt{expand}=2.38. The intention is that the resulting matrix is a suitable input to the proposal function \texttt{mvn.rw}.

Author(s)

Edward L. Ionides, Aaron A. King, Sebastian Funk

References


See Also

pomp, pfilter, MCMC proposal distributions, and the tutorials on the package website.

Examples

```r
## Not run:
library(pomp)
pompExample(ou2)

pmcmc(
pomp(ou2, dprior=Csnippet("
  lik = dnorm(alpha_2,-0.5,1,1) + dnorm(alpha_3,0.3,1,1);
  lik = (give_log) ? lik : exp(lik);
  paramnames=c("alpha.2","alpha.3")),
  Nmc=2000, Np=500, verbose=TRUE,
  proposal=mvn.rw.adaptive(rw.sd=c(alpha.2=0.01, alpha.3=0.01),
  scale.start=200, shape.start=100)) -> chain
continue(chain, Nmc=2000, proposal=mvn.rw(covmat(chain))) -> chain
plot(chain)
chain <- pmcmc(chain)
plot(chain)

library(coda)
trace <- window(conv.rec(chain,c("alpha.2","alpha.3")), start=2000)
rejectionRate(trace)
effectiveSize(trace)
autocorr.diag(trace)

summary(trace)
plot(trace)

heidel.diag(trace)
geweke.diag(trace)

## End(Not run)
```

**Description**

This function constructs a pomp object, encoding a partially-observed Markov process model together with a uni- or multi-variate time series. As such, it is central to all the package’s functionality. One implements the model by specifying some or all of its basic components. These include:

- **rprocess**, the simulator of the unobserved Markov state process;
- **dprocess**, the evaluator of the probability density function for transitions of the unobserved Markov state process;
**rmeasure**, the simulator of the observed process, conditional on the unobserved state;

**dmeasure**, the evaluator of the measurement model probability density function;

**initializer**, which samples from the distribution of the state process at the zero-time;

**rprocess** which samples from a prior probability distribution on the parameters;

**dprocess** which evaluates the prior probability density function;

**skeleton** which computes the deterministic skeleton of the unobserved state process.

The basic structure and its rationale are described in the *Journal of Statistical Software* paper, an updated version of which is to be found on the package website.

### Usage

```r
pomp(data, times, t0, ..., rprocess, dprocess, rmeasure, dmeasure, measurement.model, skeleton, skeleton.type, skelmap.delta.t, initializer, rprior, dprior, params, covar, tcovar, obsnames, statenames, paramnames, covarnames, zeronames, PACKAGE, fromEstimationScale, toEstimationScale, globals, cdir, cfile, shlib.args)
```

### Arguments

data, times required; the time series data and times at which observations are made. data should be given as a data-frame and times must indicate the column of observation times by name or index. times must be numeric and strictly increasing. Internally, data will be internally coerced to an array with storage-mode double.

In addition, a pomp object can be supplied in the data argument. In this case, the call to pomp will add element to, or replace elements of, the supplied pomp object.

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

rprocess, dprocess optional; specification of the simulator and probability density evaluation function of the unobserved state process. See below under “The Unobserved Markov State-Process Model” for details.

**Note:** it is not typically necessary (or even feasible) to define dprocess. In fact, no current pomp inference algorithm makes use of dprocess. This functionality is provided only to support future algorithm development.

rmeasure, dmeasure, measurement.model optional; specifications of the measurement model. See below under “The Measurement Model” for details.

skeleton optional: the deterministic skeleton of the unobserved state process. See below under “The Deterministic Skeleton” for details.

skeleton.type, skelmap.delta.t deprecated. These will be removed in a future release.
initializer optional; draws from the distribution of initial values of the unobserved Markov state process. Specifically, given a vector of parameters, params and an initial time, \( t_0 \), the initializer determines the state vector at time \( t_0 \). See below under “The State-Process Initializer” for details.

rprior, dprior optional; specification of the prior distribution on parameters. See below under “Specifying a Prior” for details.

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

covar, tcovar optional data frame of covariates: covar is the table of covariates (one column per variable); tcovar the name or the index of the time variable. If a covariate table is supplied, then the value of each of the covariates is interpolated as needed. The resulting interpolated values are made available to the appropriate basic components. Note that covar will be coerced internally to storage mode double. See below under “Covariates” for more details.

obsnames, statenames, paramnames, covarnames optional character vectors specifying the names of observables, state variables, parameters, and covariates, respectively. These are used only in the event that one or more of the basic components are defined using C snippets or native routines. It is usually unnecessary to specify obsnames or covarnames, as these will by default be read from data and covars, respectively.

zeronames optional character vector specifying the names of accumulator variables (see below under “Accumulator Variables”).

PACKAGE optional string giving the name of the dynamically loaded library in which any native routines are to be found. This is only useful if one or more of the model components has been specified using a precompiled dynamically loaded library; it is not useful if the components are specified using C snippets.

fromEstimationScale, toEstimationScale optional parameter transformations. Many algorithms for parameter estimation search an unconstrained space of parameters. When working with such an algorithm and a model for which the parameters are constrained, it can be useful to transform parameters. toEstimationScale and fromEstimationScale are transformations from the model scale to the estimation scale, and vice versa, respectively. See below under “Parameter Transformations” for more details.

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

cdir, cfile, shlib.args optional character variables. cdir specifies the name of the directory within which C snippet code will be compiled. By default, this is in a temporary directory specific to the running instance of R. cfile gives the name of the file (in directory cdir) into which C snippet codes will be written. By default, a random filename is used. The shlib.args can be used to pass command-line arguments to the R CMD SHLIB call that will compile the C snippets.

Any additional arguments given to pomp will be made available to each of the basic components. To prevent errors due to misspellings, a warning is issued if any such arguments are detected.
The pomp constructor function returns an object, call it \( P \), of class pomp. \( P \) contains, in addition to the data, any elements of the model that have been specified as arguments to the pomp constructor function. One can add or modify elements of \( P \) by means of further calls to pomp, using \( P \) as the first argument in such calls.

**Important note**

It is not typically necessary (or even feasible) to define all of the basic components for any given purpose. Each pomp algorithm makes use of only a subset of these components. Any algorithm requiring a component that is not present will generate an error letting you know that you have not provided a needed component.

**Using C snippets to accelerate computations**

The pomp provides a facility whereby users can define their model’s components using inline C code. Furnishing one or more C snippets as arguments to the pomp constructor causes them to be written to a C file stored in the \( R \) session’s temporary directory, which is then compiled (via \texttt{R CMD SHLIB}) into a dynamically loadable shared object file. This is then loaded as needed.

**Note to Windows and Mac users:** By default, your \( R \) installation may not support \texttt{R CMD SHLIB}. The package website contains installation instructions that explain how to enable this powerful feature of \( R \).

**General rules for writing C snippets**

In writing a C snippet one must bear in mind both the goal of the snippet, i.e., what computation it is intended to perform, and the context in which it will be executed. These are explained here in the form of general rules. Additional specific rules apply according to the function of the particular C snippet. Illustrative examples are given in the tutorials on the package website.

1. C snippets must be valid C. They will embedded verbatim in a template file which will then be compiled by a call to \texttt{R CMD SHLIB}. If the resulting file does not compile, an error message will be generated. No attempt is made by pomp to interpret this message. Typically, compilation errors are due to either invalid C syntax or undeclared variables.

2. State variables, parameters, observables, and covariates must be left undeclared within the snippet. State variables and parameters are declared via the \texttt{statenames} or \texttt{paramnames} arguments to pomp, respectively. Compiler errors that complain about undeclared state variables or parameters are usually due to failure to declare these in \texttt{statenames} or \texttt{paramnames}, as appropriate.

3. A C snippet can declare local variables. Be careful not to use names that match those of state variables, observables, or parameters. The latter must never be declared within a C snippet.

4. Names of observables are determined by their names in the data. They must be referred to in measurement model C snippets (\texttt{rmeasure} and \texttt{dmeasure}) by those names.

5. If the pomp object contains a table of covariates (see above), then the variables in the covariate table will be available, by their names, in the context within which the C snippet is executed.

6. Because the dot \( . \) has syntactic meaning in C, \( R \) variables with names containing dots (\( . \)) are replaced in the C codes by variable names in which all dots have been replaced by underscores (\( _- \)).
7. The header ‘R.h’, provided with R, will be included in the generated C file, making all of the R C API available for use in the C snippet. This makes a great many useful functions available, including all of R’s statistical distribution functions.

8. The header ‘pomp.h’, provided with pomp, will also be included, making all of the pomp C API available for use in every C snippet. Do

```c
file.show(system.file("include/pomp.h",package="pomp"))
```
to view this header file.

