Package ‘pomp’

July 5, 2020

Type Package
Title Statistical Inference for Partially Observed Markov Processes
Version 3.1
Date 2020-07-05
URL https://kingaa.github.io/pomp/
Description Tools for data analysis 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(>= 4.0.0), methods
Imports stats, graphics, digest, mvtnorm, deSolve, coda, reshape2, magrittr, plyr
Suggests ggplot2, knitr, tidyr, dplyr, subplex, nloptr
SystemRequirements For Windows users, Rtools (see https://cran.r-project.org/bin/windows/Rtools/).
License GPL-3
LazyData true
Contact kingaa at umich dot edu
BugReports https://github.com/kingaa/pomp/issues/
Encoding UTF-8
Collate 'pomp-package.R' 'aaa.R' 'pstop.R' 'undefined.R' 'csnippet.R'
    'pomp_fun.R' 'parameter_trans.R' 'covariate_table.R'
    'skeleton_spec.R' 'rrprocess_spec.R' 'safecall.R' 'pomp_class.R'
    'load.R' 'workhorses.R' 'continue.R' 'prior_spec.R'
    'dmeasure_spec.R' 'dprocess_spec.R' 'rmeasure_spec.R'
    'rinit_spec.R' 'templates.R' 'builder.R' 'pomp.R' 'probe.R'
    'abc.R' 'accumulators.R' 'kalman.R' 'pfilter.R' 'wpfilter.R'
    'proposals.R' 'pmcmc.R' 'mif2.R' 'listie.R' 'simulate.R'
    'spect.R' 'plot.R' 'bsmc2.R' 'as_data_frame.R' 'as_pomp.R'
topics documented:

'authors.R' 'bake.R' 'basic_probes.R' 'blowflies.R' 'bsflu.R'
'bsplines.R' 'coef.R' 'concat.R' 'cond_logLik.R' 'covmat.R'
'dacca.R' 'design.R' 'distributions.R' 'ebola.R'
'eff_sample_size.R' 'extract.R' 'filter_mean.R' 'filter_traj.R'
'flow.R' 'forecast.R' 'gompertz.R' 'probe_match.R'
'spect_match.R' 'summary.R' 'nlf.R' 'trajectory.R'
'traj_match.R' 'objfun.R' 'loglik.R' 'logmeanexp.R' 'lookup.R'
'measles.R' 'melt.R' 'obs.R' 'ou2.R' 'parmat.R' 'parus.R'
'pipe.R' 'pomp_examp.R' 'pred_mean.R' 'pred_var.R' 'show.R'
'print.R' 'profile_design.R' 'resample.R' 'ricker.R'
'runif_design.R' 'rw2.R' 'sannbox.R' 'saved_states.R' 'sir.R'
'slice_design.R' 'sobol.R' 'spy.R' 'states.R' 'time.R'
'timezero.R' 'traces.R' 'transformations.R' 'userdata.R'
'verhulst.R' 'window.R'

RoxygenNote 7.1.1

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],
Eamon O'Dea [ctb]

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

Repository CRAN

Date/Publication 2020-07-05 19:00:03 UTC

R topics documented:

pomp-package . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
abc . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
accumulators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
as.data.frame . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
bake . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
basic_probes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
blowflies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
bsflu . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
bsmc2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
bsplines . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
R topics documented:

coef ................................................................. 23
cond.logLik ....................................................... 24
continue .............................................................. 26
covariate_table .................................................. 26
covmat ............................................................... 27
Csnippet ............................................................. 28
dacca ................................................................. 30
design ................................................................. 32
distributions ....................................................... 34
dmeasure ............................................................ 36
dmeasure_spec ..................................................... 37
dprior ................................................................. 38
dprocess ............................................................. 39
dprocess_spec ...................................................... 40
ebola ................................................................. 41
eff.sample.size .................................................... 43
filter.mean ......................................................... 44
filter.traj .......................................................... 45
flow ................................................................. 46
forecast .............................................................. 47
gompertz ............................................................ 47
hitch ................................................................. 48
kalman ............................................................... 50
logLik ............................................................... 52
logmeanexp ......................................................... 53
measles .............................................................. 54
mif2 ................................................................. 55
nlf ................................................................. 60
obs ................................................................. 64
ou2 ................................................................. 65
parameter_trans .................................................. 66
parmat ............................................................... 68
partrans ............................................................ 69
parus ............................................................... 69
pfilter .............................................................. 70
plot ................................................................. 74
pmcmc .............................................................. 75
pomp ............................................................... 78
pomp_examples ................................................... 82
pred.mean ........................................................ 83
pred.var ........................................................... 84
print .............................................................. 85
prior_spec .......................................................... 85
probe .............................................................. 86
probe.match ....................................................... 89
proposals ........................................................ 93
ricker .............................................................. 94
rinit .............................................................. 95
The pomr 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 pomr 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

**pomp** 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 `kalman`
12. simple trajectory matching; see `traj.match`

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

**Author(s)**

Aaron A. King

**References**


See the package website, [https://kingaa.github.io/pomp/](https://kingaa.github.io/pomp/), for more references.

**See Also**

Other information on model implementation: `Csnippet`, `accumulators`, `covariate_table()`, `distributions`, `dmeasure_spec`, `dprocess_spec`, `parameter_trans()`, `prior_spec`, `rinit_spec`, `rmeasure_spec`, `rprocess_spec`, `skeleton_spec`, `transformations`, `userdata`

Other pomp parameter estimation methods: `abc()`, `bsmc2()`, `kalman`, `mif2()`, `nlf`, `pmcmc()`, `probe.match`, `spect.match`

Other elementary POMP methods: `pfilter()`, `probe()`, `simulate()`, `spect()`, `wpfilter()`
Description

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

Usage

```r
## S4 method for signature 'data.frame'
abc(
data,
Nabc = 1,
proposal,
scale,
epsilon,
probes,
params,
init,
process,
measure,
dprior,
...,
verbose = getOption("verbose", FALSE)
)

## S4 method for signature 'pomp'
abc(
data,
Nabc = 1,
proposal,
scale,
epsilon,
probes,
...,
verbose = getOption("verbose", FALSE)
)

## S4 method for signature 'probed_pomp'
abc(data, probes, ..., verbose = getOption("verbose", FALSE))

## S4 method for signature 'abcd_pomp'
abc(
data,
Nabc,
proposal,
...,
verbose = getOption("verbose", FALSE)
)
```
scale,
epsilon,
probes,
..., verbose = getOption("verbose", FALSE)
)

Arguments

**data**
either a data frame holding the time series data, or an object of class 'pomp', i.e., the output of another pomp calculation.

**Nabc**
the number of ABC iterations to perform.

**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 proposals for more information.

**scale**
named numeric vector of scales.

**epsilon**
ABC tolerance.

**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 probes are provided with the package: see basic probes.

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

**rinit**
simulator of the initial-state distribution. This can be furnished either as a C snippet, an R function, or the name of a pre-compiled native routine available in a dynamically loaded library. Setting rinit=NULL sets the initial-state simulator to its default. For more information, see ?rinit_spec.

**rprocess**
simulator of the latent state process, specified using one of the rprocess plugins. Setting rprocess=NULL removes the latent-state simulator. For more information, see ?rprocess_spec for the documentation on these plugins.

**rmeasure**
simulator of the measurement model, specified either as a C snippet, an R function, or the name of a pre-compiled native routine available in a dynamically loaded library. Setting rmeasure=NULL removes the measurement model simulator. For more information, see ?rmeasure_spec.

**dprior**
optional; prior distribution density evaluator, specified either as a C snippet, an R function, or the name of a pre-compiled native routine available in a dynamically loaded library. For more information, see ?prior_spec. Setting dprior=NULL resets the prior distribution to its default, which is a flat improper prior.

**...**
additional arguments supply new or modify existing model characteristics or components. See pomp for a full list of recognized arguments.

When named arguments not recognized by pomp are provided, these are made available to all basic components via the so-called userdata facility. This allows the user to pass information to the basic components outside of the usual routes of covariates (covar) and model parameters (params). See ?userdata for information on how to use this facility.

**verbose**
logical; if TRUE, diagnostic messages will be printed to the console.
Running ABC

abc returns an object of class ‘abcd_pomp’. One or more ‘abcd_pomp’ 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 ‘abcd_pomp’ object. By default, the same parameters used for the original ABC run are re-used (except for verbose, the default of which is 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

The following can be applied to the output of an abc operation:

- plot produces a series of diagnostic plots
- traces produces a mcmc object, to which the various coda convergence diagnostics can be applied

Author(s)

Edward L. Ionides, Aaron A. King

References


See Also

- MCMC proposals
- Other summary statistics methods: basic_probes, probe.match, probe(), spect()
- Other pomp parameter estimation methods: bsmc2(), kalman, mif2(), nlf, pmcmc(), pomp-package, probe.match, spect.match
Description

Accumulator variables

Details

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 accumvars argument will be set to zero immediately following each observation. See sir and the tutorials on the package website for examples.

See Also

sir

Other information on model implementation: Csnippet, covariate_table(), distributions, dmeasure_spec, dprocess_spec, parameter_trans(), pomp-package, prior_spec, rinit_spec, rmeasure_spec, rprocess_spec, skeleton_spec, transformations, userdata

Examples

```r
## A simple SIR model.

ewmeas %>%
  subset(time < 1952) %>%
pomp(
  times="time", t0=1948,
  rprocess=euler(
    Csnippet(""
      int nrate = 6;
      double rate[nrate]; // transition rates
      double trans[nrate]; // transition numbers
      double dW;

      // gamma noise, mean=dt, variance=(sigma^2 dt)
      dW = rgammawgamma(sigma, dt);

      // compute the transition rates
      rate[0] = mu*pop; // birth into susceptible class
      rate[1] = (iota+beta*I*dW/dt)/pop; // force of infection
      rate[2] = mu; // death from susceptible class
      rate[3] = gamma; // recovery
      rate[4] = mu; // death from infectious class
      rate[5] = mu; // death from recovered class

      // compute the transition numbers
    ")
```

trans[0] = rpois(rate[0]*dt); // births are Poisson
reulermultinom(2,S,&rate[1],dt,&trans[1]);
reulermultinom(2,I,&rate[3],dt,&trans[3]);
reulermultinom(1,R,&rate[5],dt,&trans[5]);

// balance the equations
S += trans[0]-trans[1]-trans[2];
I += trans[1]-trans[3]-trans[4];
R += trans[3]-trans[5];

),
delta.t=1/52/20
),
rinit=Csnippet("double m = pop/(S_0+I_0+R_0);
S = nearbyint(m*S_0);
I = nearbyint(m*I_0);
R = nearbyint(m*R_0);
"),

paramnames=c("mu","pop","iota","gamma","Beta","sigma",
"S_0","I_0","R_0"),

statenames=c("S","I","R"),
params=c(mu=1/50,iota=10,pop=50e6,gamma=26,Beta=400,sigma=0.1,
S_0=0.07,I_0=0.001,R_0=0.93)
) -> ew1

ev1 %>%
simulate() %>%
plot(variables=c("S","I","R"))

## A simple SIR model that tracks cumulative incidence.

ev1 %>%
pomp(
rprocess=euler(
Csnippet("int nrate = 6;
double rate[nrate]; // transition rates
double trans[nrate]; // transition numbers
double dW;

// gamma noise, mean=dt, variance=(sigma*2 dt)
dW = rgammaw(n,mean=dt);

// compute the transition rates
rate[0] = mu*pop; // birth into susceptible class
rate[1] = (iota+Beta*I*dW/dt)/pop; // force of infection
rate[2] = mu; // death from susceptible class
rate[3] = gamma; // recovery
rate[4] = mu; // death from infectious class
rate[5] = mu; // death from recovered class

// compute the transition numbers
trans[0] = rpois(rate[0]*dt); // births are Poisson

accumulators
as.data.frame

```
reulermultinom(2, S, &rate[1], dt, &trans[1]);
reulermultinom(2, I, &rate[3], dt, &trans[3]);
reulermultinom(1, R, &rate[5], dt, &trans[5]);

// balance the equations
S += trans[0] - trans[1] - trans[2];
R += trans[3] - trans[5];
H += trans[3]; // cumulative incidence
",
delta.t=1/52/20
),
rmmeasure=Csnippet("
  double mean = H*rho;
  double size = 1/tau;
  reports = rnbinom_mu(size,mean);
"),
rinit=Csnippet("
  double m = pop/(S_0+I_0+R_0);
  S = nearbyint(m*S_0);
  I = nearbyint(m*I_0);
  R = nearbyint(m*R_0);
  H = 0;
"),
paramnames=c("mu","pop","iota","gamma","Beta","sigma","tau","rho",
  "S_0","I_0","R_0"),
statenames=c("S","I","R","H"),
params=c(mu=1/50,iota=10,pop=50e6,gamma=26,
  Beta=400,sigma=0.1,tau=0.001,rho=0.6,
  S_0=0.07,I_0=0.001,R_0=0.93)
) -> ew2

ew2 %>%
simulate() %>%
plot()

## A simple SIR model that tracks weekly incidence.

ew2 %>%
pomp(accumvars="H") -> ew3

ew3 %>%
simulate() %>%
plot()
```

as.data.frame

Coerce to data.frame
Description

All pomp model objects can be recast as data frames. The contents of the resulting data frame depend on the nature of the object.

Usage

```r
## S3 method for class 'pomp'
as.data.frame(x, ...)

## S3 method for class 'pfilterd_pomp'
as.data.frame(x, ...)

## S3 method for class 'probed_pomp'
as.data.frame(x, ...)

## S3 method for class 'kalmand_pomp'
as.data.frame(x, ...)

## S3 method for class 'bsmcd_pomp'
as.data.frame(x, ...)

## S3 method for class 'pompList'
as.data.frame(x, ...)

## S3 method for class 'pfilterList'
as.data.frame(x, ...)

## S3 method for class 'abcList'
as.data.frame(x, ...)

## S3 method for class 'mif2List'
as.data.frame(x, ...)

## S3 method for class 'pmcmcList'
as.data.frame(x, ...)

## S3 method for class 'wpfilterd_pomp'
as.data.frame(x, ...)
```

Arguments

- `x` any R object.
- `...` additional arguments to be passed to or from methods.

Details

When `object` is a simple ‘pomp’ object, `as(object, "data.frame")` or `as.data.frame(object)` results in a data frame with the times, observables, states (if known), and interpolated covariates (if
When object is a `pfilterd_pomp` object, coercion to a data frame results in a data frame with the same content as for a simple `pomp`, but with conditional log likelihood and effective sample size estimates included, as well as filtering means, prediction means, and prediction variances, if these have been computed.

When object is a `probed_pomp` object, coercion to a data frame results in a data frame with the values of the probes computed on the data and on simulations.

When object is a `kalmand_pomp` object, coercion to a data frame results in a data frame with prediction means, filter means and forecasts, in addition to the data.

When object is a `bsmcd_pomp` object, coercion to a data frame results in a data frame with samples from the prior and posterior distribution. The `.id` variable distinguishes them.

When object is a `wpfilterd_pomp` object, coercion to a data frame results in a data frame with the same content as for a simple `pomp`, but with conditional log likelihood and effective sample size estimates included.

---

### Description

Tools for reproducible computations.

### Usage

```r
bake(file, expr, seed = NULL, kind = NULL, normal.kind = NULL)

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

freeze(expr, seed = NULL, kind = NULL, normal.kind = NULL)
```

### 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 object and hence be an RDS file (extension ‘rds’); for `stew`, this will contain one or more named objects and hence be an RDA file (extension ‘rda’).

- **expr**: Expression to be evaluated.

- **seed, kind, normal.kind**: optional. To set the state of the RNG. See `set.seed`. The default, `seed = NULL`, will not change the RNG state. `seed` should be a single integer. See `set.seed`. 

---
Details

On cooking shows, recipes requiring lengthy baking or stewing are prepared beforehand. The bake and stew functions perform analogously: an 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. Instead, 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.

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

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

Value

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

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

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

bake and stew return information about the time used in evaluating the expression. This is recorded in the system.time attribute of the return value. In addition, if 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)
  y <- rnorm(n=10,mean=3*x+5,sd=2)
})
```
basic_probes

))

plot(x,y)

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

## End(Not run)

---

basic_probes

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.period(var, kernel.width, transform = identity)

probe.quantile(var, probs, ...)

probe.acf(
  var,
  lags,
  type = c("covariance", "correlation"),
  transform = identity
)

probe.ccf(
  vars,
  lags,
  type = c("covariance", "correlation"),
  transform = identity
)

probe.marginal(var, ref, order = 3, diff = 1, transform = identity)

probe.nlar(var, lags, powers, transform = identity)
Arguments

- **var, vars**: character; the name(s) of the observed variable(s).
- **trim**: the fraction of observations to be trimmed (see `mean`).
- **transform**: transformation to be applied to the data before the probe is computed.
- **na.rm**: if **TRUE**, remove all NA observations prior to computing the probe.
- **kernel.width**: width of modified Daniell smoothing kernel to be used in power-spectrum computation; see `kernel`.
- **probs**: the quantile or quantiles to compute: see `quantile`.
- **...**: additional arguments passed to the underlying algorithms.
- **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.
- **type**: Compute autocorrelation or autocovariance?
- **ref**: empirical reference distribution. Simulated data will be regressed against the values of **ref**, sorted and, optionally, differenced. The resulting regression coefficients capture information about the shape of the marginal distribution. A good choice for **ref** is the data itself.
- **order**: order of polynomial regression.
- **diff**: order of differencing to perform.
- **powers**: the powers of each term (corresponding to **lags**) in the the nonlinear autoregressive model that will be fit to the actual and simulated data. See Details, below, for a precise description.

Value

A call to any one of these functions returns a probe function, suitable for use in `probe` or `probe_objfun`. 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, Aaron A. King

References


See Also

Other summary statistics methods: `abc()`, `probe.match`, `probe()`, `spect()`
blowflies

Nicholson's blowflies.

Description

blowflies is a data frame containing the data from several of Nicholson’s classic experiments with the Australian sheep blowfly, *Lucilia cuprina*.

Usage

```r
blowflies1(
  P = 3.2838,
  delta = 0.16073,
  N0 = 679.94,
  sigma.P = 1.3512,
  sigma.d = 0.74677,
  sigma.y = 0.026649
)
```

```r
blowflies2(
  P = 2.7319,
  delta = 0.17377,
  N0 = 800.31,
  sigma.P = 1.442,
  sigma.d = 0.76033,
  sigma.y = 0.010846
)
```

Arguments

- **P**: reproduction parameter
- **delta**: death rate
- **N0**: population scale factor
- **sigma.P**: intensity of \( e \) noise
- **sigma.d**: intensity of \( eps \) noise
- **sigma.y**: measurement error s.d.

Details

`blowflies1()` and `blowflies2()` construct ‘pomp’ objects encoding stochastic delay-difference equation models. The data for these come from "population I", a control culture. 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:

\[
\begin{align*}
R(t + 1) &\sim \text{Poisson}(PN(t - \tau) \exp(-N(t - \tau)/N_0)e(t + 1)\Delta t) \\
S(t + 1) &\sim \text{Binomial}(N(t), \exp(-\delta e(t + 1)\Delta t))
\end{align*}
\]

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

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

\[y(t) \sim \text{NegBin}(N(t), 1/\sigma_y^2)\]

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

Default parameter values are the MLEs as estimated by Ionides (2011).

**Value**

\texttt{blowflies1} and \texttt{blowflies2} return ‘pomp’ objects containing the actual data and two variants of the model.

**References**


**See Also**

Other pomp examples: \texttt{bsflu, dacca, ebola, gompertz, measles, ou2, parus, pomp_examples, ricker, rw2, sir_models, verhulst}

Other datasets: \texttt{bsflu, dacca, ebola, measles, parus}

**Examples**

\[\text{plot(blowflies1())}\]
\[\text{plot(blowflies2())}\]
Influenza outbreak in a boarding school

Description

An outbreak of influenza in an all-boys boarding school.

Details

Data are recorded from a 1978 flu outbreak in a closed population. The variable ‘B’ refers to boys confined to bed on the corresponding day and ‘C’ to boys in convalescence, i.e., not yet allowed back to class. In total, 763 boys were at risk of infection and, over the course of the outbreak, 512 boys spent between 3 and 7 days away from class (either in bed or convalescent). The index case was a boy who arrived at school from holiday six days before the next case.

References


See Also

sir_models

Other datasets: blowflies, dacca(), ebola, measles, parus

Other pomp examples: blowflies, dacca(), ebola, gompertz(), measles.ou2(), parus, pomp_examples, ricker(), rw2(), sir_models, verhulst()

Examples

library(magrittr)
library(tidyr)
library(ggplot2)
bsflu %>%
gather(variable,value,-date,-day) %>%
ggplot(aes(x=date,y=value,color=variable))+
geom_line()+
labs(y="number of boys",title="boarding school flu outbreak")+
theme_bw()
bsmc2

The Liu and West Bayesian particle filter

Description


Usage

```r
## S4 method for signature 'data.frame'
bsmc2(
  data,
  Np,
  smooth = 0.1,
  params,
  rprior,
  rinit,
  rprocess,
  dmeasure,
  partrans,
  ...
)

## S4 method for signature 'pomp'
bsmc2(data, Np, smooth = 0.1, ..., verbose = getOption("verbose", FALSE))
```

Arguments

data: either a data frame holding the time series data, or an object of class 'pomp', i.e., the output of another `pomp` calculation.

Np: the number of particles to use. This may be specified as a single positive integer, in which case the same number of particles will be used at each timestep. Alternatively, if one wishes the number of particles to vary across timesteps, one may specify `Np` either as a vector of positive integers of length `length(time(object,t0=TRUE))`

or as a function taking a positive integer argument. In the latter case, `Np(k)` must be a single positive integer, representing the number of particles to be used at the k-th timestep: `Np(0)` is the number of particles to use going from `timezero(object)` to `time(object)[1]`, `Np(1)`, from `timezero(object)` to `time(object)[1]`, and so on, while when `T=length(time(object))`, `Np(T)` is the number of particles to sample at the end of the time-series.

smooth: Kernel density smoothing parameter. The compensating shrinkage factor will be `sqrt(1-smooth^2)`. Thus, `smooth=0` means that no noise will be added to parameters. The general recommendation is that the value of smooth should be chosen close to 0 (e.g., shrink ~ 0.1).
params  optional; named numeric vector of parameters. This will be coerced internally to storage mode double.

rprior  optional; prior distribution sampler, specified either as a C snippet, an R function, or the name of a pre-compiled native routine available in a dynamically loaded library. For more information, see `?prior_spec`. Setting rprior=\(\text{NULL}\) removes the prior distribution sampler.

rinit  simulator of the initial-state distribution. This can be furnished either as a C snippet, an R function, or the name of a pre-compiled native routine available in a dynamically loaded library. Setting rinit=\(\text{NULL}\) sets the initial-state simulator to its default. For more information, see `?rinit_spec`.

rprocess  simulator of the latent state process, specified using one of the rprocess plugins. Setting rprocess=\(\text{NULL}\) removes the latent-state simulator. For more information, see `?rprocess_spec` for the documentation on these plugins.

dmeasure  evaluator of the measurement model density, specified either as a C snippet, an R function, or the name of a pre-compiled native routine available in a dynamically loaded library. Setting dmeasure=\(\text{NULL}\) removes the measurement density evaluator. For more information, see `?dmeasure_spec`.

partrans  optional parameter transformations, constructed using `parameter_trans`. 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. One should supply the partrans argument via a call to `parameter_trans`. For more information, see `?parameter_trans`. Setting partrans=\(\text{NULL}\) removes the parameter transformations, i.e., sets them to the identity transformation.

...  additional arguments supply new or modify existing model characteristics or components. See pomp for a full list of recognized arguments. When named arguments not recognized by pomp are provided, these are made available to all basic components via the so-called `userdata` facility. This allows the user to pass information to the basic components outside of the usual routes of covariates (covar) and model parameters (params). See `?userdata` for information on how to use this facility.

verbose  logical; if TRUE, diagnostic messages will be printed to the console.

Details

bsmc2 uses a version of the original algorithm (Liu & West 2001), but discards the auxiliary particle filter. The modification appears to give superior performance for the same amount of effort. Samples from the prior distribution are drawn using the rprior component. This is allowed to depend on elements of params, i.e., some of the elements of params can be treated as “hyperparameters”. \(N_p\) draws are made from the prior distribution.

Value

An object of class ‘bsmc2_pomp’. The following methods are available:

plot  produces diagnostic plots

as.data.frame  puts the prior and posterior samples into a data frame
bsplines

Author(s)
Michael Lavine, Matthew Ferrari, Aaron A. King, Edward L. Ionides

References

See Also
Other particle filter methods: cond.logLik(), eff.sample.size(), filter.mean(), filter.traj(), mif2(), pfilter(), pmcmc(), pred.mean(), pred.var(), saved.states(), wpfilter()

Other pomp parameter estimation methods: abc(), kalman, mif2(), nlf, pmcmc(), pomp-package, probe.match, spect.match

| bsplines | 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, deriv = 0, names = NULL)

periodic.bspline.basis(
  x,
  nbasis,
  degree = 3,
  period = 1,
  deriv = 0,
  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.
- deriv: The order of the derivative required.
coef

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.

period    The period of the requested periodic B-splines.

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`.

If `deriv>0`, the derivative of that order of each of the corresponding spline basis functions are returned.

C API

Access to the underlying C routines is available: see the `pomp C API document` for definition and documentation of the C API.

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')
```

---

**coef**

*Extract, set, or alter coefficients*

**Description**

Extract, set, or modify the estimated parameters from a fitted model.
## Usage

```r
## S4 method for signature 'listie'
coef(object, ...)

## 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 'objfun'
coef(object, ...)
```

### Arguments

- `object` an object of class ‘pomp’, or of a class extending ‘pomp’
- `...` ignored
- `pars` optional character; names of parameters to be retrieved or set.
- `transform` logical; perform parameter transformation?
- `value` numeric vector or list; values to be assigned. If `value = NULL`, the parameters are unset.

