Package ‘pomp’

July 6, 2022

Type Package

Title Statistical Inference for Partially Observed Markov Processes

Version 4.3

Date 2022-07-06

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 'pstop.R' 'undefined.R' 'package.R' 'csnippet.R' 'pomp_fun.R' 'parameter_trans.R' 'covariate_table.R' 'skeleton_spec.R' 'rprocess_spec.R' 'safecall.R' 'pomp_class.R' 'load.R' 'workhorses.R' 'continue.R' 'summary.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' 'authors.R' 'bake.R' 'basic_components.R' 'basic_probes.R' 'betabinom.R' 'blowflies.R' 'bsflu.R' 'bsplines.R' 'childhood.R' 'coef.R' 'concat.R' 'cond_logLik.R' 'covmat.R' 'dacca.R' 'design.R' 'distributions.R' 'ebola.R' 'eff_sample_size.R' 'elementary_algorithms.R' 'emeasure_spec.R' 'estimation_algorithms.R' 'extract.R' 'filter_mean.R' 'filter_traj.R' 'flow.R' 'forecast.R' 'gompertz.R' 'kf.R' 'probe_match.R' 'spect_match.R' 'nlf.R' 'trajectory.R' 'traj_match.R' 'objfun.R' 'loglik.R' 'logmeanexp.R' 'lookup.R' 'mcap.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_design.R' 'spy.R' 'states.R' 'time.R' 'timezero.R' 'traces.R' 'transformations.R' 'userdata.R' 'verhulst.R' 'vmeasure_spec.R' 'window.R'

RoxygenNote 7.2.0

NeedsCompilation yes

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

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

Repository CRAN

Date/Publication 2022-07-06 19:20:02 UTC

R topics documented:

`pomp-package` *Inference for partially observed Markov processes*

Description

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

Data analysis using pomp

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` and `wpfilter`.
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 nonlinear forecasting.
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 matching.
7. Synthetic likelihood a la Wood (2010); see `probe`.
8. A spectral probe-matching method (Reuman et al. 2006, 2008); see spectrum matching.
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 trajectory matching.

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

Structure of the package

pomp algorithms are arranged on several levels. At the top level, estimation algorithms estimate model parameters and return information needed for other aspects of inference. Elementary algorithms perform common operations on POMP models, including simulation, filtering, and application of diagnostic probes; these functions may be useful in inference, but they do not themselves perform estimation. At the lowest level, workhorse functions provide the interface to basic POMP model components. Beyond these, **pomp** provides a variety of auxiliary functions for manipulating and extracting information from ‘pomp’ objects, producing diagnostic plots, facilitating reproducible computations, and so on.

Implementing a model

The basic structure at the heart of the package is the ‘pomp object’. This is a container holding a time series of data (possibly multivariate) and a model. The model is specified by specifying some or all of its basic model components. One does this using the basic component arguments to the `pomp` constructor. One can also add, modify, or delete basic model components “on the fly” in any **pomp** function that accepts them.

Documentation and examples

The package contains a number of examples. Some of these are included in the help pages. In addition, several pre-built POMP models are included with the package. Tutorials and other documentation, including a package FAQ, are available from the package website.

Useful links

- **pomp** homepage: `https://kingaa.github.io/pomp/`
- Report bugs to: `https://github.com/kingaa/pomp/issues`
- Frequently asked questions: `https://kingaa.github.io/pomp/FAQ.html`
- User guides and tutorials: `https://kingaa.github.io/pomp/docs.html`
- **pomp** news: `https://kingaa.github.io/pomp/blog.html`

Citing pomp

Execute `citation("pomp")` to view the correct citation for publications.

Author(s)

Aaron A. King

References

A. A. King, D. Nguyen, and E. L. Ionides. Statistical inference for partially observed Markov processes via the package **pomp**. *Journal of Statistical Software* **69**(12), 1–43, 2016. An updated version of this paper is available on the package website.

See the package website for more references, including many publications that use **pomp**.

See Also

More on implementing POMP models: `Csnippet`, `accumulator variables`, `basic components`, `betabinomial`, `covariates`, `distributions`, `dmeasure specification`, `dprocess specification`, `emeasure specification`, `parameter transformations`, `pomp`, `prior specification`, `rinit specification`, `rmeasure specification`, `rprocess specification`, `skeleton specification`, `transformations`, `userdata`, `vmeasure specification`

More on **pomp** workhorse functions: `dmeasure()`, `dprior()`, `dprocess()`, `emeasure()`, `flow()`, `partrans()`, `rinit()`, `rmeasure()`, `rprior()`, `rprocess()`, `skeleton()`, `vmeasure()`, `workhorses`

More on **pomp** estimation algorithms: `approximate Bayesian computation`, `bsmc2()`, `estimation algorithms`, `mif2()`, `nonlinear forecasting`, `pmcmc()`, `probe matching`, `spectrum matching`

More on **pomp** elementary algorithms: `elementary algorithms`, `kalman`, `pfilter`(), `probe`(), `simulate`(), `spect`(), `trajectory`(), `wpfilter`()

`accumulator variables` *accumulator variables*

Description

Latent state variables that accumulate quantities through time.

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`

More on implementing POMP models: `Csnippet`, `basic components`, `betabinomial`, `covariates`, `distributions`, `dmeasure specification`, `dprocess specification`, `emeasure specification`, `parameter transformations`, `pomp-package`, `pomp`, `prior specification`, `rinit specification`, `rmeasure specification`, `rprocess specification`, `skeleton specification`, `transformations`, `userdata`, `vmeasure specification`

Examples

```
## 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 = rgammawn(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

ew1 %>%
  simulate() %>%
  plot(variables=c("S","I","R"))

## A simple SIR model that tracks cumulative incidence.

ew1 %>%
  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 = rgammawn(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];
      H += trans[3];          // cumulative incidence
    "),
    delta.t=1/52/20
    ),
    rmeasure=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()
```

`approximate Bayesian computation`

Approximate Bayesian computation

Description

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

Usage

```
## S4 method for signature 'data.frame'
abc(
  data,
  Nabc = 1,
  proposal,
  scale,
  epsilon,
  probes,
  params,
  rinit,
  rprocess,
  rmeasure,
  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,
    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. Internally, `data` will be internally coerced to an array with storage-mode `double`.
`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 specification.
`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 specification 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 specification.
`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 specification. 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:

`abc`	repeats the calculation, beginning with the last state
`continue`	continues the `abc` calculation
`plot`	produces a series of diagnostic plots
`traces`	produces an `mcmc` object, to which the various **coda** convergence diagnostics can be applied

Note for Windows users

Some Windows users report problems when using C snippets in parallel computations. These appear to arise when the temporary files created during the C snippet compilation process are not handled properly by the operating system. To circumvent this problem, use the `cdir` and `cfile` options to cause the C snippets to be written to a file of your choice, thus avoiding the use of temporary files altogether.

Author(s)

Edward L. Ionides, Aaron A. King

References

J.-M. Marin, P. Pudlo, C. P. Robert, and R. J. Ryder. Approximate Bayesian computational methods. *Statistics and Computing* **22**, 1167–1180, 2012.

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

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

See Also

More on methods based on summary statistics: `basic probes`, `nonlinear forecasting`, `probe matching`, `probe()`, `spectrum matching`, `spect()`

More on **pomp** estimation algorithms: `bsmc2()`, `estimation algorithms`, `mif2()`, `nonlinear forecasting`, `pmcmc()`, `pomp-package`, `probe matching`, `spectrum matching`

More on Markov chain Monte Carlo methods: `pmcmc()`, `proposals`

More on Bayesian methods: `bsmc2()`, `dprior()`, `pmcmc()`, `prior specification`, `rprior()`

`basic components` *Basic POMP model components.*

Description

Mathematically, the parts of a POMP model include the latent-state process transition distribution, the measurement-process distribution, the initial-state distribution, and possibly a prior parameter distribution. Algorithmically, each of these corresponds to at least two distinct operations. In particular, for each of the above parts, one sometimes needs to make a random draw from the distribution and sometimes to evaluate the density function. Accordingly, for each such component, there are two basic model components, one prefixed by a ‘r’, the other by a ‘d’, following the usual R convention.

Details

In addition to the parts listed above, **pomp** includes two additional basic model components: the deterministic skeleton, and parameter transformations that can be used to map the parameter space onto a Euclidean space for estimation purposes.

There are thus altogether eleven **basic model components**:

1. rprocess, which samples from the latent-state transition distribution,
2. dprocess, which evaluates the latent-state transition density,
3. rmeasure, which samples from the measurement distribution,
4. emeasure, which computes the conditional expectation of the measurements, given the latent states,

5. vmeasure, which computes the conditional covariance matrix of the measurements, given the latent states,
6. dmeasure, which evaluates the measurement density,
7. rprior, which samples from the prior distribution,
8. dprior, which evaluates the prior density,
9. rinit, which samples from the initial-state distribution,
10. skeleton, which evaluates the deterministic skeleton,
11. partrans, which evaluates the forward or inverse parameter transformations.

Each of these can be set or modified in the `pomp` constructor function or in any of the **pomp** elementary algorithms or estimation algorithms using an argument that matches the basic model component. A basic model component can be unset by passing `NULL` in the same way.

Help pages detailing each basic model component are provided.

See Also

workhorse functions, elementary algorithms, estimation algorithms.

More on implementing POMP models: `Csnippet`, `accumulator variables`, `betabinomial`, `covariates`, `distributions`, `dmeasure specification`, `dprocess specification`, `emeasure specification`, `parameter transformations`, `pomp-package`, `pomp`, `prior specification`, `rinit specification`, `rmeasure specification`, `rprocess specification`, `skeleton specification`, `transformations`, `userdata`, `vmeasure specification`

`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

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

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

See Also

More on methods based on summary statistics: `approximate Bayesian computation`, `nonlinear forecasting`, `probe matching`, `probe`(), `spectrum matching`, `spect`()

`betabinomial` *Beta-binomial distribution*

Description

Density and random generation for the Beta-binomial distribution with parameters `size`, `mu`, and `theta`.

Usage

```
rbetabinom(n = 1, size, prob, theta)

dbetabinom(x, size, prob, theta, log = FALSE)
```

Arguments

`n`	integer; number of random variates to generate.
`size`	`size` parameter of the binomial distribution
`prob`	mean of the Beta distribution
`theta`	Beta distribution dispersion parameter
`x`	vector of non-negative integer quantiles
`log`	logical; if TRUE, return logarithm(s) of probabilities.

Details

A variable X is Beta-binomially distributed if X $Binomial(n, P)$ where P $Beta(mu, theta)$. Using the standard (a,b) parameterization, $a = mu * theta$ and $b = (1 - mu) * theta$.

Value

`rbetabinom`	Returns a vector of length `n` containing random variates drawn from the Beta-binomial distribution.
`dbetabinom`	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`, `prob`, `theta`).

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.

See Also

More on implementing POMP models: `Csnippet`, `accumulator variables`, `basic components`, `covariates`, `distributions`, `dmeasure specification`, `dprocess specification`, `emeasure specification`, `parameter transformations`, `pomp-package`, `pomp`, `prior specification`, `rinit specification`, `rmeasure specification`, `rprocess specification`, `skeleton specification`, `transformations`, `userdata`, `vmeasure specification`

`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

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

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:

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

$$S(t+1) \sim \mathrm{Binomial}(N(t), \exp{(-\delta\epsilon(t+1)\Delta t)})$$

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

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

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

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

`blowflies1` and `blowflies2` return 'pomp' objects containing the actual data and two variants of the model.

References

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

Y. Xia and H. Tong. Feature matching in time series modeling. *Statistical Science* **26**, 21–46, 2011.

E.L. Ionides. Discussion of “Feature matching in time series modeling” by Y. Xia and H. Tong. *Statistical Science* **26**, 49–52, 2011.

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

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

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

See Also

More examples provided with **pomp**: `SIR models`, `childhood disease data`, `dacca()`, `ebola`, `gompertz()`, `ou2()`, `pomp examples`, `ricker()`, `rw2()`, `verhulst()`

More data sets provided with **pomp**: `bsflu`, `childhood disease data`, `dacca()`, `ebola`, `parus`

Examples

```
plot(blowflies1())
plot(blowflies2())
```

`bsflu` *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

Anonymous. Influenza in a boarding school. *British Medical Journal* **1**, 587, 1978.

See Also

SIR models

More data sets provided with **pomp**: `blowflies`, `childhood disease data`, `dacca`(), `ebola`, `parus`

Examples

```
if (require(tidyr) && require(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

Modified version of the Liu & West (2001) algorithm.

Usage

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

## 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. Internally, `data` will be internally coerced to an array with storage-mode `double`.
`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 specification. Setting `rprior=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=NULL` sets the initial-state simulator to its default. For more information, see rinit specification.
`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 specification 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 specification.
`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

`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". `Np` draws are made from the prior distribution.

Value

An object of class 'bsmcd_pomp'. The following methods are avaiable:

`plot` produces diagnostic plots

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

Note for Windows users

Some Windows users report problems when using C snippets in parallel computations. These appear to arise when the temporary files created during the C snippet compilation process are not handled properly by the operating system. To circumvent this problem, use the `cdir` and `cfile` options to cause the C snippets to be written to a file of your choice, thus avoiding the use of temporary files altogether.