9. Snippets of C code passed to the globals argument of pomp will be included at the head of the generated C file. This can be used to declare global variables, define useful functions, and include arbitrary header files.

The Unobserved Markov State-Process Model

Specification of process-model codes rprocess and/or dprocess is facilitated by pomp’s so-called plug-ins, which allow one to easily specify the most common kinds of process model.

**Discrete-time processes:** If the state process evolves in discrete time, specify rprocess using the discrete.time.sim plug-in. Specifically, provide

```c
rprocess = discrete.time.sim(step.fun = f, delta.t)
```
to pomp, where f is a C snippet or R function that takes simulates one step of the state process. The former is the preferred option, due to its much greater computational efficiency. The goal of such a C snippet is to replace the state variables with their new random values at the end of the time interval. Accordingly, each state variable should be over-written with its new value. In addition to the states, parameters, covariates (if any), and observables, the variables t and dt, containing respectively the time at the beginning of the step and the step’s duration, will be defined in the context in which the C snippet is executed. See below under “General rules for C snippet writing” for more details. Examples are to be found in the tutorials on the package website.

If f is given as an R function, it should have prototype

```c
f(x, t, params, delta.t, ...)
```
When f is called, x will be a named numeric vector containing the value of the state process at time t, params will be a named numeric vector containing parameters, and delta.t will be the time-step. It should return a named vector of the same length, and with the same set of names, as x, representing a draw from the distribution of the state process at time t+delta.t, conditional on its having value x at time t.

**Continuous-time processes:** If the state process evolves in continuous time, but you can use an Euler approximation, specify rprocess using the euler.sim plug-in. Furnish

```c
rprocess = euler.sim(step.fun = f, delta.t)
```
to pomp in this case. As before, f can be provided either as a C snippet or as an R function, the former resulting in much quicker computations. The form of f will be the same as above (in the discrete-time case).

If you have a procedure that allows you, given the value of the state process at any time, to simulate it at an arbitrary time in the future, use the onestep.sim plug-in. To do so, furnish

```c
rprocess = onestep.sim(step.fun = f)
```
to pomp. Again, \( f \) can be provided either as a C snippet or as an R function, the former resulting in much quicker computations. The form of \( f \) should be as above (in the discrete-time or Euler cases).

If you desire exact simulation of certain continuous-time Markov chains, an implementation of Gillespie’s algorithm (Gillespie 1977) is available, via the gillespie.sim plug-in. In this case, furnish

\[
\text{rprocess} = \text{gillespie.sim}(\text{rate.fun} = f, v, d)
\]

to pomp, where \( f \) gives the rates of the elementary events. Here, \( f \) must be an R function of the form

\[
f(j, x, t, \text{params}, \ldots)
\]

When \( f \) is called, the integer \( j \) will be the number of the elementary event (corresponding to the columns of matrices \( v \) and \( d \), see below), \( x \) will be a named numeric vector containing the value of the state process at time \( t \) and \( \text{params} \) is a named numeric vector containing parameters. \( f \) should return a single numerical value, representing the rate of that elementary event at that point in state space and time.

Matrices \( v \) and \( d \) specify the continuous-time Markov process in terms of its elementary events. Each should have dimensions \( nvar \times nevent \), 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.

A faster, but approximate, version of the Gillespie algorithm, the so-called “K-leap” method of Cai and Xu (2007), is implemented in the kleap.sim plug-in. To use it, supply

\[
\text{rprocess} = \text{kleap.sim}(\text{rate.fun}, e, v, d)
\]

to pomp, where \( \text{rate.fun} \), \( v \), and \( d \) are as above, and \( e \) gives relative error tolerances for each of the state variables. \( e \) should have length equal to the number of state variables. \( e[i] \) corresponds to row \( i \) of the \( v \) and \( d \) matrices and we must have \( 0 \leq e[i] \leq 1 \). The leap size, \( K \), is chosen so that \( K \leq \max(\min(e[i]x[i]), 1) \).

Size of time step: The simulator plug-ins discrete.time.sim, euler.sim, and onestep.sim all work by taking discrete time steps. They differ as to how this is done. Specifically,

1. onestep.sim takes a single step to go from any given time \( t1 \) to any later time \( t2 \) (\( t1 < t2 \)).

   Thus, this plug-in is designed for use in situations where a closed-form solution to the process exists.

2. To go from \( t1 \) to \( t2 \), euler.sim takes \( n \) steps of equal size, where

   \[
n = \text{ceiling}((t2-t1)/\delta t).
   \]

3. discrete.time.sim assumes that the process evolves in discrete time, where the interval between successive times is \( \delta t \). Thus, to go from \( t1 \) to \( t2 \), discrete.time.sim takes \( n \) steps of size exactly \( \delta t \), where

   \[
n = \text{floor}((t2-t1)/\delta t).
   \]

Specifying dprocess: If you have a procedure that allows you to compute the probability density of an arbitrary transition from state \( x_1 \) at time \( t_1 \) to state \( x_2 \) at time \( t_2 > t_1 \), assuming that the state remains unchanged between \( t_1 \) and \( t_2 \), then you can use the onestep.dens plug-in. This is accomplished by furnishing
dprocess = onestep.dens(dens.fun = f)

to pomp, where \( f \) is an \( R \) function with prototype

\[
f(x_1, x_2, t_1, t_2, \text{params}, \ldots)
\]

When \( f \) is called, \( x_1 \) and \( x_2 \) will be named numeric vectors containing the values of the state process at times \( t_1 \) and \( t_2 \), respectively, and \( \text{params} \) will be a named numeric vector containing parameters. \( f \) should return the log likelihood of a transition from \( x_1 \) at time \( t_1 \) to \( x_2 \) at time \( t_2 \), assuming that no intervening transitions have occurred.

To see examples, consult the tutorials on the [package website](https://www.example.com/pomp).}

### The Measurement Model

The measurement model is the link between the data and the unobserved state process. It can be specified either by using one or both of the \( \text{rprocess} \) and \( \text{dprocess} \) arguments, or via the \( \text{measurement.model} \) argument. If \( \text{measurement.model} \) is given it overrides any specification via the \( \text{rmeasure} \) or \( \text{dmeasure} \) arguments, with a warning.

The best way to specify the measurement model is by giving \( C \) snippets for \( \text{rmeasure} \) and \( \text{dmeasure} \). In writing an \( \text{rmeasure} \) \( C \) snippet, bear in mind that:

1. The goal of such a snippet is to fill the observables with random values drawn from the measurement model distribution. Accordingly, each observable should be assigned a new value.
2. In addition to the states, parameters, covariates (if any), and observables, the variable \( t \), containing the time of the observation, will be defined in the context in which the snippet is executed.

General rules for writing \( C \) snippets are provided below. The tutorials on the [package website](https://www.example.com/pomp) give examples as well.

It is also possible, though far less efficient, to specify \( \text{rmeasure} \) using an \( R \) function. In this case, specify the measurement model simulator by furnishing

\[
\text{rmeasure} = f
\]

to pomp, where \( f \) is an \( R \) function with prototype

\[
f(x, t, \text{params}, \ldots)
\]

It can also take any additional arguments if these are passed along with it in the call to pomp. When \( f \) is called,

- \( x \) will be a named numeric vector of length \( \text{nvar} \), the number of state variables.
- \( t \) will be a scalar quantity, the time at which the measurement is made.
- \( \text{params} \) will be a named numeric vector of length \( \text{npar} \), the number of parameters.

\( f \) must return a named numeric vector of length \( \text{lobs} \), the number of observable variables.

In writing a \( \text{dmeasure} \) \( C \) snippet, observe that:
1. In addition to the states, parameters, covariates (if any), and observables, the variable \( t \), containing the time of the observation, and the Boolean variable `give_log` will be defined in the context in which the snippet is executed.

2. The goal of such a snippet is to set the value of the `lik` variable to the likelihood of the data given the state. Alternatively, if `give_log == 1`, `lik` should be set to the log likelihood.

If `dmeasure` is to be provided instead as an R function, this is accomplished by supplying

```r
dmeasure = f
```

to `pomp`, where `f` has prototype

```r
f(y, x, t, params, log, ...)
```

Again, it can take additional arguments that are passed with it in the call to `pomp`. When `f` is called,

- \( y \) will be a named numeric vector of length `nobs` containing values of the observed variables;
- \( x \) will be a named numeric vector of length `nvar` containing state variables;
- `params` will be a named numeric vector of length `npar` containing parameters;
- \( t \) will be a scalar, the corresponding observation time.

