

# Package ‘pomp’

September 22, 2009

**Type** Package

**Title** Statistical inference for partially observed Markov processes

**Version** 0.25-4

**Date** 2009-09-21

**Author** Aaron A. King, Edward L. Ionides, Carles Martinez Breto, Steve Ellner, Bruce Kendall

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

**Description** Inference methods for partially-observed Markov processes

**Depends** R(>= 2.8.1), stats, methods, graphics, deSolve, subplex, mvtnorm

**License** GPL (>= 2)

**LazyLoad** true

**Repository** CRAN

**Repository/R-Forge/Project** pomp

**Repository/R-Forge/Revision** 163

**Date/Publication** 2009-09-22 16:48:38

## R topics documented:

pomp-package . . . . .	2
B-splines . . . . .	3
dmeasure-pomp . . . . .	4
dprocess-pomp . . . . .	5
euler . . . . .	6
Euler-multinomial models . . . . .	8
euler.sir . . . . .	10
init.state-pomp . . . . .	10
mif . . . . .	11
mif-class . . . . .	13

mif-methods . . . . .	15
nlf . . . . .	17
ou2 . . . . .	19
particles-mif . . . . .	19
pfilter . . . . .	20
pomp . . . . .	22
pomp-class . . . . .	28
pomp-methods . . . . .	29
rmeasure-pomp . . . . .	32
rprocess-pomp . . . . .	33
rw2 . . . . .	34
simulate-pomp . . . . .	35
skeleton-pomp . . . . .	36
slice.design . . . . .	37
sobol . . . . .	38
traj.match . . . . .	39
trajectory-pomp . . . . .	40
verhulst . . . . .	41
<b>Index</b>	<b>42</b>

---

pomp-package

*Partially-observed Markov processes*

---

## Description

The `pomp` package provides facilities for inference on time series data using partially-observed Markov processes (AKA state-space models or nonlinear stochastic dynamical systems). The user encodes a model as a `pomp` object by providing functions specifying some or all of the model's process and measurement components. The package's algorithms for fitting models to data, simulating, etc. then call these functions. At the moment, algorithms are provided for particle filtering (AKA sequential Monte Carlo or sequential importance sampling, see `pfilter`), the likelihood maximization by iterated filtering (MIF) method of Ionides, Breto, and King (PNAS, 103:18438-18443, 2006, see `mif`), and the nonlinear forecasting algorithm of Kendall, Ellner, et al. (Ecol. Monog. 75:259-276, 2005, see `nlf`). Future support for a variety of other algorithms is envisioned. A working group of the National Center for Ecological Analysis and Synthesis (NCEAS), "Inference for Mechanistic Models", is currently implementing and testing additional methods for this package.

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

## Classes

The basic class, `pomp`, is provided to encode a partially-observed Markov process together with a uni- or multi-variate data set.

**Vignettes**

The vignette ‘intro\_to\_pomp’ illustrates the facilities of the package using familiar stochastic processes. Run `vignette("intro_to_pomp")` or look at the HTML documentation to view the vignette.

**Author(s)**

Aaron A. King (kingaa at umich dot edu)

**See Also**

[pomp](#), [simulate-pomp](#), [mif](#), [nlf](#)

---

B-splines

*B-spline bases*

---

**Description**

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

**Usage**

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

**Arguments**

<code>x</code>	Vector at which the spline functions are to be evaluated.
<code>nbasis</code>	The number of basis functions to return.
<code>degree</code>	Degree of requested B-splines.
<code>period</code>	The period of the requested periodic B-splines.

**Value**

<code>bspline.basis</code>	Returns a matrix with <code>length(x)</code> rows and <code>nbasis</code> columns. Each column contains the values one of the spline basis functions.
<code>periodic.bspline.basis</code>	Returns a matrix with <code>length(x)</code> rows and <code>nbasis</code> columns. The basis functions returned are periodic with period <code>period</code> .