Author(s)

Michael Lavine, Matthew Ferrari, Aaron A. King, Edward L. Ionides

References

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

See Also

More on Bayesian methods: `approximate Bayesian computation`, `dprior()`, `pmcmc()`, `prior specification`, `rprior()`

More on full-information (i.e., likelihood-based) methods: `mif2()`, `pfilter()`, `pmcmc()`, `wpfilter()`

More on sequential Monte Carlo methods: `cond.logLik()`, `eff.sample.size()`, `filter.mean()`, `filter.traj()`, `kalman`, `mif2()`, `pfilter()`, `pmcmc()`, `pred.mean()`, `pred.var()`, `saved.states()`, `wpfilter()`

More on **pomp** estimation algorithms: `approximate Bayesian computation`, `estimation algorithms`, `mif2()`, `nonlinear forecasting`, `pmcmc()`, `pomp-package`, `probe matching`, `spectrum matching`

`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.
`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

See Also

More on interpolation: `covariates`, `lookup()`

Examples

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

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

`childhood disease data` *Historical childhood disease incidence data*

Description

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

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

References

W. P. London and J. A. Yorke, Recurrent outbreaks of measles, chickenpox and mumps: I. Seasonal variation in contact rates. *American Journal of Epidemiology* **98**, 453–468, 1973.

See Also

SIR models, `bsflu`

More data sets provided with **pomp**: `blowflies`, `bsflu`, `dacca()`, `ebola`, `parus`

More examples provided with **pomp**: `SIR models`, `blowflies`, `dacca()`, `ebola`, `gompertz()`, `ou2()`, `pomp examples`, `ricker()`, `rw2()`, `verhulst()`

Examples

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

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

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

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

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

`coef` *Extract, set, or alter coefficients*

Description

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

Usage

```
## 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, ...)

## S4 replacement method for signature 'objfun'
coef(object, pars, transform = FALSE, ...) <- value
```

Arguments

`object`	an object of class ‘pomp’, or of a class extending ‘pomp’
`...`	ignored or passed to the more primitive function
`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.

See Also

Other extraction methods: `cond.logLik()`, `covmat()`, `eff.sample.size()`, `filter.mean()`, `filter.traj()`, `forecast()`, `logLik`, `obs()`, `pred.mean()`, `pred.var()`, `saved.states()`, `spy()`, `states()`, `summary()`, `timezero()`, `time()`, `traces()`

`cond.logLik` *Conditional log likelihood*

Description

The estimated conditional log likelihood from a fitted model.

Usage

```
## 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, ...)
```

Arguments

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

Details

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

$$Y(t_k)|Y(t_1),\dots,Y(t_{k-1})$$

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

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

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

More on sequential Monte Carlo methods: `bsmc2()`, `eff.sample.size()`, `filter.mean()`, `filter.traj()`, `kalman`, `mif2()`, `pfilter()`, `pmcmc()`, `pred.mean()`, `pred.var()`, `saved.states()`, `wpfilter()`

Other extraction methods: `coef()`, `covmat()`, `eff.sample.size()`, `filter.mean()`, `filter.traj()`, `forecast()`, `logLik`, `obs()`, `pred.mean()`, `pred.var()`, `saved.states()`, `spy()`, `states()`, `summary()`, `timezero()`, `time()`, `traces()`

`continue` *Continue an iterative calculation*

Description

Continue an iterative computation where it left off.

Usage

```
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`

`covariates` *Covariates*

Description

Incorporating time-varying covariates using lookup tables.

Usage

```
## 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

More on implementing POMP models: `Csnippet`, `accumulator variables`, `basic components`, `betabinomial`, `distributions`, `dmeasure specification`, `dprocess specification`, `emeasure specification`, `parameter transformations`, `pomp-package`, `pomp`, `prior specification`, `rinit specification`, `rmeasure specification`, `rprocess specification`, `skeleton specification`, `transformations`, `userdata`, `vmeasure specification`

More on interpolation: `bsplines`, `lookup()`

`covmat` *Estimate a covariance matrix from algorithm traces*

Description

A helper function to extract a covariance matrix.

Usage

```
## 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.

Other extraction methods: `coef()`, `cond.logLik()`, `eff.sample.size()`, `filter.mean()`, `filter.traj()`, `forecast()`, `logLik`, `obs()`, `pred.mean()`, `pred.var()`, `saved.states()`, `spy()`, `states()`, `summary()`, `timezero()`, `time()`, `traces()`

`Csnippet` *C snippets*

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. TODO: Include information about linking to precompiled libraries. (e.g., Discussion #156).

C snippets are salted

To prevent collisions in parallel computations, a 'pomp' object built using C snippets is "salted" with the current time and a random number. A result is that two 'pomp' objects, built on identical codes and data, will **not** be identical as R objects, though they will be functionally identical in every respect.

Note for Windows users

Some Windows users report problems when using C snippets in parallel computations. These appear to arise when the temporary files created during the C snippet compilation process are not handled properly by the operating system. To circumvent this problem, use the `cdir` and `cfile` options to cause the C snippets to be written to a file of your choice, thus avoiding the use of temporary files altogether.

See Also

spy

More on implementing POMP models: `accumulator variables`, `basic components`, `betabinomial`, `covariates`, `distributions`, `dmeasure specification`, `dprocess specification`, `emeasure specification`, `parameter transformations`, `pomp-package`, `pomp`, `prior specification`, `rinit specification`, `rmeasure specification`, `rprocess specification`, `skeleton specification`, `transformations`, `userdata`, `vmeasure specification`

---

dacca *Model of cholera transmission for historic Bengal.*

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

`logomega`	seasonal environmental reservoir parameters
`sd_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

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

See Also

More examples provided with **pomp**: `SIR models`, `blowflies`, `childhood disease data`, `ebola`, `gompertz()`, `ou2()`, `pomp examples`, `ricker()`, `rw2()`, `verhulst()`

More data sets provided with **pomp**: `blowflies`, `bsflu`, `childhood disease data`, `ebola`, `parus`

Examples

```
## Not run:
  po <- dacca()
  plot(po)
  ## MLE:
  coef(po)
  plot(simulate(po))

## End(Not run)
```

`design` *Design matrices for pomp calculations*

Description

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

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

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

`slice_design` generates points along slices through a specified point.

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

Usage

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

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

slice_design(center, ...)

sobol_design(lower = numeric(0), upper = numeric(0), nseq)
```

Arguments

`...`	In `profile_design`, additional arguments specify the parameters over which to profile and the values of these parameters. In `slice_design`, 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="runif"` uses `runif_design`. `type="sobol"` uses `sobol_design`;
`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

`profile_design` returns a data frame with `nprof` points per profile point.

`runif_design` returns a data frame with `nseq` rows and one column for each variable named in `lower` and `upper`.

`slice_design` returns a data frame with one row per point. The 'slice' variable indicates which slice the point belongs to.

`sobol_design` returns a data frame with `nseq` rows and one column for each variable named in `lower` and `upper`.

Author(s)

Aaron A. King

References

S. Kucherenko and Y. Sytsko. Application of deterministic low-discrepancy sequences in global optimization. *Computational Optimization and Applications* **30**, 297–318, 2005. doi:10.1007/s10589-00546151.

S.G. Johnson. The **NLopt** nonlinear-optimization package. `https://github.com/stevengj/nlopt/`.

P. Bratley and B.L. Fox. Algorithm 659 Implementing Sobol's quasirandom sequence generator. *ACM Transactions on Mathematical Software* **14**, 88–100, 1988.

S. Joe and F.Y. Kuo. Remark on algorithm 659: Implementing Sobol' quasirandom sequence generator. *ACM Transactions on Mathematical Software* **29**, 49–57, 2003.

Examples

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

## Uniform random design
plot(runif_design(lower=c(a=0,b=100),upper=c(b=200,a=1),100))

## A one-parameter profile design:
x <- profile_design(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 <- profile_design(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 <- profile_design(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.
x <- slice_design(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 <- slice_design(c(A=3,B=8,C=0),A=seq(0,5,by=1),C=seq(0,5,length=11))
dim(x)
plot(x)
```

`distributions` *Probability distributions*

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 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 Δt, the number of individuals remaining alive and dying in each way is multinomially distributed:

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

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

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

In this case, we say that

$$(\Delta n_1, \ldots, \Delta n_k) \sim \mathrm{Eulermultinom}(N, r, \Delta t),$$

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

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

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

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

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 equidispersed process be given by

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

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

```
dW <- rgammawn(sigma=sigma,dt=dt)

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

or

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

He et al. (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

C. Bretó and E. L. Ionides. Compound Markov counting processe and their applications to modeling infinitesimally over-dispersed systems. *Stochastic Processes and their Applications* **121**, 2571–2591, 2011.

D. He, E.L. Ionides, & A.A. King. Plug-and-play inference for disease dynamics: measles in large and small populations as a case study. *Journal of the Royal Society Interface* **7**, 271–283, 2010.

See Also

More on implementing POMP models: `Csnippet`, `accumulator variables`, `basic components`, `betabinomial`, `covariates`, `dmeasure specification`, `dprocess specification`, `emeasure specification`, `parameter transformations`, `pomp-package`, `pomp`, `prior specification`, `rinit specification`, `rmeasure specification`, `rprocess specification`, `skeleton specification`, `transformations`, `userdata`, `vmeasure specification`

Examples

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

`dmeasure` *dmeasure*

Description

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

Usage

```
## S4 method for signature 'pomp'
dmeasure(
  object,
  y = obs(object),
  x = states(object),
  times = time(object),
  params = coef(object),
  ...,
  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 specification

More on **pomp** workhorse functions: `dprior()`, `dprocess()`, `emeasure()`, `flow()`, `partrans()`, `pomp-package`, `rinit()`, `rmeasure()`, `rprior()`, `rprocess()`, `skeleton()`, `vmeasure()`, `workhorses`

`dmeasure specification` *The measurement model density*

Description

Specification of the measurement model density function, dmeasure.

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 compute the probability density of an observation given the value of the latent state variables. Then you can furnish

```
dmeasure = 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. 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

```
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 `...`, and `log`. It can take additional arguments via the userdata 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 = TRUE` and `log = FALSE` when writing the `dmeasure` C snippet or function.

Default behavior

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

Note for Windows users

Some Windows users report problems when using C snippets in parallel computations. These appear to arise when the temporary files created during the C snippet compilation process are not handled properly by the operating system. To circumvent this problem, use the `cdir` and `cfile` options to cause the C snippets to be written to a file of your choice, thus avoiding the use of temporary files altogether.

See Also

`dmeasure`

More on implementing POMP models: `Csnippet`, `accumulator variables`, `basic components`, `betabinomial`, `covariates`, `distributions`, `dprocess specification`, `emeasure specification`, `parameter transformations`, `pomp-package`, `pomp`, `prior specification`, `rinit specification`, `rmeasure specification`, `rprocess specification`, `skeleton specification`, `transformations`, `userdata`, `vmeasure specification`

Examples

```
## We start with the pre-built Ricker example:
ricker() -> po

## To change the measurement model density, dmeasure,
## we use the 'dmeasure' argument in any 'pomp'
## elementary or estimation function.
## Here, we pass the dmeasure specification to 'pfilter'
## as an R function.

po %>%
  pfilter(
    dmeasure=function (y, N, phi, ..., log) {
      dpois(y,lambda=phi*N,log=log)
    },
    Np=100
  ) -> pf

## We can also pass it as a C snippet:

po %>%
  pfilter(
    dmeasure=Csnippet("lik = dpois(y,phi*N,give_log);"),
```

```
    paramnames="phi",
    statenames="N",
    Np=100
  ) -> pf
```

---

dprior *dprior*

Description

Evaluates the prior probability density.

Usage

```
## S4 method for signature 'pomp'
dprior(object, params = coef(object), ..., 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 specification

More on **pomp** workhorse functions: `dmeasure()`, `dprocess()`, `emeasure()`, `flow()`, `partrans()`, `pomp-package`, `rinit()`, `rmeasure()`, `rprior()`, `rprocess()`, `skeleton()`, `vmeasure()`, `workhorses`

More on Bayesian methods: `approximate Bayesian computation`, `bsmc2()`, `pmcmc()`, `prior specification`, `rprior()`

`dprocess` *dprocess*

Description

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

Usage

```
## S4 method for signature 'pomp'
dprocess(
  object,
  x = states(object),
  times = time(object),
  params = coef(object),
  ...,
  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.
`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=TRUE`) of the transition from state `x[,j,k-1]` at time `times[k-1]` to state `x[,j,k]` at time `times[k]`.

See Also

Specification of the process-model density evaluator: dprocess specification

More on **pomp** workhorse functions: `dmeasure()`, `dprior()`, `emeasure()`, `flow()`, `partrans()`, `pomp-package`, `rinit()`, `rmeasure()`, `rprior()`, `rprocess()`, `skeleton()`, `vmeasure()`, `workhorses`

`dprocess specification`

The latent state process density

Description

Specification of the latent state process density function, 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 `var` is a state variable, when `f` is called, `var_1` will value of state variable `var` at time `t_1`, `var_2` will have the value of `var` at time `t_2`. `f` should return the *log* likelihood of a transition from `x1` at time `t1` to `x2` at time `t2`, assuming that no intervening transitions have occurred.

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`).

Note for Windows users

Some Windows users report problems when using C snippets in parallel computations. These appear to arise when the temporary files created during the C snippet compilation process are not handled properly by the operating system. To circumvent this problem, use the `cdir` and `cfile` options to cause the C snippets to be written to a file of your choice, thus avoiding the use of temporary files altogether.

See Also

`dprocess`

More on implementing POMP models: `Csnippet`, `accumulator variables`, `basic components`, `betabinomial`, `covariates`, `distributions`, `dmeasure specification`, `emeasure specification`, `parameter transformations`, `pomp-package`, `pomp`, `prior specification`, `rinit specification`, `rmeasure specification`, `rprocess specification`, `skeleton specification`, `transformations`, `userdata`, `vmeasure specification`

`ebola` *Ebola outbreak, West Africa, 2014-2016*

Description

Data and models for the 2014–2016 outbreak of Ebola virus disease in West Africa.

Usage

```
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 (WHO Ebola Response Team 2014) for the appropriate country or region will be used. Providing a dataset here will override this behavior.
`timestep`	duration (in days) of Euler timestep 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 ρC_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 `cfr`.