### Details

`coef` allows one to extract the parameters from a fitted model.
`coef(object, transform=TRUE)` returns the parameters transformed onto the estimation scale.
`coef(object) <-value` sets or alters the coefficients of a ‘pomp’ object.
`coef(object, transform=TRUE) <-value` assumes that value is on the estimation scale, and applies the “from estimation scale” parameter transformation from object before altering the coefficients.

---

### Description

The estimated conditional log likelihood from a fitted model.

## Usage

```r
## S4 method for signature 'kalmand_pomp'
cond.logLik(object, ...)

## S4 method for signature 'pfilterd_pomp'
cond.logLik(object, ...)
```
## S4 method for signature 'wpfilterd_pomp'
cond.logLik(object, ...)

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

cond.loglik(...)

**Arguments**

- **object**: result of a filtering computation
- **...**: ignored

**Details**

The conditional likelihood is defined to be the value of the density of

\[ Y(t_k)|Y(t_1), \ldots, Y(t_{k-1}) \]

evaluated at \( Y(t_k) = y^*_k \). Here, \( Y(t_k) \) is the observable process, and \( y^*_k \) the data, at time \( t_k \).

Thus the conditional log likelihood at time \( t_k \) is

\[ \ell_k(\theta) = \log f[Y(t_k) = y^*_k|Y(t_1) = y^*_1, \ldots, Y(t_{k-1}) = y^*_{k-1}], \]

where \( f \) is the probability density above.

**Value**

The numerical value of the conditional log likelihood. Note that some methods compute not the log likelihood itself but instead a related quantity. To keep the code simple, the `cond.logLik` function is nevertheless used to extract this quantity.

When `object` is of class `bsmcd_pomp` (i.e., the result of a `bsmc2` computation), `cond.logLik` returns the conditional log “evidence” (see `bsmc2`).

**See Also**

Other particle filter methods: `bsmc2()`, `eff.sample.size()`, `filter.mean()`, `filter.traj()`, `mif2()`, `pfilter()`, `pmcmc()`, `pred.mean()`, `pred.var()`, `saved.states()`, `wpfilter()`
continue

Continue an iterative calculation

Description

Continue an iterative computation where it left off.

Usage

```r
continue(object, ...)  
## S4 method for signature 'abcd_pomp'
continue(object, Nabc = 1, ...)

## S4 method for signature 'pmcmcd_pomp'
continue(object, Nmcmc = 1, ...)

## S4 method for signature 'mif2d_pomp'
continue(object, Nmif = 1, ...)
```

Arguments

- `object` the result of an iterative pomp computation
- `...` additional arguments will be passed to the underlying method. This allows one to modify parameters used in the original computations.
- `Nabc` positive integer; number of additional ABC iterations to perform
- `Nmcmc` positive integer; number of additional PMCMC iterations to perform
- `Nmif` positive integer; number of additional filtering iterations to perform

See Also

mif2 pmcmc abc

covariate_table

Covariates

Description

Constructing lookup tables for time-varying covariates.

Usage

```r
## S4 method for signature 'numeric'
covariate_table(..., order = c("linear", "constant"), times)

## S4 method for signature 'character'
covariate_table(..., order = c("linear", "constant"), times)
```
Arguments

... numeric vectors or data frames containing time-varying covariates. It must be possible to bind these into a data frame.

order the order of interpolation to be used. Options are “linear” (the default) and “constant”. Setting order="linear" treats the covariates as piecewise linear functions of time; order="constant" treats them as right-continuous piecewise constant functions.

times the times corresponding to the covariates. This may be given as a vector of (non-decreasing, finite) numerical values. Alternatively, one can specify by name which of the given variables is the time variable.

Details

If the ‘pomp’ object contains covariates (specified via the covar argument), 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 covariate table 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.

See Also

lookup

Other information on model implementation: Csnippet, accumulators, distributions, dmeasure_spec, dprocess_spec, parameter_trans(), pomp-package, prior_spec, rinit_spec, rmeasure_spec, rprocess_spec, skeleton_spec, transformations, userdata

covmat Estimate a covariance matrix from algorithm traces

Description

A helper function to extract a covariance matrix.

Usage

```r
## S4 method for signature 'pmcmcd_pomp'
covmat(object, start = 1, thin = 1, expand = 2.38, ...)

## S4 method for signature 'pmcmclist'
covmat(object, start = 1, thin = 1, expand = 2.38, ...)

## S4 method for signature 'abcd_pomp'
covmat(object, start = 1, thin = 1, expand = 2.38, ...)
```
## S4 method for signature 'abcList'
covmat(object, start = 1, thin = 1, expand = 2.38, ...)

## S4 method for signature 'probed_pomp'
covmat(object, ...)

### Arguments
- object: an object extending 'pomp'
- start: the first iteration number to be used in estimating the covariance matrix. Setting thin > 1 allows for a burn-in period.
- thin: factor by which the chains are to be thinned
- expand: the expansion factor
- ...: ignored

### Value
When object is the result of a pmcmc or abc computation, covmat(object) gives the covariance matrix of the chains. This can be useful, for example, in tuning the proposal distribution.

When object is a 'probed_pomp' object (i.e., the result of a probe computation), covmat(object) returns the covariance matrix of the probes, as applied to simulated data.

### See Also
MCMC proposals.

---

### Csnippet

#### Description
Accelerating computations through inline snippets of C code

#### Usage
Csnippet(text)

#### Arguments
- text: character; text written in the C language

#### Details
pomp provides a facility whereby users can define their model’s components using inline C code. C snippets are written to a C file, by default located in the R session’s temporary directory, which is then compiled (via 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 `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 `R CMD SHLIB`. If the resulting file does not compile, an error message will be generated. Compiler messages will be displayed, but no attempt will be made by pomp to interpret them. 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 statenames or paramnames arguments to pomp, respectively. Compiler errors that complain about undeclared state variables or parameters are usually due to failure to declare these in statenames or 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. One must never declare state variables, observables, covariates, or parameters within a C snippet.

4. Names of observables must match the names given given in the data. They must be referred to in measurement model C snippets (`rmeasure` and `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 headers ‘R.h’ and ‘Rmath.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.

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.

10. INCLUDE INFORMATION ABOUT LINKING TO PRECOMPILED LIBRARIES!

See Also

Other information on model implementation: accumulators.covariate_table(), distributions, dmeasure_spec, dprocess_spec, parameter_trans(), pomp-package, prior_spec, rinit_spec, rmeasure_spec, rprocess_spec, skeleton_spec, transformations, userdata
Description

dacca constructs 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.

Usage

dacca(
  gamma = 20.8,
  eps = 19.1,
  rho = 0,
  delta = 0.02,
  deltaI = 0.06,
  clin = 1,
  alpha = 1,
  beta_trend = -0.00498,
  logbeta = c(0.747, 6.38, -3.44, 4.23, 3.33, 4.55),
  logomega = log(c(0.184, 0.0786, 0.0584, 0.00917, 0.000208, 0.0124)),
  sd_beta = 3.13,
  tau = 0.23,
  S_0 = 0.621,
  I_0 = 0.378,
  Y_0 = 0,
  R1_0 = 0.000843,
  R2_0 = 0.000972,
  R3_0 = 1.16e-07
)

Arguments

gamma recovery rate
eps rate of waning of immunity for severe infections
rho rate of waning of immunity for inapparent infections
delta baseline mortality rate
deltaI cholera mortality rate
clin fraction of infections that lead to severe infection
alpha transmission function exponent
beta_trend slope of secular trend in transmission
logbeta seasonal transmission rates

dacca

logomega seasonal environmental reservoir parameters
ds_d_beta environmental noise intensity
tau measurement error s.d.
S_0 initial susceptible fraction
I_0 initial fraction of population infected
Y_0 initial fraction of the population in the Y class
R1_0, R2_0, R3_0 initial fractions in the respective R classes

Details

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

Value

dacca returns a ‘pomp’ object containing the model, data, and MLE parameters, as estimated by King et al. (2008).

References


See Also

Other pomp examples: blowflies, bsflu, ebola, gompertz(), measles, ou2(), parus, pomp_examples, ricker(), rw2(), sir_models, verhulst()

Other datasets: blowflies, bsflu, ebola, measles, parus

Examples

po <- dacca()
plot(po)
## MLE:
coef(po)
plot(simulate(po))
Design matrices for pomp calculations

Description

These functions are useful for generating designs for the exploration of parameter space.

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

runifDesign generates a design based on random samples from a multivariate uniform distribution.

sliceDesign generates points along slices through a specified point.

sobolDesign generates a Latin hypercube design based on the Sobol' low-discrepancy sequence.

Usage

profileDesign(
  ..., 
  lower, 
  upper, 
  nprof, 
  type = c("sobol", "runif"), 
  stringsAsFactors = getOption("stringsAsFactors", FALSE)
)

runifDesign(lower = numeric(0), upper = numeric(0), nseq)

sliceDesign(center, ...)

sobolDesign(lower = numeric(0), upper = numeric(0), nseq)

Arguments

... 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.

lower, upper named numeric vectors giving the lower and upper bounds of the ranges, respectively.

nprof The number of points per profile point.

type the type of design to use. type="sobol" uses sobolDesign; type="runif" uses runifDesign.

stringsAsFactors should character vectors be converted to factors?

nseq Total number of points requested.

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**

- `profileDesign` returns a data frame with `nprof` points per profile point.
- `runifDesign` returns a data frame with `nseq` rows and one column for each variable named in `lower` and `upper`.
- `sliceDesign` returns a data frame with one row per point. The ‘slice’ variable indicates which slice the point belongs to.
- `sobolDesign` returns a data frame with `nseq` rows and one column for each variable named in `lower` and `upper`.

**Author(s)**

Aaron A. King

**References**


**Examples**

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

## Uniform random design
plot(runifDesign(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=200)
dim(x)
plot(x)

## A two-parameter profile design with random points:
x <- profileDesign(p=1:10,q=3:5,lower=c(a=0,b=0),upper=c(b=5,a=1),nprof=200,type="runif")
dim(x)
plot(x)

## A single 11-point slice through the point c(A=3,B=8,C=0) along the B direction.
```
distributions

```r
distributions
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)
```

---

### Description

**pomp** provides a number of probability distributions that have proved useful in modeling partially observed Markov processes. These include the Euler-multinomial family of distributions and the Gamma white-noise processes.

### Usage

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

### Arguments

- `n` integer; number of random variates to generate.
- `size` scalar integer; number of individuals at risk.
- `rate` numeric vector of hazard rates.
- `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.
- `sigma` numeric scalar; intensity of the Gamma white noise process.

### 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

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

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

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

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

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

or

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

He et al. (2010) 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 that have succumbed to the corresponding process.

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

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

An interface for C codes using these functions is provided by the package. Visit the package homepage to view the pomp C API document.

Author(s)

Aaron A. King

References


See Also

Other information on model implementation: Csnippet, accumulators, covariate_table(), dmeasure_spec, dprocess_spec, parameter_trans(), pomp-package, prior_spec, rinit_spec, rmeasure_spec, rprocess_spec, skeleton_spec, transformations, userdata

Examples

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

---

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

Usage

```r
## S4 method for signature 'pomp'
dmeasure(object, y, x, times, params, ..., log = FALSE)
```
Arguments
  
  object an object of class ‘pomp’, or of a class that extends ‘pomp’. This will typically be the output of pomp, simulate, or one of the pomp inference algorithms.

  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.

  x an 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. One can also pass x as a named numeric vector, which is equivalent to the nrep=1, ntimes=1 case.

  times 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 treated as an independent parameter set, in correspondence with the corresponding column of x.

  ... additional arguments are ignored.

  log if TRUE, log probabilities are returned.

Value
  
  dmeasure returns a matrix of dimensions nreps x ntimes. If d is the returned matrix, d[j,k] is the likelihood (or log likelihood if log = TRUE) of the observation y[,k] at time times[k] given the state x[,j,k].

See Also
  
  Specification of the measurement density evaluator: dmeasure_spec

  Other pomp workhorses: dprior(), dprocess(), flow(), partrans(), rinit(), rmeasure(), rprior(), rprocess(), skeleton(), workhorses
to `pomp` algorithms, where \( f \) is a C snippet or R function that implements your procedure.

Using a C snippet is much preferred, due to its much greater computational efficiency. See `Csnippet` for general rules on writing C snippets. The goal of a `dmeasure` C snippet is to fill the variable `lik` with the either the probability density or the log probability density, depending on the value of the variable `give_log`.

In writing a `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 will be defined in the context in which the snippet is executed.
2. Moreover, the Boolean variable `give_log` will be defined.
3. The goal of a `dmeasure` C snippet is to set the value of the `lik` variable to the likelihood of the data given the state, if `give_log == 0`. 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 \( f \) is a function. The arguments of \( f \) should be chosen from among the observables, state variables, parameters, covariates, and time. It must also have the arguments \( \ldots, \log \). It can take additional arguments via the `facility`. \( f \) must return a single numeric value, the probability density (or log probability density if `log = TRUE`) of \( y \) given \( x \) at time \( t \).

**Important note**

It is a common error to fail to account for both \( \log = \text{TRUE} \) and \( \log = \text{FALSE} \) when writing the `dmeasure` C snippet or function.

**Default behavior**

If `dmeasure` is left unspecified, calls to `dmeasure` will return missing values (NA).

**See Also**

Other information on model implementation: `Csnippet`, `accumulators`, `covariate_table()`, `distributions`, `dprocess_spec`, `parameter_trans()`, `pomp-package`, `prior_spec`, `rinit_spec`, `rmeasure_spec`, `rprocess_spec`, `skeleton_spec`, `transformations`, `userdata`

---

### `dprior`

**Description**

Evaluates the prior probability density.
Usage

## S4 method for signature 'pomp'
dp prior(object, params, ..., log = FALSE)

Arguments

object an object of class ‘pomp’, or of a class that extends ‘pomp’. This will typically be the output of pomp, simulate, or one of the pomp inference algorithms.

params a npar x nrep matrix of parameters. Each column is treated as an independent parameter set, in correspondence with the corresponding column of x.

... additional arguments are ignored.

log if TRUE, log probabilities are returned.

Value

The required density (or log density), as a numeric vector.

See Also

Specification of the prior density evaluator: prior_spec

Other pomp workhorses: dmeasure(), dprocess(), flow(), partrans(), rinit(), rmeasure(), rprior(), rprocess(), skeleton(), workhorses

Description

Evaluates the probability density of a sequence of consecutive state transitions.

Usage

## S4 method for signature 'pomp'
dprocess(object, x, times, params, ..., log = FALSE)

Arguments

object an object of class ‘pomp’, or of a class that extends ‘pomp’. This will typically be the output of pomp, simulate, or one of the pomp inference algorithms.

x an 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. One can also pass x as a named numeric vector, which is equivalent to the nrep=1, ntimes=1 case.

times a numeric vector (length ntimes) containing times. These must be in non-decreasing order.
dprocess_spec

params  a npar x nrep matrix of parameters. Each column is treated as an independent parameter set, in correspondence with the corresponding column of \( x \).

... additional arguments are ignored.

log  if TRUE, log probabilities are returned.

Value

dprocess returns a matrix of dimensions nrep x ntimes-1. If \( d \) is the returned matrix, \( d[j,k] \) is the likelihood (or the log likelihood if \( \log=\text{TRUE} \)) of the transition from state \( x[j,k-1] \) at time \( \text{times}[k-1] \) to state \( x[j,k] \) at time \( \text{times}[k] \).

See Also

Specification of the process-model density evaluator: dprocess_spec

Other pomp workhorses: dmeasure(), dprior(), flow(), partrans(), rinit(), rmeasure(), rprior(), rprocess(), skeleton(), workhorses

dprocess_spec  The latent state process density

Description

Specification of dprocess.

Details

Suppose 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 \( t_2 > t_1 \) under the assumption that the state remains unchanged between \( t_1 \) and \( t_2 \). Then you can furnish

\[
dprocess = f
\]

to pomp, where \( f \) is a C snippet or R function that implements your procedure. Specifically, \( f \) should compute the log probability density.

Using a C snippet is much preferred, due to its much greater computational efficiency. See Csnippet for general rules on writing C snippets. The goal of a dprocess C snippet is to fill the variable \( \loglik \) with the log probability density. In the context of such a C snippet, the parameters, and covariates will be defined, as will the times \( t_1 \) and \( t_2 \). The state variables at time \( t_1 \) will have their usual name (see statenames) with a “_1” appended. Likewise, the state variables at time \( t_2 \) will have a “_2” appended.

If \( f \) is given as an R function, it should take as arguments any or all of the state variables, parameter, covariates, and time. The state-variable and time arguments will have suffices “_1” and “_2” appended. Thus for example, if \( \text{var} \) is a state variable, when \( f \) is called, \( \text{var}_1 \) will have value of state variable \( \text{var} \) at time \( t_1 \), \( \text{var}_2 \) will have the value of \( \text{var} \) at time \( t_2 \). \( f \) 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 demos and the tutorials on the package website.
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.

Default behavior

By default, dprocess returns missing values (NA).

See Also

Other information on model implementation: Csnippet, accumulators, covariate_table(), distributions, dmeasure_spec, parameter_trans(), pomp-package, prior_spec, rinit_spec, rmeasure_spec, rprocess_spec, skeleton_spec, transformations, userdata

ebola


Description


Usage

```r
ebolaModel(
  country = c("GIN", "LBR", "SLE"),
  data = NULL,
  timestep = 1/8,
  nstageE = 3L,
  R0 = 1.4,
  rho = 0.2,
  cfr = 0.7,
  k = 0,
  index_case = 10,
  incubation_period = 11.4,
  infectious_period = 7
)
```

Arguments

country ISO symbol for the country (GIN=Guinea, LBR=Liberia, SLE=Sierra Leone).
data if NULL, the situation report data (see ebolaWHO) for the appropriate country or region will be used. Providing a dataset here will override this behavior.
timestep duration (in days) of Euler time-step for the simulations.
nstageE integer; number of incubation stages.
R0 basic reproduction ratio
rho case reporting efficiency

cfr case fatality rate

k dispersion parameter (negative binomial size parameter)

index_case number of cases on day 0 (2014-04-01)

incubation_period, infectious_period mean duration (in days) of the incubation and infectious periods.

Details

The data include monthly case counts and death reports derived from WHO situation reports, as reported by the U.S. CDC. The models are described in King et al. (2015).

The data-cleaning script is included in the R source code file ‘ebola.R’.

Model structure

The default incubation period is supposed to be Gamma distributed with shape parameter \( nstageE \) and mean 11.4 days and the case-fatality ratio (‘cfr’) is taken to be 0.7 (cf. WHO Ebola Response Team 2014). The discrete-time formula is used to calculate the corresponding \( \alpha \) (cf. He et al. 2010).

The observation model is a hierarchical model for cases and deaths:

\[
p(R_t, D_t | C_t) = p(R_t | C_t) p(D_t | C_t, R_t).
\]

Here, \( p(R_t | C_t) \) is negative binomial with mean \( \rho C_t \) and dispersion parameter \( 1/k \); \( p(D_t | C_t, R_t) \) is binomial with size \( R_t \) and probability equal to the case fatality rate \( \text{cfr} \).

References


See Also

Other datasets: blowflies, bsflu, dacca(), measles, parus

Other pomp examples: blowflies, bsflu, dacca(), gompertz(), measles.ou2(), parus. pomp_examples, ricker(), rw2(), sir_models, verhulst()
Examples

data(ebolaWA2014)

library(ggplot2)
library(tidyr)

ebolaWA2014 %>%
gather(variable,count,cases,deaths) %>%
ggplot(aes(x=date,y=count,group=country,color=country))+
  geom_line()+
  facet_grid(variable~.,scales="free_y")+
  theme_bw()+
  theme(axis.text=element_text(angle=-90))

ebolaWA2014 %>%
gather(variable,count,cases,deaths) %>%
ggplot(aes(x=date,y=count,group=variable,color=variable))+
  geom_line()+
  facet_grid(country~.,scales="free_y")+
  theme_bw()+
  theme(axis.text=element_text(angle=-90))

plot(ebolaModel(country="SLE"))
plot(ebolaModel(country="LBR"))
plot(ebolaModel(country="GIN"))

data.frame(eff.sample.size)

Description

Estimate the effective sample size of a Monte Carlo computation.

Usage

## S4 method for signature 'bsmcd_pomp'
eff.sample.size(object, ...)

## S4 method for signature 'pfilterd_pomp'
eff.sample.size(object, ...)

## S4 method for signature 'wpfilterd_pomp'
eff.sample.size(object, ...)

Arguments

object	result of a filtering computation
...	ignored
Details

Effective sample size is computed as
\[
\left( \sum_i w_{it}^2 \right)^{-1},
\]
where \( w_{it} \) is the normalized weight of particle \( i \) at time \( t \).

See Also

Other particle filter methods: `bsmc2()`, `cond.logLik()`, `filter.mean()`, `filter.traj()`, `mif2()`, `pfilter()`, `pmcmc()`, `pred.mean()`, `pred.var()`, `saved.states()`, `wpfilter()`
Description

Drawing from the smoothing distribution

Usage

```r
## S4 method for signature 'pfilterd_pomp'
filter.traj(object, vars, ...)