**Author(s)**

Aaron A. King (kingaa at umich dot edu)

**Examples**

```
x <- seq(0,2,by=0.01)
y <- bspline.basis(x,degree=3,nbasis=9)
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)
matplot(x,y,type='l')
```

---

dmeasure-pomp	<i>Evaluate the probability density of observations given underlying states in a partially-observed Markov process</i>
---------------	--

---

**Description**

The method `dmeasure` evaluates the probability density of a set of measurements given the state of the system. This function is part of the low-level interface to `pomp` objects. This help page does not give instructions on the implementation of models: see [pomp](#) for instructions.

**Usage**

```
dmeasure(object, y, x, times, params, log = FALSE, ...)
## S4 method for signature 'pomp':
dmeasure(object, y, x, times, params, log = FALSE, ...)
```

**Arguments**

<code>object</code>	an object of class <code>pomp</code> .
<code>y</code>	a rank-2 array containing observations. The dimensions of <code>y</code> are <code>nobs</code> x <code>ntimes</code> , where <code>nobs</code> is the number of observables and <code>ntimes</code> is the length of <code>times</code> .
<code>x</code>	a rank-3 array containing the states of the unobserved process. The dimensions of <code>x</code> are <code>nvars</code> x <code>nreps</code> x <code>ntimes</code> , where <code>nvars</code> is the number of state variables, <code>nreps</code> is the number of replicates, and <code>ntimes</code> is the length of <code>times</code> .
<code>times</code>	a numeric vector containing the times at which the observations were made.
<code>params</code>	a rank-2 array of parameters with columns corresponding to the columns of <code>x</code> . Note that the <code>x</code> and <code>params</code> must agree in the number of their columns.
<code>log</code>	if TRUE, log probabilities are returned.
<code>...</code>	at present, these are ignored.

**Details**

This function is essentially a wrapper around the user-supplied `dmeasure` slot of the `pomp` object. For specifications on writing such a function, see [pomp](#).

**Value**

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

**Author(s)**

Aaron A. King (kingaa at umich dot edu)

**See Also**

[pomp-class](#), [pomp](#)

---

<code>dprocess-pomp</code>	<i>Evaluate the probability density of state transitions in a Markov process</i>
----------------------------	--

---

**Description**

The method `dprocess` evaluates the probability density of a set of consecutive state transitions. This function is part of the low-level interface to `pomp` objects. This help page does not give instructions on the implementation of models: see [pomp](#) for instructions.

**Usage**

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

**Arguments**

<code>object</code>	an object of class <code>pomp</code> .
<code>x</code>	a rank-3 array containing the states of the unobserved process. The dimensions of <code>x</code> are <code>nvars</code> x <code>nreps</code> x <code>ntimes</code> , where <code>nvars</code> is the number of state variables, <code>nreps</code> is the number of replicates, and <code>ntimes</code> is the length of times.
<code>times</code>	a numeric vector containing the times corresponding to the given states.
<code>params</code>	a rank-2 array of parameters with columns corresponding to the columns of <code>x</code> . Note that the <code>x</code> and <code>params</code> must agree in the number of their columns.
<code>log</code>	if TRUE, log probabilities are returned.
<code>...</code>	at present, these are ignored.

**Details**

This function is essentially a wrapper around the user-supplied `dprocess` slot of the `pomp` object. For specifications on writing such a function, see [pomp](#).

**Value**

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

**Author(s)**

Aaron A. King (kingaa at umich dot edu)

**See Also**

[pomp-class](#), [pomp](#)

---

euler

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

---

**Description**

Plug-in facilities for implementing discrete-time Markov processes and continuous-time Markov processes using the Euler algorithm. These can be used in the `rprocess` and `dprocess` slots of `pomp`.