References

A.A. King, M. Domenech de Cellès, F.M.G. Magpantay, and P. Rohani. Avoidable errors in the modelling of outbreaks of emerging pathogens, with special reference to Ebola. *Proceedings of the Royal Society of London, Series B* **282**, 20150347, 2015.

WHO Ebola Response Team. Ebola virus disease in West Africa—the first 9 months of the epidemic and forward projections. *New England Journal of Medicine* **371**, 1481–1495, 2014.

D. He, E.L. Ionides, & A.A. King. Plug-and-play inference for disease dynamics: measles in large and small populations as a case study. *Journal of the Royal Society Interface* **7**, 271–283, 2010.

See Also

More data sets provided with **pomp**: `blowflies`, `bsflu`, `childhood disease data`, `dacca()`, `parus`

More examples provided with **pomp**: `SIR models`, `blowflies`, `childhood disease data`, `dacca()`, `gompertz()`, `ou2()`, `pomp examples`, `ricker()`, `rw2()`, `verhulst()`

Examples

```
if (require(ggplot2) && require(tidyr)) {

  data(ebolaWA2014)

  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"))
}
```

`eff.sample.size` *Effective 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

More on sequential Monte Carlo methods: `bsmc2()`, `cond.logLik()`, `filter.mean()`, `filter.traj()`, `kalman`, `mif2()`, `pfilter()`, `pmcmc()`, `pred.mean()`, `pred.var()`, `saved.states()`, `wpfilter()`

Other extraction methods: `coef()`, `cond.logLik()`, `covmat()`, `filter.mean()`, `filter.traj()`, `forecast()`, `logLik`, `obs()`, `pred.mean()`, `pred.var()`, `saved.states()`, `spy()`, `states()`, `summary()`, `timezero()`, `time()`, `traces()`

`elementary algorithms` *Elementary computations on POMP models.*

Description

In **pomp**, elementary algorithms perform POMP model operations. These operations do not themselves estimate parameters, though they may be instrumental in inference methods.

Details

There are six elementary algorithms in **pomp**:

- `simulate` which simulates from the joint distribution of latent and observed variables,
- `pfilter`, which performs a simple particle filter operation,
- `wpfilter`, which performs a weighted particle filter operation,
- `probe`, which computes a suite of user-specified summary statistics to actual and simulated data,
- `spect`, which performs a power-spectral density function computation on actual and simulated data,
- `trajectory`, which iterates or integrates the deterministic skeleton (according to whether the latter is a (discrete-time) map or a (continuous-time) vectorfield.

Help pages detailing each elementary algorithm component are provided.

See Also

basic model components, workhorse functions, estimation algorithms.

More on **pomp** elementary algorithms: `kalman`, `pfilter()`, `pomp-package`, `probe()`, `simulate()`, `spect()`, `trajectory()`, `wpfilter()`

`emeasure` *emeasure*

Description

Return the expected value of the observed variables, given values of the latent states and the parameters.

Usage

```
## S4 method for signature 'pomp'
emeasure(
  object,
  x = states(object),
  times = time(object),
  params = coef(object),
  ...
)
```

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

`emeasure` 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 expectation: emeasure specification

More on **pomp** workhorse functions: `dmeasure()`, `dprior()`, `dprocess()`, `flow()`, `partrans()`, `pomp-package`, `rinit()`, `rmeasure()`, `rprior()`, `rprocess()`, `skeleton()`, `vmeasure()`, `workhorses`

`emeasure specification` *The expectation of the measurement model*

Description

Specification of the measurement-model conditional expectation, emeasure.

Details

The measurement model is the link between the data and the unobserved state process. Some algorithms require the conditional expectation of the measurement model, given the latent state and parameters. This is supplied using the `emeasure` argument.

Suppose you have a procedure to compute this conditional expectation, given the value of the latent state variables. Then you can furnish

```
emeasure = 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 `emeasure` C snippet, bear in mind that:

1. The goal of such a snippet is to fill variables named `E_y` with the conditional expectations of observables `y`. Accordingly, there should be one assignment of `E_y` for each observable `y`.
2. In addition to the states, parameters, and covariates (if any), 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.

It is also possible, though less efficient, to specify `emeasure` using an R function. In this case, specify the measurement model expectation by furnishing

```
emeasure = 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 `...`. `f` must return a named numeric vector of length equal to the number of observable variables. The names should match those of the observable variables.

Default behavior

The default `emeasure` is undefined. It will yield missing values (`NA`).

Note for Windows users

Some Windows users report problems when using C snippets in parallel computations. These appear to arise when the temporary files created during the C snippet compilation process are not handled properly by the operating system. To circumvent this problem, use the `cdir` and `cfile` options to cause the C snippets to be written to a file of your choice, thus avoiding the use of temporary files altogether.

See Also

`emeasure`

More on implementing POMP models: `Csnippet`, `accumulator variables`, `basic components`, `betabinomial`, `covariates`, `distributions`, `dmeasure specification`, `dprocess specification`, `parameter transformations`, `pomp-package`, `pomp`, `prior specification`, `rinit specification`, `rmeasure specification`, `rprocess specification`, `skeleton specification`, `transformations`, `userdata`, `vmeasure specification`

`estimation algorithms` *Parameter estimation algorithms for POMP models.*

Description

pomp currently implements the following algorithms for estimating model parameters:

- iterated filtering (IF2)
- particle Markov chain Monte Carlo (PMCMC)
- approximate Bayesian computation (ABC)
- probe-matching via synthetic likelihood
- nonlinear forecasting
- power-spectrum matching
- Liu-West Bayesian sequential Monte Carlo
- Ensemble and ensemble-adjusted Kalman filters

Details

Help pages detailing each estimation algorithm are provided.

See Also

basic model components, workhorse functions, elementary algorithms.

More on **pomp** estimation algorithms: `approximate Bayesian computation`, `bsmc2()`, `mif2()`, `nonlinear forecasting`, `pmcmc()`, `pomp-package`, `probe matching`, `spectrum matching`

`filter.mean` *Filtering mean*

Description

The mean of the filtering distribution

Usage

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

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

Arguments

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

Details

The filtering distribution is that of

$$X(t_k)|Y(t_1) = y_1^*, \dots, Y(t_k) = y_k^*,$$

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

The filtering mean is therefore the expectation of this distribution

$$E[X(t_k)|Y(t_1) = y_1^*, \dots, Y(t_k) = y_k^*].$$

See Also

More on sequential Monte Carlo methods: `bsmc2()`, `cond.logLik()`, `eff.sample.size()`, `filter.traj()`, `kalman`, `mif2()`, `pfilter()`, `pmcmc()`, `pred.mean()`, `pred.var()`, `saved.states()`, `wpfilter()`

Other extraction methods: `coef()`, `cond.logLik()`, `covmat()`, `eff.sample.size()`, `filter.traj()`, `forecast()`, `logLik`, `obs()`, `pred.mean()`, `pred.var()`, `saved.states()`, `spy()`, `states()`, `summary()`, `timezero()`, `time()`, `traces()`

`filter.traj` *Filtering trajectories*

Description

Drawing from the smoothing distribution

Usage

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

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

## S4 method for signature 'pmcmcd_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

$$X(t_k)|Y(t_1) = y_1^*, \dots, Y(t_n) = y_n^*,$$

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

More on sequential Monte Carlo methods: `bsmc2()`, `cond.logLik()`, `eff.sample.size()`, `filter.mean()`, `kalman`, `mif2()`, `pfilter()`, `pmcmc()`, `pred.mean()`, `pred.var()`, `saved.states()`, `wpfilter()`

Other extraction methods: `coef()`, `cond.logLik()`, `covmat()`, `eff.sample.size()`, `filter.mean()`, `forecast()`, `logLik`, `obs()`, `pred.mean()`, `pred.var()`, `saved.states()`, `spy()`, `states()`, `summary()`, `timezero()`, `time()`, `traces()`

`flow` *Flow of a deterministic model*

Description

Compute the flow generated by a deterministic vectorfield or map.

Usage

```
## S4 method for signature 'pomp'
flow(
  object,
  x0,
  t0 = timezero(object),
  times = time(object),
  params = coef(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.
`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. By default, this is the zero-time (see `timezero`).
`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`. By default, this is the full set of observation times (see `time`).
`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. By default, this is the parameter vector stored in `object` (see `coef`).
`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[,j]`.

See Also

More on **pomp** workhorse functions: `dmeasure()`, `dprior()`, `dprocess()`, `emeasure()`, `partrans()`, `pomp-package`, `rinit()`, `rmeasure()`, `rprior()`, `rprocess()`, `skeleton()`, `vmeasure()`, `workhorses`

More on methods for deterministic process models: `skeleton specification`, `skeleton()`, `trajectory matching`, `trajectory()`

`forecast` *Forecast mean*

Description

Mean of the one-step-ahead forecasting distribution.

Usage

```
forecast(object, ...)

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

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

Arguments

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

See Also

Other extraction methods: `coef()`, `cond.logLik()`, `covmat()`, `eff.sample.size()`, `filter.mean()`, `filter.traj()`, `logLik`, `obs()`, `pred.mean()`, `pred.var()`, `saved.states()`, `spy()`, `states()`, `summary()`, `timezero()`, `time()`, `traces()`

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

Description

`gompertz()` constructs a ‘pomp’ object encoding a stochastic Gompertz population model with log-normal measurement error.

Usage

```
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
`times`	observation times
`t0`	zero time

Details

The state process is

$$X_{t+1} = K^{1-S} X_t^S \epsilon_t,$$

where $S = e^{-r}$ and the ϵ_t are i.i.d. lognormal random deviates with variance σ^2. The observed variables Y_t are distributed as

$$Y_t \sim \mathrm{lognormal}(\log X_t, \tau).$$

Parameters include the per-capita growth rate r, the carrying capacity K, the process noise s.d. σ, the measurement error s.d. τ, and the initial condition X_0. The ‘pomp’ object includes parameter transformations that log-transform the parameters for estimation purposes.

Value

A ‘pomp’ object with simulated data.

See Also

More examples provided with **pomp**: `SIR models`, `blowflies`, `childhood disease data`, `dacca()`, `ebola`, `ou2()`, `pomp examples`, `ricker()`, `rw2()`, `verhulst()`

Examples

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

`hitch` *Hitching C snippets and R functions to pomp_fun objects*

Description

The algorithms in **pomp** are formulated using R 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.

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` global option.
`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. 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

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

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

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

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

```
## S4 method for signature 'pomp'
eakf(data, Np, ..., 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. Internally, `data` will be internally coerced to an array with storage-mode `double`.
`Np`	integer; the number of particles to use, i.e., the size of the ensemble.
`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 specification.
`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 specification for the documentation on these plugins.
`emeasure`	the expectation of the measured variables, conditional on the latent state. This can be specified as a C snippet, an R function, or the name of a pre-compiled native routine available in a dynamically loaded library. Setting `emeasure=NULL` removes the emeasure component. For more information, see emeasure specification.
`vmeasure`	the covariance of the measured variables, conditional on the latent state. This can be specified as a C snippet, an R function, or the name of a pre-compiled native routine available in a dynamically loaded library. Setting `vmeasure=NULL` removes the vmeasure component. For more information, see vmeasure specification.
`...`	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.

Value

An object of class ‘kalmand_pomp’.

Note for Windows users

Some Windows users report problems when using C snippets in parallel computations. These appear to arise when the temporary files created during the C snippet compilation process are not handled properly by the operating system. To circumvent this problem, use the `cdir` and `cfile` options to cause the C snippets to be written to a file of your choice, thus avoiding the use of temporary files altogether.

Author(s)

Aaron A. King

References

G. Evensen. Sequential data assimilation with a nonlinear quasi-geostrophic model using Monte Carlo methods to forecast error statistics. *Journal of Geophysical Research: Oceans* **99**, 10143–10162, 1994.

J.L. Anderson. An ensemble adjustment Kalman filter for data assimilation. *Monthly Weather Review* **129**, 2884–2903, 2001.

G. Evensen. *Data assimilation: the ensemble Kalman filter*. Springer-Verlag, 2009.

See Also

`kalmanFilter`

More on sequential Monte Carlo methods: `bsmc2()`, `cond.logLik()`, `eff.sample.size()`, `filter.mean()`, `filter.traj()`, `mif2()`, `pfilter()`, `pmcmc()`, `pred.mean()`, `pred.var()`, `saved.states()`, `wpfilter()`

More on **pomp** elementary algorithms: `elementary algorithms`, `pfilter()`, `pomp-package`, `probe()`, `simulate()`, `spect()`, `trajectory()`, `wpfilter()`

`kalmanFilter` *Kalman filter*

Description

The basic Kalman filter for multivariate, linear, Gaussian processes.

Usage

```
kalmanFilter(object, X0, A, Q, C, R, tol = 1e-06)
```

Arguments

`object`	a pomp object containing data;
`X0`	length-m vector containing initial state. This is assumed known without uncertainty.
`A`	$m \times m$ latent state-process transition matrix. $E[X(t+1)\|X(t)] = A.X(t)$.
`Q`	$m \times m$ latent state-process covariance matrix. $Var[X(t+1)\|X(t)] = Q$
`C`	$n \times m$ link matrix. $E[Y(t)\|X(t)] = C.X(t)$.

`R`	$n \times n$ observation process covariance matrix. $Var[Y(t)\|X(t)] = R$
`tol`	numeric; the tolerance to be used in computing matrix pseudoinverses via singular-value decomposition. Singular values smaller than `tol` are set to zero.

