Package: pomp (via r-universe)

October 27, 2026

Type Package

Title Statistical Inference for Partially Observed Markov Processes

Version 6.5

Date 2026-10-25

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(>= 4.1.0)

Imports methods, stats, graphics, digest, mvtnorm, deSolve, coda, data.table

Suggests ggplot2, dplyr, tidyr, subplex, nloptr

SystemRequirements For Windows users, Rtools (see https://cran.r-project.org/bin/windows/Rtools/).

License GPL-3

LazyData true

BugReports https://github.com/kingaa/pomp/issues/

Encoding UTF-8

Collate 'package.R' 'pstop.R' 'undefined.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' 'dinit_spec.R' 'templates.R' 'builder.R' 'pomp.R' 'probe.R' 'abc.R' 'accumulators.R' 'melt.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' 'bake.R' 'basic_components.R'
'basic_probes.R' 'betabinom.R' 'blowflies.R' 'bsflu.R'
'bsplines.R' 'childhood.R' 'coef.R' 'conc.R' 'concat.R'
'cond_logLik.R' 'covmat.R' 'dacca.R' 'design.R' 'ebola.R'
'eff_sample_size.R' 'elementary_algorithms.R' 'emeasure_spec.R'
'estimation_algorithms.R' 'eulermultinom.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' 'obs.R' 'ou2.R' 'parmat.R'
'parus.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' 'wquant.R'

Config/roxygen2/version 8.1.1

NeedsCompilation yes

Author Aaron A. King [aut, cre] (ORCID:
<https://orcid.org/0000-0001-6159-3207>), Edward L. Ionides
[aut] (ORCID: <https://orcid.org/0000-0002-4190-0174>), Carles
Bretó [aut] (ORCID: <https://orcid.org/0000-0003-4695-4902>),
Stephen P. Ellner [ctb] (ORCID:
<https://orcid.org/0000-0002-8351-9734>), Matthew J. Ferrari
[ctb] (ORCID: <https://orcid.org/0000-0001-5251-8168>),
Sebastian Funk [ctb] (ORCID:
<https://orcid.org/0000-0002-2842-3406>), Steven G. Johnson
[ctb], Bruce E. Kendall [ctb] (ORCID:
<https://orcid.org/0000-0003-1782-8106>), Michael Lavine [ctb],
Dao Nguyen [ctb] (ORCID:
<https://orcid.org/0000-0003-2215-613X>), Eamon B. O'Dea [ctb]
(ORCID: <https://orcid.org/0000-0003-4748-683X>), Daniel C.
Reuman [ctb], Helen Wearing [ctb] (ORCID:
<https://orcid.org/0000-0002-9837-9797>), Simon N. Wood [ctb]
(ORCID: <https://orcid.org/0000-0002-2034-7453>)

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

Repository https://cran.r-universe.dev

Date/Publication 2026-10-26 20:00:02 UTC

RemoteUrl https://github.com/cran/pomp

RemoteRef HEAD

RemoteSha fdf638460b52e2af3f95303c539b3af853783946

Contents

pomp-package . . . 5
abc . . . 7
accumvars . . . 10
basic_components . . . 13
basic_probes . . . 14
betabinomial . . . 16
blowflies . . . 17
bsflu . . . 19
bsmc2 . . . 20
bsplines . . . 23
childhood_disease_data . . . 24
coef . . . 25
compartmental_models . . . 27
concat . . . 29
cond_logLik . . . 30
continue . . . 31
covariates . . . 32
covmat . . . 33
Csnippet . . . 34
dacca . . . 36
design . . . 38
dinit . . . 40
dinit_spec . . . 41
dmeasure . . . 42
dmeasure_spec . . . 43
dprior . . . 45
dprocess . . . 46
dprocess_spec . . . 47
ebola . . . 48
eff_sample_size . . . 50
elementary_algorithms . . . 51
emeasure . . . 52
emeasure_spec . . . 53
estimation_algorithms . . . 54
eulermultinom . . . 55
filter_mean . . . 58
filter_traj . . . 59
flow . . . 60
forecast . . . 61
gompertz . . . 62
hitch . . . 63
kalman . . . 65
kalmanFilter . . . 68
logLik . . . 69
logmeanexp . . . 71
lookup . . . 72

mcap . . . 73
melt . . . 74
mif2 . . . 74
nlf . . . 79
obs . . . 84
ou2 . . . 84
parameter_trans . . . 86
parmat . . . 88
partrans . . . 89
parus . . . 90
pfilter . . . 91
plot . . . 94
pmcmc . . . 96
pomp_constructor . . . 99
pomp_examples . . . 104
pred_mean . . . 106
pred_var . . . 107
prior_spec . . . 108
probe . . . 110
probe_match . . . 113
proposals . . . 117
reproducibility_tools . . . 119
ricker . . . 123
rinit . . . 124
rinit_spec . . . 125
rmeasure . . . 126
rmeasure_spec . . . 127
rprior . . . 129
rprocess . . . 130
rprocess_spec . . . 131
rw_sd . . . 135
rw2 . . . 136
sannbox . . . 137
saved_states . . . 139
simulate . . . 140
skeleton . . . 143
skeleton_spec . . . 144
spect . . . 147
spect_match . . . 150
spy . . . 154
states . . . 155
summary . . . 156
time . . . 156
timezero . . . 157
traces . . . 158
traj_match . . . 159
trajectory . . . 162
transformations . . . 166

userdata 167
verhulst 170
vmeasure 171
vmeasure_spec 172
window 174
workhorses 174
wpfilter 175
wquant 177

Index **179**

---

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 nlf.
5. The particle MCMC approach of Andrieu et al. (2010); see `pmcmc`.
6. The probe-matching method of Kendall et al. (1999, 2005); see probe_match.
7. Synthetic likelihood a la Wood (2010); see `probe`.
8. A spectral probe-matching method (Reuman et al. 2006, 2008); see spect_match.
9. Approximate Bayesian computation (Toni et al. 2009); see `abc`.
10. The approximate Bayesian sequential Monte Carlo scheme of Liu & West (2001); see `bsmc2`.
11. Ensemble and ensemble adjusted Kalman filters; see kalman.
12. Simple trajectory matching; see traj_match.

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

Structure of the package

pomp algorithms are arranged into 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 R package **pomp**. *Journal of Statistical Software* **69**(12), 1–43, 2016. doi:10.18637/jss.v069.i12. An updated version of this paper is available on the **pomp** package website.

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

See Also

Useful links:

- https://kingaa.github.io/pomp/
- Report bugs at https://github.com/kingaa/pomp/issues/

More on implementing POMP models: Csnippet(), accumvars, basic_components, betabinomial, covariates, dinit_spec, dmeasure_spec, dprocess_spec, emeasure_spec, eulermultinom, parameter_trans(), pomp_constructor, prior_spec, rinit_spec, rmeasure_spec, rprocess_spec, skeleton_spec, transformations, userdata, vmeasure_spec

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

More on **pomp** estimation algorithms: abc(), bsmc2(), estimation_algorithms, mif2(), nlf, pmcmc(), probe_match, spect_match

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

---

abc *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 coerced to an array with storage-mode double.
...	additional arguments are passed to pomp. This allows one to set, unset, or modify basic model components within a call to this function.
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.

`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. doi:10.1007/s1122201192882.

T. Toni and M. P. H. Stumpf. Simulation-based model selection for dynamical systems in systems and population biology. *Bioinformatics* **26**, 104–110, 2010. doi:10.1093/bioinformatics/btp619.

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. doi:10.1098/rsif.2008.0172.

See Also

More on methods based on summary statistics: `basic_probes`, `nlf`, `probe()`, `probe_match`, `spect()`, `spect_match`

More on **pomp** estimation algorithms: `bsmc2()`, `estimation_algorithms`, `mif2()`, `nlf`, `pmcmc()`, `pomp-package`, `probe_match`, `spect_match`

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

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

---

accumvars *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, if a is a state-variable named in the `pomp`'s `accumvars` argument, then for each interval (t_k, t_{k+1}), $k = 0, 1, 2, \ldots$, a will be set to zero at prior to any `rprocess` computation over that interval. Deterministic trajectory computation is handled slightly differently: see `flow`. For examples, see `sir` and the tutorials on the package website.

See Also

sir

More on implementing POMP models: Csnippet(), basic_components, betabinomial, covariates, dinit_spec, dmeasure_spec, dprocess_spec, emeasure_spec, eulermultinom, parameter_trans(), pomp-package, pomp_constructor, prior_spec, rinit_spec, rmeasure_spec, rprocess_spec, skeleton_spec, transformations, userdata, vmeasure_spec

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()
```

---

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 also basic model components for computing the mean and variance of the measurement process conditional on the latent-state process.

There are thus altogether twelve **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. dinit, which evaluates the initial-state density,
11. skeleton, which evaluates the deterministic skeleton,
12. 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()`, `accumvars`, `betabinomial`, `covariates`, `dinit_spec`, `dmeasure_spec`, `dprocess_spec`, `emeasure_spec`, `eulermultinom`, `parameter_trans()`, `pomp-package`, `pomp_constructor`, `prior_spec`, `rinit_spec`, `rmeasure_spec`, `rprocess_spec`, `skeleton_spec`, `transformations`, `userdata`, `vmeasure_spec`

`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. doi:10.2307/176658.

S. N. Wood Statistical inference for noisy nonlinear ecological dynamic systems. *Nature* **466**, 1102–1104, 2010. doi:10.1038/nature09319.

See Also

More on methods based on summary statistics: abc(), nlf, probe(), probe_match, spect(), spect_match

---

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 $X \sim \mathrm{Binomial}(n, P)$ where $P \sim \mathrm{Beta}(\mu, \theta)$. 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()`, `accumvars`, `basic_components`, `covariates`, `dinit_spec`, `dmeasure_spec`, `dprocess_spec`, `emeasure_spec`, `eulermultinom`, `parameter_trans()`, `pomp-package`, `pomp_constructor`, `prior_spec`, `rinit_spec`, `rmeasure_spec`, `rprocess_spec`, `skeleton_spec`, `transformations`, `userdata`, `vmeasure_spec`

`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:

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

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

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

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

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

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

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. doi:10.1101/SQB.1957.022.01.017.

Y. Xia and H. Tong. Feature matching in time series modeling. *Statistical Science* **26**, 21–46, 2011. doi:10.1214/10sts345.

E.L. Ionides. Discussion of "Feature matching in time series modeling" by Y. Xia and H. Tong. *Statistical Science* **26**, 49–52, 2011. doi:10.1214/11sts345c.

S. N. Wood Statistical inference for noisy nonlinear ecological dynamic systems. *Nature* **466**, 1102–1104, 2010. doi:10.1038/nature09319.

W.S.C. Gurney, S.P. Blythe, and R.M. Nisbet. Nicholson's blowflies revisited. *Nature* **287**, 17–21, 1980. doi:10.1038/287017a0.

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. doi:10.1007/9781461413448_19.

See Also

More examples provided with **pomp**: `childhood_disease_data`, `compartmental_models`, `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

compartmental 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 coerced to an array with storage-mode double.
...	additional arguments are passed to pomp. This allows one to set, unset, or modify basic model components within a call to this function.
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.

`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

J. Liu and M. West. Combining parameter and state estimation in simulation-based filtering. In A. Doucet, N. de Freitas, and N. J. Gordon, (eds.), *Sequential Monte Carlo Methods in Practice*, pp. 197–224. Springer, New York, 2001. doi:10.1007/9781475734379_10.

See Also

More on Bayesian methods: abc(), dprior(), pmcmc(), prior_spec, 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: abc(), estimation_algorithms, mif2(), nlf, pmcmc(), pomp-package, probe_match, spect_match

---

bsplines *B-spline bases*

Description

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

Usage

```
bspline_basis(x, nbasis, degree = 3, deriv = 0, names = NULL, rg = range(x))

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.
rg	numeric of length 2; range of the B-spline basis. To be properly specified, we must have rg[1] < rg[2].
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. doi:10.1093/oxfordjournals.aje.a121575.

See Also

compartmental models, bsflu

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

More examples provided with **pomp**: blowflies, compartmental_models, 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()`, `time()`, `timezero()`, `traces()`

`compartmental_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()
```

---

concat *Concatenate*

Description

Concatenate two or more 'pomp' objects into a list-like 'listie'.

Usage

```
## S3 method for class 'Pomp'
c(...)

concat(...)
```

Arguments

...	elements to be recursively combined into a 'listie'

Details

`concat` applied to one or more 'pomp' objects or lists of 'pomp' objects converts the list into a 'listie'. In particular, `concat(A,B,C)` is equivalent to `do.call(c,unlist(list(A,B,C)))`.

Examples

```
gompertz(sigma=2,tau=1) -> g
Np <- c(low=100,med=1000,high=10000)
lapply(
  Np,
  \(np) pfilter(g,Np=np)
) |>
  concat() -> pfs

pfs
coef(pfs)
logLik(pfs)
eff_sample_size(pfs)
cond_logLik(pfs)

pfs |> plot()
```

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, ..., format = c("numeric", "data.frame"))

## S4 method for signature 'pfilterd_pomp'
cond_logLik(object, ..., format = c("numeric", "data.frame"))

## S4 method for signature 'wpfilterd_pomp'
cond_logLik(object, ..., format = c("numeric", "data.frame"))

## S4 method for signature 'bsmcd_pomp'
cond_logLik(object, ..., format = c("numeric", "data.frame"))

## S4 method for signature 'pfilterList'
cond_logLik(object, ..., format = c("numeric", "data.frame"))
```

Arguments

object	result of a filtering computation
...	ignored
format	format of the returned object

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(), time(), timezero(), traces()

---

continue *Continue an iterative calculation*

Description

Continue an iterative computation where it left off.