`f` must return a single numeric value, the probability density of \( y \) given \( x \) at time \( t \). If `log == TRUE`, then `f` should return instead the log of the probability density. **Note: it is a common error to fail to account for both `log == TRUE` and `log == FALSE` when writing the `dmeasure` C snippet or function.**

One can also specify both the `rmeasure` and `dmeasure` components at once via the `measurementNmodel` argument. It should be a formula or list of `nobs` formulae. These are parsed internally to generate `rmeasure` and `dmeasure` functions. **Note: this is a convenience function, primarily designed to facilitate model exploration; it will typically be possible (and as a practical matter necessary) to accelerate measurement model computations by writing `dmeasure` and/or `rmeasure` using C snippets.**

**The Deterministic Skeleton**

The skeleton is a dynamical system that expresses the central tendency of the unobserved Markov state process. As such, it is not uniquely defined, but can be both interesting in itself and useful in practice. In `pomp`, the skeleton is used by `trajectory` and `traj.match`.

If the state process is a discrete-time stochastic process, then the skeleton is a discrete-time map. To specify it, provide

```r
skeleton = map(f, delta.t)
```

to `pomp`, where `f` implements the map and `delta.t` is the size of the timestep covered at one map iteration.

If the state process is a continuous-time stochastic process, then the skeleton is a vectorfield (i.e., a system of ordinary differential equations). To specify it, supply

```r
skeleton = vectorfield(f)
```
to pomp, where \( f \) implements the vectorfield, i.e., the right-hand-size of the differential equations. In either case, \( f \) can be furnished either as a C snippet (the preferred choice), or an \( R \) function. In writing a skeleton C snippet, be aware that:

1. For each state variable, there is a corresponding component of the deterministic skeleton. The goal of such a snippet is to compute all the components.
2. When the skeleton is a map, the component corresponding to state variable \( x \) is named \( \Delta x \) and is the new value of \( x \) after one iteration of the map.
3. When the skeleton is a vectorfield, the component corresponding to state variable \( x \) is named \( \Delta x \) and is the value of \( \frac{dx}{dt} \).
4. As with the other C snippets, all states, parameters and covariates, as well as the current time, \( t \), will be defined in the context within which the snippet is executed.

The tutorials on the package website give some examples.

If \( f \) is an \( R \) function, it must be of prototype

\[
f(x, t, \text{params}, \ldots)
\]

where, as usual,

- \( x \) is a numeric vector (length \( nvar \)) containing the coordinates of a point in state space at which evaluation of the skeleton is desired.
- \( t \) is a scalar value giving the time at which evaluation of the skeleton is desired.
- \( \text{params} \) is a numeric vector (length \( npar \)) holding the parameters.

As with the other basic components, \( f \) may take additional arguments, provided these are passed along with it in the call to pomp. The function \( f \) must return a numeric vector of the same length as \( x \), which contains the value of the map or vectorfield at the required point and time.

The State-Process Initializer

To fully specify the unobserved Markov state process, one must give its distribution at the zero-time (\( t_0 \)). By default, pomp assumes that this initial distribution is concentrated on a single point. In particular, any parameters in \( \text{params} \), the names of which end in “\( .NP \)”, are assumed to be initial values of states. When the state process is initialized, these are simply copied over as initial conditions. The names of the resulting state variables are obtained by dropping the “\( .0 \)” suffix.

One can override this default behavior by furnishing a value for the \texttt{initializer} argument of pomp. As usual, this can be provided either as a C snippet or as an \texttt{R} function. In the former case, bear in mind that:

1. The goal of a this snippet is the construction of a state vector, i.e., the setting of the dynamical states at time \( t_0 \).
2. In addition to the parameters and covariates (if any), the variable \( t \), containing the zero-time, will be defined in the context in which the snippet is executed.
3. \textbf{NB:} The \texttt{statenames} argument plays a particularly important role when the initializer is specified using a C snippet. In particular, every state variable must be named in \texttt{statenames}. \textbf{Failure to follow this rule will result in undefined behavior.}
If an R function is to be used, pass

\[ \text{initializer} = f \]

to \text{pomp}, where \( f \) is a function with prototype

\[ f(\text{params}, t0, \ldots) \]

When \( f \) is called,

- \text{params} will be a named numeric vector of parameters.
- \( t0 \) will be the time at which initial conditions are desired.

As usual, \( f \) may take additional arguments, provided these are passed along with it in the call to \text{pomp}. \( f \) must return a named numeric vector of initial states. It is of course important that the names of the states match the expectations of the other basic components.

Note that the state-process initializer can be either deterministic (the default) or stochastic. In the latter case, it samples from the distribution of the state process at the zero-time, \( t0 \).

**Specifying a Prior**

A prior distribution on parameters is specified by means of the \text{rprior} and/or \text{dprior} arguments to \text{pomp}. As with the other basic model components, it is preferable to specify these using C snippets. In writing a C snippet for the prior sampler (\text{rprior}), keep in mind that:

1. Within the context in which the snippet will be evaluated, only the parameters will be defined.
2. The goal of such a snippet is the replacement of parameters with values drawn from the prior distribution.
3. Hyperparameters can be included in the ordinary parameter list. Obviously, hyperparameters should not be replaced with random draws.

In writing a C snippet for the prior density function (\text{dprior}), observe that:

1. Within the context in which the snippet will be evaluated, only the parameters and \text{give\_log} will be defined.
2. The goal of such a snippet is computation of the prior probability density, or the log of same, at a given point in parameter space. This scalar value should be returned in the variable lik. When \text{give\_log} == 1, lik should contain the log of the prior probability density.
3. Hyperparameters can be included in the ordinary parameter list.

Alternatively, one can furnish R functions for one or both of these arguments. In this case, \text{rprior} must be a function of prototype

\[ f(\text{params}, \ldots) \]

that makes a draw from the prior distribution given \text{params} and returns a named vector of the same length and with the same set of names, as \text{params}. The \text{dprior} function must be of prototype

\[ f(\text{params}, \text{log} = \text{FALSE}, \ldots). \]

Its role is to evaluate the prior probability density (or log density if \text{log} == \text{TRUE}) and return that single scalar value.
Covariates

If the pomp object contains covariates (specified via the covar argument; see above), then interpolated values of the covariates will be available to each of the model components whenever it is called. In particular, variables with names as they appear in the covar data frame will be available to any C snippet. When a basic component is defined using an R function, that function will be called with an extra argument, covars, which will be a named numeric vector containing the interpolated values from the covariate table.

An exception to this rule is the prior (rprior and dprior): covariate-dependent priors are not allowed. Nor are parameter transformations permitted to depend upon covariates.

Parameter Transformations

When parameter transformations are desired, they can be integrated into the pomp object via the toEstimationScale and fromEstimationScale arguments. As with the basic model components, these should ordinarily be specified using C snippets. When doing so, note that:

1. The parameter transformation mapping a parameter vector from the scale used by the model codes to another scale is specified using the toEstimationScale argument whilst the transformation mapping a parameter vector from the alternative scale to that on which the model is defined is specified with the fromEstimationScale argument.

2. The goal of these snippets is the computation of the values of the transformed parameters. The value of transformed parameter $p$ should be assigned to variable $tp$.

3. Time-, state-, and covariate-dependent transformations are not allowed. Therefore, neither the time, nor any state variables, nor any of the covariates will be available in the context within which a parameter transformation snippet is executed.

These transformations can also be specified using R functions with arguments params and .... In this case, toEstimationScale should transform parameters from the scale that the basic components use internally to the scale used in estimation. fromEstimationScale should be the inverse of toEstimationScale.

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

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

By default, both functions are the identity transformation.

Accumulator Variables

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

It can be useful to view the C code generated by calling pomp with one or more C snippet arguments. You can set cdir and cfile to control where this code is written. Alternatively, set options(verbose=TRUE) before calling pomp. This will cause a message giving the name of the generated C file (in the session temporary directory) to be printed.

Warning

Some error checking is done by pomp, but complete error checking for arbitrary models is impossible. If the user-specified functions do not conform to the above specifications, then the results may be invalid. In particular, if both rmeasure and dmeasure are specified, the user should verify that these two functions correspond to the same probability distribution. If skeleton is specified, the user is responsible for verifying that it corresponds to a deterministic skeleton of the model.