Details

If the latent state is X, the observed variable is Y, $X(t) \in R^m$, $Y(t) \in R^n$, and

$$X(t)\ MultivariateNormal(AX(t-1), Q)$$

$$Y(t)\ MultivariateNormal(CX(t), R)$$

Then the Kalman filter computes the exact likelihood of Y given A, C, Q, and R.

Value

A named list containing the following elements:

object the 'pomp' object

A, Q, C, R as in the call

filter.mean $E[X(t)|y^*(1),\dots,y^*(t)]$

pred.mean $E[X(t)|y^*(1),\dots,y^*(t-1)]$

forecast $E[Y(t)|y^*(1),\dots,y^*(t-1)]$

cond.logLik $f(y^*(t)|y^*(1),\dots,y^*(t-1))$

logLik $f(y^*(1),\dots,y^*(T))$

See Also

`enkf`, `eakf`

Examples

```
## Not run:

  if (require(dplyr)) {

    gompertz() -> po

    po %>%
      as.data.frame() %>%
      mutate(
        logY=log(Y)
      ) %>%
      select(time,logY) %>%
      pomp(times="time",t0=0) %>%
      kalmanFilter(
        X0=c(logX=0),
        A=matrix(exp(-0.1),1,1),
        Q=matrix(0.01,1,1),
        C=matrix(1,1,1),
```

```
        R=matrix(0.01,1,1)
      ) -> kf

    po %>%
      pfilter(Np=1000) -> pf

    kf$logLik
    logLik(pf) + sum(log(obs(pf)))

  }

## End(Not run)
```

---

logLik *Log likelihood*

Description

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

Usage

```
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'
logLik(object)

## 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 'pfilterd_pomp' class (i.e., the result of a `wpfilter` computation), `logLik` retrieves the estimated log likelihood.

When `object` is of 'wpfilterd_pomp' class (i.e., the result of a `wpfilter` computation), `logLik` retrieves the estimated log likelihood.

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

When `object` is of 'kalmand_pomp' class (i.e., the result of an `eakf` or `enkf` computation), `logLik` retrieves the estimated log likelihood.

When `object` is of 'pmcmcd_pomp' class (i.e., the result of a `pmcmc` computation), `logLik` retrieves the estimated log likelihood as of the last particle filter operation.

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

When `object` is of 'spect_match_objfun' class (i.e., an objective function constructed by `spect_objfun`), `logLik` retrieves minus the spectrum mismatch.

When `object` is an NLF objective function, i.e., the result of a call to `nlf_objfun`, `logLik` retrieves the "quasi log likelihood".

See Also

Other extraction methods: `coef()`, `cond.logLik()`, `covmat()`, `eff.sample.size()`, `filter.mean()`, `filter.traj()`, `forecast()`, `obs()`, `pred.mean()`, `pred.var()`, `saved.states()`, `spy()`, `states()`, `summary()`, `timezero()`, `time()`, `traces()`

`logmeanexp` *The log-mean-exp trick*

Description

`logmeanexp` computes

$$\log\frac{1}{N}\sum_{n=1}^{N}e_i^x,$$

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

Usage

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

Arguments

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

Details

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

Value

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

Author(s)

Aaron A. King

Examples

```
## Not run: 
  ## an estimate of the log likelihood:
  po <- ricker()
  ll <- replicate(n=5,logLik(pfilter(po,Np=1000)))
  logmeanexp(ll)
  ## with standard error:
  logmeanexp(ll,se=TRUE)

## End(Not run)
```

`lookup` *Lookup table*

Description

Interpolate values from a lookup table

Usage

```
lookup(table, t)
```

Arguments

`table`	a ‘covartable’ object created by a call to `covariate_table`
`t`	numeric vector; times at which interpolated values of the covariates in `table` are required.

Details

A warning will be generated if extrapolation is performed.

Value

A numeric vector or matrix of the interpolated values.

See Also

More on interpolation: `bsplines`, `covariates`

`mcap` *Monte Carlo adjusted profile*

Description

Given a collection of points maximizing the likelihood over a range of fixed values of a focal parameter, this function constructs a profile likelihood confidence interval accommodating both Monte Carlo error in the profile and statistical uncertainty present in the likelihood function.

Usage

```
mcap(logLik, parameter, level = 0.95, span = 0.75, Ngrid = 1000)
```

Arguments

`logLik`	numeric; a vector of profile log likelihood evaluations.
`parameter`	numeric; the corresponding values of the focal parameter.
`level`	numeric; the confidence level required.
`span`	numeric; the `loess` smoothing parameter.
`Ngrid`	integer; the number of points to evaluate the smoothed profile.

Value

`mcap` returns a list including the' `loess`-smoothed profile, a quadratic approximation, and the constructed confidence interval.

Author(s)

Edward L. Ionides

References

E. L. Ionides, C. Breto, J. Park, R. A. Smith, and A. A. King. Monte Carlo profile confidence intervals for dynamic systems. *Journal of the Royal Society, Interface* **14**, 20170126, 2017.

`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. Internally, `data` will be internally coerced to an array with storage-mode `double`.
`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 `ivp(s)` can be used in this context as shorthand for `ifelse(time==time[1],s,0)`.

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

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 requested 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` operations on the result of a `mif2` computatation.
- `coef` 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,

```
rw.sd(a=0.05, b=ifelse(time==time[1],0.2,0),
      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 just before the first observation. Parameters `d` and `e`, by contrast, get perturbations of s.d. 0.2 only just 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 σ is specified using `rw.sd`, as described above. Let N be the length of the time series and α =`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 α 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.

Note for Windows users

Some Windows users report problems when using C snippets in parallel computations. These appear to arise when the temporary files created during the C snippet compilation process are not handled properly by the operating system. To circumvent this problem, use the `cdir` and `cfile` options to cause the C snippets to be written to a file of your choice, thus avoiding the use of temporary files altogether.

Author(s)

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

References

E.L. Ionides, D. Nguyen, Y. Atchadé, S. Stoev, and A.A. King. Inference for dynamic and latent variable models via iterated, perturbed Bayes maps. *Proceedings of the National Academy of Sciences* **112**, 719–724, 2015.

See Also

More on full-information (i.e., likelihood-based) methods: `bsmc2()`, `pfilter()`, `pmcmc()`, `wpfilter()`

More on sequential Monte Carlo methods: `bsmc2()`, `cond.logLik()`, `eff.sample.size()`, `filter.mean()`, `filter.traj()`, `kalman`, `pfilter()`, `pmcmc()`, `pred.mean()`, `pred.var()`, `saved.states()`, `wpfilter()`

More on **pomp** estimation algorithms: `approximate Bayesian computation`, `bsmc2()`, `estimation algorithms`, `nonlinear forecasting`, `pmcmc()`, `pomp-package`, `probe matching`, `spectrum matching`

More on maximization-based estimation methods: `nonlinear forecasting`, `probe matching`, `spectrum matching`, `trajectory matching`

`nonlinear forecasting` *Nonlinear forecasting*

Description

Parameter estimation by maximum simulated quasi-likelihood.

Usage

```
## 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")
)

## 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. Internally, `data` will be internally coerced to an array with storage-mode `double`.
`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 specification.
`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 specification 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 specification.
`...`	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. Specfically, `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.

Note for Windows users

Some Windows users report problems when using C snippets in parallel computations. These appear to arise when the temporary files created during the C snippet compilation process are not handled properly by the operating system. To circumvent this problem, use the `cdir` and `cfile` options to cause the C snippets to be written to a file of your choice, thus avoiding the use of temporary files altogether.

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

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

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

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

See Also

`optim` `subplex` `nloptr`

More on **pomp** estimation algorithms: `approximate Bayesian computation`, `bsmc2`(), `estimation algorithms`, `mif2`(), `pmcmc`(), `pomp-package`, `probe matching`, `spectrum matching`

More on methods based on summary statistics: `approximate Bayesian computation`, `basic probes`, `probe matching`, `probe`(), `spectrum matching`, `spect`()

More on maximization-based estimation methods: `mif2`(), `probe matching`, `spectrum matching`, `trajectory matching`

Examples

```
if (require(subplex)) {

  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

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

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

}
```

`obs` *obs*

Description

Extract the data array from a ‘pomp’ object.

Usage

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

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

Arguments

`object`	an object of class ‘pomp’, or of a class extending ‘pomp’
`vars`	names of variables to retrieve
`...`	ignored

See Also

Other extraction methods: `coef()`, `cond.logLik()`, `covmat()`, `eff.sample.size()`, `filter.mean()`, `filter.traj()`, `forecast()`, `logLik`, `pred.mean()`, `pred.var()`, `saved.states()`, `spy()`, `states()`, `summary()`, `timezero()`, `time()`, `traces()`

`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

```
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 α and σ are 2x2 matrices, σ is lower-triangular, and $\epsilon(t)$ is standard bivariate normal. The observation process is $Y(t) = (y_1(t), y_2(t))$, where $y_i(t) \sim \mathrm{normal}(x_i(t), \tau)$.

Value

A ‘pomp’ object with simulated data.

See Also

More examples provided with **pomp**: `SIR models`, `blowflies`, `childhood disease data`, `dacca()`, `ebola`, `gompertz()`, `pomp examples`, `ricker()`, `rw2()`, `verhulst()`

Examples

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

`parameter transformations`

Parameter transformations

Description

Equipping models with parameter transformations to ease searches in constrained parameter spaces.

Usage

```
parameter_trans(toEst, fromEst, ...)

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

## S4 method for signature 'pomp_fun,pomp_fun'
```

```
parameter_trans(toEst, fromEst, ...)

## S4 method for signature 'Csnippet,Csnippet'
parameter_trans(toEst, fromEst, ..., 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 other 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

```
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

$$\mathrm{logit}(\theta) = \log \frac{\theta}{1-\theta}.$$

The log barycentric transformation of variables $\theta_1, \dots, \theta_n$ is given by

$$\mathrm{logbarycentric}(\theta_1, \dots, \theta_n) = \left(\log \frac{\theta_1}{\sum_i \theta_i}, \dots, \log \frac{\theta_n}{\sum_i \theta_i}\right).$$

Note for Windows users

Some Windows users report problems when using C snippets in parallel computations. These appear to arise when the temporary files created during the C snippet compilation process are not handled properly by the operating system. To circumvent this problem, use the `cdir` and `cfile` options to cause the C snippets to be written to a file of your choice, thus avoiding the use of temporary files altogether.

See Also

`partrans`

More on implementing POMP models: `Csnippet`, `accumulator variables`, `basic components`, `betabinomial`, `covariates`, `distributions`, `dmeasure specification`, `dprocess specification`, `emeasure specification`, `pomp-package`, `pomp`, `prior specification`, `rinit specification`, `rmeasure specification`, `rprocess specification`, `skeleton specification`, `transformations`, `userdata`, `vmeasure specification`

`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, ...)

## S4 method for signature 'numeric'
parmat(params, nrep = 1, ..., names = NULL)

## S4 method for signature 'array'
parmat(params, nrep = 1, ..., names = NULL)

## S4 method for signature 'data.frame'
parmat(params, nrep = 1, ...)
```

Arguments

`params`	named numeric vector or matrix of parameters.
`...`	additional arguments, currently ignored.
`nrep`	number of replicates (columns) desired.
`names`	optional character; column names.

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))
trajectory(
  ricker(),
  times=seq(from=1000,to=2000,by=1),
  params=p,
  format="array"
) -> x
matplot(p["r",],x["N",,],pch='.',col='black',
  xlab=expression(log(r)),ylab="N",log='x')
```

`partrans` *partrans*

Description

Performs parameter transformations.

Usage

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

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

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`

More on **pomp** workhorse functions: `dmeasure()`, `dprior()`, `dprocess()`, `emeasure()`, `flow()`, `pomp-package`, `rinit()`, `rmeasure()`, `rprior()`, `rprocess()`, `skeleton()`, `vmeasure()`, `workhorses`

---

parus *Parus major population dynamics*

Description

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

Details

Provenance: Global Population Dynamics Database dataset #10163. (NERC Centre for Population Biology, Imperial College (2010) The Global Population Dynamics Database Version 2. `https://www.imperial.ac.uk/cpb/gpdd2/`). Original source: McCleer and Perrins (1991).

References

R. McCleery and C. Perrins. Effects of predation on the numbers of Great Tits, *Parus major*. In: C.M. Perrins, J.-D. Lebreton, and G.J.M. Hirons (eds.), *Bird Population Studies*, pp. 129–147, Oxford. Univ. Press, 1991.

See Also

More data sets provided with **pomp**: `blowflies`, `bsflu`, `childhood disease data`, `dacca`(), `ebola`

Examples

```
## Not run:
  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, ...) {
          e <- rnorm(n=1,mean=0,sd=sigma)
          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()

## End(Not run)
```

`pfilter` *Particle filter*

Description

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

Usage

```
## 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)
)

## 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)
)

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

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

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. Internally, `data` will be internally coerced to an array with storage-mode `double`.
`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 specification.
`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 specification 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 specification.
`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 `filter.traj` for more information.
`save.states`	logical. If `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 `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.

Value

An object of class 'pfilterd_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

`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

Note for Windows users

Some Windows users report problems when using C snippets in parallel computations. These appear to arise when the temporary files created during the C snippet compilation process are not handled properly by the operating system. To circumvent this problem, use the `cdir` and `cfile` options to cause the C snippets to be written to a file of your choice, thus avoiding the use of temporary files altogether.

Author(s)

Aaron A. King

References

M.S. Arulampalam, S. Maskell, N. Gordon, & T. Clapp. A tutorial on particle filters for online nonlinear, non-Gaussian Bayesian tracking. *IEEE Transactions on Signal Processing* **50**, 174–188, 2002.

A. Bhadra and E.L. Ionides. Adaptive particle allocation in iterated sequential Monte Carlo via approximating meta-models. *Statistics and Computing* **26**, 393–407, 2016.

See Also

More on **pomp** elementary algorithms: `elementary algorithms`, `kalman`, `pomp-package`, `probe()`, `simulate()`, `spect()`, `trajectory()`, `wpfilter()`

More on sequential Monte Carlo methods: `bsmc2()`, `cond.logLik()`, `eff.sample.size()`, `filter.mean()`, `filter.traj()`, `kalman`, `mif2()`, `pmcmc()`, `pred.mean()`, `pred.var()`, `saved.states()`, `wpfilter()`

More on full-information (i.e., likelihood-based) methods: `bsmc2()`, `mif2()`, `pmcmc()`, `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

as(pf,"data.frame") %>% head()
```

`plot` *pomp plotting facilities*

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, ..., pars, transform = FALSE)

## 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 the form `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.