Usage

```
## 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)

repair_lookup_table(table, t, order)
```

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.
`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

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.

`repair_lookup_table` applies `lookup` at the provided values of `t` and returns the resulting lookup table. If `order` is unsupplied, the interpolation-order from `table` is preserved. `repair_lookup_table` **should be considered experimental: its interface may change without notice**.

Value

`covariate_table` returns a lookup table suitable for inclusion of covariates in a 'pomp' object. Specifically, this is an object of class 'covartable'.

`repair_lookup_table` returns a lookup table with entries at the provided values of `t`.

Extrapolation

If `t` is outside the range of the lookup table, the values will be extrapolated, and a warning will be issued. The type of extrapolation performed will be constant or linear according to the `order` flag used when creating the table.

See Also

More on implementing POMP models: `Csnippet()`, `accumvars`, `basic_components`, `betabinomial`, `dinit_spec`, `dmeasure_spec`, `dprocess_spec`, `emeasure_spec`, `eulermultinom`, `parameter_trans()`, `pomp-package`, `pomp_constructor`, `prior_spec`, `rinit_spec`, `rmeasure_spec`, `rprocess_spec`, `skeleton_spec`, `transformations`, `userdata`, `vmeasure_spec`

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(), time(), timezero(), 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.

Linking to precompiled libraries

It is straightforward to link C snippets with precompiled C libraries. To do so, one must make sure the library's header files are included; the `globals` argument can be used for this purpose. The `shlib.args` argument can then be used to specify additional arguments to be passed to `R CMD SHLIB`. FAQ 3.7 gives an example.

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: `accumvars`, `basic_components`, `betabinomial`, `covariates`, `dinit_spec`, `dmeasure_spec`, `dprocess_spec`, `emeasure_spec`, `eulermultinom`, `parameter_trans()`, `pomp-package`, `pomp_constructor`, `prior_spec`, `rinit_spec`, `rmeasure_spec`, `rprocess_spec`, `skeleton_spec`, `transformations`, `userdata`, `vmeasure_spec`

---

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. doi:10.1038/nature07084.

See Also

More examples provided with **pomp**: blowflies, childhood_disease_data, compartmental_models, ebola, gompertz(), ou2(), pomp_examples, ricker(), rw2(), verhulst()

More data sets provided with **pomp**: blowflies, bsflu, childhood_disease_data, ebola, parus

Examples

```
# takes too long for R CMD check
 po <- dacca()
 plot(po)
 ## MLE:
 coef(po)
 plot(simulate(po))
```

---

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. doi:10.1145/42288.214372.

S. Joe and F.Y. Kuo. Remark on algorithm 659: Implementing Sobol' quasirandom sequence generator. *ACM Transactions on Mathematical Software* **29**, 49–57, 2003. doi:10.1145/641876.641879.

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)
```

`dinit` *dinit workhorse*

Description

Evaluates the initial-state density.

Usage

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

```
  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.
...	additional arguments are ignored.
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.
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.
log	if TRUE, log probabilities are returned.

Value

dinit returns a 1-D numerical array containing the likelihoods (or log likelihoods if log=TRUE). By default, t0 is the initial time defined when the 'pomp' object ws constructed.

See Also

Specification of the initial-state distribution: dinit_spec

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

---

dinit_spec *dinit specification*

Description

Specification of the initial-state distribution density evaluator, dinit.

Details

To fully specify the unobserved Markov state process, one must give its distribution at the zero-time (t0). One specifies how to evaluate the log probability density function for this distribution using the dinit 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 computation of a log likelihood, to be put into a variable named loglik.

2. In addition to the state variables, parameters, and covariates (if any), the variable `t`, containing the zero-time, will be defined in the context in which the snippet is executed.

General rules for writing C snippets can be found here.

If an R function is to be used, pass

```
dinit = f
```

to `pomp`, where `f` is a function with arguments that can include the time `t`, any or all of the model state variables, parameters, and covariates. As usual, `f` may take additional arguments, provided these are passed along with it in the call to `pomp`. `f` must return a single numeric value, 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.

See Also

`dinit`

More on implementing POMP models: `Csnippet()`, `accumvars`, `basic_components`, `betabinomial`, `covariates`, `dmeasure_spec`, `dprocess_spec`, `emeasure_spec`, `eulermultinom`, `parameter_trans()`, `pomp-package`, `pomp_constructor`, `prior_spec`, `rinit_spec`, `rmeasure_spec`, `rprocess_spec`, `skeleton_spec`, `transformations`, `userdata`, `vmeasure_spec`

---

dmeasure *dmeasure workhorse*

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.
...	additional arguments are ignored.
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.
log	if TRUE, log probabilities are returned.

Value

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

See Also

Specification of the measurement density evaluator: dmeasure_spec

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

---

dmeasure_spec *dmeasure specification*

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 predefined `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()`, `accumvars`, `basic_components`, `betabinomial`, `covariates`, `dinit_spec`, `dprocess_spec`, `emeasure_spec`, `eulermultinom`, `parameter_trans()`, `pomp-package`, `pomp_constructor`, `prior_spec`, `rinit_spec`, `rmeasure_spec`, `rprocess_spec`, `skeleton_spec`, `transformations`, `userdata`, `vmeasure_spec`

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 workhorse*

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.
`...`	additional arguments are ignored.
`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`.
`log`	if TRUE, log probabilities are returned.

Value

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

See Also

Specification of the prior density evaluator: prior_spec

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

More on Bayesian methods: `abc()`, `bsmc2()`, `pmcmc()`, `prior_spec`, `rprior()`

`dprocess` *dprocess workhorse*

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.
`...`	additional arguments are ignored.
`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`.
`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_spec

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

---

dprocess_spec *dprocess specification*

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 predefined 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()`, `accumvars`, `basic_components`, `betabinomial`, `covariates`, `dinit_spec`, `dmeasure_spec`, `emeasure_spec`, `eulermultinom`, `parameter_trans()`, `pomp-package`, `pomp_constructor`, `prior_spec`, `rinit_spec`, `rmeasure_spec`, `rprocess_spec`, `skeleton_spec`, `transformations`, `userdata`, `vmeasure_spec`

---

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. doi:10.1098/rspb.2015.0347.

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. doi:10.1056/NEJMoa1411100.

D. He, E.L. Ionides, and 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. doi:10.1098/rsif.2009.0151.

See Also

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

More examples provided with **pomp**: blowflies, childhood_disease_data, compartmental_models, dacca(), gompertz(), ou2(), pomp_examples, ricker(), rw2(), verhulst()

Examples

```
# takes too long for R CMD check
 if (require(ggplot2) && require(tidyr)) {

   ebolaWA2014 |>
     pivot_longer(c(cases,deaths)) |>
     ggplot(aes(x=date,y=value,group=name,color=name))+
     geom_line()+
     facet_grid(country~.,scales="free_y")+
     theme_bw()+
     theme(axis.text=element_text(angle=-90))

 }

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

---

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, ..., format = c("numeric", "data.frame"))

## S4 method for signature 'pfilterd_pomp'
eff_sample_size(object, ..., format = c("numeric", "data.frame"))

## S4 method for signature 'wpfilterd_pomp'
eff_sample_size(object, ..., format = c("numeric", "data.frame"))

## S4 method for signature 'pfilterList'
eff_sample_size(object, ..., format = c("numeric", "data.frame"))
```

Arguments

`object`	result of a filtering computation
`...`	ignored
`format`	format of the returned object

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()`, `time()`, `timezero()`, `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 on 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 workhorse*

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.
...	additional arguments are ignored.
x	an array containing states of the unobserved process. The dimensions of x are nvars x nrep x ntimes, where nvars is the number of state variables, nrep is the number of replicates, and ntimes is the length of times. One can also pass x as a named numeric vector, which is equivalent to the nrep=1, ntimes=1 case.
times	a numeric vector (length ntimes) containing times. These must be in non-decreasing order.
params	a npar x nrep matrix of parameters. Each column is treated as an independent parameter set, in correspondence with the corresponding column of x.

Value

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_spec

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

`emeasure_spec` *emeasure specification*

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()`, `accumvars`, `basic_components`, `betabinomial`, `covariates`, `dinit_spec`, `dmeasure_spec`, `dprocess_spec`, `eulermultinom`, `parameter_trans()`, `pomp-package`, `pomp_constructor`, `prior_spec`, `rinit_spec`, `rmeasure_spec`, `rprocess_spec`, `skeleton_spec`, `transformations`, `userdata`, `vmeasure_spec`

---

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: `abc()`, `bsmc2()`, `mif2()`, `nlf`, `pmcmc()`, `pomp-package`, `probe_match`, `spect_match`

`eulermultinom` *Eulermultinomial and Gamma-whitenoise 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)

eeulermultinom(size, rate, dt)

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:

$$(\Delta n_0, \Delta n_1, \ldots, \Delta n_K) \sim \text{Multinomial}(N; p_0, p_1, \ldots, p_K),$$

where $\Delta n_0 = N - \sum_{k=1}^{K} \Delta n_k$ is the number of individuals remaining alive and Δn_k is the number of individuals dying in way k over the interval. Here, the probability of remaining alive is

$$p_0 = \exp(-\sum_k r_k \Delta t)$$

and the probability of dying in way k is

$$p_k = \frac{r_k}{\sum_j r_j}(1 - p_0).$$

In this case, we say that

$$(\Delta n_1, \ldots, \Delta n_K) \sim \text{Eulermultinom}(N, r, \Delta t),$$

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

```
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).
```

Bretó & 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 $\Delta W \sim \text{Gamma}(\Delta t/\sigma^2, \sigma^2)$ 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 and the "Simulation-based Inference" course discusses the value of allowing for overdispersion more generally.

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`). This contains the probabilities of observing each column of `x` given the specified parameters (`size`, `rate`, `dt`).

`eeulermultinom` returns a `length(rate)`-vector containing the expected number of individuals to have succumbed to the corresponding process. With the notation above, if

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

then the k-th element of the vector `dn` will be $p_k N$.

`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 processes and their applications to modeling infinitesimally over-dispersed systems. *Stochastic Processes and their Applications* **121**, 2571–2591, 2011. doi:10.1016/j.spa.2011.07.005. D. He, E.L. Ionides, and 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. doi:10.1098/rsif.2009.0151.

See Also

More on implementing POMP models: `Csnippet()`, `accumvars`, `basic_components`, `betabinomial`, `covariates`, `dinit_spec`, `dmeasure_spec`, `dprocess_spec`, `emeasure_spec`, `parameter_trans()`, `pomp-package`, `pomp_constructor`, `prior_spec`, `rinit_spec`, `rmeasure_spec`, `rprocess_spec`, `skeleton_spec`, `transformations`, `userdata`, `vmeasure_spec`

Examples

```
## Simulate 5 realizations of Euler-multinomial random variable:

dn <- reulermultinom(5,size=100,rate=c(a=1,b=2,c=3),dt=0.1)
dn

## Compute the probability mass function at each of the 5 realizations:

deulermultinom(x=dn,size=100,rate=c(1,2,3),dt=0.1)

## Compute the expectation of an Euler-multinomial:

eeulermultinom(size=100,rate=c(a=1,b=2,c=3),dt=0.1)

## An Euler-multinomial with overdispersed transitions:

dt <- 0.1
dW <- rgammawn(sigma=0.1,dt=dt)
reulermultinom(5,size=100,rate=c(a=1,b=2,c=3),dt=dW)
```

`filter_mean` *Filtering mean*

Description

The mean of the filtering distribution

Usage

```
## S4 method for signature 'kalmand_pomp'
filter_mean(object, vars, ..., format = c("array", "data.frame"))

## S4 method for signature 'pfilterd_pomp'
filter_mean(object, vars, ..., format = c("array", "data.frame"))
```

Arguments

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

Details

The filtering distribution is that of

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

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

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

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()`, `time()`, `timezero()`, `traces()`

`filter_traj` *Filtering trajectories*

Description

Drawing from the smoothing distribution

Usage

```
## S4 method for signature 'pfilterd_pomp'
filter_traj(object, vars, ..., format = c("array", "data.frame"))

## S4 method for signature 'listie'
filter_traj(object, vars, ..., format = c("array", "data.frame"))

## S4 method for signature 'pmcmcd_pomp'
filter_traj(object, vars, ...)
```

Arguments

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

Details

The smoothing distribution is the distribution of

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

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

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

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

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

See Also

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(), time(), timezero(), traces()

`flow` *flow workhorse*

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.
`...`	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).
`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`.
`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]`.