Author(s)

Aaron A. King

References


See Also

pomp methods, pomp low-level interface

Examples

```R
## pomp encoding a stochastic Ricker model with a covariate:

pomp(data = data.frame(t = 1:100, y = NA),
    times = "t", t0 = 0,
    covar = data.frame(t=0:100,K=seq(from=50,to=200,length=100)),
    tcovar = "t",
    rprocess = discrete.time.sim(Csnippet("double e = rnorm(0,sigma);
                                         n = r*n*exp(1-n/K+e);"), delta.t = 1),
    rmeasure = Csnippet("y = rpois(n);"),
    dmeasure = Csnippet("lik = dpois(y,n,give_log);"),
    rprior = Csnippet("r = rgamma(1,1); sigma = rgamma(1,1);"),
    dprior = Csnippet("lik = dgamma(r,1,1,1) + dgamma(sigma,1,1,1);
                          if (!give_log) lik = exp(lik);"),
    initializer = Csnippet("n = n_0;"),
    toEstimationScale = Csnippet("Tr = log(r); Tsigma = log(sigma);"),
```
pomp methods

```r
fromEstimationScale = Csnippet("Tr = exp(r); Tsigma = exp(sigma);"),
paramnames = c("n_0", "r", "sigma"),
statenames = "n") -> rick

## fill it with simulated data:

rick <- simulate(rick, params = c(r=17, sigma = 0.1, n_0 = 50))
plot(rick)

## Not run:
  pompExample()
  demos(package="pomp")

## End(Not run)
```

---

### pomp methods

**Functions for manipulating, displaying, and extracting information from objects of the pomp class**

---

**Description**

This page documents the various methods that allow one to extract information from, display, plot, and modify pomp objects.

**Usage**

```r
## 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'
ob(object, vars, ...)
## S4 method for signature 'pomp'
partrans(object, params, dir = c("fromEstimationScale",
"toEstimationScale","forward","inverse"), ...)
## S4 method for signature 'pomp'
plot(x, y, variables, panel = lines,
  nc = NULL, yax.flip = FALSE,
  mar = c(0, 5.1, 0, if (yax.flip) 5.1 else 2.1),
  oma = c(6, 0, 5, 0), axes = TRUE, ...)
## S4 method for signature 'pomp'
print(x, ...)
## S4 method for signature 'pomp'
show(object)
## S4 method for signature 'pomp'
states(object, vars, ...)
## S4 method for signature 'pomp'
time(x, t0 = FALSE, ...)
```
## S4 replacement method for signature 'pomp'

time(object, t0 = FALSE, ...) <- value

## S4 method for signature 'pomp'

timezero(object, ...)

## S4 replacement method for signature 'pomp'

timezero(object, ...) <- value

## S4 method for signature 'pomp'

window(x, start, end, ...)

## S4 method for signature 'pomp'

as(object, class)

### Arguments

- **object, x**: The pomp object.
- **pars**: optional character; names of parameters to be retrieved or set.
- **vars**: optional character; names of observed variables to be retrieved.
- **transform**: optional logical; should the parameter transformations be applied?
- **value**: numeric; values to be assigned.
- **params**: a vector or matrix of parameters to be transformed.
- **dir**: direction of the transformation. *dir*="forward" applies the transformation from the “natural” scale to the “internal” scale. This is the transformation specified by the parameter.transform argument to pomp; it is stored in the ‘par.trans’ slot of object. *dir*="inverse" applies the inverse transformation (stored in the ‘par.untrans’ slot).
- **t0**: logical; if TRUE on a call to time, the zero time is prepended to the time vector; if TRUE on a call to time<-, the first element in value is taken to be the initial time.
- **start, end**: start and end times of the window.
- **class**: character; name of the class to which object should be coerced.
- **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.
- **...**: Further arguments (either ignored or passed to underlying functions).
Details

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

\[
\text{coef(object,transform=TRUE)}
\]
returns

parameter.inv.transform(coef(object)),

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

\[
\text{coef(object,pars,transform=TRUE)}
\]
returns

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

coefficients/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.

\[
\text{coef(object,transform=TRUE) <- value assigns parameter.transform(value)}
\]
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.

observations/obs/obs(object) returns the array of observations. obs(object,vars) gives just the observations of variables named in vars. vars may specify the variables by position or by name.

states/states/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.

observation times/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.

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

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

window/window/apply(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.
print  Print method.
plot  Plots the data and state trajectories (if the latter exist). Additional arguments are passed to the low-level plotting routine.
as  A pomp object can be coerced to a data frame via
    as(object,"data.frame").
The data frame contains the times, the data, and the state trajectories, if they exist.

Author(s)
    Aaron A. King

See Also
    pomp, pomp low-level interface, simulate, pfilter, probe.

Description
    simulate generates simulations of the state and measurement processes.

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

Arguments
    object  An object of class pomp.
    nsim    The number of simulations to perform. Note that the number of replicates will be nsim times ncol(params).
    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, \( t_0 \) specifies the times at which simulated observations will be made. \( t_0 \) specifies the start time (the time at which the initial conditions hold). The default for times is \( \text{times} = \text{time(object, } t_0 = \text{FALSE}) \) and \( t_0 = \text{timezero(object)} \), respectively.

as.data.frame, include.data

logical: if \( \text{as.data.frame}=\text{TRUE} \), the results are returned as a data-frame. A factor variable, ‘sim’, distinguishes one simulation from another. If, in addition, \( \text{include.data}=\text{TRUE} \), the original data are included as an additional ‘simulation’. If \( \text{as.data.frame}=\text{FALSE} \), \( \text{include.data} \) is ignored.

... further arguments that are currently ignored.

Details

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

Value

If \( \text{states}=\text{FALSE} \) and \( \text{obs}=\text{FALSE} \) (the default), a list of \( \text{nsim} \) \texttt{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 \( \text{times} \).

If \( \text{nsim}=1 \), then a single \texttt{pomp} object is returned (and not a singleton list).

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

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

If both \( \text{states}=\text{TRUE} \) and \( \text{obs}=\text{TRUE} \), then a named list is returned. It contains the state trajectories and simulated observations as above.

Author(s)