## S4 method for signature 'pfilterList'
filter.traj(object, vars, ...)

## S4 method for signature 'pmcmcdd_pomp'
filter.traj(object, vars, ...)

## S4 method for signature 'pmcmcList'
filter.traj(object, vars, ...)
```

Arguments

- `object`: result of a filtering computation
- `vars`: optional character; names of variables
- `...`: ignored

Details

The smoothing distribution is the distribution of

\[ X(t_k)|Y(t_1) = y_1^*, \ldots, Y(t_n) = y_n^* \]

where \( X(t_k) \) is the latent state process and \( Y(t_k) \) is the observable process at time \( t_k \), and \( n \) is the number of observations.

To draw samples from this distribution, one can run a number of independent particle filter (`pfilter`) operations, sampling the full trajectory of one randomly-drawn particle from each one. One should view these as weighted samples from the smoothing distribution, where the weights are the likelihoods returned by each of the `pfilter` computations.

One accomplishes this by setting `filter.traj = TRUE` in each `pfilter` computation and extracting the trajectory using the `filter.traj` command.

In particle MCMC (`pmcmc`), the tracking of an individual trajectory is performed automatically.

See Also

Other particle filter methods: `bsmc2()`, `cond.logLik()`, `eff.sample.size()`, `filter.mean()`, `mif2()`, `pfilter()`, `pmcmc()`, `pred.mean()`, `pred.var()`, `saved.states()`, `wpfilter()`
## Flow

**Flow of a deterministic model**

**Description**

Compute the flow induced by a deterministic vectorfield or map.

**Usage**

```r
## S4 method for signature 'pomp'
flow(object, x0, t0, times, params, ..., verbose = getOption("verbose", FALSE))
```

**Arguments**

- `object`: an object of class ‘pomp’, or of a class that extends ‘pomp’. This will typically be the output of `pomp`, `simulate`, or one of the `pomp` inference algorithms.
- `x0`: an array with dimensions `nvar x nrep` giving the initial conditions of the trajectories to be computed.
- `t0`: the time at which the initial conditions are assumed to hold.
- `times`: a numeric vector (length `ntimes`) containing times at which the itineraries are desired. These must be in non-decreasing order with `times[1]>t0`.
- `params`: a `npar x nrep` matrix of parameters. Each column is treated as an independent parameter set, in correspondence with the corresponding column of `x`.
- `...`: Additional arguments are passed to the ODE integrator (if the skeleton is a vectorfield) and are ignored if it is a map. See `ode` for a description of the additional arguments accepted by the ODE integrator.
- `verbose`: logical; if TRUE, diagnostic messages will be printed to the console.

**Details**

In the case of a discrete-time system (map), `flow` iterates the map to yield trajectories of the system. In the case of a continuous-time system (vectorfield), `flow` uses the numerical solvers in `deSolve` to integrate the vectorfield starting from given initial conditions.

**Value**

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

**See Also**

- `skeleton`, `trajectory`, `rprocess`
- Other pomp workhorses: `dmeasure()`, `dprior()`, `dprocess()`, `partrans()`, `rinit()`, `rmeasure()`, `rprior()`, `rprocess()`, `skeleton()`, `workhorses`
**forecast**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forecast mean</td>
<td>Mean of the one-step-ahead forecasting distribution.</td>
</tr>
</tbody>
</table>

**Usage**

```r
forecast(object, ...)
```

```
## S4 method for signature 'kalmand_pomp'
forecast(object, vars, ...)
```

**Arguments**

- **object**: result of a filtering computation
- **...**: ignored
- **vars**: optional character; names of variables

---

**gompertz**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gompertz model with log-normal observations.</td>
<td>constructs a ‘pomp’ object encoding a stochastic Gompertz population model with log-normal measurement error.</td>
</tr>
</tbody>
</table>

**Usage**

```r
gompertz(
  K = 1,
  r = 0.1,
  sigma = 0.1,
  tau = 0.1,
  X_0 = 1,
  times = 1:100,
  t0 = 0
)
```
Arguments

- $K$: carrying capacity
- $r$: growth rate
- $\sigma$: process noise intensity
- $\tau$: measurement error s.d.
- $X_0$: value of the latent state variable $X$ at the zero time
- $t$: observation times
- $t_0$: zero time

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.

Value

A ‘pomp’ object with simulated data.

See Also

Other pomp examples: blowflies, bsflu, dacca(), ebola, measles, ou2(), parus, pomp_examples, ricker(), rw2(), sir_models, verhulst()

Examples

```r
plot(gompertz())
plot(gompertz(K=2, r=0.01))
```

Description

The algorithms in pomp are formulated in terms of elementary functions that access the basic model components (rprocess, dprocess, rmeasure, dmeasure, etc.). For short, we refer to these elementary functions as “workhorses”. In implementing a model, the user specifies basic model components using functions, procedures in dynamically-linked libraries, or C snippets. Each component is then packaged into a ‘pomp_fun’ objects, which gives a uniform interface. The construction of ‘pomp_fun’ objects is handled by the hitch function, which conceptually “hitches” the workhorses to the user-defined procedures.
hitch

Usage

hitch(
    ..., templates, obsnames, statenames, paramnames, covarnames,
    PACKAGE, globals, cfile,
    cdir = getOption("pomp_cdir", NULL),
    shlib.args,
    compile = TRUE,
    verbose = getOption("verbose", FALSE)
)

Arguments

... named arguments representing the user procedures to be hitched. These can be functions, character strings naming routines in external, dynamically-linked libraries, C snippets, or NULL. The first three are converted by hitch to 'pomp_fun' objects which perform the indicated computations. NULL arguments are translated to default 'pomp_fun' objects. If any of these procedures are already 'pomp_fun' objects, they are returned unchanged.

templates named list of templates. Each workhorse must have a corresponding template. See pomp:::workhorse_templates for a list.

obsnames, statenames, paramnames, covarnames character vectors specifying the names of observable variables, latent state variables, parameters, and covariates, respectively. These are only needed if one or more of the horses are furnished as C snippets.

PACKAGE optional character; the name (without extension) of the external, 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 used for any component specified using C snippets. PACKAGE can name at most one library.

globals optional character; arbitrary C code that will be hard-coded into the shared-object library created when C snippets are provided. If no C snippets are used, globals has no effect.

cfile optional character variable. 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. If the chosen filename would result in over-writing an existing file, an error is generated.

cdir optional character variable. 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 {R} session. One can also set this directory using the pomp_cdir option.
optional character variables. Command-line arguments to the R CMD SHLIB call that compiles the C snippets.

compile logical; if FALSE, compilation of the C snippets will be postponed until they are needed.

verbose logical. Setting verbose=TRUE will cause additional information to be displayed.

Value

hitch returns a named list of length two. The element named “funs” is itself a named list of 'pomp_fun' objects, each of which corresponds to one of the horses passed in. The element named “lib” contains information on the shared-object library created using the C snippets (if any were passed to hitch). If no C snippets were passed to hitch, lib is NULL. Otherwise, it is a length-3 named list with the following elements:

name The name of the library created.

dir The directory in which the library was created. If this is NULL, the library was created in the session's temporary directory.

src A character string with the full contents of the C snippet file.

Author(s)

Aaron A. King

See Also

pomp, spy

---

**kalman**

*Ensemble Kalman filters*

**Description**

The ensemble Kalman filter and ensemble adjustment Kalman filter.

**Usage**

```r
## S4 method for signature 'data.frame'
enkf(
  data,
  Np,
  h,
  R,
  params,
  rinit,
  rprocess,
  ...,
)```
verbose = getOption("verbose", FALSE)

## S4 method for signature 'pomp'
enkf(data, Np, h, R, ..., verbose = getOption("verbose", FALSE))

## S4 method for signature 'data.frame'
eakf(
data,
    Np,
    C,
    R,
    params,
    rinit,
    rprocess,
    ...
    verbose = getOption("verbose", FALSE)
)

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

Arguments

data either a data frame holding the time series data, or an object of class ‘pomp’, i.e., the output of another pomp calculation.

Np the number of particles to use.

h function returning the expected value of the observation given the state.

R matrix; variance of the measurement noise.

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

rinit simulator of the initial-state distribution. This can be furnished either as a C snippet, an R function, or the name of a pre-compiled native routine available in a dynamically loaded library. Setting rinit=NULL sets the initial-state simulator to its default. For more information, see ?rinit_spec.

rprocess simulator of the latent state process, specified using one of the rprocess plugins. Setting rprocess=NULL removes the latent-state simulator. For more information, see ?rprocess_spec for the documentation on these plugins.

... additional arguments supply new or modify existing model characteristics or components. See pomp for a full list of recognized arguments.

When named arguments not recognized by pomp are provided, these are made available to all basic components via the so-called userdata facility. This allows the user to pass information to the basic components outside of the usual routes of covariates (covar) and model parameters (params). See ?userdata for information on how to use this facility.

verbose logical; if TRUE, diagnostic messages will be printed to the console.

C matrix converting state vector into expected value of the observation.
Value
An object of class `kalmand_pomp`.

Author(s)
Aaron A. King

References


See Also
Other pomp parameter estimation methods: `abc()`, `bsmc2()`, `mif2()`, `nlf`, `pmcmc()`, `pomp-package`, `probe.match`, `spect.match`

### logLik

**Log likelihood**

Description
Extract the estimated log likelihood (or related quantity) from a fitted model.

Usage

```r
logLik(object, ...) # S4 method for signature 'listie'
logLik(object, ...) # S4 method for signature 'pfilterd_pomp'
logLik(object) # S4 method for signature 'wpfilterd_pomp'
logLik(object) # S4 method for signature 'probed_pomp'
logLik(object) # S4 method for signature 'kalmand_pomp'
```
## S4 method for signature 'pmcmcd_pomp'
logLik(object)

## S4 method for signature 'bsmcd_pomp'
logLik(object)

## S4 method for signature 'objfun'
logLik(object)

## S4 method for signature 'spect_match_objfun'
logLik(object)

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

### Arguments

- **object**: fitted model object
- **...**: ignored

### Value

numerical value of the log likelihood. Note that some methods compute not the log likelihood itself but instead a related quantity. To keep the code simple, the `logLik` function is nevertheless used to extract this quantity.

When `object` is of 'probed_pomp' class (i.e., the result of a probe computation), `logLik` retrieves the “synthetic likelihood” (see `probe`).

When `object` is of 'bsmcd_pomp' class (i.e., the result of a bsmc2 computation), `logLik` retrieves the “log evidence” (see `bsmc2`).

When `object` is an NLF objective function, i.e., the result of a call to `nlf_objfun`, `logLik` retrieves the “quasi log likelihood” (see `nlf`).

---

### logmeanexp

**The log-mean-exp trick**

### Description

`logmeanexp` computes

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

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

\texttt{logmeanexp(x, se = FALSE)}

Arguments

- \texttt{x} numeric
- \texttt{se} logical; give approximate standard error?

Details

When \texttt{se = TRUE}, \texttt{logmeanexp} uses a jackknife estimate of the variance in \textit{log(x)}.

Value

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

Author(s)

Aaron A. King

Examples

```r
## an estimate of the log likelihood:
p0 <- ricker()
l1 <- replicate(n=5,logLik(pfilter(p0,Np=1000)))
logmeanexp(l1)
## with standard error:
logmeanexp(l1,se=TRUE)
```

measles

\textit{Historical childhood disease incidence data}

Description

\texttt{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).

\texttt{ewmeas} and \texttt{ewcitmeas} are data frames containing weekly reported cases of measles in England and Wales. \texttt{ewmeas} records the total measles reports for the whole country, 1948–1966. One questionable data point has been replaced with an NA. \texttt{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.”
References


See Also

Other datasets: blowflies, bsflu, dacca(), ebola, parus

Other pomp examples: blowflies, bsflu, dacca(), ebola, gompertz(), ou2(), parus, pomp_examples, ricker(), rw2(), sir_models, verhulst()

Examples

```r
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')
```

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

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

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

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

---

**mif2**

*Iterated filtering: maximum likelihood by iterated, perturbed Bayes maps*

**Description**

An 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

## S4 method for signature 'data.frame'
mif2(
data,
Nmif = 1,
rw.sd,
cooling.type = c("geometric", "hyperbolic"),
cooling.fraction.50,
Np,
params,
rinit,
rprocess,
dmeasure,
partrans,
...,  
verbose =getOption("verbose", FALSE)
)

## S4 method for signature 'pomp'
mif2(
data,
Nmif = 1,
rw.sd,
cooling.type = c("geometric", "hyperbolic"),
cooling.fraction.50,
Np,
...,  
verbose =getOption("verbose", FALSE)
)

## S4 method for signature 'pfilterd_pomp'
mif2(data, Nmif = 1, Np, ..., verbose =getOption("verbose", FALSE))

## S4 method for signature 'mif2d_pomp'
mif2(
data,
Nmif,
rw.sd,
cooling.type,
cooling.fraction.50,
...,  
verbose =getOption("verbose", FALSE)
Arguments

- **data**: either a data frame holding the time series data, or an object of class `pomp`, i.e., the output of another `pomp` calculation.

- **Nmif**: The number of filtering iterations to perform.

- **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 `iv(s)` can be used in this context as shorthand for `ifelse(time==time[1],s,0)`.

  Likewise, `iv(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 parameter transformations have been supplied, then the perturbations are applied on the transformed (estimation) scale.

- **cooling.type, cooling.fraction.50**: specifications for the cooling schedule, i.e., the manner and rate with 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.

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

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

- **rinit**: simulator of the initial-state distribution. This can be furnished either as a C snippet, an R function, or the name of a pre-compiled native routine available in a dynamically loaded library. Setting `rinit=NULL` sets the initial-state simulator to its default. For more information, see ?rinit_spec.

- **rprocess**: simulator of the latent state process, specified using one of the `rprocess plugins`. Setting `rprocess=NULL` removes the latent-state simulator. For more information, see ?rprocess_spec for the documentation on these plugins.
dmeasure
evaluator of the measurement model density, specified either as a C snippet, an
R function, or the name of a pre-compiled native routine available in a dynamically
loaded library. Setting dmeasure=NULL removes the measurement density
evaluator. For more information, see ?dmeasure_spec.

partrans
optional parameter transformations, constructed using parameter_trans.
Many algorithms for parameter estimation search an unconstrained space of pa-
rameters. When working with such an algorithm and a model for which the pa-
rameters are constrained, it can be useful to transform parameters. One should
supply the partrans argument via a call to parameter_trans. For more infor-
mation, see ?parameter_trans. Setting partrans=NULL removes the parameter
transformations, i.e., sets them to the identity transformation.

... additional arguments supply new or modify existing model characteristics or
components. See pomp for a full list of recognized arguments.
When named arguments not recognized by pomp are provided, these are made
available to all basic components via the so-called userdata facility. This al-
 lows the user to pass information to the basic components outside of the usual
routes of covariates (covar) and model parameters (params). See ?userdata for
information on how to use this facility.

verbose
logical; if TRUE, diagnostic messages will be printed to the console.

Value
Upon successful completion, mif2 returns an object of class 'mif2d_pomp'.

Number of particles
If Np is anything other than a constant, the user must take care that the number of particles re-
quested at the end of the time series matches that requested at the beginning. In particular, if
T=length(time(object)), then one should have Np[1]==Np[T+1]
when Np is furnished as an
integer vector and Np(0)==Np(T) when Np is furnished as a function.

Methods
The following methods are available for such an object:

continue picks up where mif2 leaves off and performs more filtering iterations.

logLik returns the so-called mif log likelihood which is the log likelihood of the perturbed model,
not of the focal model itself. To obtain the latter, it is advisable to run several pfilter opera-
tions on the result of a mif2 computatation.

coeff extracts the point estimate

eff.sample.size extracts the effective sample size of the final filtering iteration

Various other methods can be applied, including all the methods applicable to a pfilterd_pomp
object and all other pomp estimation algorithms and diagnostic methods.
Specifying the perturbations

The `rw.sd` 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 `cooling.type`="geometric", we have

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

When `cooling.type="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 IF2 iterations

To re-run a sequence of IF2 iterations, one can use the `mif2` method on a `mif2d_pomp` object. By default, the same parameters used for the original IF2 run are re-used (except for `verbose`, the default of which is shown above). If one does specify additional arguments, these will override the defaults.

Author(s)

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

References

**nlf**

*Nonlinear forecasting*

**Description**

Parameter estimation by maximum simulated quasi-likelihood.

**Usage**

```r
## S4 method for signature 'data.frame'
nlf_objfun(
  data,
  est = character(0),
  lags,
  nrbf = 4,
  ti,
  tf,
  seed = NULL,
  transform.data = identity,
  period = NA,
  tensor = TRUE,
  fail.value = NA_real_,
  params,
  rinit,
  rprocess,
  rmeasure,
  ...,,
  verbose = getOption("verbose")
)
```

```r
## S4 method for signature 'pomp'
nlf_objfun(
  data,
  est = character(0),
  lags,
  nrbf = 4,
  ti,
  tf,
  seed = NULL,
  transform.data = identity,
```
period = NA,
tensor = TRUE,
fail.value = NA,
...
verbose = getOption("verbose")
)

## S4 method for signature 'nlf_objfun'
nlf_objfun(
data, 
est, 
lags, 
nrbf, 
ti, 
tf, 
seed = NULL, 
period, 
tensor, 
transform.data, 
fail.value, 
..., 
verbose = getOption("verbose", FALSE)
)

### Arguments

data either a data frame holding the time series data, or an object of class 'pomp', i.e., the output of another pomp calculation.
est character vector; the names 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.
nrbf integer scalar; the number of radial basis functions to be used at each lag.
ti, tf required numeric values. NLF works by generating simulating long time series from the model. The simulated time series will be from ti to tf, with the same sampling frequency as the data. ti should be chosen large enough so that transient dynamics have died away. tf should be chosen large enough so that sufficiently many data points are available to estimate the nonlinear forecasting model well. An error will be generated unless the data-to-parameter ratio exceeds 10 and a warning will be given if the ratio is smaller than 30.
seed integer. When fitting, it is often best to fix the seed of the random-number generator (RNG). This is accomplished by setting seed to an integer. By default, seed = NULL, which does not alter the RNG state.
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 the data are univariate, transform.data should take a scalar and
return a scalar. If the data are multivariate, transform.data should assume a vector input and return a vector of the same length.

period numeric; period=NA means the model is nonseasonal. period > 0 is the period of seasonal forcing. period <= 0 is equivalent to period = NA.

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.

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).

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

rinit simulator of the initial-state distribution. This can be furnished either as a C snippet, an R function, or the name of a pre-compiled native routine available in a dynamically loaded library. Setting rinit=NULL sets the initial-state simulator to its default. For more information, see ?rinit_spec.

rprocess simulator of the latent state process, specified using one of the rprocess plugins. Setting rprocess=NULL removes the latent-state simulator. For more information, see ?rprocess_spec for the documentation on these plugins.

rmeasure simulator of the measurement model, specified either as a C snippet, an R function, or the name of a pre-compiled native routine available in a dynamically loaded library. Setting rmeasure=NULL removes the measurement model simulator. For more information, see ?rmeasure_spec.

... additional arguments supply new or modify existing model characteristics or components. See pomp for a full list of recognized arguments.

When named arguments not recognized by pomp are provided, these are made available to all basic components via the so-called userdata facility. This allows the user to pass information to the basic components outside of the usual routes of covariates (covar) and model parameters (params). See ?userdata for information on how to use this facility.

verbose logical; if TRUE, diagnostic messages will be printed to the console.

Details

Nonlinear forecasting (NLF) is an ‘indirect inference’ method. The NLF approximation to the log likelihood of the data series 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. The nonlinear autoregressive model is implemented as a generalized additive model (GAM), conditional on lagged values, for each observation variable. The errors are assumed multivariate normal.

The NLF objective function constructed by nlf_objfun simulates long time series (nasymp is the number of observations in the simulated times series), perhaps after allowing for a transient period (ntransient steps). It then fits the GAM for the chosen lags to the simulated time series. Finally, it computes the quasi-likelihood of the data under the fitted GAM.
NLF assumes that the observation frequency (equivalently the time between successive observations) is uniform.

Value

nlf_objfun constructs a stateful objective function for NLF estimation. Specifically, nlf_objfun returns an object of class ‘nlf_objfun’, which is a function suitable for use in an optim-like optimizer. In particular, this function takes a single numeric-vector argument that is assumed to contain the parameters named in est, in that order. When called, it will return the negative log quasilikelihood. It is a stateful function: Each time it is called, it will remember the values of the parameters and its estimate of the log quasilikelihood.

Periodically-forced systems (seasonality)

Unlike other pomp estimation methods, NLF cannot accommodate general time-dependence in the model via explicit time-dependence or dependence on time-varying covariates. However, NLF can accommodate periodic forcing. It does this by including forcing phase as a predictor in the nonlinear autoregressive model. To accomplish this, one sets period to the period of the forcing (a positive numerical value). In this case, if tensor = FALSE, the effect is to add a periodic intercept in the autoregressive model. If tensor = TRUE, by contrast, the fitted model includes time-periodic coefficients, constructed using tensor products of basis functions of observables with basis functions of time.

Important Note

Since pomp cannot guarantee that the final call an optimizer makes to the function is a call at the optimum, it cannot guarantee that the parameters stored in the function are the optimal ones. Therefore, it is a good idea to evaluate the function on the parameters returned by the optimization routine, which will ensure that these parameters are stored.

Author(s)

Stephen P. Ellner, Bruce E. Kendall, Aaron A. King

References


See Also

Other pomp parameter estimation methods: abc(), bsmc2(), kalman.mif2(), pmcmc(), pomp-package, probe.match, spect.match
Examples

library(magrittr)

ricker() %>%
  nlf_objfun(est=c("r","sigma","N_0"), lags=c(4,6),
  partrans=parameter_trans(log=c("r","sigma","N_0")),
  paramnames=c("r","sigma","N_0"),
  ti=100, tf=2000, seed=426094906L) -> m1

library(subplex)
subplex(par=log(c(20,0.5,5)), fn=m1, control=list(reltol=1e-4)) -> out

m1(out$par)
coef(m1)
plot(simulate(m1))

Description

Extract the data array from a `pomp` object.

Usage

```r
# S4 method for signature 'pomp'
obs(object, vars, ...)
```

Arguments

- `object`: an object of class `pomp`, or of a class extending `pomp`
- `vars`: names of variables to retrieve
- `...`: ignored
ou2 Two-dimensional discrete-time Ornstein-Uhlenbeck process

Description

ou2() constructs a ‘pomp’ object encoding a bivariate discrete-time Ornstein-Uhlenbeck process with noisy observations.

Usage

```r
ou2(
  alpha_1 = 0.8,
  alpha_2 = -0.5,
  alpha_3 = 0.3,
  alpha_4 = 0.9,
  sigma_1 = 3,
  sigma_2 = -0.5,
  sigma_3 = 2,
  tau = 1,
  x1_0 = -3,
  x2_0 = 4,
  times = 1:100,
  t0 = 0
)
```

Arguments

- `alpha_1, alpha_2, alpha_3, alpha_4` entries of the `alpha` matrix, in column-major order. That is, `alpha_2` is in the lower-left position.
- `sigma_1, sigma_2, sigma_3` entries of the lower-triangular `sigma` matrix. `sigma_2` is the entry in the lower-left position.
- `tau` measurement error s.d.
- `x1_0, x2_0` latent variable values at time `t0`
- `times` vector of observation times
- `t0` the zero time

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 \( y_i(t) \sim \text{normal}(x_i(t), \tau) \).
Value

A ‘pomp’ object with simulated data.

See Also

Other pomp examples: blowflies, bsflu, dacca(), ebola, gompertz(), measles, parus, pomp_examples, ricker(), rw2(), sir_models, verhulst()

Examples

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

parameter_trans  Parameter transformations

Description

Equipping models with parameter transformations.

Usage

```r
## S4 method for signature 'Csnippet,Csnippet'
parameter_trans(toEst, fromEst, ..., log, logit, barycentric)

## S4 method for signature 'missing,missing'
parameter_trans(..., log, logit, barycentric)

## S4 method for signature 'character,character'
parameter_trans(toEst, fromEst, ...)

## S4 method for signature 'function,function'
parameter_trans(toEst, fromEst, ...)
```

Arguments

toEst, fromEst  procedures that perform transformation of model parameters to and from the estimation scale, respectively. These can be furnished using C snippets, R functions, or via procedures in an external, dynamically loaded library.

...  ignored.

log  names of parameters to be log transformed.

logit  names of parameters to be logit transformed.
barycentric names of parameters to be collectively transformed according to the log barycentric transformation. **Important note:** variables to be log-barycentrically transformed must be adjacent in the parameter vector.

**Details**

When parameter transformations are desired, they can be integrated into the `pomp` object via the `partrans` arguments using the `parameter_trans` function. 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, and the inverse transformation, are specified via a call to

   `parameter_trans(toEst, fromEst)`.

2. The goal of these snippets is the transformation of the parameters from the natural scale to the estimation scale, and vice-versa. If \( p \) is the name of a variable on the natural scale, its value on the estimation scale is \( T_p \). Thus the `toEst` snippet computes \( T_p \) given \( p \) whilst the `fromEst` snippet computes \( p \) given \( T_p \).

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 chosen from among the parameters. Such an R function must also have the argument `'...'`. In this case, `toEst` should transform parameters from the scale that the basic components use internally to the scale used in estimation. `fromEst` should be the inverse of `toEst`.