Accumulator variables

When there are accumulator variables (as determined by the `accumvars` argument), their handling in the continuous-time (vectorfield) case differs from that in the discrete-time (map) case. In the latter, accumulator variables are set to zero at the beginning of each interval (t_k, t_{k+1}), $k = 0, 1, 2, \ldots$ over which flow computation is required. In the former, the flow computation proceeds over the entire set of intervals required, and accumulator variables are then differenced. That is, the value a_k of accumulator variable a at times t_k, $k = 1, 2, \ldots$ will be $A_k - A_{k-1}$, where A_k is the solution of the corresponding differential equation at t_k.

See Also

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

More on methods for deterministic process models: `skeleton()`, `skeleton_spec`, `traj_match`, `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, ..., format = c("array", "data.frame"))

## S4 method for signature 'pfilterd_pomp'
forecast(object, vars, ..., format = c("array", "data.frame"))
```

Arguments

object	result of a filtering computation
...	ignored
vars	optional character; names of variables
format	format of the returned object

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(), time(), timezero(), 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,
  seed = 299438676L
)
```

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
seed	seed of the random number generator

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

$$Y_t \sim \text{Lognormal}(\log X_t, \tau).$$

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**: `blowflies`, `childhood_disease_data`, `compartmental_models`, `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),
  on_load,
  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 or C snippet; 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.
`on_load`	optional character or C snippet: arbitrary C code that will be executed when the C snippet file is loaded. If no C snippets are used, `on_load` has no effect.
`shlib.args`	optional character variables. Command-line arguments to the `R CMD SHLIB` call that compiles the C snippets. One can, for example, specify libraries against which the C snippets are to be linked. In doing so, take care to make sure the

appropriate header files are available to the C snippets, e.g., using the `globals` argument. See `Csnippet` for more information.

`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 coerced to an array with storage-mode double.
...	additional arguments are passed to pomp. This allows one to set, unset, or modify basic model components within a call to this function.
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.

`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. doi:10.1029/94JC00572.

J.L. Anderson. An ensemble adjustment Kalman filter for data assimilation. *Monthly Weather Review* **129**, 2884–2903, 2001. doi:10.1175/15200493(2001)129<2884:AEAKFF>2.0.CO;2.

G. Evensen. *Data assimilation: the ensemble Kalman filter*. Springer-Verlag, 2009. doi:10.1007/9783642037115.

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 = rinit(object), 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

$$X_t \sim \mathrm{MVN}(AX_{t-1}, Q)$$

$$Y_t \sim \mathrm{MVN}(CX_t, R)$$

where $\mathrm{MVN}(M, V)$ denotes the multivariate normal distribution with mean M and variance V. 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 $E[X_t|y_1^*, \ldots, y_t^*]$

pred.mean $E[X_t|y_1^*, \ldots, y_{t-1}^*]$

forecast $E[Y_t|y_1^*, \ldots, y_{t-1}^*]$

cond.logLik $f(y_t^*|y_1^*, \ldots, y_{t-1}^*)$

logLik $f(y_1^*, \ldots, y_T^*)$

See Also

enkf, eakf

Examples

```
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)))

}
```

---

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()`, `time()`, `timezero()`, `traces()`

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

Description

`logmeanexp` computes

$$\log \frac{1}{n} \sum_{i=1}^{n} e^{x_i},$$

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

Usage

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

Arguments

`x`	numeric
`se`	logical; give approximate standard error?
`ess`	logical; give effective sample size?

Details

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

When `ess = TRUE`, `logmeanexp` returns an estimate of the effective sample size.

Value

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

Author(s)

Aaron A. King

Examples

```
# takes too long for R CMD check
 ## an estimate of the log likelihood:
 ricker() |>
   pfilter(Np=1000) |>
   logLik() |>
   replicate(n=5) -> ll
 logmeanexp(ll)
 ## with standard error:
 logmeanexp(ll,se=TRUE)
 ## with effective sample size
 logmeanexp(ll,ess=TRUE)
```

---

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.

Value

A numeric vector or matrix of the interpolated values.

Extrapolation

If t is outside the range of the lookup table, the values will be extrapolated, and a warning will be issued. The type of extrapolation performed will be constant or linear according to the order flag used when creating the table.

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. Bretó, 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. doi:10.1098/rsif.2017.0126.

`melt` *Melt*

Description

Convert arrays, lists, and other objects to data frames.

Usage

```
## S4 method for signature 'ANY'
melt(data, ...)

## S4 method for signature 'array'
melt(data, ...)

## S4 method for signature 'list'
melt(data, ..., level = 1)
```

Arguments

`data`	object to convert
`...`	ignored
`level`	integer; level of recursion

Details

`melt` converts its first argument to a data frame. It is a simplified version of the `melt` command provided by the no-longer maintained **reshape2** package.

An array can be melted into a data frame. In this case, the data frame will have one row per entry in the array.

A list can be melted into a data frame. This operation is recursive. A variable will be appended to distinguish the separate list entries.

`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 coerced to

an array with storage-mode `double`.

`...` additional arguments are passed to `pomp`. This allows one to set, unset, or modify basic model components within a call to this function.

`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 vector of observation times is visible (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.

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 in which the vector of observation times is defined (as '`time`'). 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),
  g=ifelse(time>=23 & time<50, 0.02, 0),
  h=ivp(0.1,lags=3:8)
)
```

results in random perturbations of parameter `a` with s.d. 0.05 at every time step, while parameters `b` and `c` both get perturbations of s.d. 0.2 only before the first observation (i.e., at the zero-time). Parameters `d` and `e`, by contrast, get perturbations of s.d. 0.2 only before the thirteenth observation. Parameter `f` gets a random perturbation of size 0.02 before every observation falling before $t = 23$, while `g` gets perturbed before all observations that fall in the interval $23 \le t < 50$. Finally, the magnitude of the perturbation of parameter `h` is applied before the third through eighth observations.

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 $\alpha =$`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. doi:10.1073/pnas.1410597112.

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: abc(), bsmc2(), estimation_algorithms, nlf, pmcmc(), pomp-package, probe_match, spect_match

More on maximization-based estimation methods: nlf, probe_match, spect_match, traj_match

---

nlf *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 coerced to an array with storage-mode `double`.
`...`	additional arguments are passed to pomp. This allows one to set, unset, or modify basic model components within a call to this function.
`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.

`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.

Warning! Objective functions based on C snippets

If you use C snippets (see `Csnippet`), a dynamically loadable library will be built. As a rule, **pomp** functions load this library as needed and unload it when it is no longer needed. The stateful objective functions are an exception to this rule. For efficiency, calls to the objective function do not execute

pompLoad or pompUnload: rather, it is assumed that pompLoad has been called before any call to the objective function. When a stateful objective function using one or more C snippets is created, pompLoad is called internally to build and load the library: therefore, within a single R session, if one creates a stateful objective function, one can freely call that objective function and (more to the point) pass it to an optimizer that calls it freely, without needing to call pompLoad. On the other hand, if one retrieves a stored objective function from a file, or passes one to another R session, one must call pompLoad before using it. **Failure to do this will typically result in a segmentation fault (i.e., it will crash the R session).**

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. doi:10.1086/286130.

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. doi:10.2307/176658.

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. doi:10.1890/034056.

See Also

optim subplex nloptr

More on **pomp** estimation algorithms: abc(), bsmc2(), estimation_algorithms, mif2(), pmcmc(), pomp-package, probe_match, spect_match

More on methods based on summary statistics: abc(), basic_probes, probe(), probe_match, spect(), spect_match

More on maximization-based estimation methods: mif2(), probe_match, spect_match, traj_match

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, ..., format = c("array", "data.frame"))

## S4 method for signature 'listie'
obs(object, vars, ..., format = c("array", "data.frame"))
```

Arguments

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

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(), time(), timezero(), 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,
  seed = 787832394L
)
```

Arguments

`alpha_1`, `alpha_2`, `alpha_3`, `alpha_4`	entries of the α 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 σ 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
`seed`	seed of the random number generator

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**: `blowflies`, `childhood_disease_data`, `compartmental_models`, `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_trans` *parameter transformations*

Description

Equipping models with parameter transformations to facilitate 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 $\theta_1, \ldots, \theta_n$ is given by

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

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()`, `accumvars`, `basic_components`, `betabinomial`, `covariates`, `dinit_spec`, `dmeasure_spec`, `dprocess_spec`, `emeasure_spec`, `eulermultinom`, `pomp-package`, `pomp_constructor`, `prior_spec`, `rinit_spec`, `rmeasure_spec`, `rprocess_spec`, `skeleton_spec`, `transformations`, `userdata`, `vmeasure_spec`

---

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

```
# takes too long for R CMD check
 ## 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 workhorse*

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`.
`...`	additional arguments are ignored.
`dir`	the direction of the transformation to perform.

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: dinit(), 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. doi:10.1093/oso/9780198577300.003.0006.

See Also

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

Examples

```
# takes too long for R CMD check
 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()
```

---

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 = c("no", "filter", "prediction", "weighted", "unweighted", "FALSE",
    "TRUE"),
  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 = c("no", "filter", "prediction", "weighted", "unweighted", "FALSE",
```

```
    "TRUE"),
  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 coerced to an array with storage-mode double.
...	additional arguments are passed to pomp. This allows one to set, unset, or modify basic model components within a call to this function.
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` character; If `save.states="no"` (the default), information on the latent states is not saved. If `save.states="filter"`, the state-vector for each filtered particle $X^F_{n,j}$ at each time n is saved. If `save.states="prediction"`, the state-vector for each prediction particle $X^P_{n,j}$ at each time n is saved, along with the corresponding weight $w_{n,j} = f_{Y_n|X_n}(y^*|X^P_{n,j};\theta)$. The options "unweighted", "weighted", TRUE, and FALSE are deprecated and will issue a warning if used, mapping to the new values for backward compatibility. The options "unweighted" and TRUE are synonymous with "filter"; the option "weighted" is synonymous with "prediction"; the option FALSE is synonymous with "no". To retrieve the saved states, apply `saved_states` to the result of the `pfilter` computation.

`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 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, and T. Clapp. A tutorial on particle filters for online nonlinear, non-Gaussian Bayesian tracking. *IEEE Transactions on Signal Processing* **50**, 174–188, 2002. doi:10.1109/78.978374.

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. doi:10.1007/s11222-0149513x.

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="filter")
fm <- filter_mean(pf) ## extract the filtering means
ft <- filter_traj(pf) ## one draw from the smoothing distribution
ss <- saved_states(pf,format="d") ## 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 coerced to an array with storage-mode `double`.
`...`	additional arguments are passed to `pomp`. This allows one to set, unset, or modify basic model components within a call to this function.
`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.

`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. doi:10.1111/j.14679868.2009.00736.x.

See Also

More on **pomp** estimation algorithms: `abc()`, `bsmc2()`, `estimation_algorithms`, `mif2()`, `nlf`, `pomp-package`, `probe_match`, `spect_match`

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: `abc()`, `proposals`

More on Bayesian methods: `abc()`, `bsmc2()`, `dprior()`, `prior_spec`, `rprior()`

`pomp_constructor` *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;

dinit which evaluates the density 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,
  dinit,
  rprocess,
  dprocess,
  rmeasure,
  dmeasure,
  emeasure,
  vmeasure,
  skeleton,
  rprior,
  dprior,
  partrans,
  covar,
  params,
  accumvars,
  obsnames,
  statenames,
  paramnames,
  covarnames,
  nstatevars,
  PACKAGE,
  globals,
  on_load,
  userdata,
  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 coerced to an array with storage-mode double.
...	additional arguments will generate an error.
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].
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.
dinit	evaluator of the initial-state density. 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 dinit=NULL removes this basic component. For more information, see dinit 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	evaluator of the probability density of transitions 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 accumvars 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. See also `nstatevars`.

`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.

`nstatevars` optional integer scalar; If C snippets or native routines are used, one can specify the number of state variables with this argument. By default, `nstatevars = length(statenames)`.

`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 or C snippet; 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.

`on_load` optional character or C snippet: arbitrary C code that will be executed when the C snippet file is loaded. If no C snippets are used, `on_load` has no effect.

`userdata` optional list; the elements of this list will be available to basic model components. 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.

`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. One can, for example, specify libraries against which the C snippets are to be linked. In doing so, take care to make sure the appropriate header files are available to the C snippets, e.g., using the `globals` argument. See `Csnippet` for more information.

`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 R package **pomp**. *Journal of Statistical Software* **69**(12), 1–43, 2016. doi:10.18637/jss.v069.i12. An updated version of this paper is available on the **pomp** package website.

See Also

More on implementing POMP models: `Csnippet()`, `accumvars`, `basic_components`, `betabinomial`, `covariates`, `dinit_spec`, `dmeasure_spec`, `dprocess_spec`, `emeasure_spec`, `eulermultinom`, `parameter_trans()`, `pomp-package`, `prior_spec`, `rinit_spec`, `rmeasure_spec`, `rprocess_spec`, `skeleton_spec`, `transformations`, `userdata`, `vmeasure_spec`

`pomp_examples` *pre-built pomp examples*

Description

Examples of pomp objects containing models and data.