**Usage**

```
euler.simulate(xstart, times, params, step.fun, delta.t, ...,
              statenames = character(0), paramnames = character(0),
              covarnames = character(0), zeronames = character(0),
              tcovar, covar, PACKAGE)
onestep.simulate(xstart, times, params, step.fun, ...,
                statenames = character(0), paramnames = character(0),
                covarnames = character(0), zeronames = character(0),
                tcovar, covar, PACKAGE)
onestep.density(x, times, params, dens.fun, ...,
               statenames = character(0), paramnames = character(0),
               covarnames = character(0), tcovar, covar, log = FALSE,
               PACKAGE)
```

**Arguments**

<code>xstart</code>	Matrix (dimensions <code>nvar x nrep</code> ) of states at initial time <code>times[1]</code> .
<code>x</code>	Matrix (dimensions <code>nvar x nrep x ntimes</code> ) of states at times <code>times</code> .
<code>times</code>	Vector of times (length <code>ntimes</code> ) at which states are required or given.
<code>params</code>	Matrix containing parameters of the model. The <code>nrep</code> columns of <code>params</code> correspond to those of <code>xstart</code> .

<code>step.fun</code>	This can be either an R function or the name of a compiled, dynamically loaded native function containing the model simulator. It should be written to take a single Euler step from a single point in state space. If it is a native function, it must be of type “ <code>pomp_onestep_sim</code> ” as defined in the header “ <code>pomp.h</code> ”, which is included with the package. For details on how to write such codes, see Details.
<code>dens.fun</code>	This can be either an R function or a compiled, dynamically loaded native function containing the model transition log probability density function. This function will be called to compute the log likelihood of the actual Euler steps. It must be of type “ <code>pomp_onestep_pdf</code> ” as defined in the header “ <code>pomp.h</code> ”, which is included with the package. For details on how to write such codes, see Details.
<code>delta.t</code>	Time interval of Euler steps.
<code>statenames, paramnames, covarnames</code>	Names of state variables, parameters, covariates, in the order they will be expected by the routine named in <code>step.fun</code> and <code>dens.fun</code> . This information is only used when the latter are implemented as compiled native functions.
<code>zeronames</code>	Names of additional variables which will be zeroed before each time in <code>times</code> . These are useful, e.g., for storing accumulations of state variables.
<code>covar, tcovar</code>	Matrix of covariates and times at which covariates are measured.
<code>log</code>	logical; if TRUE, log probabilities are given.
<code>...</code>	if <code>step.fun</code> (or <code>dens.fun</code> ) is an R function, then additional arguments will be passed to it. If <code>step.fun</code> (or <code>dens.fun</code> ) is a native routine, then additional arguments are ignored.
<code>PACKAGE</code>	an optional argument that specifies to which dynamically loaded library we restrict the search for the native routines. If this is “ <code>base</code> ”, we search in the R executable itself.

## Details

`onestep.simulate` assumes that a single call to `step.fun` will advance the state process from one time to the next. `euler.simulate` will take multiple Euler steps, each of size at most `delta.t` (see below for information on how the actual Euler step size is chosen) to get from one time to the next.

`onestep.density` assumes that no state transitions occur between consecutive times.

If `step.fun` is written as an R function, it must have at least the arguments `x`, `t`, `params`, `delta.t`, and `...`. On a call to this function, `x` will be a named vector of state variables, `t` a scalar time, and `params` a named vector of parameters. The length of the Euler step will be `delta.t`. If the argument `covars` is included and a covariate table has been included in the `pomp` object, then on a call to this function, `covars` will be filled with the values, at time `t`, of the covariates. This is accomplished via interpolation of the covariate table. Additional arguments may be given: these will be filled by the correspondingly-named elements in the `userdata` slot of the `pomp` object (see [pomp](#)).

If `step.fun` is written in a native language, it must be a function of type “`pomp_onestep_sim`” as specified in the header “`pomp.h`” included with the package (see the directory “`include`” in the installed package directory).

If `dens.fun` is written as an R function, it must have at least the arguments `x1`, `x2`, `t1`, `t2`, `params`, and `. . .`. On a call to this function, `x1` and `x2` will be named vectors of state variables at times `t1` and `t2`, respectively. The named vector `params` contains the parameters. If the argument `covars` is included and a covariate table has been included in the `pomp` object, then on a call to this function, `covars` will be filled with the values, at time `t1`, of the covariates. This is accomplished via interpolation of the covariate table. As above, any additional arguments will be filled by the correspondingly-named elements in the `userdata` slot of the `pomp` object (see [pomp](#)).