`transform`	logical; should the parameter be transformed onto the estimation scale?
`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.

`pmcmc` *The particle Markov chain Metropolis-Hastings algorithm*

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

```
## S4 method for signature 'data.frame'
pmcmc(
  data,
  Nmcmc = 1,
  proposal,
  Np,
  params,
  rinit,
  rprocess,
  dmeasure,
  dprior,
  ...,
  verbose = getOption("verbose", FALSE)
)

## S4 method for signature 'pomp'
pmcmc(
  data,
  Nmcmc = 1,
  proposal,
  Np,
  ...,
```

```
  verbose = getOption("verbose", FALSE)
)

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

## 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. Internally, `data` will be internally coerced to an array with storage-mode `double`.
`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.
`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 specification.
`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 specification 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 specification.
`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 specification. 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.

Value

An object of class 'pmcmcd_pomp'.

Methods

The following can be applied to the output of a `pmcmc` operation:

`pmcmc` repeats the calculation, beginning with the last state

`continue` continues the `pmcmc` calculation

`plot` produces a series of diagnostic plots

`filter.traj` extracts a random sample from the smoothing distribution

`traces` produces an `mcmc` object, to which the various **coda** convergence diagnostics can be applied

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.

Note for Windows users

Some Windows users report problems when using C snippets in parallel computations. These appear to arise when the temporary files created during the C snippet compilation process are not handled properly by the operating system. To circumvent this problem, use the `cdir` and `cfile` options to cause the C snippets to be written to a file of your choice, thus avoiding the use of temporary files altogether.

Author(s)

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

References

C. Andrieu, A. Doucet, and R. Holenstein. Particle Markov chain Monte Carlo methods. *Journal of the Royal Statistical Society, Series B* **72**, 269–342, 2010.

See Also

More on **pomp** estimation algorithms: `approximate Bayesian computation`, `bsmc2()`, `estimation algorithms`, `mif2()`, `nonlinear forecasting`, `pomp-package`, `probe matching`, `spectrum matching`

More on sequential Monte Carlo methods: `bsmc2()`, `cond.logLik()`, `eff.sample.size()`, `filter.mean()`, `filter.traj()`, `kalman`, `mif2()`, `pfilter()`, `pred.mean()`, `pred.var()`, `saved.states()`, `wpfilter()`

More on full-information (i.e., likelihood-based) methods: `bsmc2()`, `mif2()`, `pfilter()`, `wpfilter()`

More on Markov chain Monte Carlo methods: `approximate Bayesian computation`, `proposals`

More on Bayesian methods: `approximate Bayesian computation`, `bsmc2()`, `dprior()`, `prior specification`, `rprior()`

`pomp` *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;

emeasure, the expectation of the measurements, conditional on the latent state;

vmeasure, the covariance matrix of the measurements, conditional on the latent state;

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

```
pomp(
  data,
  times,
  t0,
  ...,
  rinit,
  rprocess,
  dprocess,
  rmeasure,
  dmeasure,
  emeasure,
  vmeasure,
  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. Internally, `data` will be internally coerced to an array with storage-mode `double`.
`times`	the sequence of observation times. `times` must indicate the column of observation times by name or index. The time vector must be numeric and non-decreasing.
`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]`.
`...`	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.
`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 specification.
`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 specification 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 specification.
`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 specification.
`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 specification.
`emeasure`	the expectation of the measured variables, conditional on the latent state. This can be specified as a C snippet, an R function, or the name of a pre-compiled native routine available in a dynamically loaded library. Setting `emeasure=NULL` removes the emeasure component. For more information, see emeasure specification.
`vmeasure`	the covariance of the measured variables, conditional on the latent state. This can be specified as a C snippet, an R function, or the name of a pre-compiled native routine available in a dynamically loaded library. Setting `vmeasure=NULL` removes the vmeasure component. For more information, see vmeasure specification.
`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` fnctions. For more information, see skeleton specification. 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. For more information, see prior specification. 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. For more information, see prior specification. 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` global 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 feasible) to define all of the basic components for any given purpose. However, each **pomp** algorithm makes use of only a subset of these components. When an algorithm requires a basic component that has not been furnished, an error is generated to let you know that you must provide the needed component to use the algorithm.

Note for Windows users

Some Windows users report problems when using C snippets in parallel computations. These appear to arise when the temporary files created during the C snippet compilation process are not handled properly by the operating system. To circumvent this problem, use the `cdir` and `cfile` options to cause the C snippets to be written to a file of your choice, thus avoiding the use of temporary files altogether.

Author(s)

Aaron A. King

References

A. A. King, D. Nguyen, and E. L. Ionides. Statistical inference for partially observed Markov processes via the package **pomp**. *Journal of Statistical Software* **69**(12), 1–43, 2016. An updated version of this paper is available on the package website.

See Also

More on implementing POMP models: `Csnippet`, `accumulator variables`, `basic components`, `betabinomial`, `covariates`, `distributions`, `dmeasure specification`, `dprocess specification`, `emeasure specification`, `parameter transformations`, `pomp-package`, `prior specification`, `rinit specification`, `rmeasure specification`, `rprocess specification`, `skeleton specification`, `transformations`, `userdata`, `vmeasure specification`

`pomp examples` *pomp_examples*

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
- `parus` Population censuses of a *Parus major* population in Wytham Wood, England.
- `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

Anonymous. Influenza in a boarding school. *British Medical Journal* **1**, 587, 1978.

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

A.A. King, M. Domenech de Cellès, F.M.G. Magpantay, and P. Rohani. Avoidable errors in the modelling of outbreaks of emerging pathogens, with special reference to Ebola. *Proceedings of the Royal Society of London, Series B* **282**, 20150347, 2015.

W. P. London and J. A. Yorke, Recurrent outbreaks of measles, chickenpox and mumps: I. Seasonal variation in contact rates. *American Journal of Epidemiology* **98**, 453–468, 1973.

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

See Also

More examples provided with **pomp**: `SIR models`, `blowflies`, `childhood disease data`, `dacca()`, `ebola`, `gompertz()`, `ou2()`, `ricker()`, `rw2()`, `verhulst()`

`pred.mean` *Prediction mean*

Description

The mean of the prediction distribution

Usage

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

## 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

$$X(t_k) | Y(t_1) = y^*_1, \dots, Y(t_{k-1}) = y^*_{k-1},$$

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

$$E[X(t_k) | Y(t_1) = y^*_1, \dots, Y(t_{k-1}) = y^*_{k-1}].$$

See Also

More on sequential Monte Carlo methods: `bsmc2()`, `cond.logLik()`, `eff.sample.size()`, `filter.mean()`, `filter.traj()`, `kalman`, `mif2()`, `pfilter()`, `pmcmc()`, `pred.var()`, `saved.states()`, `wpfilter()`

Other extraction methods: `coef()`, `cond.logLik()`, `covmat()`, `eff.sample.size()`, `filter.mean()`, `filter.traj()`, `forecast()`, `logLik`, `obs()`, `pred.var()`, `saved.states()`, `spy()`, `states()`, `summary()`, `timezero()`, `time()`, `traces()`

`pred.var` *Prediction variance*

Description

The variance of the prediction distribution

Usage

```
## 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

$$X(t_k) | Y(t_1) = y^*_1, \dots, Y(t_{k-1}) = y^*_{k-1},$$

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

$$\mathrm{Var}[X(t_k) | Y(t_1) = y^*_1, \dots, Y(t_{k-1}) = y^*_{k-1}].$$

See Also

More on sequential Monte Carlo methods: `bsmc2()`, `cond.logLik()`, `eff.sample.size()`, `filter.mean()`, `filter.traj()`, `kalman`, `mif2()`, `pfilter()`, `pmcmc()`, `pred.mean()`, `saved.states()`, `wpfilter()`

Other extraction methods: `coef()`, `cond.logLik()`, `covmat()`, `eff.sample.size()`, `filter.mean()`, `filter.traj()`, `forecast()`, `logLik`, `obs()`, `pred.mean()`, `saved.states()`, `spy()`, `states()`, `summary()`, `timezero()`, `time()`, `traces()`

`prior specification` *prior distribution*

Description

Specification of prior distributions.

Details

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

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

In writing a C snippet for the prior density function (`dprior`), observe that:

1. Within the context in which the snippet will be evaluated, only the parameters and `give_log` will be defined.
2. The goal of such a snippet is computation of the prior probability density, or the log of same, at a given point in parameter space. This scalar value should be returned in the variable `lik`. When `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, `rprior` must be a function that makes a draw from the prior distribution of the parameters and returns a named vector containing all the parameters. The only required argument of this function is `...`.

Similarly, the `dprior` function must evaluate the prior probability density (or log density if `log == TRUE`) and return that single scalar value. The only required arguments of this function are `...` and `log`.

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).

Note for Windows users

Some Windows users report problems when using C snippets in parallel computations. These appear to arise when the temporary files created during the C snippet compilation process are not handled properly by the operating system. To circumvent this problem, use the `cdir` and `cfile` options to cause the C snippets to be written to a file of your choice, thus avoiding the use of temporary files altogether.

See Also

`dprior` `rprior`

More on implementing POMP models: `Csnippet`, `accumulator variables`, `basic components`, `betabinomial`, `covariates`, `distributions`, `dmeasure specification`, `dprocess specification`, `emeasure specification`, `parameter transformations`, `pomp-package`, `pomp`, `rinit specification`, `rmeasure specification`, `rprocess specification`, `skeleton specification`, `transformations`, `userdata`, `vmeasure specification`

More on Bayesian methods: `approximate Bayesian computation`, `bsmc2()`, `dprior()`, `pmcmc()`, `rprior()`

Examples

```
## Not run:
  ## Starting with an existing pomp object

  verhulst() %>% window(end=30) -> po

  ## we add or change prior distributions using the two
  ## arguments 'rprior' and 'dprior'. Here, we introduce
  ## a Gamma prior on the 'r' parameter.
  ## We construct 'rprior' and 'dprior' using R functions.

  po %>%
    bsmc2(
      rprior=function (n_0, K0, K1, sigma, tau, r0, r1, ...) {
        c(
          n_0 = n_0,
          K = rgamma(n=1,shape=K0,scale=K1),
          r = rgamma(n=1,shape=r0,scale=r1),
          sigma = sigma,
          tau = tau
        )
      },
      dprior=function(K, K0, K1, r, r0, r1, ..., log) {
        p <- dgamma(x=c(K,r),shape=c(K0,r0),scale=c(K1,r1),log=log)
        if (log) sum(p) else prod(p)
      },
      params=c(n_0=10000,K=10000,K0=10,K1=1000,
        r=0.9,r0=0.9,r1=1,sigma=0.5,tau=0.3),
      Np=1000
    ) -> B

  ## We can also pass them as C snippets:

  po %>%
    bsmc2(
      rprior=Csnippet("
         K = rgamma(K0,K1);
         r = rgamma(r0,r1);"
      ),
      dprior=Csnippet("
```

```
          double lik1 = dgamma(K,K0,K1,give_log);
          double lik2 = dgamma(r,r0,r1,give_log);
          lik = (give_log) ? lik1+lik2 : lik1*lik2;"
       ),
       paramnames=c("K","K0","K1","r","r0","r1"),
       params=c(n_0=10000,K=10000,K0=10,K1=1000,
         r=0.9,r0=0.9,r1=1,sigma=0.5,tau=0.3),
       Np=10000
     ) -> B

   ## The prior is plotted in grey; the posterior, in blue.
   plot(B)

   B %>%
     pmcmc(Nmcmc=100,Np=1000,proposal=mvn.diag.rw(c(r=0.01,K=10))) -> Bb

   plot(Bb,pars=c("loglik","log.prior","r","K"))


## End(Not run)
```

---

probe *Probes (AKA summary statistics)*

Description

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

Usage

```
## S4 method for signature 'data.frame'
probe(
  data,
  probes,
  nsim,
  seed = NULL,
  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. Internally, `data` will be internally coerced to an array with storage-mode `double`.
`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 specification.
`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 specification 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 specification.

`...`	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.

`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.

Note for Windows users

Some Windows users report problems when using C snippets in parallel computations. These appear to arise when the temporary files created during the C snippet compilation process are not handled properly by the operating system. To circumvent this problem, use the `cdir` and `cfile` options to cause the C snippets to be written to a file of your choice, thus avoiding the use of temporary files altogether.