Aaron A. King

See Also

\texttt{pomp}

Examples

\begin{verbatim}
pompExample(ou2)
x <- simulate(ou2, seed=3495485, nsim=10)
x <- simulate(ou2, seed=3495485, nsim=10, states=TRUE, obs=TRUE)
x <- simulate(ou2, seed=3495485, nsim=10, obs=TRUE, as.data.frame=TRUE, include.data=TRUE)
\end{verbatim}
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

```r
## 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.
- **transform**: function; this transformation will be applied to the observables prior to estimation of the spectrum, and prior to any detrending.
Power spectrum computation and matching

**detrend**
- de-trending operation to perform. Options include no detrending, and subtraction of constant, linear, and quadratic trends from the data. Detrending is applied to each data series and to each model simulation independently.

**weights**
- optional. The mismatch between model and data is measured by a weighted average of mismatch at each frequency. By default, all frequencies are weighted equally. weights can be specified either as a vector (which must have length equal to the number of frequencies) or as a function of frequency. If the latter, weights(freq) must return a nonnegative weight for each frequency.

**start**
- named numeric vector; the initial guess of parameters.

**est**
- character vector; the names of parameters to be estimated.

**method**
- Optimization method. Choices are *subplex* and any of the methods used by *optim*.

**verbose**
- logical; print diagnostic messages?

**eval.only**
- logical; if TRUE, no optimization is attempted. Instead, the probe-mismatch value is simply evaluated at the start parameters.

**fail.value**
- optional scalar; if non-NA, this value is substituted for non-finite values of the objective function.

**...**
- Additional arguments. In the case of *spect*, these are currently ignored. In the case of *spect.match*, these are passed to *optim* or *subplex* in the control list.

**Details**

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

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

**Value**

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

- **kernel.width** width parameter of the smoothing kernel used.
- **transform** transformation function used.
- **freq** numeric vector of the frequencies at which the power spectrum is estimated.
- **datspec, simspec** estimated power spectra for data and simulations, respectively.
- **pvals** one-sided p-values: fraction of the simulated spectra that differ more from the mean simulated spectrum than does the data. The metric used is $L^2$ distance.
- **detrend** detrending option used.

*spect.match* returns an object of class *spect.matched.pomp*, which is derived from class *spect.pomp* and therefore has all the slots of that class. In addition, *spect.matched.pomp* objects have the following slots:
est, weights, fail.value values of the corresponding arguments in the call to spect.match.
evals number of function and gradient evaluations by the optimizer. See optim.
value Value of the objective function.
convergence, msg Convergence code and message from the optimizer. See optim.

Author(s)
Daniel C. Reuman, Cai GoGwilt, Aaron A. King

References


See Also
pomp, probe

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

ou2.bad <- ou2
ccoef(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)
Probe functions

Some useful probes for partially-observed Markov processes

Description

Several simple and configurable probes are provided with in the package. These can be used directly and as templates for 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.
a.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 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

diff

order of polynomial regression.
order of differencing to perform.

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 $y_t = \sum \beta_k y_{t-k} \tau_k + \epsilon_t$ will be fit, where $\tau_k$ are the lags and $p_k$ are the powers. The data are first centered. This function returns the regression coefficients, $\beta_k$.

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


Probes and synthetic likelihood

See Also

pomp

---

**Probes and synthetic likelihood**

*Probe a partially-observed Markov process by computing summary statistics and the synthetic likelihood.*

---

**Description**

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

**Usage**

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

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

Details

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

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

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

Value

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

**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 datvals.

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

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

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

**value** value of the objective function at the optimum.

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

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

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

**Methods**

- **plot** displays diagnostic plots.

- **summary** displays summary information.

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

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

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

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

**Author(s)**

Daniel C. Reuman, Aaron A. King

**References**


**See Also**

- `pomp`, `probe functions`, `spect`, and the tutorials on the package website.
Examples

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

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

--

**ricker**

*Ricker model with Poisson observations.*

Description

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

Details

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

See Also

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

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

**rw2**

*Two-dimensional random-walk process*

**Description**

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

**Details**

The random-walk process is fully but noisily observed.

**See Also**

`pomp`, `ou`

**Examples**

pompExample(rw2)
plot(rw2)

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

**Simulated annealing**

*Simulated annealing with box constraints.*

**Description**

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

**Usage**

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

**Arguments**

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

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

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

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

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

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

- **temp**  Starting temperature for the cooling schedule. Defaults to 1.

- **tmax**  Number of function evaluations at each temperature. Defaults to 10.

- **candidate.dist**  Function to randomly select a new candidate parameter vector. This should be a function with three arguments, the first being the current parameter vector, the second the temperature, and the third the parameter scaling. By default, \( \text{candidate.dist} = \text{function(\text{par}, \text{temp}, \text{scale}) \text{rnorm(n=length(\text{par}),mean=\text{par},sd=\text{scale*temp})}} \).

- **sched**  Cooling schedule. A function of a three arguments giving the temperature as a function of iteration number and the control parameters temp and tmax. By default, \( \text{sched} = \text{function(k, temp, tmax) temp/log(((k-1)*tmax)/tmax+exp(1))} \).

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

- **lower**, **upper**  Optional numeric vectors. These describe the lower and upper box constraints, respectively. Each can be specified either as a single scalar (common to all parameters) or as a vector of the same length as par. By default, lower=-Inf and upper=Inf, i.e., there are no constraints.

Value

sannbox returns a list with components:

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

- **convergence**  Provided for compatibility with optim; will always be 0.

- **final.params**  Last tried value of par.

- **final.value**  Value of fn corresponding to final.params.

- **par**  Best tried value of par.

- **value**  Value of fn corresponding to par.

Author(s)

Daniel Reuman, Aaron A. King
sir

See Also

traj.match, probe.match.

Description

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

Details

This and similar examples are discussed and constructed in tutorials available on the package website.

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

References


See Also

pomp and the tutorials on the package website.

Examples

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

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

pompExample(bbs)
plot(bbs)
coef(bbs)
Trajectory matching

Parameter estimation by fitting the trajectory of a model’s deterministic skeleton to data

Description

This function attempts to match trajectories of a model’s deterministic skeleton to data. Trajectory matching is equivalent to maximum likelihood estimation under the assumption that process noise is entirely absent, i.e., that all stochasticity is measurement error. Accordingly, this method uses only the skeleton and dmeasure components of a POMP model.

Usage

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

Arguments

- **object** — A `pomp` object. If object has no skeleton slot, an error will be generated.
- **start** — named numeric vector containing an initial guess for parameters. By default `start=coef(object)` if the latter exists.
- **params** — optional named numeric vector of parameters. This should contain all parameters needed by the skeleton and dmeasure slots of object. In particular, any parameters that are to be treated as fixed should be present here. Parameter values given in `params` for parameters named in `est` will be ignored. By default, `params=coef(object)` if the latter exists.
- **est** — character vector containing the names of parameters to be estimated. In the case of `traj.match.objfun`, the objective function that is constructed will assume that its argument contains the parameters in this order.
- **method** — Optimization method. Choices are `subplex`, "sannbox", and any of the methods used by `optim`.
- **transform** — logical; if TRUE, optimization is performed on the transformed scale.
- **...** — Extra arguments that will be passed either to the optimizer (`optim`, `subplex`, `nloptr`, or `sannbox`, via their control (`optim`, `subplex`, `sannbox`) or `opts` (`nloptr`) lists) or to the ODE integrator. In `traj.match`, extra arguments will be passed to the optimizer. In `traj.match.objfun`, extra arguments are passed to `trajectory`. If extra arguments are needed by both optimizer and `trajectory`, construct an objective function first using `traj.match.objfun`, then give this objective function to the optimizer.
Details

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

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

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

Value

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

- **transform, est** the values of these arguments on the call to traj.match.
- **evals** number of function and gradient evaluations by the optimizer. See optim.
- **value** value of the objective function. Larger values indicate better fit (i.e., traj.match attempts to maximize this quantity).
- **convergence, msg** convergence code and message from the optimizer. See optim.

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

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

See Also

trajectory, pomp, optim, subplex

Examples

pompExample(ou2)
true.p <- c(
  alpha.1=0.9, alpha.2=0, alpha.3=-0.4, alpha.4=0.99,
  sigma.1=2, sigma.2=0.1, sigma.3=2,
  tau=1,
  x1.0=50, x2.0=-50
)
simdata <- simulate(ou2, nsim=1, params=true.p, seed=43553)
guess.p <- true.p
Utilities for reproducibility

Tools for reproducible computations.

Description

On cooking shows, recipes requiring lengthy baking or stewing are prepared beforehand. The bake and stew functions perform analogously: an R computation is performed and stored in a named file. If the function is called again and the file is present, the computation is not executed; rather, the results are loaded from the file in which they were previously stored. Moreover, via their optional seed argument, bake and stew can control the pseudorandom-number generator (RNG) for greater reproducibility. After the computation is finished, these functions restore the pre-existing RNG state to avoid side effects.

The freeze function doesn’t save results, but does set the RNG state to the specified value and restore it after the computation is complete.

Usage

bake(file, expr, seed, kind = NULL, normal.kind = NULL)
stew(file, expr, seed, kind = NULL, normal.kind = NULL)
freeze(expr, seed, kind = NULL, normal.kind = NULL)

```r
code
```
Arguments

file
Name of the binary data file in which the result will be stored or retrieved, as appropriate. For bake, this will contain a single \texttt{R} object and hence be an RDS file (extension ‘rds’); for stew, this will contain one or more named \texttt{R} objects and hence be an RDA file (extension ‘rda’).

expr
Expression to be evaluated.

seed, kind, normal.kind
Optional. To set the state and, optionally, kind of RNG used. See \texttt{set.seed}.

Details

Both \texttt{bake} and \texttt{stew} first test to see whether \texttt{file} exists. If it does, \texttt{bake} reads it using \texttt{readRDS} and returns the resulting object. By contrast, \texttt{stew} loads the file using \texttt{load} and copies the objects it contains into the user’s workspace (or the environment of the call to \texttt{stew}).

If \texttt{file} does not exist, then both \texttt{bake} and \texttt{stew} evaluate the expression \texttt{expr}; they differ in the results that they save. \texttt{bake} saves the value of the evaluated expression to \texttt{file} as a single \texttt{R} object. The name of that object is not saved. By contrast, \texttt{stew} creates a local environment within which \texttt{expr} is evaluated; all objects in that environment are saved (by name) in \texttt{file}.

Value

\texttt{bake} returns the value of the evaluated expression \texttt{expr}. Other objects created in the evaluation of \texttt{expr} are discarded along with the temporary, local environment created for the evaluation.

The latter behavior differs from that of \texttt{stew}, which returns the names of the objects created during the evaluation of \texttt{expr}. After \texttt{stew} completes, these objects exist in the parent environment (that from which \texttt{stew} was called).

\texttt{freeze} returns the value of evaluated expression \texttt{expr}. However, \texttt{freeze} evaluates \texttt{expr} within the parent environment, so other objects created in the evaluation of \texttt{expr} will therefore exist after \texttt{freeze} completes.

All these functions return information about the time used in evaluating the expression. This is recorded in the \texttt{system.time} attribute of the return value. In addition, if \texttt{seed} is specified, information about the seed (and the kind of random-number generator used) are stored as attributes of the return value.

Author(s)

Aaron A. King

Examples