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. doi:10.1038/nature07084.

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. doi:10.1098/rspb.2015.0347.

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. doi:10.1093/oxfordjournals.aje.a121575.

A.J. Nicholson. The self-adjustment of populations to change. *Cold Spring Harbor Symposia on Quantitative Biology* **22**, 153–173, 1957. doi:10.1101/SQB.1957.022.01.017.

See Also

More examples provided with **pomp**: `blowflies`, `childhood_disease_data`, `compartmental_models`, `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, ..., format = c("array", "data.frame"))

## S4 method for signature 'pfilterd_pomp'
pred_mean(object, vars, ..., format = c("array", "data.frame"))
```

Arguments

object	result of a filtering computation
vars	optional character; names of variables
...	ignored
format	format of the returned object

Details

The prediction distribution is that of

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

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

The prediction mean is therefore the expectation of this distribution

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

See Also

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(), time(), timezero(), traces()

pred_var *Prediction variance*

Description

The variance of the prediction distribution

Usage

```
## S4 method for signature 'pfilterd_pomp'
pred_var(object, vars, ..., format = c("array", "data.frame"))
```

Arguments

object	result of a filtering computation
vars	optional character; names of variables
...	ignored
format	format of the returned object

Details

The prediction distribution is that of

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

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

The prediction variance is therefore the variance of this distribution

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

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(), time(), timezero(), traces()

`prior_spec` *prior specification*

Description

Specification of prior distributions via the rprior and dprior components.

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(), accumvars, basic_components, betabinomial, covariates, dinit_spec, dmeasure_spec, dprocess_spec, emeasure_spec, eulermultinom, parameter_trans(), pomp-package, pomp_constructor, rinit_spec, rmeasure_spec, rprocess_spec, skeleton_spec, transformations, userdata, vmeasure_spec

More on Bayesian methods: abc(), bsmc2(), dprior(), pmcmc(), rprior()

Examples

```
# takes too long for R CMD check
 ## 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"))
```

---

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 coerced to an array with storage-mode double.
...	additional arguments are passed to pomp. This allows one to set, unset, or modify basic model components within a call to this function.
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 set, the pseudorandom number generator (RNG) will be initialized with seed. The RNG will be restored to its original state afterward.
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.
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. doi:10.2307/176658.

S. N. Wood Statistical inference for noisy nonlinear ecological dynamic systems. *Nature* **466**, 1102–1104, 2010. doi:10.1038/nature09319.

See Also

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

More on methods based on summary statistics: `abc()`, `basic_probes`, `nlf`, `probe_match`, `spect()`, `spect_match`

`probe_match` *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 coerced to an array with storage-mode double.
...	additional arguments are passed to pomp. This allows one to set, unset, or modify basic model components within a call to this function.
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.

`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.

Warning! Objective functions based on C snippets

If you use C snippets (see Csnippet), a dynamically loadable library will be built. As a rule, **pomp** functions load this library as needed and unload it when it is no longer needed. The stateful objective functions are an exception to this rule. For efficiency, calls to the objective function do not execute pompLoad or pompUnload: rather, it is assumed that pompLoad has been called before any call to the objective function. When a stateful objective function using one or more C snippets is created, pompLoad is called internally to build and load the library: therefore, within a single R session, if one creates a stateful objective function, one can freely call that objective function and (more to the point) pass it to an optimizer that calls it freely, without needing to call pompLoad. On the other hand, if one retrieves a stored objective function from a file, or passes one to another R session, one must call pompLoad before using it. **Failure to do this will typically result in a segmentation fault (i.e., it will crash the R session).**

Author(s)

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. doi:10.2307/176658.

S. N. Wood Statistical inference for noisy nonlinear ecological dynamic systems. *Nature* **466**, 1102–1104, 2010. doi:10.1038/nature09319.

See Also

optim subplex nloptr

More on methods based on summary statistics: abc(), basic_probes, nlf, probe(), spect(), spect_match

More on **pomp** estimation algorithms: abc(), bsmc2(), estimation_algorithms, mif2(), nlf, pmcmc(), pomp-package, spect_match

More on maximization-based estimation methods: mif2(), nlf, spect_match, traj_match

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. doi:10.1198/jcgs.2009.06134.

See Also

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

`reproducibility_tools` *Tools for reproducible computations*

Description

Archiving of computations and control of the random-number generator.

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,
  timing = TRUE,
  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()
)

append_data(
```

```
  data,
  file,
  overwrite = FALSE,
  dir = getOption("pomp_archive_dir", getwd())
)
```

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'); for append_data, this will be a CSV file.
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. If it does not exist, this directory will be created (with a message).
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.
data	data frame
overwrite	logical; if TRUE, data are written to file, replacing any existing contents. If FALSE, the data is appended to the existing contents of file.

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.

append_data returns a data frame containing the new contents of file, invisibly.

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**: `blowflies`, `childhood_disease_data`, `compartmental_models`, `dacca()`, `ebola`, `gompertz()`, `ou2()`, `pomp_examples`, `rw2()`, `verhulst()`

Examples

```
po <- ricker()
plot(po)
coef(po)
simulate(po) |> plot()
 # takes too long for R CMD check
  ## 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 workhorse*

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.
`...`	additional arguments are ignored.
`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`.

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

_spec of the initial-state distribution: rinit_spec

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

`rinit_spec` *rinit specification*

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(), accumvars, basic_components, betabinomial, covariates, dinit_spec, dmeasure_spec, dprocess_spec, emeasure_spec, eulermultinom, parameter_trans(), pomp-package, pomp_constructor, prior_spec, rmeasure_spec, rprocess_spec, skeleton_spec, transformations, userdata, vmeasure_spec

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 workhorse*

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.
...	additional arguments are ignored.
x	an array containing states of the unobserved process. The dimensions of x are nvars x nrep x ntimes, where nvars is the number of state variables, nrep is the number of replicates, and ntimes is the length of times. One can also pass x as a named numeric vector, which is equivalent to the nrep=1, ntimes=1 case.
times	a numeric vector (length ntimes) containing times. These must be in non-decreasing order.
params	a npar x nrep matrix of parameters. Each column is treated as an independent parameter set, in correspondence with the corresponding column of x.

Value

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

See Also

Specification of the measurement-model simulator: rmeasure_spec

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

---

rmeasure_spec *rmeasure specification*

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()`, `accumvars`, `basic_components`, `betabinomial`, `covariates`, `dinit_spec`, `dmeasure_spec`, `dprocess_spec`, `emeasure_spec`, `eulermultinom`, `parameter_trans()`, `pomp-package`, `pomp_constructor`, `prior_spec`, `rinit_spec`, `rprocess_spec`, `skeleton_spec`, `transformations`, `userdata`, `vmeasure_spec`

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 workhorse*

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

Value

A numeric matrix containing the required samples.

See Also

Specification of the prior distribution simulator: prior_spec

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

More on Bayesian methods: abc(), bsmc2(), dprior(), pmcmc(), prior_spec

---

rprocess *rprocess workhorse*

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.
...	additional arguments are ignored.
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.

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_spec

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

---

rprocess_spec *rprocess specification*

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 furnished as an R function or as a C snippet. If `f` is an R function, its arguments should come from the state variables, parameters, covariates, and time. It must also have the arguments '`j`' and '`...`'. When `f` is called, the integer `j` will indicate the elementary event (corresponding to the column the matrix `v`, see below). `f` should return a single numerical value, representing the rate of that elementary event at that point in state space and time. If `f` is supplied as a C snippet, the parameters, latent state variables, covariates, and time will be visible in the context wherein the snippet is executed, as will the integer '`j`'. The purpose of the C snippet is to fill the double-precision variable '`rate`' with the corresponding event rate.

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.

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()`, `accumvars`, `basic_components`, `betabinomial`, `covariates`, `dinit_spec`, `dmeasure_spec`, `dprocess_spec`, `emeasure_spec`, `eulermultinom`, `parameter_trans()`, `pomp-package`, `pomp_constructor`, `prior_spec`, `rinit_spec`, `rmeasure_spec`, `skeleton_spec`, `transformations`, `userdata`, `vmeasure_spec`

---

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,
  seed = 1376784970L
)
```

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
seed	seed of the random number generator

Details

The random-walk process is fully but noisily observed.

Value

A 'pomp' object containing simulated data.

See Also

More examples provided with **pomp**: blowflies, childhood_disease_data, compartmental_models, 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, ..., format = c("list", "data.frame"))

## S4 method for signature 'pfilterList'
saved_states(object, ..., format = c("list", "data.frame"))
```

Arguments

object	result of a filtering computation
...	ignored
format	character; format of the returned object (see below).

Details

When one calls pfilter with save.states="filter" or save.states="prediction", the latent state vector associated with each particle is saved. This can be extracted by calling saved_states on the 'pfilterd.pomp' object. If the filtered particles are saved, these particles are *unweighted*, saved *after* resampling using their normalized weights. If the argument save.states="prediction" was used, the particles correspond to simulations from rprocess, and their corresponding unnormalized weights are included in the output.

Value

According to the format argument, the saved states are returned either as a list or a data frame.

If format="data.frame", then the returned data frame holds the state variables and (optionally) the unnormalized log weight of each particle at each observation time. The .id variable distinguishes particles.

If format="list" and pfilter was called with save.states="unweighted" or save.states="TRUE", the returned list contains one element per observation time. Each element consists of a matrix, with one row for each state variable and one column for each particle. If pfilter was called with save.states="weighted", the list itself contains two lists: the first holds the particles as above, the second holds the corresponding unnormalized log weights. In particular, it has one element per observation time; each element is the vector of per-particle log weights.

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(), time(), timezero(), 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(
  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 '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

object	optional; if present, it should be a data frame or a 'pomp' object.
nsim	The number of simulations to perform. Note that the number of replicates will be nsim times ncol(params).
seed	optional integer; if set, the pseudorandom number generator (RNG) will be initialized with seed. The RNG will be restored to its original state afterward.
...	additional arguments are passed to pomp.
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, if `params` is a vector, or a list of "pomp" objects, if `params` is a matrix with more than one column. Each of these will be identical to `object` except in that the latent states and observations will have been replaced by their 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"`.

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

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()`

skeleton *skeleton workhorse*

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 `flow` and `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.
`...`	additional arguments are ignored.
`x`	an array containing states of the unobserved process. The dimensions of `x` are `nvars` x `nrep` x `ntimes`, where `nvars` is the number of state variables, `nrep` is the number of replicates, and `ntimes` is the length of `times`. One can also pass `x` as a named numeric vector, which is equivalent to the `nrep=1`, `ntimes=1` case.
`times`	a numeric vector (length `ntimes`) containing times. These must be in non-decreasing order.
`params`	a `npar` x `nrep` matrix of parameters. Each column is treated as an independent parameter set, in correspondence with the corresponding column of `x`.

Value

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

See Also

Specification of the deterministic skeleton: skeleton_spec

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

More on methods for deterministic process models: flow(), skeleton_spec, traj_match, trajectory()

---

skeleton_spec *skeleton specification*

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.

Masking of `map`

Other packages (most notably the **tidyverse** package **purrr**) have functions named 'map'. Beware that, if you load one of these packages after you load **pomp**, the **pomp** function `map` described here will be masked. You can always access the **pomp** function by calling `pomp::map`.

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()`, `accumvars`, `basic_components`, `betabinomial`, `covariates`, `dinit_spec`, `dmeasure_spec`, `dprocess_spec`, `emeasure_spec`, `eulermultinom`, `parameter_trans()`, `pomp-package`, `pomp_constructor`, `prior_spec`, `rinit_spec`, `rmeasure_spec`, `rprocess_spec`, `transformations`, `userdata`, `vmeasure_spec`

More on methods for deterministic process models: `flow()`, `skeleton()`, `traj_match`, `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 coerced to an array with storage-mode double.
...	additional arguments are passed to pomp. This allows one to set, unset, or modify basic model components within a call to this function.
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.

`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. doi:10.1073/pnas.0608571103.

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. doi:10.1111/j.14610248.2008.01194.x.

See Also

More on methods based on summary statistics: abc(), basic_probes, nlf, probe(), probe_match, spect_match

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

---

spect_match *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 coerced to an array with storage-mode double.
...	additional arguments are passed to pomp. This allows one to set, unset, or modify basic model components within a call to this function.
est	character vector; the names of parameters to be estimated.
weights	optional numeric or function. The mismatch between model and data is measured by a weighted average of mismatch at each frequency. By default, all frequencies are weighted equally. weights can be specified either as a vector (which must have length equal to the number of frequencies) or as a function of frequency. If the latter, weights(freq) must return a nonnegative weight for each frequency.

fail.value	optional numeric scalar; if non-NA, this value is substituted for non-finite values of the objective function. It should be a large number (i.e., bigger than any legitimate values the objective function is likely to take).
vars	optional; names of observed variables for which the power spectrum will be computed. By default, the spectrum will be computed for all observables.
kernel.width	width parameter for the smoothing kernel used for calculating the estimate of the spectrum.
nsim	the number of model simulations to be computed.
seed	integer. When fitting, it is often best to fix the seed of the random-number generator (RNG). This is accomplished by setting seed to an integer. By default, seed = NULL, which does not alter the RNG state.
transform.data	function; this transformation will be applied to the observables prior to estimation of the spectrum, and prior to any detrending.
detrend	de-trending operation to perform. Options include no detrending, and subtraction of constant, linear, and quadratic trends from the data. Detrending is applied to each data series and to each model simulation independently.
params	optional; named numeric vector of parameters. This will be coerced internally to storage mode double.
rinit	simulator of the initial-state distribution. This can be furnished either as a C snippet, an R function, or the name of a pre-compiled native routine available in a dynamically loaded library. Setting rinit=NULL sets the initial-state simulator to its default. For more information, see rinit 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.
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.

Warning! Objective functions based on C snippets

If you use C snippets (see `Csnippet`), a dynamically loadable library will be built. As a rule, **pomp** functions load this library as needed and unload it when it is no longer needed. The stateful objective functions are an exception to this rule. For efficiency, calls to the objective function do not execute `pompLoad` or `pompUnload`: rather, it is assumed that `pompLoad` has been called before any call to the objective function. When a stateful objective function using one or more C snippets is created, `pompLoad` is called internally to build and load the library: therefore, within a single R session, if one creates a stateful objective function, one can freely call that objective function and (more to the point) pass it to an optimizer that calls it freely, without needing to call `pompLoad`. On the other hand, if one retrieves a stored objective function from a file, or passes one to another R session, one must call `pompLoad` before using it. **Failure to do this will typically result in a segmentation fault (i.e., it will crash the R session).**

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. doi:10.1073/pnas.0608571103.

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. doi:10.1111/j.14610248.2008.01194.x.

See Also

spect optim subplex nloptr

More on **pomp** estimation algorithms: abc(), bsmc2(), estimation_algorithms, mif2(), nlf, pmcmc(), pomp-package, probe_match

More on methods based on summary statistics: abc(), basic_probes, nlf, probe(), probe_match, spect()

More on maximization-based estimation methods: mif2(), nlf, probe_match, traj_match

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(), time(), timezero(), traces()

Examples