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)
```

One can use the log and logit arguments of `parameter_trans` to name variables that should be log-transformed or logit-transformed, respectively. The barycentric argument can name sets of parameters that should be log-barycentric transformed.

Note that using the log, logit, or barycentric arguments causes C snippets to be generated. Therefore, you must make sure that variables named in any of these arguments are also mentioned in `paramnames` at the same time.

The logit transform is defined by

\[
\text{logit}(\theta) = \log \frac{\theta}{1 - \theta}.
\]

The log barycentric transformation of variables \( \theta_1, \ldots, \theta_n \) is given by

\[
\text{logbarycentric}(\theta_1, \ldots, \theta_n) = \left( \log \frac{\theta_1}{\sum_i \theta_i}, \ldots, \log \frac{\theta_n}{\sum_i \theta_i} \right).
\]
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
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')
```

See Also

Other information on model implementation: Csnippet, accumulators, covariate_table(), distributions, dmeasure_spec, dprocess_spec, pomp-package, prior_spec, rinit_spec, rmeasure_spec, rprocess_spec, skeleton_spec, transformations, userdata
partrans

Description

Performs parameter transformations.

Usage

```r
## S4 method for signature 'pomp'
partrans(object, params, dir = c("fromEst", "toEst"), ...)
```

Arguments

- `object`: an object of class ‘pomp’, or of a class that extends ‘pomp’. This will typically be the output of `pomp`, `simulate`, or one of the `pomp` inference algorithms.
- `params`: a `npar x nrep` matrix of parameters. Each column is treated as an independent parameter set, in correspondence with the corresponding column of `x`.
- `dir`: the direction of the transformation to perform.
- `...`: additional arguments are ignored.

Value

If `dir=fromEst`, the parameters in `params` are assumed to be on the estimation scale and are transformed onto the natural scale. If `dir=toEst`, they are transformed onto the estimation scale. In both cases, the parameters are returned as a named numeric vector or an array with rownames, as appropriate.

See Also

Specification of parameter transformations: `parameter_trans`
Other pomp workhorses: `dmeasure()`, `dprior()`, `dprocess()`, `flow()`, `rinit()`, `rmeasure()`, `rprior()`, `rprocess()`, `skeleton()`, `workhorses`

parus

Parus major population dynamics

Description

Size of a population of great tits (Parus major) from Wytham Wood, near Oxford.

Details

References


See Also

Other datasets: blowflies, bsflu, dacca(), ebola, measles

Other pomp examples: blowflies, bsflu, dacca(), ebola.gompertz(), measles.ou2(), pomp_examples, ricker(), rw2(), sir_models, verhulst()

Examples

```r
parus %>%
pfilter(Np=1000, times="year", t0=1960, 
params=c(K=190, r=2.7, sigma=0.2, theta=0.05, N.0=148),
 rprocess=discrete_time(
   function (r, K, sigma, N, ...) {
     N = c(N = exp(log(N)+r*(1-N/K)+e))
   },
   delta.t=1
 ),
 rmeasure=function (N, theta, ...) {
     c(pop=rnbinom(n=1,size=1/theta,mu=N+1e-10))
 },
 dmeasure=function (pop, N, theta, ..., log) {
     dnbinom(x=pop,mu=N+1e-10,size=1/theta,log=log)
 },
 partrans=parameter_trans(log=c("sigma","theta","N.0","r","K")),
 paramnames=c("sigma","theta","N.0","r","K")
 ) -> pf

pf %>% logLik()

pf %>% simulate() %>% plot()
```

---

**pfilter**

*Particle filter*

**Description**

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

```r
## S4 method for signature 'data.frame'
pfilter(
data,  
Np,  
params,  
rinit,  
rprocess,  
dmeasure,  
pred.mean = FALSE,  
pred.var = FALSE,  
filter.mean = FALSE,  
filter.traj = FALSE,  
save.states = FALSE,  
...,  
verbose =getOption("verbose", FALSE)
)
```

```r
## S4 method for signature 'pomp'
pfilter(
data,  
Np,  
pred.mean = FALSE,  
pred.var = FALSE,  
filter.mean = FALSE,  
filter.traj = FALSE,  
save.states = FALSE,  
...,  
verbose =getOption("verbose", FALSE)
)
```

```r
## S4 method for signature 'pfilterd_pomp'
pfilter(data, Np, ..., verbose =getOption("verbose", FALSE))
```

```r
## S4 method for signature 'objfun'
pfilter(data, ...)
```

Arguments

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

**params**

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

**rinit**

simulator of the initial-state distribution. This can be furnished either as a C snippet, an R function, or the name of a pre-compiled native routine available in a dynamically loaded library. Setting \( \text{rinit=NULL} \) sets the initial-state simulator to its default. For more information, see \(?\text{rinit_spec}\).

**rprocess**

simulator of the latent state process, specified using one of the rprocess plugins. Setting \( \text{rprocess=NULL} \) removes the latent-state simulator. For more information, see \(?\text{rprocess_spec}\) for the documentation on these plugins.

**dmeasure**

evaluator of the measurement model density, specified either as a C snippet, an R function, or the name of a pre-compiled native routine available in a dynamically loaded library. Setting \( \text{dmeasure=NULL} \) removes the measurement density evaluator. For more information, see \(?\text{dmeasure_spec}\).

**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. See \texttt{filter.traj} for more information.

**save.states**

logical. If \( \text{save.states=TRUE} \), the state-vector for each particle at each time is saved.

... additional arguments supply new or modify existing model characteristics or components. See \texttt{pomp} for a full list of recognized arguments.

When named arguments not recognized by \texttt{pomp} are provided, these are made available to all basic components via the so-called \texttt{userdata} facility. This allows the user to pass information to the basic components outside of the usual routes of covariates (\texttt{covar}) and model parameters (\texttt{params}). See \texttt{?userdata} for information on how to use this facility.

**verbose**

logical; if TRUE, diagnostic messages will be printed to the console.

**Value**

An object of class \('\text{pfilterd_pomp}\)', which extends class \('\text{pomp}'\). Information can be extracted from this object using the methods documented below.

**Methods**

\texttt{logLik} \ \texttt{the estimated log likelihood}
cond.logLik  the estimated conditional log likelihood
eff.sample.size  the (time-dependent) estimated effective sample size
pred.mean, pred.var  the mean and variance of the approximate prediction distribution
filter.mean  the mean of the filtering distribution
filter.traj  retrieve one particle trajectory. Useful for building up the smoothing distribution.
saved.states  retrieve list of saved states.
as.data.frame  coerce to a data frame
plot  diagnostic plots

Author(s)

Aaron A. King

References


See Also

Other elementary POMP methods: pomp-package, probe(), simulate(), spect(), wpfilter()

Other particle filter methods: bsmc2(), cond.logLik(), eff.sample.size(), filter.mean(),
filter.traj(), mif2(), pmcmc(), pred.mean(), pred.var(), saved.states(), wpfilter()

Examples

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, filter.traj=TRUE, save.states=TRUE)
fm <- filter.mean(pf)  ## extract the filtering means
ft <- filter.traj(pf)  ## one draw from the smoothing distribution
ss <- saved.states(pf)  ## the latent-state portion of each particle
## Plotting

### Description

Diagnostic plots.

### Usage

```
## S4 method for signature 'pomp_plottable'
plot(
    x,
    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 'Pmcmc'
plot(x, ..., pars)
```

```
## S4 method for signature 'Abc'
plot(x, ..., pars, scatter = FALSE)
```

```
## S4 method for signature 'Mif2'
plot(x, y, ...)
```

```
## S4 method for signature 'probed_pomp'
plot(x, y, ...)
```

```
## S4 method for signature 'spectd_pomp'
plot(
    x,
    ..., 
    max.plots.per.page = 4,
    plot.data = TRUE,
    quantiles = c(0.025, 0.25, 0.5, 0.75, 0.975),
    quantile.styles = list(lwd = 1, lty = 1, col = "gray70"),
    data.styles = list(lwd = 2, lty = 2, col = "black")
)
```

```
## S4 method for signature 'bsmcd_pomp'
plot(x, pars, thin, ...)
```
## S4 method for signature 'probe_match_objfun'
plot(x, y, ...)

## S4 method for signature 'spect_match_objfun'
plot(x, y, ...)

### Arguments

- **x**: the object to plot
- **variables**: optional character; names of variables to be displayed
- **panel**: 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 mar` and `oma` settings. Modify with care!
- **axes**: logical; indicates if x- and y- axes should be drawn
- **...**: ignored or passed to low-level plotting functions
- **pars**: names of parameters.
- **scatter**: logical; if FALSE, traces of the parameters named in pars will be plotted against ABC iteration number. If TRUE, the traces will be displayed or as a scatterplot.
- **y**: ignored
- **max.plots.per.page**: positive integer; maximum number of plots on a page
- **plot.data**: logical; should the data spectrum be included?
- **quantiles**: numeric; quantiles to display
- **quantile.styles**: list; plot styles to use for quantiles
- **data.styles**: list; plot styles to use for data
- **thin**: integer; when the number of samples is very large, it can be helpful to plot a random subsample: thin specifies the size of this subsample.

### 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 'data.frame'
pmcmc(
  data,
  Nmcmc = 1,
  proposal,
  Np,
  params,
  rinit,
  rprocess,
  dmeasure,
  dprior,
  ...
  ,
  verbose = getOption("verbose", FALSE)
)
```

```r
## S4 method for signature 'pomp'
pmcmc(
  data,
  Nmcmc = 1,
  proposal,
  Np,
  ..., 
  verbose = getOption("verbose", FALSE)
)
```

```r
## S4 method for signature 'pfilterd_pomp'
pmcmc(
  data,
  Nmcmc = 1,
  proposal,
  Np,
  ..., 
  verbose = getOption("verbose", FALSE)
)
```

```r
## S4 method for signature 'pmcmcd_pomp'
pmcmc(data, Nmcmc, proposal, ..., verbose = getOption("verbose", FALSE))
```

Arguments

data either a data frame holding the time series data, or an object of class ‘pomp’, i.e., the output of another pomp calculation.

Nmcmc The number of PMCMC iterations to perform.

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 proposals for more information.
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. Al-
ternatively, if one wishes the number of particles to vary across timesteps, one
may specify \( N_p \) either as a vector of positive integers of length

\[
\text{length(time(object, t0=TRUE))}
\]
or as a function taking a positive integer argument. In the latter case, \( N_p(k) \)
must be a single positive integer, representing the number of particles to be
used at the \( k \)-th timestep: \( N_p(0) \) is the number of particles to use going from
timezero(object) to time(object)[1], \( N_p(1) \), from timezero(object) to
time(object)[1], and so on, while when \( T=\text{length(time(object))} \), \( N_p(T) \)
is the number of particles to sample at the end of the time-series.

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

**rinit**
simulator of the initial-state distribution. This can be furnished either as a C
snippet, an \( \mathcal{R} \) function, or the name of a pre-compiled native routine available in
a dynamically loaded library. Setting \( \text{rinit=NULL} \) sets the initial-state simulator
to its default. For more information, see ?rinit_spec.

**rprocess**
simulator of the latent state process, specified using one of the rprocess plugins.
Setting \( \text{rprocess=NULL} \) removes the latent-state simulator. For more information,
see ?rprocess_spec for the documentation on these plugins.

**dmeasure**
evaluator of the measurement model density, specified either as a C snippet, an
\( \mathcal{R} \) function, or the name of a pre-compiled native routine available in a dynami-
cally loaded library. Setting \( \text{dmeasure=NULL} \) removes the measurement density
evaluator. For more information, see ?dmeasure_spec.

**dprior**
optional; prior distribution density evaluator, specified either as a C snippet, an \( \mathcal{R} \)
function, or the name of a pre-compiled native routine available in a dynamically
loaded library. For more information, see ?prior_spec. Setting \( \text{dprior=NULL} \)
resets the prior distribution to its default, which is a flat improper prior.

**...**
additional arguments supply new or modify existing model characteristics or
components. See pomp for a full list of recognized arguments.
When named arguments not recognized by pomp are provided, these are made
available to all basic components via the so-called userdata facility. This al-
 lows the user to pass information to the basic components outside of the usual
routes of covariates (covar) and model parameters (params). See ?userdata for
information on how to use this facility.

**verbose**
logical; if TRUE, diagnostic messages will be printed to the console.

**Value**
An object of class ‘pmcmcd_pomp’.

**Re-running PMCMC Iterations**
To re-run a sequence of PMCMC iterations, one can use the pmcmc method on a ‘pmcmc’ object.
By default, the same parameters used for the original PMCMC run are re-used (except for verbose,
the default of which is shown above). If one does specify additional arguments, these will override
the defaults.
Constructor of the basic pomp object

Description

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

- **rinit**, which samples from the distribution of the state process at the zero-time;
- **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;
- **rprior**, which samples from a prior probability distribution on the parameters;
- **dprior**, which evaluates the prior probability density function;
- **skeleton**, which computes the deterministic skeleton of the unobserved state process;
- **partrans**, which performs parameter transformations.

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,  
...,  
rinit,  
rprocess,  
dprocess,  
rmeasure,  
dmeasure,  
skeleton,  
rprior,  
dprior,  
partrans,  
covar,  
params,  
accumvars,  
obsnames,  
statenames,  
paramnames,  
covarnames,  
PACKAGE,  
globals,  
cdir = getOption("pomp_cdir", NULL),  
cfile,  
shlib.args,  
compile = TRUE,  
verbose = getOption("verbose", FALSE)
)
```

Arguments

data: either a data frame holding the time series data, or an object of class ‘pomp’, i.e., the output of another pomp calculation.
times: the times at which observations are made. times must indicate the column of observation times by name or index. The time vector must be numeric and non-decreasing. Internally, data will be internally coerced to an array with storage-mode double.
t0: The zero-time, i.e., the time of the initial state. This must be no later than the time of the first observation, i.e., \( t0 \leq \text{times}[1] \).

... additional arguments supply new or modify existing model characteristics or components. See pomp for a full list of recognized arguments.

When named arguments not recognized by pomp are provided, these are made available to all basic components via the so-called userdata facility. This allows the user to pass information to the basic components outside of the usual
routes of covariates (covar) and model parameters (params). See ?userdata for information on how to use this facility.

**rinit**
simulator of the initial-state distribution. This can be furnished either as a C snippet, an R function, or the name of a pre-compiled native routine available in a dynamically loaded library. Setting rinit=NULL sets the initial-state simulator to its default. For more information, see ?rinit_spec.

**rprocess**
simulator of the latent state process, specified using one of the rprocess plugins. Setting rprocess=NULL removes the latent-state simulator. For more information, see ?rprocess_spec for the documentation on these plugins.

**dprocess**
optional; specification of the probability density evaluation function of the unobserved state process. Setting dprocess=NULL removes the latent-state density evaluator. For more information, see ?dprocess_spec.

**rmeasure**
simulator of the measurement model, specified either as a C snippet, an R function, or the name of a pre-compiled native routine available in a dynamically loaded library. Setting rmeasure=NULL removes the measurement model simulator. Setting dmeasure=NULL removes the measurement density evaluator. For more information, see ?dmeasure_spec.

**dmeasure**
evaluator of the measurement model density, specified either as a C snippet, an R function, or the name of a pre-compiled native routine available in a dynamically loaded library. Setting dmeasure=NULL removes the measurement density evaluator. For more information, see ?dmeasure_spec.

**skeleton**
optional; the deterministic skeleton of the unobserved state process. Depending on whether the model operates in continuous or discrete time, this is either a vectorfield or a map. Accordingly, this is supplied using either the vectorfield or map functions. For more information, see ?skeleton_spec. Setting skeleton=NULL removes the deterministic skeleton.

**rprior**
optional; prior distribution sampler, specified either as a C snippet, an R function, or the name of a pre-compiled native routine available in a dynamically loaded library. Setting rprior=NULL removes the prior distribution sampler.

**dprior**
optional; prior distribution density evaluator, specified either as a C snippet, an R function, or the name of a pre-compiled native routine available in a dynamically loaded library. Setting dprior=NULL resets the prior distribution to its default, which is a flat improper prior.

**partrans**
optional parameter transformations, constructed using parameter_trans. 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. One should supply the partrans argument via a call to parameter_trans. For more information, see ?parameter_trans. Setting partrans=NULL removes the parameter transformations, i.e., sets them to the identity transformation.

**covar**
optional covariate table, constructed using covariate_table. 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. See the documentation for covariate_table for details.
params: optional; named numeric vector of parameters. This will be coerced internally to storage mode `double`.

accumvars: optional character vector; contains the names of accumulator variables. See `?accumulators` for a definition and discussion of accumulator variables.

obsnames: optional character vector; names of the observables. It is not usually necessary to specify `obsnames` since, by default, these are read from the names of the data variables.

statenames: optional character vector; names of the latent state variables. It is typically only necessary to supply `statenames` when C snippets are in use.

paramnames: optional character vector; names of model parameters. It is typically only necessary to supply `paramnames` when C snippets are in use.

covarnames: optional character vector; names of the covariates. It is not usually necessary to specify `covarnames` since, by default, these are read from the names of the covariates.

PACKAGE: optional character; the name (without extension) of the external, 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 used for any component specified using C snippets. PACKAGE can name at most one library.

globals: optional character; arbitrary C code that will be hard-coded into the shared-object library created when C snippets are provided. If no C snippets are used, `globals` has no effect.

cdir: optional character variable. `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 R session. One can also set this directory using the `pomp_cdir` option.

cfile: optional character variable. `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. If the chosen filename would result in over-writing an existing file, an error is generated.

shlib.args: optional character variables. Command-line arguments to the `R CMD SHLIB` call that compiles the C snippets.

compile: logical; if `FALSE`, compilation of the C snippets will be postponed until they are needed.

verbose: logical; if `TRUE`, diagnostic messages will be printed to the console.

**Details**

Each basic component is supplied via an argument of the same name. These can be given in the call to `pomp`, or to many of the package’s other functions. In any case, the effect is the same: to add, remove, or modify the basic component.

Each basic component can be furnished using C snippets, R functions, or pre-compiled native routine available in user-provided dynamically loaded libraries.
Value

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. One can pass \( P \) to most of the pomp package methods via their data argument.

Note

It is not typically necessary (or indeed often 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. FIXME

Author(s)

Aaron A. King

References


Description

Pre-built POMP examples

Details

pomp includes a number of pre-built examples of pomp objects and data that can be analyzed using pomp methods. These include:

- blowflies Data from Nicholson’s experiments with sheep blowfly populations
- blowflies1() A pomp object with some of the blowfly data together with a discrete delay equation model.
- blowflies2() A variant of blowflies1.
- bsflu Data from an outbreak of influenza in a boarding school.
- dacca() Fifty years of census and cholera mortality data, together with a stochastic differential equation transmission model (King et al. 2008).
- ebolaModel() Data from the 2014 West Africa outbreak of Ebola virus disease, together with simple transmission models (King et al. 2015).
- gompertz() The Gompertz population dynamics model, with simulated data.
**LondonYorke**  Data on incidence of several childhood diseases (London and Yorke 1973)

**ewmeas**  Measles incidence data from England and Wales

**ewcitmeas**  Measles incidence data from 7 English cities

**ou2()**  A 2-D Ornstein-Uhlenbeck process with simulated data


**ricker**  The Ricker population dynamics model, with simulated data

**rw2**  A 2-D Brownian motion model, with simulated data.

**sir()**  A simple continuous-time Markov chain SIR model, coded using Euler-multinomial steps, with simulated data.

**sir2()**  A simple continuous-time Markov chain SIR model, coded using Gillespie’s algorithm, with simulated data.

**verhulst()**  The Verhulst-Pearl (logistic) model, a continuous-time model of population dynamics, with simulated data

See also the tutorials on the package website for more examples.

**References**


**See Also**

Other pomp examples: *blowflies, bsflu, dacca*, *ebola, gompertz*, *measles, ou2, parus, ricker, rw2, sir_models, verhulst*
Usage

```r
## S4 method for signature 'kalmand_pomp'
pred.mean(object, vars, ...)
```

```r
## S4 method for signature 'pfilterd_pomp'
pred.mean(object, vars, ...)
```

Arguments

- **object**: result of a filtering computation
- **vars**: optional character; names of variables
- **...**: ignored

Details

The prediction distribution is that of

\[ X(t_k)|Y(t_1) = y_1^*, \ldots, Y(t_{k-1}) = y_{k-1}^* \]

where \( X(t_k), Y(t_k) \) are the latent state and observable processes, respectively, and \( y_k^* \) is the data, at time \( t_k \).

The prediction mean is therefore the expectation of this distribution

\[ E[X(t_k)|Y(t_1) = y_1^*, \ldots, Y(t_{k-1}) = y_{k-1}^*]. \]

See Also

Other particle filter methods: `bsmc2()`, `cond.logLik()`, `eff.sample.size()`, `filter.mean()`, `filter.traj()`, `mif2()`, `pfilter()`, `pmcmc()`, `pred.var()`, `saved.states()`, `wpfilter()`

---

**pred.var**

*Prediction variance*

Description

The variance of the prediction distribution

Usage

```r
## S4 method for signature 'pfilterd_pomp'
pred.var(object, vars, ...)
```

Arguments

- **object**: result of a filtering computation
- **vars**: optional character; names of variables
- **...**: ignored
Details

The prediction distribution is that of

\[ X(t_k)|Y(t_1) = y_1^*, \ldots, Y(t_{k-1}) = y_{k-1}^* , \]

where \( X(t_k), Y(t_k) \) are the latent state and observable processes, respectively, and \( y_k^* \) is the data, at time \( t_k \).

The prediction variance is therefore the variance of this distribution

\[ \text{Var}[X(t_k)|Y(t_1) = y_1^*, \ldots, Y(t_{k-1}) = y_{k-1}^*]. \]

See Also

Other particle filter methods: `bsmc2()`, `cond.logLik()`, `eff.sample.size()`, `filter.mean()`, `filter.traj()`, `mif2()`, `pfilter()`, `pmcmc()`, `pred.mean()`, `saved.states()`, `wpfilter()`
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 `give_log == 1`, `lik` should contain the log of the prior probability density.

3. Hyperparameters can be included in the ordinary parameter list.

**General rules for writing C snippets can be found here.**

Alternatively, one can furnish R functions for one or both of these arguments. In this case, `r prior` must be a function of prototype

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

that makes a draw from the prior distribution given `params` and returns a named vector of the same length and with the same set of names, as `params`. The `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 `log == TRUE`) and return that single scalar value.

**Default behavior**

By default, the prior is assumed flat and improper. In particular, `dprior` returns 1 (0 if `log = TRUE`) for every parameter set. Since it is impossible to simulate from a flat improper prior, `rprocess` returns missing values (`NA`s).