If `dens.fun` is written in a native language, it must be a function of type "pomp\_onestep\_pdf" as defined in the header "pomp.h" included with the package (see the directory "include" in the installed package directory).

### Value

`euler.simulate` and `onestep.simulate` each return a `nvar x nrep x ntimes` array, where `nvar` is the number of state variables, `nrep` is the number of replicate simulations (= number of columns of `xstart` and `params`), and `ntimes` is the length of `times`. If `x` is this array, `x[, , 1]` will be identical to `xstart`; the rownames of `x` and `xstart` will also coincide.

`onestep.density` returns a `nrep x ntimes-1` array. If `f` is this array, `f[i, j]` is the likelihood of a transition from `x[i, j]` to `x[i, j+1]` in exactly one Euler step of duration `times[j+1]-times[j]`.

### Author(s)

Aaron A. King (kingaa at umich dot edu)

### See Also

[eulermultinom](#), [pomp](#)

### Examples

```
## an example showing how to use these functions to implement a seasonal SIR model is contain
## in the 'examples' directory
## Not run:
edit(file=system.file("examples/euler_sir.R", package="pomp"))
## End(Not run)
```

---

Euler-multinomial models

*Euler-multinomial models*

---

### Description

Density and random-deviate generation for the Euler-multinomial death process with parameters `size`, `rate`, and `dt`.

**Usage**

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

**Arguments**

<code>n</code>	integer; number of random variates to generate.
<code>size</code>	scalar integer; number of individuals at risk.
<code>rate</code>	numeric vector of hazard rates.
<code>dt</code>	numeric scalar; duration of Euler step.
<code>x</code>	Matrix or vector containing number of individuals that have succumbed to each death process.
<code>log</code>	logical; if TRUE, return logarithm(s) of probabilities.

**Details**

Direct access to the underlying C routines is available: see the header file “pomp.h”, included with the package.

**Value**

`reulermultinom`  
Returns a `length(rate)` by `n` matrix. Each column is a different random draw. Each row contains the numbers of individuals succumbed to the corresponding process.

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

**Author(s)**

Aaron A. King (kingaa at umich dot edu)

**See Also**

[euler](#)

**Examples**

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

---

`euler.sir`*Seasonal SIR model implemented as an Euler-multinomial model*

---

**Description**

`euler.sir` is a `pomp` object encoding a simple seasonal SIR model.

**Usage**

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

**See Also**

[pomp-class](#) and the vignettes

**Examples**

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

---

`init.state-pomp`*Return a matrix of initial conditions given a vector of parameters and an initial time.*

---

**Description**

The method `init.state` returns a vector of initial conditions for the state process when given a vector of parameters `params` and an initial time `t0`. This function is part of the low-level interface to `pomp` objects. This help page does not give instructions on the implementation of models: see [pomp](#) for instructions.

**Usage**

```
init.state(object, params, t0, ...)
## S4 method for signature 'pomp':
init.state(object, params, t0, ...)
```

**Arguments**

<code>object</code>	an object of class <code>pomp</code> .
<code>params</code>	a named vector of parameters.
<code>t0</code>	the initial time at which initial states are requested.
<code>...</code>	at present, these are ignored.

**Value**

Returns a matrix of initial states (with rownames).

**Author(s)**

Aaron A. King (kingaa at umich dot edu)

**See Also**

[pomp-class](#)

**Examples**

```
data(ou2)
coef(ou2)
init.state(ou2)

data(euler.sir)
coef(euler.sir)
init.state(euler.sir)
```

---

mif

*The MIF algorithm*


---

**Description**

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

**Usage**

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

**Arguments**

object	An object of class pomp.
Nmif	The number of MIF iterations to perform.
start	The initial guess of the parameters. This must be a named vector.