```
ricker() |> spy()

sir() |> spy()

sir2() |> spy()
```

---

states *Latent states*

Description

Extract the latent states from a 'pomp' object.

Usage

```
## S4 method for signature 'pomp'
states(object, vars, ..., format = c("array", "data.frame"))

## S4 method for signature 'listie'
states(object, vars, ..., format = c("array", "data.frame"))
```

Arguments

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

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(), time(), timezero(), 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(), time(), timezero(), traces()

time *Methods to extract and manipulate the obseration 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()`, `time()`, `timezero()`

---

traj_match *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 coerced to an array with storage-mode double.
...	additional arguments will modify the model structure
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.

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.

Warning! Objective functions based on C snippets

If you use C snippets (see Csnippet), a dynamically loadable library will be built. As a rule, **pomp** functions load this library as needed and unload it when it is no longer needed. The stateful objective functions are an exception to this rule. For efficiency, calls to the objective function do not execute pompLoad or pompUnload: rather, it is assumed that pompLoad has been called before any call to

the objective function. When a stateful objective function using one or more C snippets is created, pompLoad is called internally to build and load the library: therefore, within a single R session, if one creates a stateful objective function, one can freely call that objective function and (more to the point) pass it to an optimizer that calls it freely, without needing to call pompLoad. On the other hand, if one retrieves a stored objective function from a file, or passes one to another R session, one must call pompLoad before using it. **Failure to do this will typically result in a segmentation fault (i.e., it will crash the R session).**

See Also

optim, subplex, nloptr

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

More on maximization-based estimation methods: mif2(), nlf, probe_match, spect_match

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()

}
```

---

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

`...`	additional arguments are passed to pomp.
`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	a named numeric vector or a matrix with rownames containing the parameters at which the simulations are to be performed.
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.
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 has been furnished to trajectory) or as a 'pompList' object (if a matrix of 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 vectorfield; 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(), skeleton_spec, traj_match

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)


# takes too long for R CMD check
## 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')
```

`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 $X \in R^n_+$, to a vector $Y \in R^n$, where

$$Y_i = \log \frac{X_i}{\sum_j X_j}.$$

The transformation is not one-to-one. However, for each $c > 0$, it maps the simplex $\{X \in R^n_+ : \sum_i X_i = c\}$ bijectively onto 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, $\{X \in R^n_+ : \sum_i X_i = 1\}$. 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()`, `accumvars`, `basic_components`, `betabinomial`, `covariates`, `dinit_spec`, `dmeasure_spec`, `dprocess_spec`, `emeasure_spec`, `eulermultinom`, `parameter_trans()`, `pomp-package`, `pomp_constructor`, `prior_spec`, `rinit_spec`, `rmeasure_spec`, `rprocess_spec`, `skeleton_spec`, `userdata`, `vmeasure_spec`

`userdata` *Facilities for making additional information available to basic model 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, userdata = list(theta = 42), ...)
```

where the `...` represent other arguments.

When the basic component is specified via a C snippet

A C snippet implementation of the aforementioned measurement model is:

```
f <- Csnippet(r"{
 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 that, by using R string literals (`r"{}"`) we avoid 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, userdata = list(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 often more efficient, 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(), accumvars, basic_components, betabinomial, covariates, dinit_spec, dmeasure_spec, dprocess_spec, emeasure_spec, eulermultinom, parameter_trans(), pomp-package, pomp_constructor, prior_spec, rinit_spec, rmeasure_spec, rprocess_spec, skeleton_spec, transformations, vmeasure_spec

Examples

```
## The familiar Ricker example.
## Suppose that for some reason we wish to pass 'phi'
## via the userdata facility instead of as a parameter.

## C snippet approach:

simulate(times=1:100,t0=0,
  userdata=list(phi=as.double(100)),
  params=c(r=3.8,sigma=0.3,N.0=7),
  rprocess=discrete_time(
    step.fun=Csnippet(r"{
    double e = (sigma > 0.0) ? rnorm(0,sigma) : 0.0;
    N = r*N*exp(-N+e);}"
    ),
    delta.t=1
  ),
  rmeasure=Csnippet(r"{
     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(r"{
    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,
  userdata=list(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,
  seed = 73658676L
)
```

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
`seed`	seed of the random number generator

Details

A stochastic version of the Verhulst-Pearl logistic model. This evolves in continuous time, according to the stochastic differential equation

$$dn_t = r\, n_t \, \left(1 - \frac{n_t}{K}\right)\, dt + \sigma\, n_t \, dW_t.$$

Numerically, we simulate the stochastic dynamics using an Euler approximation.

The measurements are assumed to be log-normally distributed:

$$N_t \sim \text{Lognormal}\left(\log n_t, \tau\right).$$

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**: blowflies, childhood_disease_data, compartmental_models, dacca(), ebola, gompertz(), ou2(), pomp_examples, ricker(), rw2()

Examples

```
# takes too long for R CMD check
 verhulst() -> po
 plot(po)
 plot(simulate(po))
 pfilter(po,Np=1000) -> pf
 logLik(pf)
 spy(po)
```

---

vmeasure *vmeasure workhorse*

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.
...	additional arguments are ignored.
x	an array containing states of the unobserved process. The dimensions of x are nvars x nrep x ntimes, where nvars is the number of state variables, nrep is the number of replicates, and ntimes is the length of times. One can also pass x as a named numeric vector, which is equivalent to the nrep=1, ntimes=1 case.
times	a numeric vector (length ntimes) containing times. These must be in non-decreasing order.
params	a npar x nrep matrix of parameters. Each column is treated as an independent parameter set, in correspondence with the corresponding column of x.

Value

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_spec

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

---

vmeasure_spec *vmeasure specification*

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()`, `accumvars`, `basic_components`, `betabinomial`, `covariates`, `dinit_spec`, `dmeasure_spec`, `dprocess_spec`, `emeasure_spec`, `eulermultinom`, `parameter_trans()`, `pomp-package`, `pomp_constructor`, `prior_spec`, `rinit_spec`, `rmeasure_spec`, `rprocess_spec`, `skeleton_spec`, `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

`rinit` which samples from the initial-state distribution,

`dinit` which evaluates the initial-state density,

`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: `dinit()`, `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 coerced to an array with storage-mode double.
...	additional arguments are passed to pomp. This allows one to set, unset, or modify basic model components within a call to this function.
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.
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, and T. Clapp. A tutorial on particle filters for online nonlinear, non-Gaussian Bayesian tracking. *IEEE Transactions on Signal Processing* **50**, 174–188, 2002. doi:10.1109/78.978374.

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()`

---

wquant *Weighted quantile function*

Description

Estimate weighted quantiles.

Usage

```
wquant(
  x,
  weights = rep(1, length(x)),
  probs = c(`0%` = 0, `25%` = 0.25, `50%` = 0.5, `75%` = 0.75, `100%` = 1)
)
```

Arguments

x	numeric; a vector of data.
weights	numeric; vector of weights.
probs	numeric; desired quantiles.

Details

wquant estimates quantiles of weighted data using the estimator of Harrell & Davis (1982), with improvements recommended by Andrey Akinshin (2023).

Value

wquant returns a vector containing the estimated quantiles. If probs has names, these are inherited.

Author(s)

Aaron A. King

References

F. E. Harrell and C. E. Davis. A new distribution-free quantile estimator. *Biometrika* **69**, 635–640, 1982. doi:10.1093/biomet/69.3.635.

A. Akinshin. Weighted quantile estimators. arXiv:2304.07265, 2023. doi:10.48550/arxiv.2304.07265.

Examples