**See Also**

Other information on model implementation: `Csnippet, accumulators, covariate_table(), distributions, dmeasure_spec, dprocess_spec, parameter_trans(), pomp-package, rinit_spec, rmeasure_spec, rprocess_spec, skeleton_spec, transformations, userdata`

---

**probe**

Probes (AKA summary statistics)

**Description**

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

**Usage**

```r
## S4 method for signature 'data.frame'
probe(
  data,
  probes,
  nsim,
  seed = NULL,
)```

---
probe

params,
  rinit,
  rprocess,
  rmeasure,
  ...,  
  verbose = getOption("verbose", FALSE)
)

## S4 method for signature 'pomp'
probe(
  data,
  probes,
  nsim,
  seed = NULL,
  ...,  
  verbose = getOption("verbose", FALSE)
)

## S4 method for signature 'probed_pomp'
probe(
  data,
  probes,
  nsim,
  seed = NULL,
  ...,  
  verbose = getOption("verbose", FALSE)
)

## S4 method for signature 'probe_match_objfun'
probe(data, seed, ..., verbose = getOption("verbose", FALSE))

## S4 method for signature 'objfun'
probe(data, seed = NULL, ...)

Arguments

data either a data frame holding the time series data, or an object of class 'pomp', i.e., the output of another pomp calculation.

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 probes are provided with the package: see basic probes.

nsim the number of model simulations to be computed.

seed optional integer; if non-NULL, the random number generator will be initialized with this seed for simulations. See simulate.

params optional; named numeric vector of parameters. This will be coerced internally to storage mode double.
**rinit** simulator of the initial-state distribution. This can be furnished either as a C snippet, an R function, or the name of a pre-compiled native routine available in a dynamically loaded library. Setting rinit=NULL sets the initial-state simulator to its default. For more information, see ?rinit_spec.

**rprocess** simulator of the latent state process, specified using one of the rprocess plugins. Setting rprocess=NULL removes the latent-state simulator. For more information, see ?rprocess_spec for the documentation on these plugins.

**rmeasure** simulator of the measurement model, specified either as a C snippet, an R function, or the name of a pre-compiled native routine available in a dynamically loaded library. Setting rmeasure=NULL removes the measurement model simulator. For more information, see ?rmeasure_spec.

... additional arguments supply new or modify existing model characteristics or components. See pomp for a full list of recognized arguments.

When named arguments not recognized by pomp are provided, these are made available to all basic components via the so-called userdata facility. This allows the user to pass information to the basic components outside of the usual routes of covariates (covar) and model parameters (params). See ?userdata for information on how to use this facility.

**verbose** logical; if TRUE, diagnostic messages will be printed to the console.

**Details**

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.

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’.

When probe operates on a probe-matching objective function (a ‘probe_match_objfun’ object), by default, the random-number generator seed is fixed at the value given when the objective function was constructed. Specifying NULL or an integer for seed overrides this behavior.

**Value**

probe returns an object of class ‘probed_pomp’, which contains the data and the model, together with the results of the probe calculation.

**Methods**

The following methods are available.

**plot** displays diagnostic plots.

**summary** displays summary information. The summary includes quantiles (fractions of simulations with probe values less than those realized on the data) and the corresponding two-sided p-values. In addition, the “synthetic likelihood” (Wood 2010) is computed, under the assumption that the probe values are multivariate-normally distributed.
**probe.match**

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

as.data.frame coerces a `probed_pomp` to a `data.frame`. The latter contains the realized values of the probes on the data and on the simulations. The variable `.id` indicates whether the probes are from the data or simulations.

**Author(s)**

Daniel C. Reuman, Aaron A. King

**References**


**See Also**

Other elementary POMP methods: `pfilter()`, `pomp-package`, `simulate()`, `spect()`, `wpfilter()`

Other summary statistics methods: `abc()`, `basic_probes`, `probe.match`, `spect()`

---

**probe.match**

*Probe matching*

**Description**

Estimation of parameters by maximum synthetic likelihood

**Usage**

```r
## S4 method for signature 'data.frame'
probe_objfun(
    data,
    est = character(0),
    fail.value = NA,
    probes,
    nsim,
    seed = NULL,
    params,
    rinit,
    rprocess,
    rmeasure,
    partrans,
    ...
    verbose = getOption("verbose", FALSE)
```

---

## probe.match

**Probe matching**

Estimation of parameters by maximum synthetic likelihood

**Usage**

```r
## S4 method for signature 'data.frame'
probe_objfun(
    data,
    est = character(0),
    fail.value = NA,
    probes,
    nsim,
    seed = NULL,
    params,
    rinit,
    rprocess,
    rmeasure,
    partrans,
    ...
    verbose = getOption("verbose", FALSE)
```
### S4 method for signature 'pomp'

```r
probe_objfun(
  data,
  est = character(0),
  fail.value = NA,
  probes,
  nsim,
  seed = NULL,
  ...
)
```

### S4 method for signature 'probed_pomp'

```r
probe_objfun(
  data,
  est = character(0),
  fail.value = NA,
  probes,
  nsim,
  seed = NULL,
  ...
)
```

### S4 method for signature 'probe_match_objfun'

```r
probe_objfun(
  data,
  est,
  fail.value,
  seed = NULL,
  ...
)
```

#### Arguments

- **data**: either a data frame holding the time series data, or an object of class ‘pomp’, i.e., the output of another `pomp` calculation.
- **est**: character vector; the names of parameters to be estimated.
- **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).
- **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 probes are provided with the package: see basic probes.
**probe.match**

- **nsim**
  - the number of model simulations to be computed.

- **seed**
  - integer. When fitting, it is often best to fix the seed of the random-number generator (RNG). This is accomplished by setting seed to an integer. By default, seed = NULL, which does not alter the RNG state.

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

- **rinit**
  - simulator of the initial-state distribution. This can be furnished either as a C snippet, an R function, or the name of a pre-compiled native routine available in a dynamically loaded library. Setting rinit=NULL sets the initial-state simulator to its default. For more information, see ?rinit_spec.

- **rprocess**
  - simulator of the latent state process, specified using one of the rprocess plugins. Setting rprocess=NULL removes the latent-state simulator. For more information, see ?rprocess spec for the documentation on these plugins.

- **rmeasure**
  - simulator of the measurement model, specified either as a C snippet, an R function, or the name of a pre-compiled native routine available in a dynamically loaded library. Setting rmeasure=NULL removes the measurement model simulator. For more information, see ?rmeasure spec.

- **partrans**
  - optional parameter transformations, constructed using parameter_trans. 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. One should supply the partrans argument via a call to parameter_trans. For more information, see ?parameter_trans. Setting partrans=NULL removes the parameter transformations, i.e., sets them to the identity transformation.

Additional arguments supply new or modify existing model characteristics or components. See pomp for a full list of recognized arguments.

When named arguments not recognized by pomp are provided, these are made available to all basic components via the so-called userdata facility. This allows the user to pass information to the basic components outside of the usual routes of covariates (covar) and model parameters (params). See ?userdata for information on how to use this facility.

- **verbose**
  - logical; if TRUE, diagnostic messages will be printed to the console.

**Details**

In probe-matching, one attempts to minimize the discrepancy between simulated and actual data, as measured by a set of summary statistics called probes. In pomp, this discrepancy is measured using the “synthetic likelihood” as defined by Wood (2010).

**Value**

probe_objfun constructs a stateful objective function for probe matching. Specifically, probe_objfun returns an object of class ‘probe_match_objfun’, which is a function suitable for use in an optim-like optimizer. In particular, this function takes a single numeric-vector argument that is assumed to contain the parameters named in est, in that order. When called, it will return the negative synthetic log likelihood for the probes specified. It is a stateful function: Each time it is called, it will remember the values of the parameters and its estimate of the synthetic likelihood.
Important Note

Since pomp cannot guarantee that the final call an optimizer makes to the function is a call at the optimum, it cannot guarantee that the parameters stored in the function are the optimal ones. Therefore, it is a good idea to evaluate the function on the parameters returned by the optimization routine, which will ensure that these parameters are stored.

Author(s)

Aaron A. King

See Also

optim subplex nloptr

Other summary statistics methods: abc(), basic_probes, probe(), spect()  
Other pomp parameter estimation methods: abc(), bsmc2(), kalman, mif2(), nlf, pmcmc(), pomp-package, spect.match

Examples

library(magrittr)

gompertz() -> po

## A list of probes:
plist <- list(
  mean=probe.mean("Y", trim=0.1, transform=sqrt),
  sd=probe.sd("Y", transform=sqrt),
  probe.marginal("Y", ref=obs(po)),
  probe.acf("Y", lags=c(1,3,5), type="correlation", transform=sqrt),
  probe.quantile("Y", prob=c(0.25,0.75), na.rm=TRUE)
)

## Construct the probe-matching objective function.
## Here, we just want to estimate 'K'.
po %>%
  probe_objfun(probes=plist, nsim=100, seed=5069977, est="K") -> f

## Any numerical optimizer can be used to minimize 'f'.
library(subplex)

subplex(fn=f, par=0.4, control=list(reltol=1e-5)) -> out

## Call the objective one last time on the optimal parameters:
f(out$par)

## There are 'plot' and 'summary' methods:
f %>% as("probed_pomp") %>% plot()
f %>% summary()
f %>% probe() %>% plot()

## One can modify the objective function with another call
## to 'probe_objfun':

f %>% probe_objfun(est=c("r","K")) -> f1

---

### MCMC proposal distributions

#### Description

Functions to construct proposal distributions for use with MCMC methods.

#### Usage

- `mvn.diag.rw(rw.sd)`
- `mvn.rw(rw.var)`
- `mvn.rw.adaptive(
  rw.sd,
  rw.var,
  scale.start = NA,
  scale.cooling = 0.999,
  shape.start = NA,
  target = 0.234,
  max.scaling = 50
)

#### Arguments

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

Author(s)

Aaron A. King, Sebastian Funk

References


See Also

pmcmc, abc

Description

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

Usage

ricker(r = exp(3.8), sigma = 0.3, phi = 10, c = 1, N_0 = 7)

Arguments

r intrinsic growth rate
sigma environmental process noise s.d.
phi sampling rate
c density dependence parameter
N_0 initial condition

Details

The state process is $N_{t+1} = rN_t \exp(-cN_t + e_t)$, 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($\phi N_t$).

Value

A ‘pomp’ object containing the Ricker model and simulated data.
See Also

Other pomp examples: blowflies, bsflu, dacca(), ebola, gompertz(), measles, ou2(), parus, pomp_examples, rw2(), sir_models, verhulst()

Examples

```r
po <- ricker()
plot(po)
coef(po)
simulate(po) %>% plot()
```

Description

Samples from the initial-state distribution.

Usage

```r
## S4 method for signature 'pomp'
rintit(object, params, t0, nsim = 1, ...)
```

Arguments

- **object**: an object of class ‘pomp’, or of a class that extends ‘pomp’. This will typically be the output of pomp, simulate, or one of the pomp inference algorithms.
- **params**: a npar x nrep matrix of parameters. Each column is treated as an independent parameter set, in correspondence with the corresponding column of x.
- **t0**: the initial time, i.e., the time corresponding to the initial-state distribution.
- **nsim**: optional integer; the number of initial states to simulate per column of params.
- **...**: additional arguments are ignored.

Value

`rintit` returns an nvar x nsim*ncol(params) 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 constructed.

See Also

Specification of the initial-state distribution: rinit_spec

Other pomp workhorses: dmeasure(), dprior(), dprocess(), flow(), partrans(), rmeasure(), rprior(), rprocess(), skeleton(), workhorses
The initial-state distribution

Description

Specification of rinit

Details

To fully specify the unobserved Markov state process, one must give its distribution at the zero-time ($t_0$). One does this by furnishing a value for the rinit argument. As usual, this can be provided either as a C snippet or as an 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. NB: The statenames argument plays a particularly important role when the rinit is specified using a C snippet. In particular, every state variable must be named in statenames. Failure to follow this rule will result in undefined behavior.

General rules for writing C snippets can be found here.

If an R function is to be used, pass

\[ rinit = f \]

to pomp, where $f$ is a function with arguments that can include the initial time $t_0$, any of the model parameters, and any covariates. As usual, $f$ may take additional arguments, provided these are passed along with it in the call to 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 rinit can be either deterministic (as in the default) or stochastic. In the latter case, it samples from the distribution of the state process at the zero-time, $t_0$.

Default behavior

By default, pomp assumes that the initial distribution is concentrated on a single point. In particular, any parameters in params, the names of which end in “\_0” or “.0”, 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 suffix.

See Also

Other information on model implementation: Csnippet, accumulators, covariate_table(), distributions, dmeasure_spec, dprocess_spec, parameter_trans(), pomp-package, prior_spec, rmeasure_spec, rprocess_spec, skeleton_spec, transformations, userdata
Examples

```r
## We set up a trivial process model:
trivial <- function (X, Y, ...) {
  c(X = X+1, Y = Y-1)
}

## We specify \code{rinit} with a function that
## sets state variables X and Y to the values in
## parameters X0, Y0:
f <- function (X0, Y0, ...) {
  c(X = X0, Y = Y0)
}

plot(simulate(times=1:5,t0=0,params=c(X0=3,Y0=-7),
             rinit=f,rprocess=onestep(trivial)))

## A function that depends on covariate P and
## time t0, as well as parameter X0:
g <- function (t0, X0, P, ...) {
  c(X = X0, Y = P + sin(2*pi*t0))
}

plot(simulate(times=1:5,t0=0,params=c(X0=3,Y0=-7),
             covar=covariate_table(t=0:10,P=3:13,times="t"),
             rinit=g,rprocess=onestep(trivial)))
```

Description

Sample from the measurement model distribution, given values of the latent states and the parameters.

Usage

```r
## S4 method for signature 'pomp'
rmeasure(object, x, times, params, ...)
```

Arguments

- **object**: an object of class ‘pomp’, or of a class that extends ‘pomp’. This will typically be the output of pomp, simulate, or one of the pomp inference algorithms.
- **x**: an 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. One can also pass x as a named numeric vector, which is equivalent to the nrep=1, ntimes=1 case.
times: 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 treated as an independent parameter set, in correspondence with the corresponding column of x.

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

See Also
Specification of the measurement-model simulator: rmeasure_spec
Other pomp workhorses: dmeasure(), dprior(), dprocess(), flow(), partrans(), rinit(), rprior(), rprocess(), skeleton(), workhorses

rmeasure_spec

The measurement-model simulator

Description
Specification of rmeasure

Details
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 rmeasure and dmeasure arguments.

Suppose you have a procedure to simulate observations given the value of the latent state variables. Then you can furnish

rmeasure = f
to pomp algorithms, where f is a C snippet or R function that implements your procedure.

Using a C snippet is much preferred, due to its much greater computational efficiency. See Csnippet for general rules on writing C snippets.

In writing an 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.

The demos and the tutorials on the package website give examples as well.

It is also possible, though far less efficient, to specify rmeasure using an R function. In this case, specify the measurement model simulator by furnishing
$$rmeasure = f$$

to pomp, where \( f \) is an R function. The arguments of \( f \) should be chosen from among the state variables, parameters, covariates, and time. It must also have the argument \( \ldots \) \( f \) must return a named numeric vector of length equal to the number of observable variables.

**Default behavior**

The default \( rmeasure \) is undefined. It will yield missing values (NA).

**See Also**

Other information on model implementation: Csnippet, accumulators, covariate_table(), distributions, dmeasure_spec, dprocess_spec, parameter_trans(), pomp-package, prior_spec, rinit_spec, rprocess_spec, skeleton_spec, transformations, userdata

---

**Description**

Sample from the prior probability distribution.

**Usage**

```r
## S4 method for signature 'pomp'
rprior(object, params, ...)
```

**Arguments**

- `object`: an object of class ‘pomp’, or of a class that extends ‘pomp’. This will typically be the output of pomp, simulate, or one of the pomp inference algorithms.
- `params`: a npar \( \times \) nrep matrix of parameters. Each column is treated as an independent parameter set, in correspondence with the corresponding column of \( x \).
- `\ldots`: additional arguments are ignored.

**Value**

A numeric matrix containing the required samples.

**See Also**

Specification of the prior distribution simulator: prior_spec

Other pomp workhorses: dmeasure(), dprior(), dprocess(), flow(), partrans(), rinit(), rmeasure(), rprocess(), skeleton(), workhorses
Description

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

Usage

## S4 method for signature 'pomp'

rprocess(object, x0, t0, times, params, ...)

Arguments

object an object of class ‘pomp’, or of a class that extends ‘pomp’. This will typically be the output of pomp, simulate, or one of the pomp inference algorithms.

x0 an nvar x nrep matrix containing the starting state of the system. Columns of x0 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.

t0 the initial time, i.e., the time corresponding to the state in x0.

times 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 treated as an independent parameter set, in correspondence with the corresponding column of x0.

... additional arguments are ignored.

Details

When rprocess is called, t0 is taken to be the initial time (i.e., that corresponding to x0). The values in times are the times at which the state of the simulated processes are required.

Value

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

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

where nvars is the number of state variables (=nrow(x0)), nrep is the number of independent realizations simulated (=ncol(x0)), 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]. The rownames of x will correspond to those of x0.

See Also

Specification of the process-model simulator: rprocess_spec

Other pomp workhorses: dmeasure(), dprior(), dprocess(), flow(), partrans(), rinit(), rmeasure(), rprior(), skeleton(), workhorses
rprocess_spec

The latent state process simulator

Description

Specification of rprocess using “plugins”.

Usage

onestep(step.fun)

discrete_time(step.fun, delta.t = 1)

euler(step.fun, delta.t)

gillespie(rate.fun, v, hmax = Inf)

gillespie_hl(..., .pre = "", .post = "", hmax = Inf)

Arguments

step.fun     a C snippet, an R function, or the name of a native routine in a shared-object library. This gives a procedure by which one simulates a single step of the latent state process.
delta.t      positive numerical value; for euler and discrete_time, the size of the step to take
rate.fun     a C snippet, an R function, or the name of a native routine in a shared-object library. This gives a procedure by which one computes the event-rate of the elementary events in the continuous-time latent Markov chain.
v            integer matrix; giving the stoichiometry of the continuous-time latent Markov process. It should have dimensions nvar x nevent, where nvar is the number of state variables and nevent is the number of elementary events. v describes the changes that occur in each elementary event: it will usually comprise the values 1, -1, and 0 according to whether a state variable is incremented, decremented, or unchanged in an elementary event. The rows of v may be unnamed or named. If the rows are unnamed, they are assumed to be in the same order as the vector of state variables returned by rinit. If the rows are named, the names of the state variables returned by rinit will be matched to the rows of v to ensure a correct mapping. If any of the row names of v cannot be found among the state variables or if any row names of v are duplicated, an error will occur.
hmax         maximum time step allowed (see below)
...          individual C snippets corresponding to elementary events
.pre, .post  C snippets (see Details)
Discrete-time processes

If the state process evolves in discrete time, specify \texttt{rprocess} using the \texttt{discrete\_time} plug-in. Specifically, provide

\[
\texttt{rprocess = discrete\_time(step.fun = f, delta.t)},
\]

where \( f \) is a C snippet or R function that 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 \texttt{Csnippet} for general rules on writing C snippets. Examples are to be found in the tutorials on the package website.

If \( f \) is given as an R function, its arguments should come from the state variables, parameters, covariates, and time. It may also take the argument ‘\( \texttt{delta.t} \)’: when called, the latter will be the time-step. It must also have the argument ‘\( \ldots \)’. It should return a named vector of length equal to the number of state variables, representing a draw from the distribution of the state process at time \( t+\texttt{delta.t} \) conditional on its value at time \( t \).

Continuous-time processes

If the state process evolves in continuous time, but you can use an Euler approximation, implement \texttt{rprocess} using the \texttt{euler} plug-in. Specify

\[
\texttt{rprocess = euler(step.fun = f, delta.t)}
\]

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 \texttt{onestep} plug-in. To do so, specify

\[
\texttt{rprocess = onestep(step.fun = f)}.
\]

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).

Size of time step

The simulator plug-ins \texttt{discrete\_time}, \texttt{euler}, and \texttt{onestep} all work by taking discrete time steps. They differ as to how this is done. Specifically,

1. \texttt{onestep} takes a single step to go from any given time \( t_1 \) to any later time \( t_2 \) (\( t_1 < t_2 \)). Thus, this plug-in is designed for use in situations where a closed-form solution to the process exists.
2. To go from \( t_1 \) to \( t_2 \), \texttt{euler} takes \( n \) steps of equal size, where

\[
n = \text{ceiling}((t_2-t_1)/\texttt{delta.t}).
\]
3. **discrete_time** assumes that the process evolves in discrete time, where the interval between successive times is \( \Delta t \). Thus, to go from \( t_1 \) to \( t_2 \), **discrete_time** takes \( n \) steps of size exactly \( \Delta t \), where

\[
n = \text{floor}((t_2-t_1)/\Delta t).
\]

### Exact (event-driven) simulations

If you desire exact simulation of certain continuous-time Markov chains, an implementation of Gillespie’s algorithm (Gillespie 1977) is available, via the gillespie and gillespie_hl plug-ins. The former allows for the rate function to be provided as an \( \mathcal{R} \) function or a single C snippet, while the latter provides a means of specifying the elementary events via a list of C snippets.

A high-level interface to the simulator is provided by gillespie_hl. To use it, supply

\[
rprocess = \text{gillespie}\_hl(..., \ .pre = "", \ .post = "", \ hmax = Inf)
\]

to pomp. Each argument in \( ... \) corresponds to a single elementary event and should be a list containing two elements. The first should be a string or C snippet; the second should be a named integer vector. The variable rate will exist in the context of the C snippet, as will the parameter, state variables, covariates, and the time \( t \). The C snippet should assign to the variable rate the corresponding elementary event rate.

The named integer vector specifies the changes to the state variables corresponding to the elementary event. There should be named value for each of the state variables returned by rinit. The arguments \( \cdot\pre \) and \( \cdot\post \) can be used to provide C code that will run respectively before and after the elementary-event snippets. These hooks can be useful for avoiding duplication of code that performs calculations needed to obtain several of the different event rates.

Here’s how a simple birth-death model might be specified:

\[
\text{gillespie}\_hl(
    \text{birth}=\text{list}("rate = b*N;",c(N=1)),
    \text{death}=\text{list}("rate = m*N;",c(N=-1))
)
\]

In the above, the state variable \( N \) represents the population size and parameters \( b, m \) are the birth and death rates, respectively.

To use the lower-level gillespie interface, furnish

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

to pomp, where \( f \) gives the rates of the elementary events. Here, \( f \) may be an \( \mathcal{R} \) function with prototype

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

When \( f \) is called, the integer \( j \) will be the number of the elementary event (corresponding to the column the matrix \( v \), 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.
Here, the stoichiometric matrix \( v \) specifies the continuous-time Markov process in terms of its elementary events. It should have dimensions \( n\text{var} \times n\text{event} \), where \( n\text{var} \) is the number of state variables and \( n\text{event} \) 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. The rows of \( v \) should have names corresponding to the state variables. If any of the row names of \( v \) cannot be found among the state variables or if any row names of \( v \) are duplicated, an error will occur.

It is also possible to provide a C snippet via the \( \text{rate.fun} \) argument to \texttt{gillespie}. Such a snippet should assign the correct value to a rate variable depending on the value of \( j \). The same variables will be available as for the C code provided to \texttt{gillespie_hl}. This lower-level interface may be preferable if it is easier to write code that calculates the correct rate based on \( j \) rather than to write a snippet for each possible value of \( j \). For example, if the number of possible values of \( j \) is large and the rates vary according to a few simple rules, the lower-level interface may provide the easier way of specifying the model.

When the process is non-autonomous (i.e., the event rates depend explicitly on time), it can be useful to set \( \text{hmax} \) to the maximum step that will be taken. By default, the elementary event rates will be recomputed at least once per observation interval.

**Default behavior**

The default \texttt{rprocess} is undefined. It will yield missing values (NA) for all state variables.

**See Also**

Other information on model implementation: \texttt{Csnippet}, \texttt{accumulators}, \texttt{covariate_table()}, \texttt{distributions}, \texttt{dmeasure_spec}, \texttt{dprocess_spec}, \texttt{parameter_trans()}, \texttt{pomp-package}, \texttt{prior_spec}, \texttt{rinit_spec}, \texttt{rmeasure_spec}, \texttt{skeleton_spec}, \texttt{transformations}, \texttt{userdata}

---

**Description**

Specifying random-walk intensities.

**Usage**

\texttt{rw.sd(...)}

**Arguments**

\( ... \quad \text{Specification of the random-walk intensities (as standard deviations).} \)

**Details**

See \texttt{mif2} for details.
**rw2**

*Two-dimensional random-walk process*

**Description**

rw2 constructs a ‘pomp’ object encoding a 2-D Gaussian random walk.

**Usage**

```r
rw2(x1_0 = 0, x2_0 = 0, s1 = 1, s2 = 3, tau = 1, times = 1:100, t0 = 0)
```

**Arguments**

- `x1_0, x2_0`: initial conditions (i.e., latent state variable values at the zero time `t0`)
- `s1, s2`: random walk intensities
- `tau`: observation error s.d.
- `times`: observation times
- `t0`: zero time

**Details**

The random-walk process is fully but noisily observed.

**Value**

A ‘pomp’ object containing simulated data.

**See Also**

Other pomp examples: `blowflies, bsflu, dacca(), ebola, gompertz(), measles, ou2(), parus, pomp_examples, ricker(), sir_models, verhulst()`

**Examples**

```r
library(ggplot2)

rw2() %>% plot()

rw2(s1=1,s2=1,tau=0.1) %>%
  simulate(nsim=10, format="d") %>%
  ggplot(aes(x=y1, y=y2, group=.id, color=.id)) +
  geom_path() +
  guides(color=FALSE) +
  theme_bw()
```
Simulated annealing with box constraints.

Description
A straightforward implementation of simulated annealing with box constraints.

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

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

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

- **trace**: Non-negative integer. If positive, tracing information on the progress of the optimization is produced. Higher values may produce more tracing information.
- **fnscale**: An overall scaling to be applied to the value of fn during optimization. If negative, turns the problem into a maximization problem. Optimization is performed on fn(par)/fnscale.
- **parscale**: A vector of scaling values for the parameters. Optimization is performed on par/parscale and these should be comparable in the sense that a unit change in any element produces about a unit change in the scaled value.
- **maxit**: The total number of function evaluations: there is no other stopping criterion. Defaults to 10000.
- **temp**: Starting temperature for the cooling schedule. Defaults to 1.
- **tmax**: Number of function evaluations at each temperature. Defaults to 10.
- **candidate.dist**: Function to randomly select a new candidate parameter vector. This should be a function with three arguments, the first being the current parameter vector, the second the temperature, and the third the parameter scaling. By default, candidate.dist is

\[
\text{function}(\text{par}, \text{temp}, \text{scale}) \\
\quad \text{rnorm}(n=\text{length(par)}, \text{mean}=\text{par}, \text{sd}=\text{scale} \times \text{temp}).
\]

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

\[
\text{function}(k, \text{temp}, \text{tmax}) \text{ temp}/\log((k-1)/\%tmax) \times \text{tmax} + \exp(1))
\]
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

**See Also**

`traj.match, probe.match`.

---

**saved.states**

**Saved states**

**Description**

Retrieve latent state trajectories from a particle filter calculation.

**Usage**

```r
## S4 method for signature 'pfilterd_pomp'
saved.states(object, ...)