Author(s)

Daniel C. Reuman, Aaron A. King

References

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

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

See Also

More on **pomp** elementary algorithms: `elementary algorithms`, `kalman`, `pfilter()`, `pomp-package`, `simulate()`, `spect()`, `trajectory()`, `wpfilter()`

More on methods based on summary statistics: `approximate Bayesian computation`, `basic probes`, `nonlinear forecasting`, `probe matching`, `spectrum matching`, `spect()`

`probe matching` *Probe matching*

Description

Estimation of parameters by maximum synthetic likelihood

Usage

```
## 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'
probe_objfun(
```

```
  data,
  est = character(0),
  fail.value = NA,
  probes,
  nsim,
  seed = NULL,
  ...,
  verbose = getOption("verbose", FALSE)
)

## S4 method for signature 'probed_pomp'
probe_objfun(
  data,
  est = character(0),
  fail.value = NA,
  probes,
  nsim,
  seed = NULL,
  ...,
  verbose = getOption("verbose", FALSE)
)

## S4 method for signature 'probe_match_objfun'
probe_objfun(
  data,
  est,
  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. Internally, `data` will be internally coerced to an array with storage-mode `double`.
`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.
`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 specification.
`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 specification 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 specification.
`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.

Note for Windows users

Some Windows users report problems when using C snippets in parallel computations. These appear to arise when the temporary files created during the C snippet compilation process are not handled properly by the operating system. To circumvent this problem, use the `cdir` and `cfile` options to cause the C snippets to be written to a file of your choice, thus avoiding the use of temporary files altogether.

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`

More on methods based on summary statistics: `approximate Bayesian computation`, `basic probes`, `nonlinear forecasting`, `probe()`, `spectrum matching`, `spect()`

More on **pomp** estimation algorithms: `approximate Bayesian computation`, `bsmc2()`, `estimation algorithms`, `mif2()`, `nonlinear forecasting`, `pmcmc()`, `pomp-package`, `spectrum matching`

More on maximization-based estimation methods: `mif2()`, `nonlinear forecasting`, `spectrum matching`, `trajectory matching`

Examples

```
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'.
```

```
if (require(subplex)) {

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

} else {

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

}

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

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

## One can convert an objective function to a data frame:
f %>% as("data.frame") %>% head()
f %>% as("probed_pomp") %>% as("data.frame") %>% head()

f %>% probe() %>% plot()

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

f %>% probe_objfun(est=c("r","K")) -> f1
optim(fn=f1,par=c(0.3,0.3),control=list(reltol=1e-5)) -> out
f1(out$par)
coef(f1)
```

`proposals` *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

G.O. Roberts and J.S. Rosenthal. Examples of adaptive MCMC. *Journal of Computational and Graphical Statistics* **18**, 349–367, 2009.

See Also

More on Markov chain Monte Carlo methods: `approximate Bayesian computation`, `pmcmc()`

`reproducibility tools` *Tools for reproducible computations.*

Description

Bake, stew, and freeze assist in the construction of reproducible computations.

Usage

```
bake(
  file,
  expr,
  seed = NULL,
  kind = NULL,
  normal.kind = NULL,
  dependson = NULL,
  info = FALSE,
  timing = TRUE,
  dir = getOption("pomp_archive_dir", getwd())
)

stew(
  file,
  expr,
  seed = NULL,
  kind = NULL,
  normal.kind = NULL,
  dependson = NULL,
  info = FALSE,
  dir = getOption("pomp_archive_dir", getwd())
)

freeze(
  expr,
  seed = NULL,
  kind = NULL,
  normal.kind = NULL,
  envir = parent.frame(),
 enclos = if (is.list(envir) || is.pairlist(envir)) parent.frame() else baseenv()
)
```

Arguments

`file`	Name of the archive 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 and of the RNG. The default, `seed = NULL`, will not change the RNG state. `seed` should be a single integer. See `set.seed` for more information.
`dependson`	arbitrary R object (optional). Variables on which the computation in `expr` depends. A hash of these objects will be archived in `file`, along with the results of evaluation `expr`. When `bake` or `stew` are called and `file` exists, the hash of these objects will be compared against the archived hash; recomputation is forced when these do not match. The dependencies should be specified as unquoted symbols: use a list if there are multiple dependencies. See the note below about avoiding using 'pomp' objects as dependencies.
`info`	logical. If `TRUE`, the "ingredients" of the calculation are returned as a list. In the case of `bake`, this list is the "ingredients" attribute of the returned object. In the case of `stew`, this list is a hidden object named ".ingredients", located in the environment within which `stew` was called.
`timing`	logical. If `TRUE`, the time required for the computation is returned. This is returned as the "system.time" attribute of the returned object.
`dir`	Directory holding archive files; by default, this is the current working directory. This can also be set using the global option `pomp_archive_dir`.
`envir`	the `environment` in which `expr` is to be evaluated. May also be `NULL`, a list, a data frame, a pairlist or an integer as specified to `sys.call`.
`enclos`	Relevant when `envir` is a (pair)list or a data frame. Specifies the enclosure, i.e., where R looks for objects not found in `envir`. This can be `NULL` (interpreted as the base package environment, `baseenv()`) or an environment.

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 archived 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 archive. 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`. `bake` and `stew` also store information about the code executed, the dependencies, and the state of the random-number generator (if the latter is controlled) in the archive file. Re-computation is triggered if any of these things change.

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 are copied into the environment in 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` store information about the code executed, the dependencies, and the state of the random-number generator in the archive file. In the case of `bake`, this is recorded in the "ingredients" attribute (`attr(.,"ingredients")`); in the `stew` case, this is recorded in an object, ".ingredients", in the archive. This information is returned only if `info=TRUE`.

The time required for execution is also recorded. `bake` stores this in the "system.time" attribute of the archived R object; `stew` does so in a hidden variable named `.system.time`. The timing is obtained using `system.time`.

Avoid using 'pomp' objects as dependencies

Note that when a 'pomp' object is built with one or more C snippets, the resulting code is "salted" with a random element to prevent collisions in parallel computations. As a result, two such 'pomp' objects will never match perfectly, even if the codes and data used to construct them are identical. Therefore, avoid using 'pomp' objects as dependencies in `bake` and `stew`.

Compatibility with older versions

With **pomp** version 3.4.4.2, the behavior of `bake` and `stew` changed. In particular, older versions did no dependency checking, and did not check to see whether `expr` had changed. Accordingly, the archive files written by older versions have a format that is not compatible with the newer ones. When an archive file in the old format is encountered, it will be updated to the new format, with a warning message. **Note that this will overwrite existing archive files!** However, there will be no loss of information.

Author(s)

Aaron A. King

Examples

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

  bake(file="example1.rds",{
    x <- runif(1000)
    mean(x)
```

```
  })

  bake(file="example1.rds",{
    a <- 3
    x <- runif(1000)
    mean(x)
  })

  a <- 5
  b <- 2

  stew(file="example2.rda",
    dependson=list(a,b),{
      x <- runif(10)
      y <- rnorm(n=10,mean=a*x+b,sd=2)
    })

  plot(x,y)

  set.seed(11)
  runif(2)
  freeze(runif(3),seed=5886730)
  runif(2)
  freeze(runif(3),seed=5886730)
  runif(2)

  set.seed(11)
  runif(2)
  runif(2)
  runif(2)


## End(Not run)
```

`ricker` *Ricker model with Poisson observations.*

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

Value

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

See Also

More examples provided with **pomp**: `SIR models`, `blowflies`, `childhood disease data`, `dacca()`, `ebola`, `gompertz()`, `ou2()`, `pomp examples`, `rw2()`, `verhulst()`

Examples

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

  ## 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))
  trajectory(
    ricker(),
    times=seq(from=1000,to=2000,by=1),
    params=p,
    format="array"
  ) -> x
  matplot(p["r",],x["N",,],pch='.',col='black',
    xlab=expression(log(r)),ylab="N",log='x')
```

`rinit` *rinit*

Description

Samples from the initial-state distribution.

Usage

```
## S4 method for signature 'pomp'
rinit(object, params = coef(object), t0 = timezero(object), 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

`rinit` 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 specification

More on **pomp** workhorse functions: `dmeasure()`, `dprior()`, `dprocess()`, `emeasure()`, `flow()`, `partrans()`, `pomp-package`, `rmeasure()`, `rprior()`, `rprocess()`, `skeleton()`, `vmeasure()`, `workhorses`

`rinit specification` *The initial-state distribution*

Description

Specification of the initial-state distribution simulator, rinit.

Details

To fully specify the unobserved Markov state process, one must give its distribution at the zero-time (`t0`). 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 `t0`, 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, `t0`.

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.

Note for Windows users

Some Windows users report problems when using C snippets in parallel computations. These appear to arise when the temporary files created during the C snippet compilation process are not handled properly by the operating system. To circumvent this problem, use the `cdir` and `cfile` options to cause the C snippets to be written to a file of your choice, thus avoiding the use of temporary files altogether.

See Also

`rinit`

More on implementing POMP models: `Csnippet`, `accumulator variables`, `basic components`, `betabinomial`, `covariates`, `distributions`, `dmeasure specification`, `dprocess specification`, `emeasure specification`, `parameter transformations`, `pomp-package`, `pomp`, `prior specification`, `rmeasure specification`, `rprocess specification`, `skeleton specification`, `transformations`, `userdata`, `vmeasure specification`

Examples

```
## Starting with an existing pomp object

verhulst() -> po

## we add or change the initial-state simulator,
## rinit, using the 'rinit' argument in any 'pomp'
## elementary or estimation function (or in the
## 'pomp' constructor itself).
## Here, we pass the rinit specification to 'simulate'
## as an R function.

po %>%
  simulate(
```

```
      rinit=function (n_0, ...) {
        c(n=rpois(n=1,lambda=n_0))
      }
    ) -> sim

  ## We can also pass it as a C snippet:

  po %>%
    simulate(
      rinit=Csnippet("n = rpois(n_0);"),
      paramnames="n_0",
      statenames="n"
    ) -> sim
```

`rmeasure` *rmeasure*

Description

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

Usage

```
## S4 method for signature 'pomp'
rmeasure(
  object,
  x = states(object),
  times = time(object),
  params = coef(object),
  ...
)
```

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

`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 specification

More on **pomp** workhorse functions: `dmeasure()`, `dprior()`, `dprocess()`, `emeasure()`, `flow()`, `partrans()`, `pomp-package`, `rinit()`, `rprior()`, `rprocess()`, `skeleton()`, `vmeasure()`, `workhorses`

`rmeasure specification`

The measurement-model simulator

Description

Specification of the measurement-model simulator, 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, and covariates (if any), 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.

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 `...`. `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`).

Note for Windows users

Some Windows users report problems when using C snippets in parallel computations. These appear to arise when the temporary files created during the C snippet compilation process are not handled properly by the operating system. To circumvent this problem, use the `cdir` and `cfile` options to cause the C snippets to be written to a file of your choice, thus avoiding the use of temporary files altogether.

See Also

`rmeasure`

More on implementing POMP models: `Csnippet`, `accumulator variables`, `basic components`, `betabinomial`, `covariates`, `distributions`, `dmeasure specification`, `dprocess specification`, `emeasure specification`, `parameter transformations`, `pomp-package`, `pomp`, `prior specification`, `rinit specification`, `rprocess specification`, `skeleton specification`, `transformations`, `userdata`, `vmeasure specification`

Examples

```
## We start with the pre-built Ricker example:

ricker() -> po

## To change the measurement model simulator, rmeasure,
## we use the 'rmeasure' argument in any 'pomp'
## elementary or estimation function.
## Here, we pass the rmeasure specification to 'simulate'
## as an R function.

po %>%
  simulate(
    rmeasure=function (N, phi, ...) {
      c(y=rpois(n=1,lambda=phi*N))
    }
  ) -> sim

## We can also pass it as a C snippet:

po %>%
  simulate(
    rmeasure=Csnippet("y = rpois(phi*N);"),
    paramnames="phi",
    statenames="N"
  ) -> sim
```

`rprior` *rprior*

Description

Sample from the prior probability distribution.

Usage

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

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.

Value

A numeric matrix containing the required samples.

See Also

Specification of the prior distribution simulator: prior specification

More on **pomp** workhorse functions: `dmeasure()`, `dprior()`, `dprocess()`, `emeasure()`, `flow()`, `partrans()`, `pomp-package`, `rinit()`, `rmeasure()`, `rprocess()`, `skeleton()`, `vmeasure()`, `workhorses`

More on Bayesian methods: `approximate Bayesian computation`, `bsmc2()`, `dprior()`, `pmcmc()`, `prior specification`

`rprocess` *rprocess*

Description

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

Usage

```
## S4 method for signature 'pomp'
rprocess(
  object,
  x0 = rinit(object),
  t0 = timezero(object),
  times = time(object),
  params = coef(object),
  ...
)
```

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

`dim(x)=c(nvars,nrep,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 specification

More on **pomp** workhorse functions: `dmeasure()`, `dprior()`, `dprocess()`, `emeasure()`, `flow()`, `partrans()`, `pomp-package`, `rinit()`, `rmeasure()`, `rprior()`, `skeleton()`, `vmeasure()`, `workhorses`

`rprocess specification` *The latent state process simulator*

Description

Specification of the latent state process simulator, rprocess.