```
x <- c(1,1,1,2,2,3,3,3,3,4,5,5,6,6,6)
quantile(x)
wquant(x)
wquant(c(1,2,3,4,5,6),weights=c(3,2,4,1,2,3))
wquant(c(1,2,3,4,5),c(1,0,0,1,1))
```

Index

∗ **Bayesian methods**
abc, 7
bsmc2, 20
dprior, 45
pmcmc, 96
prior_spec, 108
rprior, 129
∗ **Kalman filter**
kalman, 65
kalmanFilter, 68
∗ **MCMC methods**
abc, 7
pmcmc, 96
proposals, 117
∗ **approximate Bayesian computation**
abc, 7
∗ **basic model components**
basic_components, 13
∗ **covariates**
covariates, 32
∗ **deterministic methods**
flow, 60
skeleton, 143
skeleton_spec, 144
traj_match, 159
trajectory, 162
∗ **diagnostics**
basic_probes, 14
∗ **distribution**
eulermultinom, 55
∗ **elementary algorithms**
elementary_algorithms, 51
kalman, 65
pfilter, 91
pomp-package, 5
probe, 110
simulate, 140
spect, 147
trajectory, 162
wpfilter, 175
∗ **estimation methods**
abc, 7
bsmc2, 20
estimation_algorithms, 54
mif2, 74
nlf, 79
pmcmc, 96
pomp-package, 5
probe_match, 113
spect_match, 150
∗ **extending the pomp package**
dinit, 40
dmeasure, 42
dprior, 45
dprocess, 46
flow, 60
hitch, 63
partrans, 89
rinit, 124
rmeasure, 126
rprior, 129
rprocess, 130
skeleton, 143
workhorses, 174
∗ **extraction methods**
coef, 25
cond_logLik, 30
covmat, 33
eff_sample_size, 50
filter_mean, 58
filter_traj, 59
forecast, 61
logLik, 69
obs, 84
pred_mean, 106
pred_var, 107
saved_states, 139
spy, 154

states, 155
summary, 156
time, 156
timezero, 157
traces, 158
* **full-information methods**
bsmc2, 20
mif2, 74
pfilter, 91
pmcmc, 96
wpfilter, 175
* **implementation information**
accumvars, 10
basic_components, 13
betabinomial, 16
covariates, 32
Csnippet, 34
dinit_spec, 41
dmeasure_spec, 43
dprocess_spec, 47
emeasure_spec, 53
eulermultinom, 55
parameter_trans, 86
pomp-package, 5
pomp_constructor, 99
prior_spec, 108
rinit_spec, 125
rmeasure_spec, 127
rprocess_spec, 131
skeleton_spec, 144
transformations, 166
userdata, 167
vmeasure_spec, 172
* **interpolation**
bsplines, 23
covariates, 32
lookup, 72
* **low-level interface**
dinit, 40
dmeasure, 42
dprior, 45
dprocess, 46
flow, 60
hitch, 63
partrans, 89
rinit, 124
rmeasure, 126
rprior, 129
rprocess, 130
skeleton, 143
workhorses, 174
* **methods based on maximization**
mif2, 74
nlf, 79
probe_match, 113
spect_match, 150
traj_match, 159
* **models**
blowflies, 17
compartmental_models, 27
dacca, 36
gompertz, 62
ou2, 84
pomp-package, 5
pomp_examples, 104
ricker, 123
rw2, 136
* **multivariate**
pomp-package, 5
* **nonlinear forecasting**
nlf, 79
* **optimize**
sannbox, 137
* **parameter transformations**
transformations, 166
* **particle filter methods**
bsmc2, 20
cond_logLik, 30
eff_sample_size, 50
filter_mean, 58
filter_traj, 59
kalman, 65
mif2, 74
pfilter, 91
pmcmc, 96
pred_mean, 106
pred_var, 107
saved_states, 139
wpfilter, 175
* **pomp datasets**
blowflies, 17
bsflu, 19
childhood_disease_data, 24
dacca, 36
ebola, 48
parus, 90

∗ **pomp examples**
blowflies, 17
childhood_disease_data, 24
compartmental_models, 27
dacca, 36
ebola, 48
gompertz, 62
ou2, 84
pomp_examples, 104
ricker, 123
rw2, 136
verhulst, 170
∗ **pomp workhorses**
dinit, 40
dmeasure, 42
dprior, 45
dprocess, 46
emeasure, 52
flow, 60
partrans, 89
pomp-package, 5
rinit, 124
rmeasure, 126
rprior, 129
rprocess, 130
skeleton, 143
vmeasure, 171
workhorses, 174
∗ **power-spectrum matching**
spect_match, 150
∗ **probability distributions**
betabinomial, 16
eulermultinom, 55
∗ **probe matching**
probe_match, 113
∗ **profile likelihood**
design, 38
mcap, 73
∗ **reproducibility**
reproducibility_tools, 119
∗ **search design**
design, 38
∗ **smooth**
bsplines, 23
∗ **splines**
bsplines, 23
∗ **summary statistic-based methods**
abc, 7
basic_probes, 14
nlf, 79
probe, 110
probe_match, 113
spect, 147
spect_match, 150
∗ **synthetic likelihood**
probe, 110
probe_match, 113
∗ **trajectory matching**
traj_match, 159
∗ **ts**
pomp-package, 5

abc, *5*, 7, *31*, *168*
abc(), *7*, *16*, *23*, *46*, *54*, *79*, *83*, *99*, *109*, *113*, *116*, *119*, *130*, *150*, *154*
abc,abcd_pomp-method *(*abc*)*, 7
abc,ANY-method *(*abc*)*, 7
abc,data.frame-method *(*abc*)*, 7
abc,missing-method *(*abc*)*, 7
abc,pomp-method *(*abc*)*, 7
abc,probed_pomp-method *(*abc*)*, 7
accumvars, *7*, 10, *14*, *17*, *33*, *36*, *42*, *44*, *48*, *54*, *57*, *61*, *88*, *102*, *104*, *109*, *126*, *128*, *135*, *145*, *167*, *169*, *173*
append_data *(*reproducibility_tools*)*, 119
approximate Bayesian computation (ABC), *54*
as.data.frame, *22*, *93*, *177*

bake *(*reproducibility_tools*)*, 119
baseenv, *120*
basic component arguments, *6*
basic model component, *167*
basic model components, *6*, *8*, *21*, *52*, *54*, *63*, *66*, *76*, *80*, *87*, *92*, *97*, *108*, *111*, *114*, *148*, *151*, *175, 176*
basic POMP model components, *6*
basic probes, *8*, *111*, *114*
basic_components, *7*, *11*, 13, *17*, *33*, *36*, *42*, *44*, *48*, *54*, *57*, *88*, *104*, *109*, *126*, *128*, *135*, *145*, *167*, *169*, *173*
basic_probes, *10*, 14, *83*, *113*, *116*, *150*, *154*
betabinomial, *7*, *11*, *14*, 16, *33*, *36*, *42*, *44*, *48*, *54*, *57*, *88*, *104*, *109*, *126*, *128*, *135*, *145*, *167*, *169*, *173*
blowflies, 17, *20*, *25*, *28*, *38*, *50*, *63*, *85*, *90*, *104, 105*, *123*, *136*, *171*

blowflies1, *104*
blowflies1 *(*blowflies*)*, 17
blowflies2, *104*
blowflies2 *(*blowflies*)*, 17
bsflu, 19, *19*, *25*, *38*, *50*, *90*, *105*
bsmc2, *5*, 20, *30*, *70*
bsmc2(), *7*, *10*, *31*, *46*, *51*, *54*, *58*, *60*, *67*, *79*, *83*, *94*, *99*, *106, 107*, *109*, *116*, *130*, *140*, *154*, *177*
bsmc2,ANY-method *(*bsmc2*)*, 20
bsmc2,data.frame-method *(*bsmc2*)*, 20
bsmc2,missing-method *(*bsmc2*)*, 20
bsmc2,pomp-method *(*bsmc2*)*, 20
bspline_basis *(*bsplines*)*, 23
bsplines, 23, *33*, *72*

c *(*concat*)*, 29
C snippets, *121*
cdir, *10*, *22*, *36*, *42*, *44*, *48*, *54*, *67*, *78*, *82*, *88*, *93*, *99*, *104*, *108*, *112*, *115*, *125*, *128*, *135*, *142*, *145*, *149*, *153*, *161*, *173*, *177*
cfile, *10*, *22*, *36*, *42*, *44*, *48*, *54*, *67*, *78*, *82*, *88*, *93*, *99*, *104*, *108*, *112*, *115*, *125*, *128*, *135*, *142*, *145*, *149*, *153*, *161*, *173*, *177*
childhood_disease_data, *19, 20*, 24, *28*, *38*, *50*, *63*, *85*, *90*, *105*, *123*, *136*, *171*
coef, 25, *60*, *77*
coef(), *31*, *34*, *51*, *58*, *60*, *62*, *71*, *84*, *106, 107*, *140*, *155–159*
coef,listie-method *(*coef*)*, 25
coef,objfun-method *(*coef*)*, 25
coef,pomp-method *(*coef*)*, 25
coef<- *(*coef*)*, 25
coef<-,missing-method *(*coef*)*, 25
coef<-,objfun-method *(*coef*)*, 25
coef<-,pomp-method *(*coef*)*, 25
compartmental models, *20*, *25*
compartmental_models, *19*, *25*, 27, *38*, *50*, *63*, *85*, *105*, *123*, *136*, *171*
concat, 29
cond_logLik, 30, *93*, *177*
cond_logLik(), *23*, *26*, *34*, *51*, *58*, *60*, *62*, *67*, *71*, *79*, *84*, *94*, *99*, *106, 107*, *140*, *155–159*, *177*
cond_logLik,ANY-method *(*cond_logLik*)*, 30
cond_logLik,bsmcd_pomp-method *(*cond_logLik*)*, 30
cond_logLik,kalmand_pomp-method *(*cond_logLik*)*, 30
cond_logLik,missing-method *(*cond_logLik*)*, 30
cond_logLik,pfilterd_pomp-method *(*cond_logLik*)*, 30
cond_logLik,pfilterList-method *(*cond_logLik*)*, 30
cond_logLik,wpfilterd_pomp-method *(*cond_logLik*)*, 30
constructor function, *14*
continue, *9*, 31, *77*, *98*
continue,abcd_pomp-method *(*continue*)*, 31
continue,ANY-method *(*continue*)*, 31
continue,mif2d_pomp-method *(*continue*)*, 31
continue,missing-method *(*continue*)*, 31
continue,pmcmcd_pomp-method *(*continue*)*, 31
covariate_table, *32*, *72*, *102*
covariate_table *(*covariates*)*, 32
covariate_table,ANY-method *(*covariates*)*, 32
covariate_table,character-method *(*covariates*)*, 32
covariate_table,missing-method *(*covariates*)*, 32
covariate_table,numeric-method *(*covariates*)*, 32
covariates, *7*, *11*, *14*, *17*, *24*, 32, *36*, *42*, *44*, *48*, *54*, *57*, *72*, *88*, *104*, *109*, *126*, *128*, *135*, *145*, *167*, *169*, *173*
covmat, 33
covmat(), *26*, *31*, *51*, *58*, *60*, *62*, *71*, *84*, *106, 107*, *140*, *155–159*
covmat,abcd_pomp-method *(*covmat*)*, 33
covmat,abcList-method *(*covmat*)*, 33
covmat,ANY-method *(*covmat*)*, 33
covmat,missing-method *(*covmat*)*, 33
covmat,pmcmcd_pomp-method *(*covmat*)*, 33
covmat,pmcmcList-method *(*covmat*)*, 33
covmat,probed_pomp-method *(*covmat*)*, 33
Csnippet, 34, *44*, *47*, *53*, *65*, *82*, *103*, *116*, *128*, *132*, *153*, *161*, *173*
Csnippet(), *7*, *11*, *14*, *17*, *33*, *42*, *44*, *48*, *54*, *57*, *88*, *104*, *109*, *126*, *128*, *135*, *145*, *167*, *169*, *173*

dacca, 36, *105*

dacca(), *19, 20*, *25*, *28*, *50*, *63*, *85*, *90*, *105*, *123*, *136*, *171*
dbetabinom *(*betabinomial*)*, 16
design, 38
deSolve, *61*, *164*
deulermultinom *(*eulermultinom*)*, 55
dinit, *14*, 40, *42*, *174*
dinit specification, *101*
dinit(), *7*, *43*, *46, 47*, *53*, *61*, *90*, *124*, *127*, *130, 131*, *143*, *172*, *175*
dinit,ANY-method *(*dinit*)*, 40
dinit,missing-method *(*dinit*)*, 40
dinit,pomp-method *(*dinit*)*, 40
dinit_spec, *7*, *11*, *14*, *17*, *33*, *36*, 41, *41*, *44*, *48*, *54*, *57*, *88*, *104*, *109*, *126*, *128*, *135*, *145*, *167*, *169*, *173*
discrete_time *(*rprocess_spec*)*, 131
dmeasure, *14*, 42, *44*, *174*
dmeasure specification, *22*, *77*, *92*, *98*, *101*, *160*, *176*
dmeasure(), *7*, *41*, *46, 47*, *53*, *61*, *90*, *124*, *127*, *130, 131*, *143*, *172*, *175*
dmeasure,ANY-method *(*dmeasure*)*, 42
dmeasure,missing-method *(*dmeasure*)*, 42
dmeasure,pomp-method *(*dmeasure*)*, 42
dmeasure_spec, *7*, *11*, *14*, *17*, *33*, *36*, *42*, 43, *43*, *48*, *54*, *57*, *88*, *104*, *109*, *126*, *128*, *135*, *145*, *167*, *169*, *173*
dprior, *14*, 45, *109*, *174*
dprior(), *7*, *10*, *23*, *41*, *43*, *47*, *53*, *61*, *90*, *99*, *109*, *124*, *127*, *130, 131*, *143*, *172*, *175*
dprior,ANY-method *(*dprior*)*, 45
dprior,missing-method *(*dprior*)*, 45
dprior,pomp-method *(*dprior*)*, 45
dprocess, *14*, 46, *48*, *174*
dprocess specification, *101*
dprocess(), *7*, *41*, *43*, *46*, *53*, *61*, *90*, *124*, *127*, *130, 131*, *143*, *172*, *175*
dprocess,ANY-method *(*dprocess*)*, 46
dprocess,missing-method *(*dprocess*)*, 46
dprocess,pomp-method *(*dprocess*)*, 46