## S4 method for signature 'pfilterList'
saved.states(object, ...)
```

**Arguments**

- `object` result of a filtering computation
- `...` ignored
Details

When one calls `pfilter` with `save.states=TRUE`, the latent state vector associated with each particle is saved. This can be extracted by calling `saved.states` on the `pfilterd.pomp` object.

Value

The saved states are returned in the form of a list, with one element per time-point. Each element consists of a matrix, with one row for each state variable and one column for each particle.

See Also

Other particle filter methods: `bsmc2()`, `cond.logLik()`, `eff.sample.size()`, `filter.mean()`, `filter.traj()`, `mif2()`, `pfilter()`, `pmcmc()`, `pred.mean()`, `pred.var()`, `wpfilter()`.

Description

`simulate` generates simulations of the state and measurement processes.

Usage

```r
## S4 method for signature 'missing'
simulate(
  nsim = 1,
  seed = NULL,
  times,
  t0,
  params,
  rinit,
  rprocess,
  rmeasure,
  format = c("pomps", "arrays", "data.frame"),
  include.data = FALSE,
  ...
  verbose = getOption("verbose", FALSE)
)

## S4 method for signature 'data.frame'
simulate(
  object,
  nsim = 1,
  seed = NULL,
  times,
  t0,
  params,
  ...,
  verbose = getOption("verbose", FALSE)
)
```
simulate

rinit,
rprocess,
rmeasure,
format = c("pomps", "arrays", "data.frame"),
include.data = FALSE,
...
verbose = getOption("verbose", FALSE)
)

## S4 method for signature 'pomp'
simulate(
  object,
  nsim = 1,
  seed = NULL,
  format = c("pomps", "arrays", "data.frame"),
  include.data = FALSE,
  ...
  verbose = getOption("verbose", FALSE)
)

## S4 method for signature 'objfun'
simulate(object, nsim = 1, seed = NULL, ...)

Arguments

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.

times
  the times at which observations are made. times must indicate the column
  of observation times by name or index. The time vector must be numeric and non-
  decreasing. Internally, data will be internally coerced to an array with storage-
  mode double.

t0
  The zero-time, i.e., the time of the initial state. This must be no later than the
  time of the first observation, i.e., t0 <= times[1].

params
  a named numeric vector or a matrix with rownames containing the parameters
  at which the simulations are to be performed.

rinit
  simulator of the initial-state distribution. This can be furnished either as a C
  snippet, an R function, or the name of a pre-compiled native routine available in
  a dynamically loaded library. Setting rinit=NULL sets the initial-state simulator
  to its default. For more information, see ?rinit_spec.

rprocess
  simulator of the latent state process, specified using one of the rprocess plugins.
  Setting rprocess=NULL removes the latent-state simulator. For more informa-
  tion, see ?rprocess_spec for the documentation on these plugins.

rmeasure
  simulator of the measurement model, specified either as a C snippet, an R func-
  tion, or the name of a pre-compiled native routine available in a dynamically
loaded library. Setting rmeasure=NULL removes the measurement model simulator. For more information, see ?rmeasure_spec.

format
the format in which to return the results.
format = "pumps" causes the results to be returned as a single “pomp” object, identical to object except for the latent states and observations, which have been replaced by the simulated values.
format = "arrays" causes the results to be returned as a list of two arrays. The “states” element will contain the simulated state trajectories in a rank-3 array with dimensions nvar x (ncol(params)*nsim) x ntimes. Here, nvar is the number of state variables and ntimes the length of the argument times. The “obs” element will contain the simulated data, returned as a rank-3 array with dimensions nobs x (ncol(params)*nsim) x ntimes. Here, nobs is the number of observables.
format = "data.frame" causes the results to be returned as a single data frame containing the time, states, and observations. An ordered factor variable, ‘.id’, distinguishes one simulation from another.

include.data
if TRUE, the original data are included (with .id = "rep"). This option is ignored unless format = "data.frame".

... additional arguments supply new or modify existing model characteristics or components. See pomp for a full list of recognized arguments.
When named arguments not recognized by pomp are provided, these are made available to all basic components via the so-called userdata facility. This allows the user to pass information to the basic components outside of the usual routes of covariates (covar) and model parameters (params). See ?userdata for information on how to use this facility.

verbose
logical; if TRUE, diagnostic messages will be printed to the console.

object
optional; if present, it should be the output of one of pomp’s methods

Value
A single “pomp” object, a “pompList” object, a named list of two arrays, or a data frame, according to the format option.
If params is a matrix, each column is treated as a distinct parameter set. In this case, if nsim=1, then simulate will return one simulation for each parameter set. If nsim>1, then simulate will yield nsim simulations for each parameter set. These will be ordered such that the first ncol(params) simulations represent one simulation from each of the distinct parameter sets, the second ncol(params) simulations represent a second simulation from each, and so on.
Adding column names to params can be helpful.

Author(s)
Aaron A. King

See Also
Other elementary POMP methods: pfilter(), pomp-package, probe(), spect(), wpfilter()
Description

Simple SIR-type models implemented in various ways.

Usage

```r
sir(
  gamma = 26,
  mu = 0.02,
  iota = 0.01,
  beta1 = 400,
  beta2 = 480,
  beta3 = 320,
  beta_sd = 0.001,
  rho = 0.6,
  pop = 2100000,
  S_0 = 26/400,
  I_0 = 0.001,
  R_0 = 1 - S_0 - I_0,
  t0 = 0,
  times = seq(from = t0 + 1/52, to = t0 + 4, by = 1/52),
  seed = 329343545,
  delta.t = 1/52/20
)
```

```r
sir2(
  gamma = 24,
  mu = 1/70,
  iota = 0.1,
  beta1 = 330,
  beta2 = 410,
  beta3 = 490,
  rho = 0.1,
  pop = 1e+06,
  S_0 = 0.05,
  I_0 = 1e-04,
  R_0 = 1 - S_0 - I_0,
  t0 = 0,
  times = seq(from = t0 + 1/12, to = t0 + 10, by = 1/12),
  seed = 1772464524
)
```

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gamma</td>
<td>recovery rate</td>
</tr>
</tbody>
</table>
Details

`sir()` produces a ‘pomp’ object encoding a simple seasonal SIR model with simulated data. Simulation is performed using an Euler multinomial approximation.

`sir2()` has the same model implemented using Gillespie’s algorithm.

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

Value

These functions return ‘pomp’ objects containing simulated data.

See Also

Other pomp examples: `blowflies.bsflu.dacca()`, `ebola.gompertz()`, `measles.ou2()`, `parus.pomp_examples.ricker()`, `rw2()`, `verhulst()`

Examples

```r
po <- sir()
plot(po)
coef(po)

po <- sir2()
plot(po)
plot(simulate(window(po,end=3)))
coef(po)
```
### Description

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 vector field. NB: skeleton just evaluates the deterministic skeleton; it does not iterate or integrate (see `trajectory` for this).

### Usage

```r
## S4 method for signature 'pomp'
skeleton(object, x, times, params, ...)
```

### Arguments

- **object**: an object of class 'pomp', or of a class that extends 'pomp'. This will typically be the output of `pomp`, `simulate`, or one of the `pomp` inference algorithms.
- **x**: an 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`. One can also pass `x` as a named numeric vector, which is equivalent to the `nrep=1, ntimes=1` case.
- **times**: 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 treated as an independent parameter set, in correspondence with the corresponding column of `x`.
- **...**: additional arguments are ignored.

### Value

`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]`.

### See Also

Specification of the deterministic skeleton: `skeleton_spec`

Other pomp workhorses: `dmeasure()`, `dprior()`, `dprocess()`, `flow()`, `partrans()`, `rinit()`, `rmeasure()`, `rprior()`, `rprocess()`, `workhorses`
The deterministic skeleton of a model

Description

Specification of skeleton.

Usage

\[
\begin{align*}
\text{vectorfield}(f) \\
\text{map}(f, \text{delta}.t = 1)
\end{align*}
\]

Arguments

- \(f\): procedure for evaluating the deterministic skeleton. This can be a C snippet, an \texttt{R} function, or the name of a native routine in a dynamically linked library.
- \(\text{delta}.t\): positive numerical value; the size of the discrete time step corresponding to an application of the map.

Details

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 \texttt{pomp}, the skeleton is used by \texttt{trajectory} and \texttt{traj.match}.

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

\[
\text{skeleton} = \text{map}(f, \text{delta}.t)
\]

to \texttt{pomp}, where \(f\) implements the map and \(\text{delta}.t\) 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

\[
\text{skeleton} = \text{vectorfield}(f)
\]

to \texttt{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 \texttt{R} function. General rules for writing C snippets can be found here. 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 \(Dx\) 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 \( D_x \) and is the value of \( dx/dt \).

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, its arguments should be taken from among the state variables, parameters, covariates, and time. It must also take the argument ‘\( \ldots \)’. As with the other basic components, \( f \) may take additional arguments, provided these are passed along with it in the call to \( \text{pomp} \). The function \( f \) must return a numeric vector of the same length as the number of state variables, which contains the value of the map or vectorfield at the required point and time.

**Default behavior**

The default skeleton is undefined. It will yield missing values (NA) for all state variables.

**See Also**

Other information on model implementation: \( \text{Csnippet, accumulators, covariate_table()} \), \( \text{distributions, dmeasure_spec, dprocess_spec, parameter_trans()} \), \( \text{pomp-package, prior_spec, rinit_spec, rmeasure_spec, rprocess_spec, transformations, userdata} \)

---

**Description**

Power spectrum computation and spectrum-matching for partially-observed Markov processes.

**Usage**

```r
## S4 method for signature 'data.frame'
spect(
data, 
vars, 
kernel.width, 
nsim, 
seed = NULL, 
transform.data = identity, 
detrend = c("none", "mean", "linear", "quadratic"), 
params, 
rinit, 
rprocess, 
rmeasure, 
..., 
verbose = getOption("verbose", FALSE)
)
```

---

**spect**

*Power spectrum*

---

Power spectrum computation and spectrum-matching for partially-observed Markov processes.
## S4 method for signature 'pomp'
spect(
data, vars, kernel.width, nsim, seed = NULL, transform.data = identity, detrend = c("none", "mean", "linear", "quadratic"), ...
, verbose =getOption("verbose", FALSE)
)

## S4 method for signature 'spectd_pomp'
spect(
data, vars, kernel.width, nsim, seed = NULL, transform.data, detrend, ...
, verbose =getOption("verbose", FALSE)
)

## S4 method for signature 'spect_match_objfun'
spect(data, seed, ..., verbose = getOption("verbose", FALSE))

## S4 method for signature 'objfun'
spect(data, seed = NULL, ...)

**Arguments**

- **data**: either a data frame holding the time series data, or an object of class ‘pomp’, i.e., the output of another pomp calculation.
- **vars**: optional; names of observed variables for which the power spectrum will be computed. 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.data**: function; this transformation will be applied to the observables prior to estimation of the spectrum, and prior to any detrending.
detrend  de-trending operation to perform. Options include no detrending, and subtraction of constant, linear, and quadratic trends from the data. Detrending is applied to each data series and to each model simulation independently.

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

rinit  simulator of the initial-state distribution. This can be furnished either as a C snippet, an R function, or the name of a pre-compiled native routine available in a dynamically loaded library. Setting rinit=NULL sets the initial-state simulator to its default. For more information, see ?rinit_spec.

rprocess  simulator of the latent state process, specified using one of the rprocess plugins. Setting rprocess=NULL removes the latent-state simulator. For more information, see ?rprocess_spec for the documentation on these plugins.

rmeasure  simulator of the measurement model, specified either as a C snippet, an R function, or the name of a pre-compiled native routine available in a dynamically loaded library. Setting rmeasure=NULL removes the measurement model simulator. For more information, see ?rmeasure_spec.

...  additional arguments supply new or modify existing model characteristics or components. See pomp for a full list of recognized arguments.

verbose  logical; if TRUE, diagnostic messages will be printed to the console.

Details

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).

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 'spectd_pomp'.

When spect operates on a spectrum-matching objective function (a 'spect_match_objfun' object), by default, the random-number generator seed is fixed at the value given when the objective function was constructed. Specifying NULL or an integer for seed overrides this behavior.

Value

An object of class 'spectd_pomp', which contains the model, the data, and the results of the spect computation. The following methods are available:

- **plot**  produces some diagnostic plots
- **summary**  displays a summary
- **logLik**  gives a measure of the agreement of the power spectra
Author(s)

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

References


See Also

Other summary statistics methods: `abc()`, `basic_probes`, `probe.match`, `probe()`

Other elementary POMP methods: `pfilter()`, `pomp-package`, `probe()`, `simulate()`, `wpfilter()`

spect.match  
*Spectrum matching*

Description

Estimation of parameters by matching power spectra

Usage

```r
## S4 method for signature 'data.frame'
spect_objfun(
  data,
  est = character(0),
  weights = 1,
  fail.value = NA,
  vars,
  kernel.width,
  nsim,
  seed = NULL,
  transform.data = identity,
  detrend = c("none", "mean", "linear", "quadratic"),
  params,
  rinit,
  rprocess,
  rmeasure,
  partrans,
  ...
  verbose = getOption("verbose", FALSE)
)
```
## S4 method for signature 'pomp'
spect_objfun(
  data,
  est = character(0),
  weights = 1,
  fail.value = NA,
  vars,
  kernel.width,
  nsim,
  seed = NULL,
  transform.data = identity,
  detrend = c("none", "mean", "linear", "quadratic"),
  ..., 
  verbose = getOption("verbose", FALSE)
)

## S4 method for signature 'spectd_pomp'
spect_objfun(
  data,
  est = character(0),
  weights = 1,
  fail.value = NA,
  vars,
  kernel.width,
  nsim,
  seed = NULL,
  transform.data = identity,
  detrend,
  ..., 
  verbose = getOption("verbose", FALSE)
)

## S4 method for signature 'spect_match_objfun'
spect_objfun(
  data,
  est,
  weights,
  fail.value,
  seed = NULL,
  ..., 
  verbose = getOption("verbose", FALSE)
)

**Arguments**

- **data**: either a data frame holding the time series data, or an object of class `pomp`, i.e., the output of another `pomp` calculation.
- **est**: character vector; the names of parameters to be estimated.
weights

optional numeric or function. 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.

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).

vars

optional; names of observed variables for which the power spectrum will be computed. 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

the number of model simulations to be computed.

seed

integer. When fitting, it is often best to fix the seed of the random-number generator (RNG). This is accomplished by setting seed to an integer. By default, seed = NULL, which does not alter the RNG state.

transform.data

function; this transformation will be applied to the observables prior to estimation of the spectrum, and prior to any detrending.

detrend

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

params

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

rinit

simulator of the initial-state distribution. This can be furnished either as a C snippet, an R function, or the name of a pre-compiled native routine available in a dynamically loaded library. Setting rinit=NULL sets the initial-state simulator to its default. For more information, see ?rinit_spec.

rprocess

simulator of the latent state process, specified using one of the rprocess plugins. Setting rprocess=NULL removes the latent-state simulator. For more information, see ?rprocess_spec for the documentation on these plugins.

rmeasure

simulator of the measurement model, specified either as a C snippet, an R function, or the name of a pre-compiled native routine available in a dynamically loaded library. Setting rmeasure=NULL removes the measurement model simulator. For more information, see ?rmeasure_spec.

partrans

optional parameter transformations, constructed using parameter_trans. 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. One should supply the partrans argument via a call to parameter_trans. For more information, see ?parameter_trans. Setting partrans=NULL removes the parameter transformations, i.e., sets them to the identity transformation.

...  

additional arguments supply new or modify existing model characteristics or components. See pomp for a full list of recognized arguments.
When named arguments not recognized by \texttt{pomp} are provided, these are made available to all basic components via the so-called \texttt{userdata} facility. This allows the user to pass information to the basic components outside of the usual routes of covariates (\texttt{covar}) and model parameters (\texttt{params}). See \texttt{?userdata} for information on how to use this facility.

\texttt{verbose} logical; if \texttt{TRUE}, diagnostic messages will be printed to the console.

\textbf{Details}

In spectrum matching, one attempts to minimize the discrepancy between a \texttt{POMP} model's predictions and data, as measured in the frequency domain by the power spectrum.

\texttt{spect_objfun} constructs an objective function that measures the discrepancy. It can be passed to any one of a variety of numerical optimization routines, which will adjust model parameters to minimize the discrepancies between the power spectrum of model simulations and that of the data.

\textbf{Value}

\texttt{spect_objfun} constructs a stateful objective function for spectrum matching. Specifically, \texttt{spect_objfun} returns an object of class \texttt{'spect_match_objfun'}, which is a function suitable for use in an \texttt{optim}-like optimizer. This function takes a single numeric-vector argument that is assumed to contain the parameters named in \texttt{est}, in that order. When called, it will return the (optionally weighted) $L^2$ distance between the data spectrum and simulated spectra. It is a stateful function: Each time it is called, it will remember the values of the parameters and the discrepancy measure.

\textbf{Important Note}

Since \texttt{pomp} cannot guarantee that the final call an optimizer makes to the function is a call at the optimum, it cannot guarantee that the parameters stored in the function are the optimal ones. Therefore, it is a good idea to evaluate the function on the parameters returned by the optimization routine, which will ensure that these parameters are stored.

\textbf{See Also}

\texttt{spect optim subplex nloptr}

Other pomp parameter estimation methods: \texttt{abc()}, \texttt{bsmc2()}, \texttt{kalman}, \texttt{mif2()}, \texttt{nlf}, \texttt{pmcmc()}, \texttt{pomp-package}, \texttt{probe.match}

\textbf{Examples}

```
library(magrittr)

ricker() %>%
  spect_objfun(
    est=c("r","sigma","N_0"),
    partrans=parameter_trans(log=c("r","sigma","N_0")),
    paramnames=c("r","sigma","N_0"),
    kernel.width=3,
    nsim=100,
  )
```
seed=5069977
  ) -> f

f(log(c(20,0.3,10)))
f %>% spect() %>% plot()

library(subplex)
subplex(fn=f,par=log(c(20,0.3,10)),control=list(reltol=1e-5)) -> out
f(out$par)

f %>% summary()
f %>% spect() %>% plot()

spy Spy

Description
Peek into the inside of one of pomp’s objects.

Usage
## S4 method for signature 'pomp'
spy(object)

Arguments
object the object whose structure we wish to examine

states Latent states

Description
Extract the latent states from a ‘pomp’ object.

Usage
## S4 method for signature 'pomp'
states(object, vars, ...)

Arguments
object an object of class ‘pomp’, or of a class extending ‘pomp’
vars names of variables to retrieve
... ignored
**Summary methods**

**Description**
Display a summary of a fitted model object.

**Usage**
```
## S4 method for signature 'probed_pomp'
summary(object, ...)

## S4 method for signature 'spectd_pomp'
summary(object, ...)

## S4 method for signature 'objfun'
summary(object, ...)
```

**Arguments**
- **object**: a fitted model object
- **...**: ignored

**Methods to manipulate the observation times**

**Description**
Get and set the vector of observation times.

**Usage**
```
## S4 method for signature 'pomp'
time(x, t0 = FALSE, ...)

## S4 replacement method for signature 'pomp'
time(object, t0 = FALSE, ...) <- value
```

**Arguments**
- **x**: a 'pomp' object
- **t0**: logical; should the zero time be included?
- **...**: ignored
- **object**: a 'pomp' object
- **value**: numeric vector; the new vector of times
Details

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

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

---

timezero The zero time

Description

Get and set the zero-time.

Usage

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

## S4 replacement method for signature 'pomp'
timezero(object, ...) <- value

Arguments

| object | an object of class ‘pomp’, or of a class that extends ‘pomp’ |
| ...    | ignored |
| value  | numeric; the new zero-time value |

Value

the value of the zero time

---

traces Traces

Description

Retrieve the history of an iterative calculation.
Usage

## S4 method for signature 'mif2d_pomp'
traces(object, pars, transform = FALSE, ...)

## S4 method for signature 'mif2List'
traces(object, pars, ...)

## S4 method for signature 'abcd_pomp'
traces(object, pars, ...)

## S4 method for signature 'abclist'
traces(object, pars, ...)

## S4 method for signature 'pmcmcd_pomp'
traces(object, pars, ...)

## S4 method for signature 'pmcmcList'
traces(object, pars, ...)

Arguments

object an object of class extending ‘pomp’, the result of the application of a parameter estimation algorithm
pars names of parameters
transform logical; should the traces be transformed back onto the natural scale?
... ignored or (in the case of the listie, passed to the more primitive function)

Details

Note that pmcmc does not currently support parameter transformations.

Value

When object is the result of a mif2 calculation, traces(object, pars, transform = FALSE) returns the traces of the parameters named in pars. By default, the traces of all parameters are returned. Note that, if the computation was performed with transformed parameters, the traces are on the estimation scale. If transform=TRUE, the parameters are transformed from the estimation scale onto the natural scale.

When object is a ‘abcd_pomp’, traces(object) extracts the traces as a coda::mcmc.
When object is a ‘abclist’, traces(object) extracts the traces as a coda::mcmc.list.
When object is a ‘pmcmcd_pomp’, traces(object) extracts the traces as a coda::mcmc.
When object is a ‘pmcmcList’, traces(object) extracts the traces as a coda::mcmc.list.
**Description**

Estimation of parameters for deterministic POMP models

**Usage**

```r
## S4 method for signature 'data.frame'
traj_objfun(  
data,  
est = character(0),  
fail.value = NA,  
ode_control = list(),  
params,  
rinit,  
skeleton,  
dmeasure,  
partrans,  
...,  
verbose = getOption("verbose", FALSE)
)

## S4 method for signature 'pomp'
traj_objfun(  
data,  
est = character(0),  
fail.value = NA,  
ode_control = list(),  
...,  
verbose = getOption("verbose", FALSE)
)