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 `rprocess` using the `discrete_time` plug-in. Specifically, provide

```
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 `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 '`delta.t`'; when called, the latter will be the timestep. It must also have the argument '`...`'. 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+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 `rprocess` using the `euler` plug-in. Specify

```
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 `onestep` plug-in. To do so, specify

```
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 `discrete_time`, `euler`, and `onestep` all work by taking discrete time steps. They differ as to how this is done. Specifically,

1. `onestep` takes a single step to go from any given time `t1` to any later time `t2` (`t1 < t2`). Thus, this plug-in is designed for use in situations where a closed-form solution to the process exists.
2. To go from `t1` to `t2`, `euler` takes `n` steps of equal size, where

   ```
   n = ceiling((t2-t1)/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 `t1` to `t2`, `discrete_time` takes `n` steps of size exactly `delta.t`, where

   ```
   n = floor((t2-t1)/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 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 = 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 `.pre` and `.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:

```
gillespie_hl(
    birth=list("rate = b*N;",c(N=1)),
    death=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 = gillespie(rate.fun = f, v, hmax = Inf)
```

to `pomp`, where `f` gives the rates of the elementary events. Here, `f` may be an R function of the form

```
f(j, x, t, params, ...)
```

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 `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 `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` 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 `rate.fun` argument to `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 `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 `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 `rprocess` is undefined. It will yield missing values (`NA`) for all state variables.

Note for Windows users

Some Windows users report problems when using C snippets in parallel computations. These appear to arise when the temporary files created during the C snippet compilation process are not handled properly by the operating system. To circumvent this problem, use the `cdir` and `cfile` options to cause the C snippets to be written to a file of your choice, thus avoiding the use of temporary files altogether.

See Also

`rprocess`

More on implementing POMP models: `Csnippet`, `accumulator variables`, `basic components`, `betabinomial`, `covariates`, `distributions`, `dmeasure specification`, `dprocess specification`, `emeasure specification`, `parameter transformations`, `pomp-package`, `pomp`, `prior specification`, `rinit specification`, `rmeasure specification`, `skeleton specification`, `transformations`, `userdata`, `vmeasure specification`

`rw.sd` *rw.sd*

Description

Specifying random-walk intensities.

Usage

```
rw.sd(...)
```

Arguments

`...`	Specification of the random-walk intensities (as standard deviations).

Details

See `mif2` for details.

See Also

`mif2`

`rw2` *Two-dimensional random-walk process*

Description

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

Usage

```
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

More examples provided with **pomp**: `SIR models`, `blowflies`, `childhood disease data`, `dacca()`, `ebola`, `gompertz()`, `ou2()`, `pomp examples`, `ricker()`, `verhulst()`

Examples

```
if (require(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="none")+
    theme_bw()

}
```

`sannbox` *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

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

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

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

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

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

trajectory matching, probe matching, spectrum matching, nonlinear forecasting.

`saved.states` *Saved states*

Description

Retrieve latent state trajectories from a particle filter calculation.

Usage

```
## 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

More on sequential Monte Carlo methods: `bsmc2()`, `cond.logLik()`, `eff.sample.size()`, `filter.mean()`, `filter.traj()`, `kalman`, `mif2()`, `pfilter()`, `pmcmc()`, `pred.mean()`, `pred.var()`, `wpfilter()`

Other extraction methods: `coef()`, `cond.logLik()`, `covmat()`, `eff.sample.size()`, `filter.mean()`, `filter.traj()`, `forecast()`, `logLik`, `obs()`, `pred.mean()`, `pred.var()`, `spy()`, `states()`, `summary()`, `timezero()`, `time()`, `traces()`

`simulate` *Simulations of a partially-observed Markov process*

Description

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

Usage

```
## 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,
  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 sequence of observation times. `times` must indicate the column of observation times by name or index. The time vector must be numeric and non-decreasing.
`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 specification.
`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 specification 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 specification.
`format`	the format in which to return the results. `format = "pomps"` 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 and covariates (if any) are included (with `.id = "data"`). 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 a data frame or a 'pomp' object.

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.

Note for Windows users

Some Windows users report problems when using C snippets in parallel computations. These appear to arise when the temporary files created during the C snippet compilation process are not handled properly by the operating system. To circumvent this problem, use the `cdir` and `cfile` options to cause the C snippets to be written to a file of your choice, thus avoiding the use of temporary files altogether.

Author(s)

Aaron A. King

See Also

More on **pomp** elementary algorithms: `elementary algorithms`, `kalman`, `pfilter()`, `pomp-package`, `probe()`, `spect()`, `trajectory()`, `wpfilter()`

`SIR models` *Compartmental epidemiological models*

Description

Simple SIR-type models implemented in various ways.

Usage

```
sir(
  gamma = 26,
  mu = 0.02,
  iota = 0.01,
  beta1 = 400,
  beta2 = 480,
  beta3 = 320,
  beta_sd = 0.001,
  rho = 0.6,
  k = 0.1,
  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
)

sir2(
  gamma = 24,
  mu = 1/70,
  iota = 0.1,
  beta1 = 330,
  beta2 = 410,
  beta3 = 490,
  rho = 0.1,
  k = 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

`gamma`	recovery rate
`mu`	death rate (assumed equal to the birth rate)
`iota`	infection import rate
`beta1, beta2, beta3`	seasonal contact rates
`beta_sd`	environmental noise intensity
`rho`	reporting efficiency
`k`	reporting overdispersion parameter (reciprocal of the negative-binomial *size* parameter)
`pop`	overall host population size
`S_0, I_0, R_0`	the fractions of the host population that are susceptible, infectious, and recovered, respectively, at time zero.
`t0`	zero time
`times`	observation times
`seed`	seed of the random number generator
`delta.t`	Euler step size

Details

`sir()` producees 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.

In both cases the measurement model is negative binomial: `reports` is distributed as a negative binomial random variable with mean equal to `rho*cases` and size equal to `1/k`.

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

More examples provided with **pomp**: `blowflies`, `childhood disease data`, `dacca()`, `ebola`, `gompertz()`, `ou2()`, `pomp examples`, `ricker()`, `rw2()`, `verhulst()`

Examples

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

po <- sir2()
```

```
plot(po)
plot(simulate(window(po,end=3)))
coef(po)

po %>% as.data.frame() %>% head()
```

`skeleton` *skeleton*

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

Usage

```
## S4 method for signature 'pomp'
skeleton(
  object,
  x = states(object),
  times = time(object),
  params = coef(object),
  ...
)
```

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 specification

More on **pomp** workhorse functions: `dmeasure()`, `dprior()`, `dprocess()`, `emeasure()`, `flow()`, `partrans()`, `pomp-package`, `rinit()`, `rmeasure()`, `rprior()`, `rprocess()`, `vmeasure()`, `workhorses`

More on methods for deterministic process models: `flow()`, `skeleton specification`, `trajectory matching`, `trajectory()`

`skeleton specification`

The deterministic skeleton of a model

Description

Specification of the deterministic skeleton.

Usage

```
vectorfield(f)

map(f, delta.t = 1)
```

Arguments

`f`	procedure for evaluating the deterministic skeleton This can be a C snippet, an R function, or the name of a native routine in a dynamically linked library.
`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 **pomp**, the skeleton is used by `trajectory` and `traj_objfun`.

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

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

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

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

```
skeleton = vectorfield(f)
```

to `pomp`, where `f` implements the vectorfield, i.e., the right-hand-size of the differential equations.

In either case, `f` can be furnished either as a C snippet (the preferred choice), or an R function. 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 `Dx` 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.
5. **NB:** When the skeleton is a map, the duration of the timestep will **not** be defined in the context within which the snippet is executed. When the skeleton is a vectorfield, of course, no timestep is defined. In this regard, C snippets for the skeleton and rprocess components differ.

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 '`...`'. As with the other basic components, `f` may take additional arguments, provided these are passed along with it in the call to `pomp`. The function `f` must return a numeric vector of the same length as 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.

Note for Windows users

Some Windows users report problems when using C snippets in parallel computations. These appear to arise when the temporary files created during the C snippet compilation process are not handled properly by the operating system. To circumvent this problem, use the `cdir` and `cfile` options to cause the C snippets to be written to a file of your choice, thus avoiding the use of temporary files altogether.

See Also

`skeleton`

More on implementing POMP models: `Csnippet`, `accumulator variables`, `basic components`, `betabinomial`, `covariates`, `distributions`, `dmeasure specification`, `dprocess specification`, `emeasure specification`, `parameter transformations`, `pomp-package`, `pomp`, `prior specification`, `rinit specification`, `rmeasure specification`, `rprocess specification`, `transformations`, `userdata`, `vmeasure specification`

More on methods for deterministic process models: `flow()`, `skeleton()`, `trajectory matching`, `trajectory()`

Examples

```
## Starting with an existing pomp object,
## e.g., the continuous-time Verhulst-Pearl model,

verhulst() -> po

## we add or change the deterministic skeleton
## using the 'skeleton' argument in any 'pomp'
## elementary or estimation function
## (or in the 'pomp' constructor itself).
## Here, we pass the skeleton specification
## to 'trajectory' as an R function.
## Since this is a continuous-time POMP, the
## skeleton is a vectorfield.

po %>%
  trajectory(
    skeleton=vectorfield(
      function(r, K, n, ...) {
        c(n=r*n*(1-n/K))
      }
    ),
    format="data.frame"
  ) -> traj

## We can also pass it as a C snippet:

po %>%
  traj_objfun(
    skeleton=vectorfield(Csnippet("Dn=r*n*(1-n/K);")),
    paramnames=c("r","K"),
    statenames="n"
  ) -> ofun

ofun()

## For a discrete-time POMP, the deterministic skeleton
## is a map.  For example,

gompertz() -> po

po %>%
  traj_objfun(
    skeleton=map(
      Csnippet("
        double dt = 1.0;
        double s = exp(-r*dt);
        DX = pow(K,(1-s))*pow(X,s);"
      ), delta.t=1
    ),
    paramnames=c("r","K"),
```

```
    statenames=c("X")
  ) -> ofun

ofun()
```

---

spect *Power spectrum*

Description

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

Usage

```
## 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)
)

## 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. Internally, `data` will be internally coerced to an array with storage-mode `double`.
`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 specification.
`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 specification 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 specification.
`...`	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

`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

Note for Windows users

Some Windows users report problems when using C snippets in parallel computations. These appear to arise when the temporary files created during the C snippet compilation process are not handled properly by the operating system. To circumvent this problem, use the `cdir` and `cfile` options to cause the C snippets to be written to a file of your choice, thus avoiding the use of temporary files altogether.

Author(s)

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

References

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

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

See Also

More on methods based on summary statistics: `approximate Bayesian computation`, `basic probes`, `nonlinear forecasting`, `probe matching`, `probe()`, `spectrum matching`

More on **pomp** elementary algorithms: `elementary algorithms`, `kalman`, `pfilter()`, `pomp-package`, `probe()`, `simulate()`, `trajectory()`, `wpfilter()`

`spectrum matching` *Spectrum matching*

Description

Estimation of parameters by matching power spectra

Usage

```
## 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. Internally, `data` will be internally coerced to an array with storage-mode `double`.
`est`	character vector; the names of parameters to be estimated.
`weights`	optional numeric or function. The mismatch between model and data is mea-

	sured 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 specification.
`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 specification 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 specification.
`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 spectrum matching, one attempts to minimize the discrepancy between a POMP model's predictions and data, as measured in the frequency domain by the power spectrum.

`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.

Value

`spect_objfun` constructs a stateful objective function for spectrum matching. Specifically, `spect_objfun` returns an object of class 'spect_match_objfun', which is a function suitable for use in an `optim`-like optimizer. 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 (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.

Note for Windows users

Some Windows users report problems when using C snippets in parallel computations. These appear to arise when the temporary files created during the C snippet compilation process are not handled properly by the operating system. To circumvent this problem, use the `cdir` and `cfile` options to cause the C snippets to be written to a file of your choice, thus avoiding the use of temporary files altogether.

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.

See Also

`spect` `optim` `subplex` `nloptr`

More on **pomp** estimation algorithms: `approximate Bayesian computation`, `bsmc2()`, `estimation algorithms`, `mif2()`, `nonlinear forecasting`, `pmcmc()`, `pomp-package`, `probe matching`

More on methods based on summary statistics: `approximate Bayesian computation`, `basic probes`, `nonlinear forecasting`, `probe matching`, `probe()`, `spect()`

More on maximization-based estimation methods: `mif2()`, `nonlinear forecasting`, `probe matching`, `trajectory matching`

Examples

```
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()

if (require(subplex)) {
  subplex(fn=f,par=log(c(20,0.3,10)),control=list(reltol=1e-5)) -> out
} else {
  optim(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

See Also

Csnippet

Other extraction methods: `coef()`, `cond.logLik()`, `covmat()`, `eff.sample.size()`, `filter.mean()`, `filter.traj()`, `forecast()`, `logLik`, `obs()`, `pred.mean()`, `pred.var()`, `saved.states()`, `states()`, `summary()`, `timezero()`, `time()`, `traces()`

`states` *Latent states*

Description

Extract the latent states from a 'pomp' object.

Usage

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

## S4 method for signature 'listie'
states(object, vars, ...)
```

Arguments

`object`	an object of class 'pomp', or of a class extending 'pomp'
`vars`	names of variables to retrieve
`...`	ignored

See Also

Other extraction methods: `coef()`, `cond.logLik()`, `covmat()`, `eff.sample.size()`, `filter.mean()`, `filter.traj()`, `forecast()`, `logLik`, `obs()`, `pred.mean()`, `pred.var()`, `saved.states()`, `spy()`, `summary()`, `timezero()`, `time()`, `traces()`

`summary` *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 or passed to the more primitive function

See Also

Other extraction methods: `coef()`, `cond.logLik()`, `covmat()`, `eff.sample.size()`, `filter.mean()`, `filter.traj()`, `forecast()`, `logLik`, `obs()`, `pred.mean()`, `pred.var()`, `saved.states()`, `spy()`, `states()`, `timezero()`, `time()`, `traces()`

`time` *Methods to extract and 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

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

Arguments

`x`	a ‘pomp’ object
`t0`	logical; should the zero time be included?
`...`	ignored or passed to the more primitive function
`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.