dprocess_spec, *7*, *11*, *14*, *17*, *33*, *36*, *42*, *44*, 47, *47*, *54*, *57*, *88*, *104*, *109*, *126*, *128*, *135*, *145*, *167*, *169*, *173*

eakf, *69, 70*
eakf *(*kalman*)*, 65
eakf,ANY-method *(*kalman*)*, 65
eakf,data.frame-method *(*kalman*)*, 65
eakf,missing-method *(*kalman*)*, 65
eakf,pomp-method *(*kalman*)*, 65
ebola, *19, 20*, *25*, *28*, *38*, 48, *63*, *85*, *90*, *105*, *123*, *136*, *171*
ebolaModel, *105*
ebolaModel *(*ebola*)*, 48
ebolaWA2014 *(*ebola*)*, 48
eeulermultinom *(*eulermultinom*)*, 55
eff_sample_size, 50, *77*, *93*, *177*
eff_sample_size(), *23*, *26*, *31*, *34*, *58*, *60*, *62*, *67*, *71*, *79*, *84*, *94*, *99*, *106, 107*, *140*, *155–159*, *177*
eff_sample_size,ANY-method *(*eff_sample_size*)*, 50
eff_sample_size,bsmcd_pomp-method *(*eff_sample_size*)*, 50
eff_sample_size,missing-method *(*eff_sample_size*)*, 50
eff_sample_size,pfilterd_pomp-method *(*eff_sample_size*)*, 50
eff_sample_size,pfilterList-method *(*eff_sample_size*)*, 50
eff_sample_size,wpfilterd_pomp-method *(*eff_sample_size*)*, 50
Elementary algorithms, *6*
elementary algorithms, *14*, *54*, *175*
elementary_algorithms, *7*, 51, *67*, *94*, *113*, *142*, *150*, *164*, *177*
emeasure, *14*, 52, *54*, *174*
emeasure specification, *66*, *101*
emeasure(), *7*, *41*, *43*, *46, 47*, *61*, *90*, *124*, *127*, *130, 131*, *143*, *172*, *175*
emeasure,ANY-method *(*emeasure*)*, 52
emeasure,missing-method *(*emeasure*)*, 52
emeasure,pomp-method *(*emeasure*)*, 52
emeasure_spec, *7*, *11*, *14*, *17*, *33*, *36*, *42*, *44*, *48*, 53, *53*, *57*, *88*, *104*, *109*, *126*, *128*, *135*, *145*, *167*, *169*, *173*
enkf, *69, 70*
enkf *(*kalman*)*, 65
enkf,ANY-method *(*kalman*)*, 65
enkf,data.frame-method *(*kalman*)*, 65
enkf,kalmand_pomp-method *(*kalman*)*, 65
enkf,missing-method *(*kalman*)*, 65
enkf,pomp-method *(*kalman*)*, 65
Ensemble and ensemble-adjusted Kalman filters, *54*

environment, *120*
estimation algorithms, *6*, *14*, *52*, *175*
estimation_algorithms, *7*, *10*, *23*, 54, *79*, *83*, *99*, *116*, *154*
euler *(*rprocess_spec*)*, 131
eulermultinom, *7*, *11*, *14*, *17*, *33*, *36*, *42*, *44*, *48*, *54*, 55, *88*, *104*, *109*, *126*, *128*, *135*, *145*, *167*, *169*, *173*
ewcitmeas, *105*
ewcitmeas *(*childhood_disease_data*)*, 24
ewmeas, *105*
ewmeas *(*childhood_disease_data*)*, 24
expit *(*transformations*)*, 166

facilitating reproducible computations, *6*
filter_mean, 58, *93*
filter_mean(), *23*, *26*, *31*, *34*, *51*, *60*, *62*, *67*, *71*, *79*, *84*, *94*, *99*, *106, 107*, *140*, *155–159*, *177*
filter_mean,ANY-method *(*filter_mean*)*, 58
filter_mean,kalmand_pomp-method *(*filter_mean*)*, 58
filter_mean,missing-method *(*filter_mean*)*, 58
filter_mean,pfilterd_pomp-method *(*filter_mean*)*, 58
filter_traj, 59, *93*, *98*
filter_traj(), *23*, *26*, *31*, *34*, *51*, *58*, *62*, *67*, *71*, *79*, *84*, *94*, *99*, *106, 107*, *140*, *155–159*, *177*
filter_traj,ANY-method *(*filter_traj*)*, 59
filter_traj,listie-method *(*filter_traj*)*, 59
filter_traj,missing-method *(*filter_traj*)*, 59
filter_traj,pfilterd_pomp-method *(*filter_traj*)*, 59
filter_traj,pmcmcd_pomp-method *(*filter_traj*)*, 59
flow, *10*, 60, *143*, *174*
flow(), *7*, *41*, *43*, *46, 47*, *53*, *90*, *124*, *127*, *130, 131*, *143–145*, *162*, *164*, *172*, *175*
flow,ANY-method *(*flow*)*, 60
flow,missing-method *(*flow*)*, 60
flow,pomp-method *(*flow*)*, 60
forecast, 61
forecast(), *26*, *31*, *34*, *51*, *58*, *60*, *71*, *84*, *106, 107*, *140*, *155–159*
forecast,ANY-method *(*forecast*)*, 61
forecast,kalmand_pomp-method *(*forecast*)*, 61
forecast,missing-method *(*forecast*)*, 61
forecast,pfilterd_pomp-method *(*forecast*)*, 61
freeze *(*reproducibility_tools*)*, 119

General rules for writing C snippets can be found here, *42*, *108*, *125*, *144*
gillespie *(*rprocess_spec*)*, 131
gillespie_hl *(*rprocess_spec*)*, 131
gompertz, 62, *105*
gompertz(), *19*, *25*, *28*, *38*, *50*, *85*, *105*, *123*, *136*, *171*

here for a definition of this term, *167*
hitch, 63

inv_log_barycentric *(*transformations*)*, 166
iterated filtering (IF2), *54*

kalman, *5*, *7*, *23*, *31*, *51, 52*, *58*, *60*, 65, *79*, *94*, *99*, *106, 107*, *113*, *140*, *142*, *150*, *164*, *177*
kalmanFilter, *67*, 68
kernel, *15*

Liu-West Bayesian sequential Monte Carlo, *54*
load, *121*
loess, *73*
log_barycentric *(*transformations*)*, 166
logit *(*transformations*)*, 166
logLik, 69, *77*, *93*, *177*
logLik(), *26*, *31*, *34*, *51*, *58*, *60*, *62*, *84*, *106, 107*, *140*, *155–159*
logLik,ANY-method *(*logLik*)*, 69
logLik,bsmcd_pomp-method *(*logLik*)*, 69
logLik,kalmand_pomp-method *(*logLik*)*, 69
logLik,listie-method *(*logLik*)*, 69
logLik,missing-method *(*logLik*)*, 69
logLik,nlf_objfun-method *(*logLik*)*, 69
logLik,objfun-method *(*logLik*)*, 69
logLik,pfilterd_pomp-method *(*logLik*)*, 69

logLik,pmcmcd_pomp-method (logLik), 69
logLik,probed_pomp-method (logLik), 69
logLik,spect_match_objfun-method (logLik), 69
logLik,wpfilterd_pomp-method (logLik), 69
logmeanexp, 71
LondonYorke, *105*
LondonYorke (childhood_disease_data), 24
lookup, *32*, 72
lookup(), *24*, *33*

map, *102*, *160*, *164*
map (skeleton_spec), 144
mcap, 73
mcmc, *9*, *98*
MCMC proposals, *8*, *34*, *97*
mean, *15*
melt, 74
melt,ANY-method (melt), 74
melt,array-method (melt), 74
melt,list-method (melt), 74
melt,missing-method (melt), 74
mif2, *5*, *31*, 74, *135*, *159*, *168*
mif2(), *7*, *10*, *23*, *31*, *51*, *54*, *58*, *60*, *67*, *83*, *94*, *99*, *106*, *107*, *116*, *140*, *154*, *162*, *177*
mif2,ANY-method (mif2), 74
mif2,data.frame-method (mif2), 74
mif2,mif2d_pomp-method (mif2), 74
mif2,missing-method (mif2), 74
mif2,pfilterd_pomp-method (mif2), 74
mif2,pomp-method (mif2), 74
mvn_diag_rw (proposals), 117
mvn_rw (proposals), 117
mvn_rw_adaptive (proposals), 117

nlf, *5*, *7*, *10*, *16*, *23*, *54*, 79, *79*, *99*, *113*, *116*, *150*, *154*, *162*
nlf_objfun, *71*
nlf_objfun (nlf), 79
nlf_objfun,ANY-method (nlf), 79
nlf_objfun,data.frame-method (nlf), 79
nlf_objfun,missing-method (nlf), 79
nlf_objfun,nlf_objfun-method (nlf), 79
nlf_objfun,pomp-method (nlf), 79
nloptr, *83*, *116*, *154*, *162*
nonlinear forecasting, *54*, *138*

obs, *16*, 84
obs(), *26*, *31*, *34*, *51*, *58*, *60*, *62*, *71*, *106*, *107*, *140*, *155–159*
obs,ANY-method (obs), 84
obs,listie-method (obs), 84
obs,missing-method (obs), 84
obs,pomp-method (obs), 84
ode, *60*, *160*, *164*
onestep (rprocess_spec), 131
optim, *82*, *83*, *115*, *116*, *138*, *153*, *154*, *161*, *162*
ou2, 84, *105*
ou2(), *19*, *25*, *28*, *38*, *50*, *63*, *105*, *123*, *136*, *171*

par, *96*
parameter_trans, *22*, *77*, 86, *90*, *102*, *115*, *152*, *160*, *161*, *166*
parameter_trans(), *7*, *11*, *14*, *17*, *33*, *36*, *42*, *44*, *48*, *54*, *57*, *104*, *109*, *126*, *128*, *135*, *145*, *167*, *169*, *173*
parameter_trans,ANY,ANY-method (parameter_trans), 86
parameter_trans,ANY,missing-method (parameter_trans), 86
parameter_trans,character,character-method (parameter_trans), 86
parameter_trans,Csnippet,Csnippet-method (parameter_trans), 86
parameter_trans,function,function-method (parameter_trans), 86
parameter_trans,missing,ANY-method (parameter_trans), 86
parameter_trans,missing,missing-method (parameter_trans), 86
parameter_trans,NULL,NULL-method (parameter_trans), 86
parameter_trans,pomp_fun,pomp_fun-method (parameter_trans), 86
parmat, 88
parmat,ANY-method (parmat), 88
parmat,array-method (parmat), 88
parmat,data.frame-method (parmat), 88
parmat,missing-method (parmat), 88
parmat,numeric-method (parmat), 88
particle Markov chain Monte Carlo (PMCMC), *54*
partrans, *14*, *88*, 89, *174*

partrans(), *7*, *41*, *43*, *46, 47*, *53*, *61*, *124*, *127*, *130, 131*, *143*, *172*, *175*
partrans,ANY-method *(*partrans*)*, 89
partrans,missing-method *(*partrans*)*, 89
partrans,objfun-method *(*partrans*)*, 89
partrans,pomp-method *(*partrans*)*, 89
parus, *19, 20*, *25*, *38*, *50*, 90, *105*
paste, *23*
periodic_bspline_basis *(*bsplines*)*, 23
pfilter, *5*, *51*, *59*, *77*, 91, *139*, *168*
pfilter(), *7*, *23*, *31*, *51, 52*, *58*, *60*, *67*, *79*, *99*, *106, 107*, *113*, *140*, *142*, *150*, *164*, *177*
pfilter,ANY-method *(*pfilter*)*, 91
pfilter,data.frame-method *(*pfilter*)*, 91
pfilter,missing-method *(*pfilter*)*, 91
pfilter,objfun-method *(*pfilter*)*, 91
pfilter,pfilterd_pomp-method *(*pfilter*)*, 91
pfilter,pomp-method *(*pfilter*)*, 91
pfilterd_pomp, *77*
plot, *22*, *93*, 94, *177*
plot,Abc-method *(*plot*)*, 94
plot,bsmcd_pomp-method *(*plot*)*, 94
plot,Mif2-method *(*plot*)*, 94
plot,missing-method *(*plot*)*, 94
plot,Pmcmc-method *(*plot*)*, 94
plot,pomp-method *(*plot*)*, 94
plot,pomp_plottable-method *(*plot*)*, 94
plot,probe_match_objfun-method *(*plot*)*, 94
plot,probed_pomp-method *(*plot*)*, 94
plot,spect_match_objfun-method *(*plot*)*, 94
plot,spectd_pomp-method *(*plot*)*, 94
pmcmc, *5*, *31*, *59*, *70*, 96, *158*
pmcmc(), *7*, *10*, *23*, *31*, *46*, *51*, *54*, *58*, *60*, *67*, *79*, *83*, *94*, *106, 107*, *109*, *116*, *119*, *130*, *140*, *154*, *177*
pmcmc,ANY-method *(*pmcmc*)*, 96
pmcmc,data.frame-method *(*pmcmc*)*, 96
pmcmc,missing-method *(*pmcmc*)*, 96
pmcmc,pfilterd_pomp-method *(*pmcmc*)*, 96
pmcmc,pmcmcd_pomp-method *(*pmcmc*)*, 96
pmcmc,pomp-method *(*pmcmc*)*, 96
pomp, *6*, *8*, *21*, *65, 66*, *76*, *80*, *92*, *97*, *111*, *114*, *141*, *148*, *151*, *163*, *176*
pomp *(*pomp_constructor*)*, 99
pomp,package *(*pomp-package*)*, 5
pomp-package, 5
pomp_constructor, *7*, *11*, *14*, *17*, *33*, *36*, *42*, *44*, *48*, *54*, *57*, *88*, 99, *109*, *126*, *128*, *135*, *145*, *167*, *169*, *173*
pomp_examples, *19*, *25*, *28*, *38*, *50*, *63*, *85*, 104, *123*, *136*, *171*
pompLoad, *83*, *116*, *153*, *161, 162*
pompUnload, *83*, *116*, *153*, *161*
power-spectrum matching, *54*
pred_mean, *93*, 106
pred_mean(), *23*, *26*, *31*, *34*, *51*, *58*, *60*, *62*, *67*, *71*, *79*, *84*, *94*, *99*, *107*, *140*, *155–159*, *177*
pred_mean,ANY-method *(*pred_mean*)*, 106
pred_mean,kalmand_pomp-method *(*pred_mean*)*, 106
pred_mean,missing-method *(*pred_mean*)*, 106
pred_mean,pfilterd_pomp-method *(*pred_mean*)*, 106