## S4 method for signature 'traj_match_objfun'
traj_objfun(  
data,  
est,  
fail.value,  
node_control,  
...,  
verbose = getOption("verbose", FALSE)
)
```

**Arguments**

- **data**: either a data frame holding the time series data, or an object of class ‘pomp’, i.e., the output of another pomp calculation.
est         character vector; the names of parameters to be estimated.
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).
ode_control optional list; the elements of this list will be passed to ode.
params       optional; named numeric vector of parameters. This will be coerced internally to storage mode double.
rinit      simulator of the initial-state distribution. This can be furnished either as a C snippet, an R function, or the name of a pre-compiled native routine available in a dynamically loaded library. Setting rinit=NULL sets the initial-state simulator to its default. For more information, see ?rinit_spec.
skeleton    optional; the deterministic skeleton of the unobserved state process. Depending on whether the model operates in continuous or discrete time, this is either a vectorfield or a map. Accordingly, this is supplied using either the vectorfield or map functions. For more information, see ?skeleton_spec. Setting skeleton=NULL removes the deterministic skeleton.
dmeasure    evaluator of the measurement model density, specified either as a C snippet, an R function, or the name of a pre-compiled native routine available in a dynamically loaded library. Setting dmeasure=NULL removes the measurement density evaluator. For more information, see ?dmeasure_spec.
partrans   optional parameter transformations, constructed using parameter_trans. 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. One should supply the partrans argument via a call to parameter_trans. For more information, see ?parameter_trans. Setting partrans=NULL removes the parameter transformations, i.e., sets them to the identity transformation.
...          additional arguments will modify the model structure
verbose     logical; if TRUE, diagnostic messages will be printed to the console.

Details

In trajectory matching, one attempts to minimize the discrepancy between a POMP model’s predictions and data under the assumption that the latent state process is deterministic and all discrepancies between model and data are due to measurement error. The measurement model likelihood (dmeasure), or rather its negative, is the natural measure of the discrepancy.

Trajectory matching is a generalization of the traditional nonlinear least squares approach. In particular, if, on some scale, measurement errors are normal with constant variance, then trajectory matching is equivalent to least squares on that particular scale.

traj_objfun constructs an objective function that evaluates the likelihood function. It can be passed to any one of a variety of numerical optimization routines, which will adjust model parameters to minimize the discrepancies between the power spectrum of model simulations and that of the data.
Value

\texttt{traj\_objfun} constructs a stateful objective function for spectrum matching. Specifically, \texttt{traj\_objfun} returns an object of class ‘traj\_match\_objfun’, which is a function suitable for use in an \texttt{optim}-like optimizer. In particular, this function takes a single numeric-vector argument that is assumed to contain the parameters named in \texttt{est}, in that order. When called, it will return the negative log likelihood. It is a stateful function: Each time it is called, it will remember the values of the parameters and its estimate of the log likelihood.

Important Note

Since \texttt{pomp} cannot guarantee that the final call an optimizer makes to the function is a call \textit{at} the optimum, it cannot guarantee that the parameters stored in the function are the optimal ones. Therefore, it is a good idea to evaluate the function on the parameters returned by the optimization routine, which will ensure that these parameters are stored.

See Also

\texttt{trajectory}, \texttt{optim}, \texttt{subplex}, \texttt{nloptr}

Examples

```r
library(magrittr)

ricker() %>%
  traj_objfun(
    est=c("r","sigma","N_0"),
    partrans=parameter_trans(log=c("r","sigma","N_0")),
    paramnames=c("r","sigma","N_0"),
  ) -> f

f(log(c(20,0.3,10)))

library(subplex)

subplex(fn=f,par=log(c(20,0.3,10)),control=list(reltol=1e-5)) -> out

f(out$par)

library(ggplot2)

f %>%
  trajectory(format="data.frame") %>%
  ggplot(aes(x=time,y=N))+geom_line()+theme_bw()
```
**Trajectory of a deterministic model**

**Description**

Compute trajectories of the deterministic skeleton of a Markov process.

**Usage**

```r
## S4 method for signature 'pomp'
trajectory(
  object,
  params,
  times,
  t0,
  format = c("array", "data.frame"),
  ..., 
  verbose = getOption("verbose", FALSE)
)

## S4 method for signature 'traj_match_objfun'
trajectory(object, ..., verbose = getOption("verbose", FALSE))
```

**Arguments**

- **object**: an object of class `pomp`, or of a class that extends `pomp`. This will typically be the output of `pomp`, `simulate`, or one of the `pomp` inference algorithms.
- **params**: a `npar x nrep` matrix of parameters. Each column is treated as an independent parameter set, in correspondence with the corresponding column of `x`.
- **times**: a numeric vector (length `ntimes`) containing times at which the itineraries are desired. These must be in non-decreasing order with `times[1]>t0`.
- **t0**: the time at which the initial conditions are assumed to hold.
- **format**: the format in which to return the results.
  - `format = "array"` causes the trajectories to be returned in a rank-3 array with dimensions `nvar x ncol(params) x ntimes`. Here, `nvar` is the number of state variables and `ntimes` the length of the argument times.
  - `format = "data.frame"` causes the results to be returned as a single data frame containing the time and states. An ordered factor variable, `.id`, distinguishes the trajectories from one another.
- **...**: Additional arguments are passed to the ODE integrator (if the skeleton is a vectorfield) and are ignored if it is a map. See `ode` for a description of the additional arguments accepted by the ODE integrator.
- **verbose**: logical; if TRUE, diagnostic messages will be printed to the console.
**Details**

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.

Note that the handling of ... in trajectory differs from that of most other functions in **pomp**. In particular, it is not possible to modify the model structure in a call to trajectory.

**Value**

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

**See Also**

skeleton, flow

---

**transformations**

**Transformations**

**Description**

Some useful parameter transformations.

**Usage**

logit(p)

expit(x)

log_barycentric(X)

inv_log_barycentric(Y)

**Arguments**

- `p` numeric; a quantity in [0,1].
- `x` numeric; the log odds ratio.
- `X` numeric; a vector containing the quantities to be transformed according to the log-barycentric transformation.
- `Y` numeric; a vector containing the log fractions.
Parameter transformations can be used in many cases to recast constrained optimization problems as unconstrained problems. Although there are no limits to the transformations one can implement using the parameter_trans facility, pomp provides a few ready-built functions to implement some very commonly useful ones.

The logit transformation takes a probability \( p \) to its log odds, \( \log \frac{p}{1-p} \). It maps the unit interval \([0, 1]\) into the extended real line \([-\infty, \infty]\).

The inverse of the logit transformation is the expit transformation.

The log-barycentric transformation takes a vector \( X_i, i = 1, \ldots, n \), to a vector \( Y_i \), where

\[
Y_i = \log \frac{X_i}{\sum_j X_j}
\]

If \( X \) is an \( n \)-vector, it takes every simplex defined by \( \sum_i X_i = c \), \( c \) constant, to \( n \)-dimensional Euclidean space \( \mathbb{R}^n \).

The inverse of the log-barycentric transformation is implemented as \( \text{inv\_log\_barycentric} \). Note that it is not a true inverse, in the sense that it takes \( \mathbb{R}^n \) to the unit simplex, \( \sum_i X_i = 1 \). Thus,

\[
\log\text{barycentric}(\text{inv\_log\_barycentric}(Y)) == Y,
\]

but

\[
\text{inv\_log\_barycentric}(\log\text{barycentric}(X)) == X
\]

only if \( \text{sum}(X) == 1 \).

See Also

Other information on model implementation: Csnippet, accumulators, covariate_table(), distributions, dmeasure_spec, dprocess_spec, parameter_trans(), pomp-package, prior_spec, rinit_spec, rmeasure_spec, rprocess_spec, skeleton_spec, userdata

---

**userdata**

*Facilities for making additional information to basic components*

**Description**

When POMP basic components need information they can’t get from parameters or covariates.
Details

It can happen that one desires to pass information to one of the POMP model basic components (see here for a definition of this term) outside of the standard routes (i.e., via model parameters or covariates). pomp provides facilities for this purpose. We refer to the objects one wishes to pass in this way as user data.

The following will apply to every basic model component. For the sake of definiteness, however, we’ll use the rmeasure component as an example. To be even more specific, the measurement model we wish to implement is

\[ y_1 \sim \text{Poisson}(x_1+\theta), \quad y_2 \sim \text{Poisson}(x_2+\theta), \]

where \( \theta \) is a parameter. Although it would be very easy (and indeed far preferable) to include \( \theta \) among the ordinary parameters (by including it in params), we will assume here that we have some reason for not wanting to do so.

Now, we have the choice of providing rmeasure in one of three ways:

1. as an R function,
2. as a C snippet, or
3. as a procedure in an external, dynamically loaded library.

We’ll deal with these three cases in turn.

When the basic component is specified as an R function

We can implement a simulator for the aforementioned measurement model so:

```r
f <- function (t, x, params, theta, ...) {
  y <- rpois(n=2,x[c("x1","x2")]+theta)
  setNames(y,c("y1","y2"))
}
```

So far, so good, but how do we get \( \theta \) to this function? We simply provide an additional argument to whichever pomp algorithm we are employing (e.g., simulate, pfiler, mif2, abc, etc.). For example:

```r
simulate(..., rmeasure = f, theta = 42, ...)
```

where the ... represent the other simulate arguments we might want to supply. When we do so, a message will be generated, informing us that \( \theta \) is available for use by the POMP basic components. This warning helps forestall accidental triggering of this facility due to typographical error.

When the basic component is specified via a C snippet

A C snippet implementation of the aforementioned measurement model is:

```c
f <- Csnippet("
  double theta = *(get_userdata_double("theta"));
  
")
```
Here, the call to get_userdata_double retrieves a pointer to the stored value of theta. Note the need to escape the quotes in the C snippet text.

It is possible to store and retrieve integer objects also, using get_userdata_int.

One must take care that one stores the user data with the appropriate storage type. For example, it is wise to wrap floating point scalars and vectors with as.double and integers with as.integer.

In the present example, our call to simulate might look like

```r
simulate(..., rmeasure = f, theta = as.double(42), ...)
```

Since the two functions `get_userdata_double` and `get_userdata_int` return pointers, it is trivial to pass vectors of double-precision and integers.

A simpler and more elegant approach is afforded by the `globals` argument (see below).

**When the basic component is specified via an external library**

The rules are essentially the same as for C snippets. typedef declarations for the `get_userdata_double` and `get_userdata_int` are given in the 'pomp.h' header file and these two routines are registered so that they can be retrieved via a call to `R_GetCCallable`. See the Writing R extensions manual for more information.

**Setting globals**

The use of the userdata facilities incurs a run-time cost. It is faster and more elegant, when using C snippets, to put the needed objects directly into the C snippet library. The `globals` argument does this. See the example below.

**See Also**

Other information on model implementation: `Csnippet`, `accumulators`, `covariate_table()`, `distributions`, `dmeasure_spec`, `dprocess_spec`, `parameter_trans()`. `pomp-package`, `prior_spec`, `rinit_spec`, `rmeasure_spec`, `rprocess_spec`, `skeleton_spec`, `transformations`

**Examples**

```r
## The familiar Ricker example
## For some bizarre reason, we wish to pass 'phi' via the userdata facility.

## C snippet approach:

simulate(times=1:100,t0=0, phi=as.double(100),
params=c(r=3.8,sigma=0.3,N.0=7),
rprocess=discrete_time(
  step.fun=Csnippet("
    double e = (sigma > 0.0) ? rnorm(0,sigma) : 0.0;
    N = r*N*exp(-N+e);
  ")
),
  delta.t=1
),
rmeasure=Csnippet("
double phi = *(get_userdata_double("phi"));
y = rpois(phi*N);
},
paramnames=c("r","sigma"),
statenames="N",
obsnames="y"
) -> rick1

## The same problem solved using 'globals':
simulate(times=1:100,t0=0,
globals=Csnippet("static double phi = 100;"),
params=c(r=3.8,sigma=0.3,N.0=7),
rprocess=discrete_time(
    step.fun=Csnippet(
        double e = (sigma > 0.0) ? rnorm(0,sigma) : 0.0;
        N = r*N*exp(-N+e);
    ),
    delta.t=1
),
rmeasure=Csnippet(" y = rpois(phi*N);"
),
paramnames=c("r","sigma"),
statenames="N",
obsnames="y"
) -> rick2

## Finally, the R function approach:
simulate(times=1:100,t0=0,
phi=100,
params=c(r=3.8,sigma=0.3,N_0=7),
rprocess=discrete_time(
    step.fun=function (r, N, sigma, ...) {
        e <- rnorm(n=1,mean=0,sd=sigma)
        c(N=r*N*exp(-N+e))
    },
    delta.t=1
),
rmeasure=function(phi, N, ...) {
    c(y=rpois(n=1,lambda=phi*N))
}
) -> rick3

---

**verhulst**  
**Verhulst-Pearl model**

**Description**

The Verhulst-Pearl (logistic) model of population growth.
verhulst

Usage

verhulst(n_0 = 10000, K = 10000, r = 0.9, sigma = 0.4, tau = 0.1, dt = 0.01)

Arguments

n_0  initial condition
K  carrying capacity
r  intrinsic growth rate
sigma  environmental process noise s.d.
tau  measurement error s.d.
dt  Euler time-step

Details

A stochastic version of the Verhulst-Pearl logistic model. This evolves in continuous time, according to the stochastic differential equation

\[ dn = r n \left(1 - \frac{n}{K}\right) dt + \sigma n dW. \]

Numerically, we simulate the stochastic dynamics using an Euler approximation.

The measurements are assumed to be log-normally distributed.

Value

A ‘pomp’ object containing the model and simulated data. The following basic components are included in the ‘pomp’ object: ‘rinit’, ‘rprocess’, ‘rmeasure’, ‘dmeasure’, and ‘skeleton’.

See Also

Other pomp examples: blowflies, bsflu, dacca(), ebola, gompertz(), measles, ou2(), parus, pomp_examples, ricker(), rw2(), sir_models

Examples

verhulst() -> po
plot(po)
plot(simulate(po))
pfilter(po,Np=1000) -> pf
logLik(pf)
spy(po)
window

Description

Restrict to a portion of a time series.

Usage

```r
## S4 method for signature 'pomp'
window(x, start, end, ...)
```

Arguments

- `x`: a ‘pomp’ object or object of class extending ‘pomp’
- `start, end`: the left and right ends of the window, in units of time
- `...`: ignored

Description

Workhorse functions for the `pomp` algorithms.

These functions mediate the interface between the user’s model and the package algorithms. They are low-level functions that do the work needed by the package’s inference methods.

Details

They include

- `dmeasure` which evaluates the measurement model density,
- `rmeasure` which samples from the measurement model distribution,
- `dprocess` which evaluates the process model density,
- `rprocess` which samples from the process model distribution,
- `dprior` which evaluates the prior probability density,
- `rprior` which samples from the prior distribution,
- `skeleton` which evaluates the model’s deterministic skeleton,
- `flow` which iterates or integrates the deterministic skeleton to yield trajectories,
- `partrans` which performs parameter transformations associated with the model.

Author(s)

Aaron A. King
wpfilter

See Also

```
simulate, trajectory, pfilter, probe
```

Other pomp workhorses: `dmeasure()`, `dprior()`, `dprocess()`, `flow()`, `partrans()`, `rinit()`, `rmeasure()`, `rprior()`, `rprocess()`, `skeleton()`

---

**wpfilter**  
Weighted particle filter

---

**Description**

A sequential importance sampling (particle filter) algorithm. Unlike in `pfilter`, resampling is performed only when triggered by deficiency in the effective sample size.

**Usage**

```r
## S4 method for signature 'data.frame'
wpfilter(
data, 
Np, 
params, 
rinit, 
rprocess, 
dmeasure, 
trigger = 1, 
target = 0.5, 
..., 
verbose = getOption("verbose", FALSE)
)

## S4 method for signature 'pomp'
wpfilter(
data, 
Np, 
trigger = 1, 
target = 0.5, 
..., 
verbose = getOption("verbose", FALSE)
)

## S4 method for signature 'wpfilterd_pomp'
wpfilter(data, Np, trigger, target, ..., verbose = getOption("verbose", FALSE))
```

**Arguments**

- `data`: either a data frame holding the time series data, or an object of class `pomp`, i.e., the output of another `pomp` calculation.
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 \( N_p \) either as a vector of positive integers of length
\[
\text{length}(\text{time}(\text{object}, t0=\text{TRUE}))
\]
or as a function taking a positive integer argument. In the latter case, \( N_p(k) \) must be a single positive integer, representing the number of particles to be used at the \( k \)-th timestep: \( N_p(0) \) is the number of particles to use going from \( \text{timezero} \text{(object)} \) to \( \text{time} \text{(object)}[1] \), \( N_p(1) \), from \( \text{timezero} \text{(object)} \) to \( \text{time} \text{(object)}[1] \), and so on, while when \( T=\text{length}(\text{time}(\text{object})) \), \( N_p(T) \) is the number of particles to sample at the end of the time-series.

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

simulator of the initial-state distribution. This can be furnished either as a C snippet, an \text{R} function, or the name of a pre-compiled native routine available in a dynamically loaded library. Setting \( \text{rinit} = \text{NULL} \) sets the initial-state simulator to its default. For more information, see \?\text{rinit\_spec}.

simulator of the latent state process, specified using one of the \text{rprocess} plugins. Setting \( \text{rprocess} = \text{NULL} \) removes the latent-state simulator. For more information, see \?\text{rprocess\_spec} for the documentation on these plugins.

evaluator of the measurement model density, specified either as a C snippet, an \text{R} function, or the name of a pre-compiled native routine available in a dynamically loaded library. Setting \( \text{dmeasure} = \text{NULL} \) removes the measurement density evaluator. For more information, see \?\text{dmeasure\_spec}.

numeric; if the effective sample size becomes smaller than \( \text{trigger} \times N_p \), resampling is triggered.

numeric; target power.

additional arguments supply new or modify existing model characteristics or components. See \text{pomp} for a full list of recognized arguments. When named arguments not recognized by \text{pomp} are provided, these are made available to all basic components via the so-called \text{userdata} facility. This allows the user to pass information to the basic components outside of the usual routes of covariates (\text{covar}) and model parameters (\text{params}). See \?\text{userdata} for information on how to use this facility.

logical; if TRUE, diagnostic messages will be printed to the console.

This function is experimental and should be considered in alpha stage. Both interface and underlying algorithms may change without warning at any time. Please explore the function and give feedback via the \text{pomp} Issues page.

An object of class ‘wpfilterd\_pomp’, which extends class ‘pomp’. Information can be extracted from this object using the methods documented below.
Methods

- **logLik**: the estimated log likelihood
- **cond.logLik**: the estimated conditional log likelihood
- **eff.sample.size**: the (time-dependent) estimated effective sample size
- **as.data.frame**: coerce to a data frame
- **plot**: diagnostic plots

Author(s)

Aaron A. King

References


See Also

Other elementary POMP methods: `pfilter()`, `pomp-package`, `probe()`, `simulate()`, `spect()`
Other particle filter methods: `bsmc2()`, `cond.logLik()`, `eff.sample.size()`, `filter.mean()`, `filter.traj()`, `mif2()`, `pfilter()`, `pmcmc()`, `pred.mean()`, `pred.var()`, `saved.states()`
Index

* datasets
  blowflies, 17
  bsflu, 19
  dacca, 30
  ebola, 41
  gompertz, 47
  measles, 54
  ou2, 65
  parus, 69
  pomp_examples, 82
  ricker, 94
  rw2, 105
  sir_models, 111
* design
  design, 32
* diagnostics
  basic_probes, 15
* distribution
  distributions, 34
* elementary POMP methods
  pfilter, 70
  pomp-package, 4
  probe, 86
  simulate, 108
  spect, 115
  wpfilter, 137
* extending the pomp package
  hitch, 48
  workhorses, 136
* information on model implementation
  accumulators, 9
  covariate_table, 26
  Csnippet, 28
  distributions, 34
  dmeasure_spec, 37
  dprocess_spec, 40
  parameter_trans, 66
  pomp-package, 4
  prior_spec, 85
  rinit_spec, 96
  rmeasure_spec, 98
  rprocess_spec, 101
  skeleton_spec, 114
  transformations, 130
  userdata, 131
* low-level interface
  hitch, 48
  workhorses, 136
* models
  blowflies, 17
  dacca, 30
  gompertz, 47
  ou2, 65
  pomp-package, 4
  pomp_examples, 82
  ricker, 94
  rw2, 105
  sir_models, 111
* multivariate
  pomp-package, 4
* optimize
  sannbox, 106
* parameter transformations
  transformations, 130
* particle filter methods
  bsmc2, 20
  cond.logLik, 24
  eff.sample.size, 43
  filter.mean, 44
  filter.traj, 45
  mif2, 55
  pfilter, 70
  pmcmc, 75
  pred.mean, 83
  pred.var, 84
  saved.states, 107
  wpfilter, 137
* particle filtering methods

140
INDEX 141

kalman, 50

* pomp examples
  blowflies, 17
  bsflu, 19
  dacca, 30
  ebola, 41
  gompertz, 47
  measles, 54
  ou2, 65
  parus, 69
  pomp_examples, 82
  ricker, 94
  rw2, 105
  sir_models, 111
  verhulst, 134

* pomp parameter estimation methods
  abc, 6
  bsmc2, 20
  kalman, 50
  mif2, 55
  nlf, 60
  pmcmc, 75
  pomp-package, 4
  probe.match, 89
  spect.match, 118

* pomp workhorses
  dmeasure, 36
  dprior, 38
  dprocess, 39
  flow, 46
  partrans, 69
  rinit, 95
  rmeasure, 97
  rprior, 99
  rprocess, 100
  skeleton, 113
  workhorses, 136

* probability distributions
  distributions, 34

* smooth
  bsplines, 22

* summary statistics methods
  abc, 6
  basic_probes, 15
  probe, 86
  probe.match, 89
  spect, 115