See Also

Other extraction methods: `coef()`, `cond.logLik()`, `covmat()`, `eff.sample.size()`, `filter.mean()`, `filter.traj()`, `forecast()`, `logLik`, `obs()`, `pred.mean()`, `pred.var()`, `saved.states()`, `spy()`, `states()`, `summary()`, `timezero()`, `traces()`

`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 or passed to the more primitive function
`value`	numeric; the new zero-time value

Value

the value of the zero time

See Also

Other extraction methods: `coef()`, `cond.logLik()`, `covmat()`, `eff.sample.size()`, `filter.mean()`, `filter.traj()`, `forecast()`, `logLik`, `obs()`, `pred.mean()`, `pred.var()`, `saved.states()`, `spy()`, `states()`, `summary()`, `time()`, `traces()`

`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 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)` returns the traces of the parameters named in `pars`. By default, the traces of all parameters are returned. If `transform=TRUE`, the parameters are transformed from the natural scale to the estimation 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`.

See Also

Other extraction methods: `coef()`, `cond.logLik()`, `covmat()`, `eff.sample.size()`, `filter.mean()`, `filter.traj()`, `forecast()`, `logLik`, `obs()`, `pred.mean()`, `pred.var()`, `saved.states()`, `spy()`, `states()`, `summary()`, `timezero()`, `time()`

`trajectory` *Trajectory of a deterministic model*

Description

Compute trajectories of the deterministic skeleton of a Markov process.

Usage

```
## S4 method for signature 'missing'
trajectory(
  t0,
  times,
  params,
  skeleton,
  rinit,
  ...,
  ode_control = list(),
  format = c("pomps", "array", "data.frame"),
  verbose = getOption("verbose", FALSE)
)

## S4 method for signature 'data.frame'
trajectory(
  object,
  ...,
  t0,
  times,
  params,
  skeleton,
  rinit,
  ode_control = list(),
  format = c("pomps", "array", "data.frame"),
  verbose = getOption("verbose", FALSE)
)

## S4 method for signature 'pomp'
trajectory(
```

```
  object,
  params,
  ...,
  skeleton,
  rinit,
  ode_control = list(),
  format = c("pomps", "array", "data.frame"),
  verbose = getOption("verbose", FALSE)
)

## S4 method for signature 'traj_match_objfun'
trajectory(object, ..., verbose = getOption("verbose", FALSE))
```

Arguments

`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]`.
`times`	the sequence of observation times. `times` must indicate the column of observation times by name or index. The time vector must be numeric and non-decreasing.
`params`	optional; named numeric vector of parameters. This will be coerced internally to storage mode `double`.
`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` fnctions. For more information, see skeleton specification. Setting `skeleton=NULL` removes the deterministic skeleton.
`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 specification.
`...`	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.
`ode_control`	optional list; the elements of this list will be passed to `ode` if the skeleton is a vectorfield, and ignored if it is a map.
`format`	the format in which to return the results. `format = "pomps"` causes the trajectories to be returned as a single 'pomp' object (if a single parameter vector have been furnished to `trajectory`) or as a 'pompList' object (if multiple parameters have been furnished). In each of these, the `states` slot will have been replaced by the computed trajectory. Use `states` to view these.

	`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`. Thus if `x` is the returned array, `x[i,j,k]` is the i-th component of the state vector at time `times[k]` given parameters `params[,j]`. `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.
`verbose`	logical; if `TRUE`, diagnostic messages will be printed to the console.
`object`	optional; if present, it should be a data frame or a 'pomp' object.

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.

Value

The `format` option controls the nature of the return value of `trajectory`. See above for details.

See Also

More on **pomp** elementary algorithms: `elementary algorithms`, `kalman`, `pfilter()`, `pomp-package`, `probe()`, `simulate()`, `spect()`, `wpfilter()`

More on methods for deterministic process models: `flow()`, `skeleton specification`, `skeleton()`, `trajectory matching`

Examples

```
## The basic components needed to compute trajectories
## of a deterministic dynamical system are
## rinit and skeleton.

## The following specifies these for a simple continuous-time
## model: dx/dt = r (1+e cos(t)) x

trajectory(
  t0 = 0, times = seq(1,30,by=0.1),
  rinit = function (x0, ...) {
    c(x = x0)
  },
  skeleton = vectorfield(
    function (r, e, t, x, ...) {
      c(x=r*(1+e*cos(t))*x)
    }
  ),
  params = c(r=1,e=3,x0=1)
) -> po
```

```
plot(po,log='y')

## In the case of a discrete-time skeleton,
## we use the 'map' function.  For example,
## the following computes a trajectory from
## the dynamical system with skeleton
## x -> x exp(r sin(omega t)).

trajectory(
  t0 = 0, times=seq(1,100),
  rinit = function (x0, ...) {
    c(x = x0)
  },
  skeleton = map(
    function (r, t, x, omega, ...) {
      c(x=x*exp(r*sin(omega*t)))
    },
    delta.t=1
  ),
  params = c(r=1,x0=1,omega=4)
) -> po

plot(po)



## 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))
trajectory(
  ricker(),
  times=seq(from=1000,to=2000,by=1),
  params=p,
  format="array"
) -> x
matplot(p["r",],x["N",,],pch='.',col='black',
  xlab=expression(log(r)),ylab="N",log='x')
```

`trajectory matching` *Trajectory matching*

Description

Estimation of parameters for deterministic POMP models via trajectory matching.

Usage

```
## 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,
  ode_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. Internally, `data` will be internally coerced to an array with storage-mode `double`.
`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` if the skeleton is a vectorfield, and ignored if it is a map.
`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 specification.
`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` fnctions. For more information, see skeleton specification. 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 specification.
`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

`traj_objfun` constructs a stateful objective function for spectrum matching. Specifically, `traj_objfun` returns an object of class ‘traj_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 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.

Note for Windows users

Some Windows users report problems when using C snippets in parallel computations. These appear to arise when the temporary files created during the C snippet compilation process are not handled properly by the operating system. To circumvent this problem, use the `cdir` and `cfile` options to cause the C snippets to be written to a file of your choice, thus avoiding the use of temporary files altogether.

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.

See Also

`optim`, `subplex`, `nloptr`

More on methods for deterministic process models: `flow()`, `skeleton specification`, `skeleton()`, `trajectory()`

More on maximization-based estimation methods: `mif2()`, `nonlinear forecasting`, `probe matching`, `spectrum matching`

Examples

```
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)))

if (require(subplex)) {
  subplex(fn=f,par=log(c(20,0.3,10)),control=list(reltol=1e-5)) -> out
} else {
  optim(fn=f,par=log(c(20,0.3,10)),control=list(reltol=1e-5)) -> out
}

f(out$par)

if (require(ggplot2)) {

  f %>%
    trajectory(format="data.frame") %>%
    ggplot(aes(x=time,y=N))+geom_line()+theme_bw()

}
```

`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.

Details

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` facilty, **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, $i=1,\dots,n$, 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 R^n.

The inverse of the log-barycentric transformation is implemented as `inv_log_barycentric`. Note that it is not a true inverse, in the sense that it takes R^n to the *unit* simplex, $\sum_i X_i = 1$. Thus,

```
log_barycentric(inv_log_barycentric(Y)) == Y,
```

but

```
inv_log_barycentric(log_barycentric(X)) == X
```

only if `sum(X) == 1`.

See Also

More on implementing POMP models: `Csnippet`, `accumulator variables`, `basic components`, `betabinomial`, `covariates`, `distributions`, `dmeasure specification`, `dprocess specification`, `emeasure specification`, `parameter transformations`, `pomp-package`, `pomp`, `prior specification`, `rinit specification`, `rmeasure specification`, `rprocess specification`, `skeleton specification`, `userdata`, `vmeasure specification`

`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

```
y1 ~ Poisson(x1+theta),  y2 ~ Poisson(x2+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:

```
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`, `pfilter`, `mif2`, `abc`, etc.). For example:

```
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:

```
f <- Csnippet("
 double theta = *(get_userdata_double(\"theta\"));
 y1 = rpois(x1+theta); y2 = rpois(x2+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

```
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

More on implementing POMP models: `Csnippet`, `accumulator variables`, `basic components`, `betabinomial`, `covariates`, `distributions`, `dmeasure specification`, `dprocess specification`, `emeasure specification`, `parameter transformations`, `pomp-package`, `pomp`, `prior specification`, `rinit specification`, `rmeasure specification`, `rprocess specification`, `skeleton specification`, `transformations`, `vmeasure specification`

Examples

```
## 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.

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 timestep

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

More examples provided with **pomp**: `SIR models`, `blowflies`, `childhood disease data`, `dacca()`, `ebola`, `gompertz()`, `ou2()`, `pomp examples`, `ricker()`, `rw2()`

Examples

```
## Not run: 
  verhulst() -> po
  plot(po)
  plot(simulate(po))
  pfilter(po,Np=1000) -> pf
  logLik(pf)
  spy(po)

## End(Not run)
```

`vmeasure` *vmeasure*

Description

Return the covariance matrix of the observed variables, given values of the latent states and the parameters.

Usage

```
## S4 method for signature 'pomp'
vmeasure(
  object,
  x = states(object),
  times = time(object),
  params = coef(object),
  ...
)
```

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

`vmeasure` returns a rank-4 array of dimensions `nobs` x `nobs` x `nrep` x `ntimes`, where `nobs` is the number of observed variables. If `v` is the returned array, `v[,,j,k]` contains the covariance matrix at time `times[k]` given the state `x[,j,k]`.

See Also

Specification of the measurement-model covariance matrix: vmeasure specification

More on **pomp** workhorse functions: `dmeasure()`, `dprior()`, `dprocess()`, `emeasure()`, `flow()`, `partrans()`, `pomp-package`, `rinit()`, `rmeasure()`, `rprior()`, `rprocess()`, `skeleton()`, `workhorses`

`vmeasure specification`

The variance of the measurement model

Description

Specification of the measurement-model covariance matrix, vmeasure.

Details

The measurement model is the link between the data and the unobserved state process. Some algorithms require the conditional covariance of the measurement model, given the latent state and parameters. This is supplied using the `vmeasure` argument.

Suppose you have a procedure to compute this conditional covariance matrix, given the value of the latent state variables. Then you can furnish

```
vmeasure = 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 a `vmeasure` C snippet, bear in mind that:

1. The goal of such a snippet is to fill variables named `V_y_z` with the conditional covariances of observables `y`, `z`. Accordingly, there should be one assignment of `V_y_z` and one assignment of `V_z_y` for each pair of observables `y` and `z`.
2. In addition to the states, parameters, and covariates (if any), 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.

It is also possible, though less efficient, to specify `vmeasure` using an R function. In this case, specify it by furnishing

```
vmeasure = 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 `...`. `f` must return a square matrix of dimension equal to the number of observable variables. The row- and column-names of this matrix should match the names of the observable variables. The matrix should of course be symmetric.

Default behavior

The default `vmeasure` is undefined. It will yield missing values (`NA`).

Note for Windows users

Some Windows users report problems when using C snippets in parallel computations. These appear to arise when the temporary files created during the C snippet compilation process are not handled properly by the operating system. To circumvent this problem, use the `cdir` and `cfile` options to cause the C snippets to be written to a file of your choice, thus avoiding the use of temporary files altogether.

See Also

`vmeasure`

More on implementing POMP models: `Csnippet`, `accumulator variables`, `basic components`, `betabinomial`, `covariates`, `distributions`, `dmeasure specification`, `dprocess specification`, `emeasure specification`, `parameter transformations`, `pomp-package`, `pomp`, `prior specification`, `rinit specification`, `rmeasure specification`, `rprocess specification`, `skeleton specification`, `transformations`, `userdata`

`window` *Window*

Description

Restrict to a portion of a time series.

Usage

```
## 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

`workhorses` *Workhorse functions for the* **pomp** *algorithms.*

Description

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,

`emeasure` which computes the expectation of the observed variables conditional on the latent state,

`vmeasure` which computes the covariance matrix of the observed variables conditional on the latent state,

`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

See Also

basic model components, elementary algorithms, estimation algorithms

More on **pomp** workhorse functions: `dmeasure()`, `dprior()`, `dprocess()`, `emeasure()`, `flow()`, `partrans()`, `pomp-package`, `rinit()`, `rmeasure()`, `rprior()`, `rprocess()`, `skeleton()`, `vmeasure()`

`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

```
## 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. Internally, `data` will be internally coerced to an array with storage-mode `double`.
`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 specification.
`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 specification 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 specification.
`trigger`	numeric; if the effective sample size becomes smaller than `trigger * Np`, resampling is triggered.
`target`	numeric; target power.
`...`	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

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 pomp Issues page.

Value

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

Note for Windows users

Some Windows users report problems when using C snippets in parallel computations. These appear to arise when the temporary files created during the C snippet compilation process are not handled properly by the operating system. To circumvent this problem, use the `cdir` and `cfile` options to cause the C snippets to be written to a file of your choice, thus avoiding the use of temporary files altogether.

Author(s)

Aaron A. King

References

M.S. Arulampalam, S. Maskell, N. Gordon, & T. Clapp. A tutorial on particle filters for online nonlinear, non-Gaussian Bayesian tracking. *IEEE Transactions on Signal Processing* **50**, 174–188, 2002.

See Also

More on **pomp** elementary algorithms: `elementary algorithms`, `kalman`, `pfilter()`, `pomp-package`, `probe()`, `simulate()`, `spect()`, `trajectory()`

More on sequential Monte Carlo methods: `bsmc2()`, `cond.logLik()`, `eff.sample.size()`, `filter.mean()`, `filter.traj()`, `kalman`, `mif2()`, `pfilter()`, `pmcmc()`, `pred.mean()`, `pred.var()`, `saved.states()`

More on full-information (i.e., likelihood-based) methods: `bsmc2()`, `mif2()`, `pfilter()`, `pmcmc()`

Index