```r
## Not run:
bake(file="example1.rds",{
  x <- runif(1000)
  mean(x)
})

stew(file="example2.rda",{
  x <- runif(10)
}
```
Utilities for reproducibility

```
y <- rnorm(n=10, mean=3*x+5, sd=2)
```

```
plot(x,y)
```

````
## End(Not run)
```

```
freeze(runif(3), seed=5886730)
freeze(runif(3), seed=5886730)
```
Index

*Topic **datasets**
- blowflies, 10
- Childhood disease incidence data, 11
- dacca, 12
- Example pomp models, 19
- gompertz, 20
- ou2, 36
- pomp-package, 3
- ricker, 68
- rw2, 69
- sir, 71

*Topic **design**
- design, 13

*Topic **distribution**
- eulermultinom, 17

*Topic **interface**
- pomp constructor, 43

*Topic **internals**
- Ensemble Kalman filters, 15

*Topic **models**
- blowflies, 10
- dacca, 12
- Example pomp models, 19
- gompertz, 20
- ou2, 36
- pomp-package, 43
- POMP simulation, 58
- pomp-package, 3
- ricker, 68
- rw2, 69
- sir, 71

*Topic **optimize**
- Iterated filtering, 21
- Iterated filtering 2, 24
- Nonlinear forecasting, 34
- Power spectrum computation and matching, 60
- Probes and synthetic likelihood, 65

- Simulated annealing, 69
- Trajectory matching, 72

*Topic **programming**
- Low-level-interface, 29
- MCMC proposal distributions, 33
- pomp constructor, 43
- pomp methods, 55

*Topic **smooth**
- B-splines, 7

*Topic **ts**
- Approximate Bayesian computation, 5
- Bayesian sequential Monte Carlo, 8
- Ensemble Kalman filters, 15
- Iterated filtering, 21
- Iterated filtering 2, 24
- Nonlinear forecasting, 34
- Particle filter, 37
- Particle Markov Chain Monte Carlo, 41
- pomp constructor, 43
- pomp methods, 55
- POMP simulation, 58
- pomp-package, 3
- Power spectrum computation and matching, 60
- Probe functions, 63
- Probes and synthetic likelihood, 65
- Trajectory matching, 72

[,abcList-method (Approximate Bayesian computation), 5
[,mif2List-method (Iterated filtering 2), 24
[,mifList-method (Iterated filtering), 21
[,pcmcmList-method (Particle Markov Chain Monte Carlo), 41
[-abcList (Approximate Bayesian computation), 5
[-mif2List (Iterated filtering 2), 24

77
[-miflist (Iterated filtering), 21
[-pmcmcList (Particle Markov Chain
Monte Carlo), 41
$.bsmcd.pomp-method (Bayesian
sequential Monte Carlo), 8
$.kalmand.pomp-method (Ensemble Kalman
filters), 15
$.nlfd.pomp-method (Nonlinear
forecasting), 34
$.pfilterd.pomp-method (Particle
filter), 37
$.probe.matched.pomp-method (Probes
and synthetic likelihood), 65
$.probed.pomp-method (Probes and
synthetic likelihood), 65
$.traj.matched.pomp-method (Trajectory
matching), 72
$-bsmcd.pomp (Bayesian sequential
Monte Carlo), 8
$-kalmand.pomp (Ensemble Kalman
filters), 15
$-nlfd.pomp (Nonlinear forecasting), 34
$-pfilterd.pomp (Particle filter), 37
$.probe.matched.pomp (Probes and
synthetic likelihood), 65
$.probed.pomp (Probes and synthetic
likelihood), 65
$.traj.matched.pomp (Trajectory
matching), 72

ABC (Approximate Bayesian computation),
5
abc, 3, 34
abc (Approximate Bayesian computation),
5
abc,abc-method (Approximate Bayesian
computation), 5
abc,ANY-method (Approximate Bayesian
computation), 5
abc,missing-method (Approximate
Bayesian computation), 5
abc,pomp-method (Approximate Bayesian
computation), 5
abc,probed.pomp-method (Approximate
Bayesian computation), 5
abc-abc (Approximate Bayesian
computation), 5
abc-class (Approximate Bayesian
computation), 5
abc-methods (Approximate Bayesian
computation), 5
abc-pomp (Approximate Bayesian
computation), 5
abc-probed.pomp (Approximate Bayesian
computation), 5
abcList-class (Approximate Bayesian
computation), 5
accumulator variables (pomp
constructor), 43
Approximate Bayesian computation, 5
as,kalmand.pomp-method (Ensemble
Kalman filters), 15
as,pfilterd.pomp-method (Particle
filter), 37
as,pomp-method (pomp methods), 55
as,probed.pomp-method (Probes and
synthetic likelihood), 65
as.data.frame.kalmand.pomp (Ensemble
Kalman filters), 15
as.data.frame.pfilterd.pomp (Particle
filter), 37
as.data.frame.pomp (pomp methods), 55

B-splines, 7
bake (Utilities for reproducibility), 74
basic.probes (Probe functions), 63
Bayesian sequential Monte Carlo, 8
bbs, 20
bbs (sir), 71
blowflies, 10, 20
blowflies1 (blowflies), 10
blowflies2 (blowflies), 10
bsmc (Bayesian sequential Monte Carlo),
8
bsmc,ANY-method (Bayesian sequential
Monte Carlo), 8
bsmc,missing-method (Bayesian
sequential Monte Carlo), 8
bsmc,pomp-method (Bayesian sequential
Monte Carlo), 8
bsmc-pomp (Bayesian sequential Monte
Carlo), 8
bsmc2, 3, 5, 40
bsmc2 (Bayesian sequential Monte
Carlo), 8
bsmc2,ANY-method (Bayesian sequential
Monte Carlo), 8
bsmc2, missing-method (Bayesian sequential Monte Carlo), 8
bsmc2, pomp-method (Bayesian sequential Monte Carlo), 8
bsmc2-pomp (Bayesian sequential Monte Carlo), 8
bspline.basis (B-splines), 7
c, abc-method (Approximate Bayesian computation), 5
c, abcList-method (Approximate Bayesian computation), 5
c, mif-method (Iterated filtering), 21
c, mif2d.pomp-method (Iterated filtering 2), 24
c, mif2List-method (Iterated filtering 2), 24
coef, mifList-method (Iterated filtering), 21
c, pmcmc-method (Particle Markov Chain Monte Carlo), 41
c, pmcmcList-method (Particle Markov Chain Monte Carlo), 41
c-abc (Approximate Bayesian computation), 5
c-abcList (Approximate Bayesian computation), 5
c-mif (Iterated filtering), 21
c-mif2d.pomp (Iterated filtering 2), 24
c-mif2List (Iterated filtering 2), 24
c-mifList (Iterated filtering), 21
c-pmcmc (Particle Markov Chain Monte Carlo), 41
c-pmcmcList (Particle Markov Chain Monte Carlo), 41
Childhood disease incidence data, 11
coef, mif2List-method (Iterated filtering 2), 24
coef, mifList-method (Iterated filtering), 21
coef, pomp-method (pomp methods), 55
coeff-pomp (pomp methods), 55
coef.rec-mif2List (Iterated filtering 2), 24
coef.rec-mifList (Iterated filtering), 21
coeff<- (pomp methods), 55
coeff<- , pomp-method (pomp methods), 55
coeff<-`-pomp (pomp methods), 55
coeff, kalmand.pomp, data.frame-method (Ensemble Kalman filters), 15
coef, pfilterd.pomp, data.frame-method (Particle filter), 37
coef, pomp, data.frame-method (pomp methods), 55
coef, probed.pomp, data.frame-method (Probes and synthetic likelihood), 65
cond.logLik (Particle filter), 37
cond.logLik, kalmand.pomp-method (Ensemble Kalman filters), 15
cond.logLik, pfilterd.pomp-method (Particle filter), 37
cond.logLik, kalmand.pomp (Ensemble Kalman filters), 15
continue (Iterated filtering 2), 24
continue, abc-method (Approximate Bayesian computation), 5
continue, ANY-method (Iterated filtering 2), 24
continue, mif-method (Iterated filtering), 21
continue, mif2d.pomp-method (Iterated filtering 2), 24
continue, missing-method (Iterated filtering 2), 24
continue, pmcmc-method (Particle Markov Chain Monte Carlo), 41
continue-abc (Approximate Bayesian computation), 5
continue-mif (Iterated filtering), 21
continue-mif2d.pomp (Iterated filtering 2), 24
continue-pmcmc (Particle Markov Chain Monte Carlo), 41
conv.rec (Iterated filtering), 21
conv.rec, abc-method (Approximate Bayesian computation), 5
conv.rec, abcList-method (Approximate Bayesian computation), 5
conv.rec, mif-method (Iterated filtering), 21
conv.rec, mif2d.pomp-method (Iterated filtering 2), 24
conv.rec, mif2List-method (Iterated
filtering R, 24
conv.rec.mifList-method (Iterated filtering), 21
conv.rec.pmcmc-method (Particle Markov Chain Monte Carlo), 41
conv.rec.pmcmcList-method (Particle Markov Chain Monte Carlo), 41
covmat (Approximate Bayesian computation), 5