* ts
  pomp-package, 4
  ?accumulators, 81
  ?dmeasureSpec, 21, 58, 72, 77, 80, 127, 138
  ?dpriorSpec, 80
  ?parameterTrans, 21, 58, 80, 91, 120, 127
  ?rinitSpec, 7, 21, 77, 80
  ?rmeasureSpec, 7, 21, 51, 57, 62, 72, 77, 80, 88,
    91, 109, 117, 120, 127, 138
  ?rprocessSpec, 7, 62, 80, 88, 91, 110, 117, 120
  ?skeletonSpec, 80, 127
  ?userdata, 7, 21, 51, 58, 62, 72, 77, 80, 88,
    91, 110, 117, 121, 138
  abcs, 5, 6, 16, 22, 26, 52, 60, 63, 78, 89, 92, 94,
    118, 121, 132
  abcs, abcd_pomp-method (abc), 6
  abcs, ANY-method (abc), 6
  abcs, data_frame-method (abc), 6
  abcs, missing-method (abc), 6
  abcs, pomp-method (abc), 6
  abcs, probed_pomp-method (abc), 6
  abcs-data-frame (abc), 6
  abcs-pomp (abc), 6
  accumulators, 5, 9, 27, 29, 36, 38, 41, 68, 86,
    96, 99, 104, 115, 131, 133
  accumvars (accumulators), 9
  as, bsmcd_pomp-method (as.data.frame), 11
  as, Csnippet (Csnippet), 28
  as, kalman_pomp-method (as.data.frame), 11
  as, listie-method (as.data.frame), 11
  as, pfilterd_pomp-method (as.data.frame), 11
  as, probed_pomp-method (as.data.frame), 11
  as, wprocfilterd_pomp-method (as.data.frame), 11
  as, csnippet (Csnippet), 28
  as.data.frame, 11, 21, 73, 139
  bake, 13
  basic model component, 132
  basic probes, 7, 87, 90
  basic_probes, 8, 15, 89, 92, 118
  blowflies, 17, 19, 31, 42, 48, 55, 66, 70, 82,
    83, 95, 105, 112, 135
  blowflies1, 82
  blowflies1 (blowflies), 17
blowflies2, 82
blowflies2 (blowflies), 17
bsflu, 18, 19, 31, 42, 48, 55, 66, 70, 82, 83, 95, 105, 112, 135
bsmc2, 5, 8, 20, 25, 44, 45, 52, 53, 60, 63, 73, 78, 84, 85, 92, 108, 121, 139
bsmc2, ANY-method (bsmc2), 20
bsmc2, data.frame-method (bsmc2), 20
bsmc2, missing-method (bsmc2), 20
bsmc2, pomp-method (bsmc2), 20
bsmc2-pomp (bsmc2), 20
bspline.basis (bsplines), 22
bsplines, 22
coef, 23, 58
coef, listie-method (coef), 23
coef, missing-method (coef), 23
coef, objfun-method (coef), 23
coef, pomp-method (coef), 23
coef, listie (coef), 23
coef, objfun (coef), 23
coef, pomp (coef), 23
coef <- (coef), 23
coef <-, missing-method (coef), 23
coef <-, pomp-method (coef), 23
coef <-, pomp (coef), 23
coerce, bsmcd_pomp, data.frame-method
(as.data.frame), 11
coerce, Csnippet, character-method
(Csnippet), 28
coerce, kalmand_pomp, data.frame-method
(as.data.frame), 11
coerce, listie, data.frame-method
(as.data.frame), 11
coerce, objfun, data.frame-method
(as.data.frame), 11
coerce, pfilterd_pomp, data.frame-method
(as.data.frame), 11
coerce, pomp, data.frame-method
(as.data.frame), 11
coerce, probe_match_objfun, pfilterd_pomp-method
(probe), 86
coerce, probed_pomp, data.frame-method
(as.data.frame), 11
coerce, spect_match_objfun, spectd_pomp-method
(spect), 115
coerce, wpfilterd_pomp, data.frame-method
(as.data.frame), 11
coerce-objfun-data.frame
(as.data.frame), 11
coerce-pomp-data.frame (as.data.frame), 11
coerce-probe_match_objfun-probed_pomp
(probe), 86
coerce-spect_match_objfun-spectd_pomp
(spect), 115
cond.logLik, 22, 24, 44, 45, 60, 73, 78, 84, 85, 108, 139
cond.logLik (cond.logLik), 24
cond.logLik, ANY-method (cond.logLik), 24
cond.logLik, bsmcd_pomp-method
(cond.logLik), 24
cond.logLik, kalmand_pomp-method
(cond.logLik), 24
cond.logLik, missing-method
(cond.logLik), 24
cond.logLik, pfilterd_pomp-method
(cond.logLik), 24
cond.logLik, wpfilterd_pomp-method
(cond.logLik), 24
cond.logLik-bsmcd_pomp (cond.logLik), 24
cond.logLik-kalmand_pomp (cond.logLik), 24
cond.logLik-pfilterd_pomp
(cond.logLik), 24
cond.logLik-wpfilterd_pomp
(cond.logLik), 24
continue, 26, 58
continue, abcd_pomp-method (continue), 26
continue, ANY-method (continue), 26
continue, mif2d_pomp-method (continue), 26
continue, missing-method (continue), 26
continue, pmcmc-d_pomp-method (continue), 26
continue-abcd_pomp (continue), 26
continue-mif2d_pomp (continue), 26
continue-pmcmc-d_pomp (continue), 26
covariate_table, 5, 9, 26, 29, 36, 38, 41, 68, 80, 86, 96, 99, 104, 115, 131, 133
covariate_table, ANY-method
(covariate_table), 26
covariate_table, character-method
(covariate_table), 26
covariate_table, missing-method
(covariate_table), 26
covariate_table, numeric-method (covariate_table), 26
covariate_table, character, covariate_table, character-method (covariate_table), 26
covariate_table, numeric, covariate_table, numeric-method (covariate_table), 26
covmat, 27
covmat, abcd_pomp-method (covmat), 27
covmat, abclist-method (covmat), 27
covmat, ANY-method (covmat), 27
covmat, missing-method (covmat), 27
covmat, pmcmd_pomp-method (covmat), 27
covmat, pmcmclist-method (covmat), 27
covmat, probed_pomp-method (covmat), 27
covmat, abcd_pomp (covmat), 27
covmat, abclist (covmat), 27
covmat, pmcmd_pomp (covmat), 27
covmat, pmcmclist (covmat), 27
covmat, probed_pomp (covmat), 27
Csnippet, 5, 9, 27, 28, 36, 38, 40, 41, 68, 86, 96, 98, 99, 102, 104, 115, 131, 133
Csnippet-class (Csnippet), 28
dacca, 18, 19, 30, 42, 48, 55, 66, 70, 82, 83, 95, 105, 112, 135
design, 32
desolve, 46, 130
deuermultinom (distributions), 34
discrete_time (rprocess_spec), 101
distributions, 5, 9, 27, 29, 34, 38, 41, 68, 86, 96, 99, 104, 115, 131, 133
dmeasure, 36, 38–40, 46, 69, 95, 98–100, 113, 136, 137
dmeasure, ANY-method (dmeasure), 36
dmeasure, missing-method (dmeasure), 36
dmeasure, pomp-method (dmeasure), 36
dmeasure-pomp (dmeasure), 36
dmeasure_spec, 5, 9, 27, 29, 36, 37, 37, 41, 68, 86, 96, 99, 104, 115, 131, 133
dprior, 37, 38, 40, 46, 69, 95, 98–100, 113, 136, 137
dprior, ANY-method (dprior), 38
dprior, missing-method (dprior), 38
dprior, pomp-method (dprior), 38
dprior-pomp (dprior), 38
dprocess, 37, 39, 39, 46, 69, 95, 98–100, 113, 136, 137

dprocess, ANY-method (dprocess), 39
dprocess, missing-method (dprocess), 39
dprocess, pomp-method (dprocess), 39
dprocess-pomp (dprocess), 39
dprocess-pomp (dprocess), 39
dprocess-pomp (dprocess), 39
deak (kalman), 50
deak, ANY-method (kalman), 50
deak, data.frame-method (kalman), 50
deak, missing-method (kalman), 50
deak, pomp-method (kalman), 50
deak-data.frame (kalman), 50
deak-pomp (kalman), 50
ebola, 18, 19, 31, 41, 48, 55, 66, 70, 83, 95, 105, 112, 135

ebolaModel, 82
ebolaModel (ebola), 41
ebolaWA2014 (ebola), 41
eff.sample.size, 22, 25, 43, 44, 45, 58, 60, 73, 78, 84, 85, 108, 139

eff.sample.size, ANY-method (eff.sample.size), 43
eff.sample.size, bsmcmd_pomp-method (eff.sample.size), 43
eff.sample.size, missing-method (eff.sample.size), 43
eff.sample.size, pfilterd_pomp-method (eff.sample.size), 43
eff.sample.size, wfilterd_pomp-method (eff.sample.size), 43
eff.sample.size-bsmcmd_pomp (eff.sample.size), 43
eff.sample.size-pfilterd_pomp (eff.sample.size), 43
eff.sample.size-wfilterd_pomp (eff.sample.size), 43
eenkf (kalman), 50
eneakf, ANY-method (kalman), 50
eneakf, data.frame-method (kalman), 50
eneakf, missing-method (kalman), 50
eneakf, pomp-method (kalman), 50
eneakf-data.frame (kalman), 50
eneakf-pomp (kalman), 50
eneuler (rprocess_spec), 101
ewcitmeas, 83
ewcitmeas (measles), 54
ewmeas, 83
ewmeas (measles), 54
expit (transformations), 130
filter.mean, ANY-method (filter.mean), 44
filter.mean, kalmand_pomp-method (filter.mean), 44
filter.mean, missing-method (filter.mean), 44
filter.mean, pfiltrold_pomp-method (filter.mean), 44
filter.mean = kalmand_pomp (filter.mean), 44
filter.mean-pfiltrold_pomp (filter.mean), 44
filter.traj, ANY-method (filter.traj), 45
filter.traj, missing-method (filter.traj), 45
filter.traj, pfiltrold_pomp-method (filter.traj), 45
filter.traj, pfilterList-method (filter.traj), 45
filter.traj, pmcmcd_pomp-method (filter.traj), 45
filter.traj-pfiltrold_pomp (filter.traj), 45
filter.traj-pfilterList (filter.traj), 45
filter.traj-pmcmc_pomp (filter.traj), 45
filter.traj-pmcmcList (filter.traj), 45
flow, ANY-method (flow), 46
flow, missing-method (flow), 46
flow, pomp-method (flow), 46
flow-pomp (flow), 46
forecast, 47
forecast, ANY-method (forecast), 47
forecast, kalmand_pomp-method (forecast), 47
forecast, missing-method (forecast), 47
forecast-kalmand_pomp (forecast), 47
freeze (bake), 13
General rules for writing C snippets can be found here, 86, 96, 114
gillespie (rprocess_spec), 101
gillespie.hl (rprocess_spec), 101
gompertz, 18, 19, 31, 42, 47, 55, 66, 70, 82, 83, 95, 105, 112, 135
here for a definition of this term, 132
hitch, 48
inv_log_barycentric (transformations), 130
kalman, 5, 8, 22, 50, 60, 63, 78, 92, 121
kernel, 16
load, 14
log_barycentric (transformations), 130
logit (transformations), 130
logLik, 52, 58, 72, 139
logLik, ANY-method (logLik), 52
logLik, bsmcd_pomp-method (logLik), 52
logLik, kalmand_pomp-method (logLik), 52
logLik, listie-method (logLik), 52
logLik, missing-method (logLik), 52
logLik, nlf_objfun-method (logLik), 52
logLik, objfun-method (logLik), 52
logLik, pfiltrold_pomp-method (logLik), 52
logLik, pmcmcd_pomp-method (logLik), 52
logLik, probed_pomp-method (logLik), 52
logLik, spect_match_objfun-method (logLik), 52
logLik, wpfiltrold_pomp-method (logLik), 52
logLik-bsmc_pomp (logLik), 52
logLik-kalmand_pomp (logLik), 52
logLik-nlf_objfun (logLik), 52
logLik-objfun (logLik), 52
logLik-pfiltrold_pomp (logLik), 52
logLik-pmcmc_pomp (logLik), 52
logLik-probed_pomp (logLik), 52
logLik-spect_match_objfun (logLik), 52
logLik-wpfiltrold_pomp (logLik), 52
logmeanexp, 53
LondonYorke, 83
LondonYorke (measles), 54
map, 80, 127
map (skeleton_spec), 114
mcmc, 8
MCMC proposals, 7, 8, 28, 76, 78
mean, 16
measles, 18, 19, 31, 42, 48, 54, 66, 70, 83, 95, 105, 112, 135
mif2, 5, 8, 22, 25, 26, 44, 45, 52, 55, 63, 73, 78, 84, 85, 92, 104, 105, 108, 121, 125, 132, 139
mif2,ANY-method (mif2), 55
mif2,data.frame-method (mif2), 55
mif2,mif2d_pomp-method (mif2), 55
mif2,missing-method (mif2), 55
mif2,pfilterd_pomp-method (mif2), 55
mif2,pomp-method (mif2), 55
mif2-data.frame (mif2), 55
mif2-mif2d_pomp (mif2), 55
mif2-pfilterd_pomp (mif2), 55
mif2-pomp (mif2), 55
mvn.diag.rw (proposals), 93
mvn.rw (proposals), 93
nlf, 5, 8, 22, 52, 53, 60, 60, 78, 92, 121
nlf_objfun (nlf), 60
nlf_objfun,ANY-method (nlf), 60
nlf_objfun,data.frame-method (nlf), 60
nlf_objfun,missing-method (nlf), 60
nlf_objfun,nlf_objfun-method (nlf), 60
nlf_objfun,pfilterd_pomp-method (nlf), 60
nlf_objfun-data.frame (nlf), 60
nlf_objfun-nlf_objfun-method (nlf), 60
nloptr, 92, 121, 128
obs, 16, 64
obs,pomp-method (obs), 64
ode, 46, 127, 129
onestep (rprocess_spec), 101
optim, 63, 91, 92, 107, 121, 128
ou2, 18, 19, 31, 42, 48, 55, 65, 70, 83, 95, 105, 112, 135
par, 75
parameter_trans, 5, 9, 21, 27, 29, 36, 38, 41, 58, 66, 69, 80, 86, 91, 96, 99, 104, 115, 120, 127, 131, 133
parameter_trans,ANY,ANY-method (parameter_trans), 66
parameter_trans,ANY,missing-method (parameter_trans), 66
parameter_trans,character,character-method (parameter_trans), 66
parameter_trans,Csnippet,Csnippet-method (parameter_trans), 66
parameter_trans,missing,ANY-method (parameter_trans), 66
parameter_trans,missing,missing-method (parameter_trans), 66
parameter_trans,NULL,NULL-method (parameter_trans), 66
parameter_trans,pomp_fun,pomp_fun-method (parameter_trans), 66
parameter_trans-character,character (parameter_trans), 66
parameter_trans-Csnippet,Csnippet (parameter_trans), 66
parameter_trans-function,function (parameter_trans), 66
parameter_trans-function,missing (parameter_trans), 66
partrans, 68
partrans, 37, 39, 40, 46, 69, 95, 98–100, 113, 136, 137
partrans,ANY-method (partrans), 69
partrans,missing-method (partrans), 69
partrans,pomp-method (partrans), 69
partrans-pomp (partrans), 69
parus, 18, 19, 31, 42, 48, 55, 66, 69, 83, 95, 105, 112, 135
paste, 23
periodic.bspline.basis (bsplines), 22
pfilter, 5, 22, 25, 44, 45, 58, 60, 70, 78, 84, 85, 89, 108, 110, 118, 132, 137, 139
pfilter,ANY-method (pfilter), 70
pfilter,data.frame-method (pfilter), 70
pfilter,missing-method (pfilter), 70
pfilter,objectfun-method (pfilter), 70
pfilter,pfilterd_pomp-method (pfilter), 70
pfilter,pomp-method (pfilter), 70
pfilter-data.frame (pfilter), 70
pfilter-objectfun (pfilter), 70
pfilter,pfilterd_pomp (pfilter), 70
pfilter,pomp (pfilter), 70
pfilterd_pomp, 58
pfilterd_pomp (pfilter), 70
pfilterd_pomp-class (pfilter), 70
plot, 21, 73, 74, 139
plot, Abc-method (plot), 74
plot, abcd_pomp-method (plot), 74
plot, abclist-method (plot), 74
plot, bsmcd_pomp-method (plot), 74
plot, Mif2-method (plot), 74
plot, mif2d_pomp-method (plot), 74
plot, mif2list-method (plot), 74
plot, missing-method (plot), 74
plot, pfilterd_pomp-method (plot), 74
plot, pmcmc-method (plot), 74
plot, pmcmc_pomp-method (plot), 74
plot, pomp-method (plot), 74
plot, pomp_plottable-method (plot), 74
plot, probe_match_objfun-method (plot), 74
plot, probed_pomp-method (plot), 74
plot, spect_match_objfun-method (plot), 74
plot, spectd_pomp-method (plot), 74
plot, wpfilterd_pomp-method (plot), 74
plot, Abc (plot), 74
plot, bsmcd_pomp (plot), 74
plot, Mif2 (plot), 74
plot, pfilterd_pomp (plot), 74
plot, pmcmc (plot), 74
plot, pomp (plot), 74
plot, probe_match_objfun (plot), 74
plot, probed_pomp (plot), 74
plot, spect_match_objfun (plot), 74
plot, spectd_pomp (plot), 74
plot, wpfilterd_pomp (plot), 74
pmcmc, 5, 8, 22, 25, 26, 44, 45, 52, 60, 63, 73, 75, 84, 85, 92, 94, 108, 121, 125, 139
pmcmc, ANY-method (pmcmc), 75
pmcmc, data.frame-method (pmcmc), 75
pmcmc, missing-method (pmcmc), 75
pmcmc, pfilterd_pomp-method (pmcmc), 75
pmcmc, pmcmc_pomp-method (pmcmc), 75
pmcmc, pomp-method (pmcmc), 75
pmcmc, data.frame (pmcmc), 75
pmcmc-pfilterd_pomp (pmcmc), 75
pmcmc-pmcmc_pomp (pmcmc), 75
pmcmc-pomp (pmcmc), 75
pomp, 7, 21, 50, 51, 58, 62, 72, 77, 78, 79, 88, 91, 110, 117, 120, 121, 138
pomp, package (pomp-package), 4
pomp-package, 4
pomp_example (pomp_examples), 82
pomp_examples, 18, 19, 31, 42, 48, 55, 66, 70, 82, 95, 105, 112, 135
pompExample (pomp_examples), 82
pompExamples (pomp_examples), 82
pred.mean, 22, 25, 44, 45, 60, 73, 78, 83, 85, 108, 139
pred.mean, ANY-method (pred.mean), 83
pred.mean, kalmand_pomp-method (pred.mean), 83
pred.mean, missing-method (pred.mean), 83
pred.mean, pfilterd_pomp-method (pred.mean), 83
pred.mean, kalmand_pomp (pred.mean), 83
pred.mean-pfilterd_pomp (pred.mean), 83
pred.var, 22, 25, 44, 45, 60, 73, 78, 84, 84, 108, 139
pred.var, ANY-method (pred.var), 84
pred.var, missing-method (pred.var), 84
pred.var-pfilterd_pomp (pred.var), 84
pred.var, print (pred.var), 85
print, listie-method (print), 85
print, pomp_fun-method (print), 85
print, unshovable-method (print), 85
prior_spec, 5, 9, 27, 29, 36, 38, 39, 41, 68, 85, 96, 99, 104, 115, 131, 133
probe, 5, 8, 16, 53, 73, 86, 92, 110, 118, 137, 139
probe, ANY-method (probe), 86
probe, data.frame-method (probe), 86
probe, missing-method (probe), 86
probe, objfun-method (probe), 86
probe, pomp-method (probe), 86
probe, probe_match_objfun-method (probe), 86
probe, probed_pomp-method (probe), 86
probe, acf (basic_probes), 15
probe, ccf (basic_probes), 15
probe, marginal (basic_probes), 15
probes

INDEX

probe.mean (basic_probes), 15
probe.median (basic_probes), 15
probe.nlar (basic_probes), 15
probe.period (basic_probes), 15
probe.quantile (basic_probes), 15
probe.sd (basic_probes), 15
probe.var (basic_probes), 15
probe_objfun (probe.match), 89
probe_objfun.Any-method (probe.match), 89
probe_objfun.data.frame-method (probe.match), 89
probe_objfun.missing-method (probe.match), 89
probe_objfun.pomp-method (probe.match), 89
probe_objfun.probe_match_objfun-method (probe.match), 89
probe_objfun.probed_pomp-method (probe.match), 89
profileDesign (design), 32
proposals, 93
quantile, 16
rmeasure, 37, 39, 40, 46, 69, 95, 97, 99, 100, 113, 136, 137
rmeasure.Any-method (rmeasure), 97
rmeasure.missing-method (rmeasure), 97
rmeasure.pomp-method (rmeasure), 97
rmeasure-pomp (rmeasure), 97
rmeasure_spec, 5, 9, 27, 29, 36, 38, 41, 68, 86, 96, 98, 98, 104, 115, 131, 133
rprocess, 37, 39, 40, 46, 69, 95, 98, 99, 100, 113, 136, 137
rprocess.Any-method (rprocess), 99
rprocess.missing-method (rprocess), 99
rprocess.pomp-method (rprocess), 99
rprocess-pomp (rprocess), 99
rprocess_spec, 5, 9, 27, 29, 36, 38, 41, 68, 86, 96, 99, 100, 115, 131, 133
rprocess plugins, 7, 21, 51, 57, 62, 72, 77, 80, 88, 91, 109, 117, 120, 138
rprocess.Any-method (rprocess), 100
rprocess.missing-method (rprocess), 100
rprocess.pomp-method (rprocess), 100
rprocess-pomp (rprocess), 100
rprocess_spec, 5, 9, 27, 29, 36, 38, 41, 68, 86, 96, 99, 100, 101, 115, 131, 133
runifDesign, 32
runifDesign (design), 32
rW2, 18, 19, 31, 42, 48, 55, 66, 70, 83, 95, 105, 112, 135
sannbox, 106
saved.states, 22, 25, 44, 45, 60, 73, 78, 84, 85, 107, 139
saved.states.Any-method (saved.states), 107
saved.states.missing-method (saved.states), 107
saved.states.pfilter_pomp-method (saved.states), 107
saved.states.pfilter_pomp-list-method (saved.states), 107
saved.states.pfilter_pomp (saved.states), 107
saved.states.pfilter_pomp-list (saved.states), 107
see ?rprocess_spec for the documentation on these plugins, 7, 21, 51, 57, 62, 72, 77, 80, 88, 91, 109, 117, 120, 138
R CMD SHLIB, 28, 29
readRDS, 14
reulearmultinom (distributions), 34
rgamma (distributions), 34
ricker, 18, 19, 31, 42, 48, 55, 66, 70, 83, 94, 105, 112, 135
rinit, 37, 39, 40, 46, 69, 95, 98–100, 113, 137
rinit.Any-method (rinit), 95
rinit.missing-method (rinit), 95
rinit.pomp-method (rinit), 95
rinit-pomp (rinit), 95
rinit_spec, 5, 9, 27, 29, 36, 38, 41, 68, 86, 95, 96, 99, 104, 115, 131, 133
rmeasure, 37, 39, 40, 46, 69, 95, 97, 99, 100, 113, 136, 137
rmeasure.Any-method (rmeasure), 97
rmeasure.missing-method (rmeasure), 97
rmeasure.pomp-method (rmeasure), 97
rmeasure-pomp (rmeasure), 97
rmeasure_spec, 5, 9, 27, 29, 36, 38, 41, 68, 86, 96, 98, 98, 104, 115, 131, 133
rprocess, 37, 39, 40, 46, 69, 95, 98, 99, 100, 113, 136, 137
rprocess.Any-method (rprocess), 99
rprocess.missing-method (rprocess), 99
rprocess.pomp-method (rprocess), 99
rprocess-pomp (rprocess), 99
rprocess_spec, 5, 9, 27, 29, 36, 38, 41, 68, 86, 96, 99, 100, 115, 131, 133
runifDesign, 32
runifDesign (design), 32
rW2, 18, 19, 31, 42, 48, 55, 66, 70, 83, 95, 105, 112, 135
sannbox, 106
saved.states, 22, 25, 44, 45, 60, 73, 78, 84, 85, 107, 139
saved.states.Any-method (saved.states), 107
saved.states.missing-method (saved.states), 107
saved.states.pfilter_pomp-method (saved.states), 107
saved.states.pfilter_pomp-list-method (saved.states), 107
saved.states.pfilter_pomp (saved.states), 107
saved.states.pfilter_pomp-list (saved.states), 107
see ?rprocess_spec for the documentation on these plugins, 7, 21, 51, 57, 62, 72, 77, 80, 88, 91, 109, 117, 120, 138
timezero, pomp-method (timezero), 124
timezero-pomp (timezero), 124
timezero<-(timezero), 124
timezero<-, ANY-method (timezero), 124
timezero<-, missing-method (timezero), 124
timezero<-, pomp-method (timezero), 124
traces, 124
traces, Abc-method (traces), 124
traces, abcd_pomp-method (traces), 124
traces, abclist-method (traces), 124
traces, ANY-method (traces), 124
traces, Mif2-method (traces), 124
traces, mif2d_pomp-method (traces), 124
traces, mif2List-method (traces), 124
traces, missing-method (traces), 124
traces, Pmcmc-method (traces), 124
traces, pmcncl_pomp-method (traces), 124
traces, pmcnclList-method (traces), 124
traces=Abc (traces), 124
traces=abcd_pomp (traces), 124
traces=abclist (traces), 124
traces=Mif2 (traces), 124
traces=mif2d_pomp (traces), 124
traces=mif2List (traces), 124
traces=Pmcmc (traces), 124
traces=pmcncl_pomp (traces), 124
traces=pmcnclList (traces), 124
traj.match, 5, 107, 114, 126
traj.objfun (traj.match), 126
traj.objfun, ANY-method (traj.match), 126
traj.objfun, data.frame-method (traj.match), 126
traj.objfun, missing-method (traj.match), 126
traj.objfun, pomp-method (traj.match), 126
traj.objfun, traj_match_objfun-method (traj.match), 126
traj.objfun, data.frame (traj.match), 126
traj.objfun, pomp (traj.match), 126
traj.objfun-traj_match_objfun (traj.match), 126
trajectory, 46, 113, 114, 128, 129, 137
trajectory, ANY-method (trajectory), 129
trajectory, missing-method (trajectory), 129
trajectory, pomp-method (trajectory), 129
trajectory, traj_match_objfun-method (trajectory), 129
trajectory-pomp (trajectory), 129
trajectory-traj_match_objfun (trajectory), 129
transformations, 5, 9, 27, 29, 36, 38, 41, 68, 86, 96, 99, 104, 115, 130, 133
userdata, 5, 9, 27, 29, 36, 38, 41, 68, 86, 96, 99, 104, 115, 131, 133
vectorfield, 80, 127
vectorfield (skelton_spec), 114
verhulst, 18, 19, 31, 42, 48, 55, 66, 70, 83, 95, 105, 112, 134
window, 136
window, pomp-method (window), 136
window=pomp (window), 136
workhorses, 37, 39, 40, 46, 48, 69, 95, 98–100, 113, 136
wpfilter, 5, 22, 25, 44, 45, 60, 73, 78, 84, 85, 89, 108, 110, 118, 137
wpfilter, ANY-method (wpfilter), 137
wpfilter, data.frame-method (wpfilter), 137
wpfilter, missing-method (wpfilter), 137
wpfilter, pomp-method (wpfilter), 137
wpfilter, wpfilterd_pomp-method (wpfilter), 137
wpfilter, data.frame (wpfilter), 137
wpfilter-pomp (wpfilter), 137
wpfilter-wpfilterd_pomp (wpfilter), 137
wpfilterd_pomp (wpfilter), 137
wpfilterd_pomp-class (wpfilter), 137