pred_var, *93*, 107
pred_var(), *23*, *26*, *31*, *34*, *51*, *58*, *60*, *62*, *67*, *71*, *79*, *84*, *94*, *99*, *106*, *140*, *155–159*, *177*
pred_var,ANY-method *(*pred_var*)*, 107
pred_var,missing-method *(*pred_var*)*, 107
pred_var,pfilterd_pomp-method *(*pred_var*)*, 107
prior specification, *9*, *21*, *98*, *102*
prior_spec, *7*, *10, 11*, *14*, *17*, *23*, *33*, *36*, *42*, *44*, *46*, *48*, *54*, *57*, *88*, *99*, *104*, 108, *126*, *128*, *130*, *135*, *145*, *167*, *169*, *173*
priors *(*prior_spec*)*, 108
probe, *5*, *16*, *51*, *70*, 110
probe matching, *138*
probe(), *7*, *10*, *16*, *52*, *67*, *83*, *94*, *116*, *142*, *150*, *154*, *164*, *177*
probe,ANY-method *(*probe*)*, 110
probe,data.frame-method *(*probe*)*, 110
probe,missing-method *(*probe*)*, 110
probe,objfun-method *(*probe*)*, 110
probe,pomp-method *(*probe*)*, 110
probe,probe_match_objfun-method *(*probe*)*, 110
probe,probed_pomp-method *(*probe*)*, 110
probe-matching via synthetic

likelihood, *54*
probe_acf *(*basic_probes*)*, 14
probe_ccf *(*basic_probes*)*, 14
probe_marginal *(*basic_probes*)*, 14
probe_match, *5*, *7*, *10*, *16*, *23*, *54*, *79*, *83*, *99*, 113, *113*, *150*, *154*, *162*
probe_mean *(*basic_probes*)*, 14
probe_median *(*basic_probes*)*, 14
probe_nlar *(*basic_probes*)*, 14
probe_objfun, *16*
probe_objfun *(*probe_match*)*, 113
probe_objfun,ANY-method *(*probe_match*)*, 113
probe_objfun,data.frame-method *(*probe_match*)*, 113
probe_objfun,missing-method *(*probe_match*)*, 113
probe_objfun,pomp-method *(*probe_match*)*, 113
probe_objfun,probe_match_objfun-method *(*probe_match*)*, 113
probe_objfun,probed_pomp-method *(*probe_match*)*, 113
probe_period *(*basic_probes*)*, 14
probe_quantile *(*basic_probes*)*, 14
probe_sd *(*basic_probes*)*, 14
probe_var *(*basic_probes*)*, 14
profile_design *(*design*)*, 38
proposals, *10*, *99*, 117

quantile, *15*

rbetabinom *(*betabinomial*)*, 16
readRDS, *121*
repair_lookup_table *(*covariates*)*, 32
reproducibility_tools, 119
reulermultinom *(*eulermultinom*)*, 55
rgammawn *(*eulermultinom*)*, 55
ricker, *105*, 123
ricker(), *19*, *25*, *28*, *38*, *50*, *63*, *85*, *105*, *136*, *171*
rinit, *14*, 124, *126*, *174*
rinit specification, *9*, *21*, *66*, *76*, *81*, *92*, *98*, *101*, *111*, *115*, *141*, *148*, *152*, *160*, *164*, *176*
rinit(), *7*, *41*, *43*, *46*, *47*, *53*, *61*, *90*, *127*, *130*, *131*, *143*, *172*, *175*
rinit,ANY-method *(*rinit*)*, 124
rinit,missing-method *(*rinit*)*, 124
rinit,pomp-method *(*rinit*)*, 124
rinit_spec, *7*, *11*, *14*, *17*, *33*, *36*, *42*, *44*, *48*, *54*, *57*, *88*, *104*, *109*, *124*, 125, *128*, *135*, *145*, *167*, *169*, *173*
rmeasure, *14*, 126, *128*, *174*
rmeasure specification, *9*, *81*, *101*, *111*, *115*, *141*, *149*, *152*
rmeasure(), *7*, *41*, *43*, *46*, *47*, *53*, *61*, *90*, *124*, *130*, *131*, *143*, *172*, *175*
rmeasure,ANY-method *(*rmeasure*)*, 126
rmeasure,missing-method *(*rmeasure*)*, 126
rmeasure,pomp-method *(*rmeasure*)*, 126
rmeasure_spec, *7*, *11*, *14*, *17*, *33*, *36*, *42*, *44*, *48*, *54*, *57*, *88*, *104*, *109*, *126*, 127, *127*, *135*, *145*, *167*, *169*, *173*
rprior, *14*, *109*, 129, *174*
rprior(), *7*, *10*, *23*, *41*, *43*, *46*, *47*, *53*, *61*, *90*, *99*, *109*, *124*, *127*, *131*, *143*, *172*, *175*
rprior,ANY-method *(*rprior*)*, 129
rprior,missing-method *(*rprior*)*, 129
rprior,pomp-method *(*rprior*)*, 129
rprocess, *10*, *14*, 130, *135*, *174*
rprocess plugins, *9*, *21*, *66*, *76*, *81*, *92*, *98*, *101*, *111*, *115*, *141*, *148*, *152*, *176*
rprocess specification for the documentation on these plugins, *9*, *21*, *66*, *76*, *81*, *92*, *98*, *101*, *111*, *115*, *141*, *148*, *152*, *176*
rprocess(), *7*, *41*, *43*, *46*, *47*, *53*, *61*, *90*, *124*, *127*, *130*, *143*, *172*, *175*
rprocess,ANY-method *(*rprocess*)*, 130
rprocess,missing-method *(*rprocess*)*, 130
rprocess,pomp-method *(*rprocess*)*, 130
rprocess_spec, *7*, *11*, *14*, *17*, *33*, *36*, *42*, *44*, *48*, *54*, *57*, *88*, *104*, *109*, *126*, *128*, 131, *131*, *145*, *167*, *169*, *173*
runif_design, *39*
runif_design *(*design*)*, 38
rw2, *105*, 136
rw2(), *19*, *25*, *28*, *38*, *50*, *63*, *85*, *105*, *123*, *171*
rw_sd, *76*, 135

sannbox, 137
saved_states, *93*, 139
saved_states(), *23*, *26*, *31*, *34*, *51*, *58*, *60*, *62*, *67*, *71*, *79*, *84*, *94*, *99*, *106*, *107*, *155–159*, *177*
saved_states,ANY-method *(*saved_states*)*, 139

saved_states,missing-method (saved_states), 139
saved_states,pfilterd_pomp-method (saved_states), 139
saved_states,pfilterList-method (saved_states), 139
set.seed, *120*
several pre-built POMP models, *6*
simulate, *5*, *51*, *112*, 140, *148*, *168*
simulate(), *7*, *52*, *67*, *94*, *113*, *150*, *164*, *177*
simulate,data.frame-method (simulate), 140
simulate,missing-method (simulate), 140
simulate,objfun-method (simulate), 140
simulate,pomp-method (simulate), 140
sir, *10*, *11*, *105*
sir (compartmental_models), 27
sir2, *105*
sir2 (compartmental_models), 27
SIR_models (compartmental_models), 27
skeleton, *14*, 143, *145*, *174*
skeleton specification, *102*, *160*, *164*
skeleton(), *7*, *41*, *43*, *46*, *47*, *53*, *61*, *90*, *124*, *127*, *130*, *131*, *145*, *162*, *164*, *172*, *175*
skeleton,ANY-method (skeleton), 143
skeleton,missing-method (skeleton), 143
skeleton,pomp-method (skeleton), 143
skeleton_spec, *7*, *11*, *14*, *17*, *33*, *36*, *42*, *44*, *48*, *54*, *57*, *61*, *88*, *104*, *109*, *126*, *128*, *135*, *143*, 144, *144*, *162*, *164*, *167*, *169*, *173*
slice_design (design), 38
sobol_design, *39*
sobol_design (design), 38
spect, *51*, 147, *154*
spect(), *7*, *10*, *16*, *52*, *67*, *83*, *94*, *113*, *116*, *142*, *154*, *164*, *177*
spect,ANY-method (spect), 147
spect,data.frame-method (spect), 147
spect,missing-method (spect), 147
spect,objfun-method (spect), 147
spect,pomp-method (spect), 147
spect,spect_match_objfun-method (spect), 147
spect,spectd_pomp-method (spect), 147
spect_match, *5*, *7*, *10*, *16*, *23*, *54*, *79*, *83*, *99*, *113*, *116*, 150, *150*, *162*
spect_objfun, *71*
spect_objfun (spect_match), 150
spect_objfun,ANY-method (spect_match), 150
spect_objfun,data.frame-method (spect_match), 150
spect_objfun,missing-method (spect_match), 150
spect_objfun,pomp-method (spect_match), 150
spect_objfun,spect_match_objfun-method (spect_match), 150
spect_objfun,spectd_pomp-method (spect_match), 150
spectrum matching, *138*
sprintf, *23*
spy, *65*, 154
spy(), *26*, *31*, *34*, *51*, *58*, *60*, *62*, *71*, *84*, *106*, *107*, *140*, *155–159*
spy,ANY-method (spy), 154
spy,missing-method (spy), 154
spy,pomp-method (spy), 154
states, 155, *164*
states(), *26*, *31*, *34*, *51*, *58*, *60*, *62*, *71*, *84*, *106*, *107*, *140*, *155–159*
states,ANY-method (states), 155
states,listie-method (states), 155
states,missing-method (states), 155
states,pomp-method (states), 155
stew (reproducibility_tools), 119
subplex, *83*, *116*, *154*, *162*
summary, 156
summary(), *26*, *31*, *34*, *51*, *58*, *60*, *62*, *71*, *84*, *106*, *107*, *140*, *155*, *157–159*
summary,objfun-method (summary), 156
summary,probed_pomp-method (summary), 156
summary,spectd_pomp-method (summary), 156
sys.call, *120*
system.time, *121*

time, *60*, 156
time(), *26*, *31*, *34*, *51*, *58*, *60*, *62*, *71*, *84*, *106*, *107*, *140*, *155*, *156*, *158*, *159*
time,listie-method (time), 156
time,missing-method (time), 156
time,pomp-method (time), 156
time<- (time), 156

time<-,pomp-method (time), 156
timezero, 60, 157
timezero(), 26, 31, 34, 51, 58, 60, 62, 71, 84, 106, 107, 140, 155–157, 159
timezero,ANY-method (timezero), 157
timezero,missing-method (timezero), 157
timezero,pomp-method (timezero), 157
timezero<- (timezero), 157
timezero<-,ANY-method (timezero), 157
timezero<-,missing-method (timezero), 157
timezero<-,pomp-method (timezero), 157
traces, 9, 98, 158
traces(), 26, 31, 34, 51, 58, 60, 62, 71, 84, 106, 107, 140, 155–158
traces,abcd_pomp-method (traces), 158
traces,abcList-method (traces), 158
traces,ANY-method (traces), 158
traces,mif2d_pomp-method (traces), 158
traces,mif2List-method (traces), 158
traces,missing-method (traces), 158
traces,pmcmcd_pomp-method (traces), 158
traces,pmcmcList-method (traces), 158
traj_match, 5, 61, 79, 83, 116, 144, 145, 154, 159, 164
traj_objfun, 144
traj_objfun (traj_match), 159
traj_objfun,ANY-method (traj_match), 159
traj_objfun,data.frame-method (traj_match), 159
traj_objfun,missing-method (traj_match), 159
traj_objfun,pomp-method (traj_match), 159
traj_objfun,traj_match_objfun-method (traj_match), 159
trajectory, 51, 143, 144, 162
trajectory matching, 138
trajectory(), 7, 52, 61, 67, 94, 113, 142, 144, 145, 150, 162, 177
trajectory,ANY-method (trajectory), 162
trajectory,data.frame-method (trajectory), 162
trajectory,missing-method (trajectory), 162
trajectory,pomp-method (trajectory), 162
trajectory,traj_match_objfun-method (trajectory), 162
transformations, 7, 11, 14, 17, 33, 36, 42, 44, 48, 54, 57, 88, 104, 109, 126, 128, 135, 145, 166, 169, 173
userdata, 7, 11, 14, 17, 33, 36, 42, 44, 48, 54, 57, 88, 103, 104, 109, 126, 128, 135, 145, 167, 167, 173
userdata facility, 44
vectorfield, 102, 160, 164
vectorfield (skeleton_spec), 144
verhulst, 105, 170
verhulst(), 19, 25, 28, 38, 50, 63, 85, 105, 123, 136
vmeasure, 14, 171, 173, 174
vmeasure specification, 67, 101
vmeasure(), 7, 41, 43, 46, 47, 53, 61, 90, 124, 127, 130, 131, 143, 175
vmeasure,ANY-method (vmeasure), 171
vmeasure,missing-method (vmeasure), 171
vmeasure,pomp-method (vmeasure), 171
vmeasure_spec, 7, 11, 14, 17, 33, 36, 42, 44, 48, 54, 57, 88, 104, 109, 126, 128, 135, 145, 167, 169, 172, 172
window, 174
window,pomp-method (window), 174
workhorse functions, 6, 14, 52, 54
workhorses, 7, 41, 43, 46, 47, 53, 61, 63, 90, 124, 127, 130, 131, 143, 172, 174
wpfilter, 5, 51, 175
wpfilter(), 7, 23, 31, 51, 52, 58, 60, 67, 79, 94, 99, 106, 107, 113, 140, 142, 150, 164
wpfilter,ANY-method (wpfilter), 175
wpfilter,data.frame-method (wpfilter), 175
wpfilter,missing-method (wpfilter), 175
wpfilter,pomp-method (wpfilter), 175
wpfilter,wpfilterd_pomp-method (wpfilter), 175
wquant, 177