covmat,abc-method (Approximate Bayesian computation), 5
covmat,abcList-method (Approximate Bayesian computation), 5
covmat,pmcmc-method (Particle Markov Chain Monte Carlo), 41
covmat,pmcmcList-method (Particle Markov Chain Monte Carlo), 41
covmat-abc (Approximate Bayesian computation), 5
covmat-abcList (Approximate Bayesian computation), 5
covmat-pmcmc (Particle Markov Chain Monte Carlo), 41
covmat-pmcmcList (Particle Markov Chain Monte Carlo), 41
csnippet (pomp constructor), 43
csnippet-class (pomp constructor), 43
dacca, 12, 20
design, 13
desolve, 32
deuermultinom (euermultinom), 17
discrete.time.sim (pomp constructor), 43
dmeasure (Low-level-interface), 29
dmeasure, pomp-method (Low-level-interface), 29
dmeasure-pomp (Low-level-interface), 29
dprior (Low-level-interface), 29
dprior,pomp-method (Low-level-interface), 29
dprior-pomp (Low-level-interface), 29
dprocess (Low-level-interface), 29
dprocess,pomp-method (Low-level-interface), 29
dprocess-pomp (Low-level-interface), 29
eakf (Ensemble Kalman filters), 15
eakf,ANY-method (Ensemble Kalman filters), 15
eakf,missing-method (Ensemble Kalman filters), 15
eakf,pomp-method (Ensemble Kalman filters), 15
eakf-pomp (Ensemble Kalman filters), 15
eff.sample.size (Particle filter), 37
eff.sample.size,pfilterd.pomp-method (Particle filter), 37
eulermultinom, 17
eucitmeas (Childhood disease incidence data), 11
ewmeas (Childhood disease incidence data), 11
Example pomp models, 19
logLik, nlfd.pomp-method (Nonlinear forecasting), 34
logLik, pfilterd.pomp-method (Particle filter), 37
logLik, pmcmc-method (Particle Markov Chain Monte Carlo), 41
logLik, pmcmcList-method (Particle Markov Chain Monte Carlo), 41
logLik, probed.pomp-method (Probes and synthetic likelihood), 65
logLik, traj.matched.pomp-method (Trajectory matching), 72
logLik - kalmand.pomp (Ensemble Kalman filters), 15
logLik - nlfd.pomp (Nonlinear forecasting), 34
logLik - pfilterd.pomp (Particle filter), 37
logLik - pmcmc (Particle Markov Chain Monte Carlo), 41
logLik - pmcmcList (Particle Markov Chain Monte Carlo), 41
logLik - probed.pomp (Probes and synthetic likelihood), 65
logLik - traj.matched.pomp (Trajectory matching), 72
logmeanexp, 28
LondonYorke (Childhood disease incidence data), 11
Low-level-interface, 29
map (pomp constructor), 43
mcmc, 42
MCMC proposal distributions, 7, 33, 43
MCMC proposal functions, 5, 41
MCMC proposal functions (MCMC proposal distributions), 33
mcmc.list, 42
mean, 63
mif, 5, 25, 27, 40
mif (Iterated filtering), 21
mif, ANY-method (Iterated filtering), 21
mif, mif-method (Iterated filtering), 21
mif, missing-method (Iterated filtering), 21
mif, pfilterd.pomp-method (Iterated filtering), 21
mif, pomp-method (Iterated filtering), 21
mif-class (Iterated filtering), 21
mif-methods (Iterated filtering), 21
mif-mif (Iterated filtering), 21
mif-pfilterd.pomp (Iterated filtering), 21
mif-pomp (Iterated filtering), 21
mif2, 3, 21, 24
mif2 (Iterated filtering 2), 24
mif2, ANY-method (Iterated filtering 2), 24
mif2, mif2d.pomp-method (Iterated filtering 2), 24
mif2, missing-method (Iterated filtering 2), 24
mif2, pfilterd.pomp-method (Iterated filtering 2), 24
mif2, pomp-method (Iterated filtering 2), 24
mif2-mif2d.pomp (Iterated filtering 2), 24
mif2-pfilterd.pomp (Iterated filtering 2), 24
mif2-pomp (Iterated filtering 2), 24
mif2d.pomp-class (Iterated filtering 2), 24
mif2d.pomp-methods (Iterated filtering 2), 24
mif2List-class (Iterated filtering 2), 24
mifList-class (Iterated filtering), 21
mvn.diag.rw (MCMC proposal distributions), 33
mvn.rw, 6, 42
mvn.rw (MCMC proposal distributions), 33
nloptr, 66, 67, 72
Nonlinear forecasting, 34

obs, 64
obs (pomp methods), 55
obs, pomp-method (pomp methods), 55
obs-pomp (pomp methods), 55
ode, 30, 32
onestep.dens (pomp constructor), 43
onestep.sim (pomp constructor), 43
optim, 35, 61, 62, 66, 67, 70, 72, 73
ou2, 20, 36, 69

parmat, 37
Particle filter, 37
particle filter (Particle filter), 37
Particle Markov Chain Monte Carlo, 41
partrans, 32
partrans (pomp methods), 55
partrans, pomp-method (pomp methods), 55
partrans-pomp (pomp methods), 55
paste, 7
periodic.bspline.basis (B-splines), 7
pfilter, 3, 5, 10, 17, 22, 24, 25, 27, 43, 58
pfilter (Particle filter), 37
pfilter, ANY-method (Particle filter), 37
pfilter, missing-method (Particle filter), 37
pfilter, pfilterd.pomp-method (Particle filter), 37
pfilter, pomp-method (Particle filter), 37
pfilter-pfilterd.pomp (Particle filter), 37
pfilter-pomp (Particle filter), 37
pfilterd.pomp, 23, 26, 39
pfilterd.pomp (Particle filter), 37
pfilterd.pomp-class (Particle filter), 37
plot, abc-method (Approximate Bayesian computation), 5
plot, abcList-method (Approximate Bayesian computation), 5
plot, bsmcd.pomp-method (Bayesian sequential Monte Carlo), 8
plot, mif-method (Iterated filtering), 21
plot, mif2d.pomp-method (Iterated filtering 2), 24
INDEX

plot,mif2List-method (Iterated filtering), 24
plot,mifList-method (Iterated filtering), 21
plot,pmmc-method (Particle Markov Chain Monte Carlo), 41
plot,pmmcList-method (Particle Markov Chain Monte Carlo), 41
plot,pomp-method (pomp methods), 55
plot,probe.matched.pomp-method (Probes and synthetic likelihood), 65
plot,probed.pomp-method (Probes and synthetic likelihood), 65
plot,spect.matched.pomp-method (Probes and synthetic likelihood), 65
plot,spect.pomp-method (Probes and synthetic likelihood), 65
plot,abc (Approximate Bayesian computation), 5
plot,abcList (Approximate Bayesian computation), 5
plot-bsmc.pomp (Bayesian sequential Monte Carlo), 8
plot-mif (Iterated Bayesian computation), 21
plot-mif2.pomp (Iterated filtering), 24
plot-mif2List (Iterated filtering), 24
plot-mifList (Iterated filtering), 21
plot-pmmc (Particle Markov Chain Monte Carlo), 41
plot-pmmcList (Particle Markov Chain Monte Carlo), 41
plot-pomp (pomp methods), 55
plot-probe.matched.pomp (Probes and synthetic likelihood), 65
plot-probed.pomp (Probes and synthetic likelihood), 65
plot-spect.pomp (Probes and synthetic likelihood), 65
plug-ins (pomp constructor), 43
pmmc, 3, 5, 34, 40
pmmc (Particle Markov Chain Monte Carlo), 41
pmmc,ANY-method (Particle Markov Chain Monte Carlo), 41
pmmc,missing-method (Particle Markov Chain Monte Carlo), 41
pmmc,pfilterd.pomp-method (Particle Markov Chain Monte Carlo), 41
pmmc,pmc.mc-method (Particle Markov Chain Monte Carlo), 41
pmmc,pomp-method (Particle Markov Chain Monte Carlo), 41
pmmc-class (Particle Markov Chain Monte Carlo), 41
pmmc-methods (Particle Markov Chain Monte Carlo), 41
pmmc-pfilterd.pomp (Particle Markov Chain Monte Carlo), 41
pmmc-pmc.method (Particle Markov Chain Monte Carlo), 41
pmmc-pmc (Particle Markov Chain Monte Carlo), 41
pmmc-pmcList (Particle Markov Chain Monte Carlo), 41
pomp (pomp constructor), 43
pomp constructor, 43
pomp constructor function, 3
pomp low-level interface, 5, 54, 58
pomp low-level interface (Low-level-interface), 29
pomp methods, 32, 54, 55
pomp package (pomp-package), 3
POMP simulation, 58
pomp-class (pomp constructor), 43
pomp-methods (pomp methods), 55
pomp-package, 3
pompExample, 4
pompExample (Example pomp models), 19
pompLoad (Low-level-interface), 29
pompLoad,pomp-method (Low-level-interface), 29
pompUnload (Low-level-interface), 29
pompunload, 29
pompUnload (Low-level-interface), 29
pompUnload,pomp-method (Low-level-interface), 29
pompUnload= (Low-level-interface), 29
Power spectrum computation and matching, 60
Power spectrum computation and matching (Power spectrum computation and matching), 60
pred.mean (Particle filter), 37
pred.mean, kalmand.pomp-method
(Ensemble Kalman filters), 15
pred.mean, pfilterd.pomp-method
(Particle filter), 37
pred.mean - kalmand.pomp (Ensemble
Kalman filters), 15
pred.mean - pfilterd.pomp (Particle
filter), 37
pred.var (Particle filter), 37
pred.var, pfilterd.pomp-method
(Particle filter), 37
pred.var - pfilterd.pomp (Particle
filter), 37
print, pomp-method (pomp methods), 55
print - pomp (pomp methods), 55
probe, 3, 5–7, 58, 62, 64
probe (Probes and synthetic
likelihood), 65
 Probe functions, 63
probe functions, 66, 67
probe functions (Probe functions), 63
probe, ANY-method (Probes and synthetic
likelihood), 65
probe, missing-method (Probes and
synthetic likelihood), 65
probe, pomp-method (Probes and
synthetic likelihood), 65
probe, probed.pomp-method (Probes and
synthetic likelihood), 65
probe - pomp (Probes and synthetic
likelihood), 65
probe - probed.pomp (Probes and
synthetic likelihood), 65
probe.acf (Probe functions), 63
probe.ccf (Probe functions), 63
probe.marginal (Probe functions), 63
probe.match, 3, 64, 71
probe.match (Probes and synthetic
likelihood), 65
probe.match, ANY-method (Probes and
synthetic likelihood), 65
probe.match, missing-method (Probes and
synthetic likelihood), 65
probe.match, pomp-method (Probes and
synthetic likelihood), 65
probe.match, probed.matched.pomp-method
(Probes and synthetic
likelihood), 65
probe.match, probed.pomp-method (Probes and synthetic
likelihood), 65
probe.match - pomp (Probes and synthetic
likelihood), 65
probe.match - probed.pomp (Probes and synthetic
likelihood), 65
probe.match - probed.pomp (Probes and synthetic
likelihood), 65
probe.match - objfun (Probes and synthetic
likelihood), 65
probe.match - objfun, ANY-method (Probes and synthetic
likelihood), 65
probe.match - objfun, missing-method
(Probes and synthetic
likelihood), 65
probe.match - objfun, pomp-method (Probes and synthetic
likelihood), 65
probe.match - objfun, probed.pomp-method
(Probes and synthetic
likelihood), 65
probe.matched.pomp, 66
probe.matched.pomp-class (Probes and synthetic
likelihood), 65
probe.matched.pomp-methods (Probes and synthetic
likelihood), 65
probe.match, probed.pomp-method (Probes and synthetic
likelihood), 65
probe.match, probed.pomp-method (Probes and synthetic
likelihood), 65
probe.match, pomp-method (Probes and synthetic
likelihood), 65
probe.match, probed.pomp (Probes and synthetic
likelihood), 66
probe.matched.pomp, 66
profileDesign (design), 13
quantile, 63
readRDS, 75
INDEX

reulermultinom (eulermultinom), 17
rgammawn (eulermultinom), 17
ricker, 20, 68
rmeasure (Low-level-interface), 29
rmeasure, pomp-method (Low-level-interface), 29
rmeasure-pomp (Low-level-interface), 29
rprior, 9
rprior (Low-level-interface), 29
rprior, pomp-method (Low-level-interface), 29
rprocess (Low-level-interface), 29
rprocess, pomp-method (Low-level-interface), 29
rw.sd (Iterated filtering 2), 24
rw2, 20, 69

sannbox, 66, 72, 73
sannbox (Simulated annealing), 69
sequential Monte Carlo (Particle filter), 37
set.seed, 75
show, pomp-method (pomp methods), 55
show-pomp (pomp methods), 55
simulate, 3, 5, 58, 60, 66
simulate, ANY-method (POMP simulation), 58
simulate, missing-method (POMP simulation), 58
simulate, pomp-method (POMP simulation), 58
simulate-pomp, 66
simulate-pomp (POMP simulation), 58
Simulated annealing, 69
sir, 71
skeleton (Low-level-interface), 29
skeleton, pomp-method (Low-level-interface), 29
skeleton-pomp (Low-level-interface), 29
sliceDesign (design), 13
SMC (Particle filter), 37
sobol (design), 13
sobolDesign (design), 13
spect, 67
spect (Power spectrum computation and matching), 60
spect, ANY-method (Power spectrum computation and matching), 60
spect, missing-method (Power spectrum computation and matching), 60
spect, pomp-method (Power spectrum computation and matching), 60
spect, spect.pomp-method (Power spectrum computation and matching), 60
spect-pomp (Power spectrum computation and matching), 60
spect-spect.pomp (Power spectrum computation and matching), 60
spect.match, 3
spect.match (Power spectrum computation and matching), 60
spect.match, ANY-method (Power spectrum computation and matching), 60
spect.match, missing-method (Power spectrum computation and matching), 60
spect.match, pomp-method (Power spectrum computation and matching), 60
spect.match, spect.pomp-method (Power spectrum computation and matching), 60
spect.match, spect-spect.pomp (Power spectrum computation and matching), 60
spect.matched.pomp, 61
spect.matched.pomp-class (Power spectrum computation and matching), 60
spect.matched.pomp-methods (Probes and synthetic likelihood), 65
spect.pomp, 61
spect.pomp-class (Power spectrum computation and matching), 60
spect.pomp-methods (Probes and synthetic likelihood), 65
sprintf, 7
states (pomp methods), 55
states, pomp-method (pomp methods), 55
states-pomp (pomp methods), 55
stew (Utilities for reproducibility), 74
subplex, 35, 61, 66, 72, 73
index

summary.probe.matched.pomp-method (Probes and synthetic likelihood), 65
summary.probed.pomp-method (Probes and synthetic likelihood), 65
summary.spect.matched.pomp-method (Probes and synthetic likelihood), 65
summary.spect.pomp-method (Probes and synthetic likelihood), 65
summary.traj.matched.pomp-method (Trajectory matching), 72
summary.probe.matched.pomp (Probes and synthetic likelihood), 65
summary.probed.pomp (Probes and synthetic likelihood), 65
summary.spect.matched.pomp (Probes and synthetic likelihood), 65
summary.spect.pomp (Probes and synthetic likelihood), 65
summary.traj.matched.pomp (Trajectory matching), 72
summary.probed.pomp (Probes and synthetic likelihood), 65
summary.spect.matched.pomp (Probes and synthetic likelihood), 65
summary.spect.pomp (Probes and synthetic likelihood), 65
summary.traj.matched.pomp (Trajectory matching), 72

The pomp package (pomp-package), 3
time.pomp-method (pomp methods), 55
time<-.pomp-method (pomp methods), 55
timezero.pomp-method (pomp methods), 55
timezero.pomp (pomp methods), 55
timezero<-.pomp-method (pomp methods), 55
timezero<-.pomp (pomp methods), 55
traj.match, 3, 5, 50, 71
traj.match (Trajectory matching), 72
traj.match, ANY-method (Trajectory matching), 72
traj.match, missing-method (Trajectory matching), 72
traj.match, pomp-method (Trajectory matching), 72
traj.match, traj.matched.pomp-method (Trajectory matching), 72
traj.match-pomp (Trajectory matching), 72

traj.match-traj.matched.pomp (Trajectory matching), 72
traj.match.objfun (Trajectory matching), 72
traj.match.objfun, ANY-method (Trajectory matching), 72
traj.match.objfun, missing-method (Trajectory matching), 72
traj.match.objfun.pomp-method (Trajectory matching), 72
traj.match.objfun-pomp (Trajectory matching), 72
traj.match.objfun-class (Trajectory matching), 72

trajectory, 50, 72, 73
trajectory (Low-level-interface), 29
Trajectory matching, 72
trajectory, pomp-method (Low-level-interface), 29

Utilities for reproducibility, 74
values (Probes and synthetic likelihood), 65
values, probe.matched.pomp-method (Probes and synthetic likelihood), 65
values, probed.pomp-method (Probes and synthetic likelihood), 65
values, probe.matched.pomp (Probes and synthetic likelihood), 65
values, probed.pomp (Probes and synthetic likelihood), 65

vectorfield (pomp constructor), 43

window, pomp-method (pomp methods), 55
window-pomp (pomp methods), 55