<code>pars</code>	optional character vector naming the ordinary parameters to be estimated. Every parameter named in <code>pars</code> must have a positive random-walk standard deviation specified in <code>rw.sd</code> . Leaving <code>pars</code> unspecified is equivalent to setting it equal to the names of all parameters with a positive value of <code>rw.sd</code> that are not <code>ivps</code> .
<code>ivps</code>	optional character vector naming the initial-value parameters to be estimated. Every parameter named in <code>ivps</code> must have a positive random-walk standard deviation specified in <code>rw.sd</code> .
<code>particles</code>	Function of prototype <code>particles(Np, center, sd, ...)</code> which sets up the initial particle matrix by drawing a sample of size <code>Np</code> from the initial particle distribution centered at <code>center</code> and of width <code>sd</code> . If <code>particles</code> is not supplied by the user, the default behavior is to draw the particles from a multivariate normal distribution with mean <code>center</code> and standard deviation <code>sd</code> .
<code>rw.sd</code>	numeric vector with names; the intensity of the random walk to be applied to parameters. The random walk is only applied to parameters named in <code>pars</code> (i.e., not to those named in <code>ivps</code> ). The algorithm requires that the random walk be nontrivial, so each element in <code>rw.sd[pars]</code> must be positive. <code>rw.sd</code> is also used to scale the initial-value parameters (via the <code>particles</code> function). Therefore, each element of <code>rw.sd[ivps]</code> must be positive. The following must be satisfied: <code>names(rw.sd)</code> must be a subset of <code>names(start)</code> , <code>rw.sd</code> must be non-negative (zeros are simply ignored), the name of every positive element of <code>rw.sd</code> must be in either <code>pars</code> or <code>ivps</code> .
<code>Np</code>	a positive integer; the number of particles to use in filtering
<code>ic.lag</code>	a positive integer; the timepoint for fixed-lag smoothing of initial-value parameters
<code>var.factor</code>	a positive number; the scaling coefficient relating the width of the initial particle distribution to <code>rw.sd</code> . The width of the initial distribution of particles will be <code>random.walk.sd*var.factor</code> .
<code>cooling.factor</code>	a positive number not greater than 1; the exponential cooling factor, <code>alpha</code> .
<code>weighted</code>	Should a weighted average be used? If <code>weighted=F</code> , the MIF update is not used; instead, an unweighted average of the filtering means is used for the update.
<code>tol</code>	Particles with log likelihood below <code>tol</code> are considered to be "lost". A filtering failure occurs when, at some time point, all particles are lost.
<code>warn</code>	Should a warning be generated when a filtering failure occurs?
<code>max.fail</code>	Maximum number of filtering failures permitted. If the number of failures exceeds this number, execution will terminate with an error.
<code>verbose</code>	logical; if TRUE, print progress reports.
<code>...</code>	Additional arguments that can be used to override the defaults.

### Re-running MIF Iterations

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

### Continuing MIF Iterations

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

### Details

If `particles` is not specified, the default behavior is to draw the particles from a multivariate normal distribution. **It is the user's responsibility to ensure that, if the optional `particles` argument is given, that the `particles` function satisfies the following conditions:**

`particles` has at least the following arguments: `Np`, `center`, `sd`, and `...`. `Np` may be assumed to be a positive integer; `center` and `sd` will be named vectors of the same length. Additional arguments may be specified; these will be filled with the elements of the `userdata` slot of the underlying `pomp` object (see [pomp-class](#)).

`particles` returns a  $\text{length}(\text{center}) \times N_p$  matrix with rownames matching the names of `center` and `sd`. Each column represents a distinct particle.

The center of the particle distribution returned by `particles` should be `center`. The width of the particle distribution should vary monotonically with `sd`. In particular, when `sd=0`, the `particles` should return matrices with  $N_p$  identical columns, each corresponding to the parameters specified in `center`.

### Author(s)

Aaron A. King (kingaa at umich dot edu)

### References

- E. L. Ionides, C. Bretó, & A. A. King, Inference for nonlinear dynamical systems, Proc. Natl. Acad. Sci. U.S.A., 103:18438–18443, 2006.
- A. A. King, E. L. Ionides, M. Pascual, and M. J. Bouma, Inapparent infections and cholera dynamics, Nature, 454:877–880, 2008.

### See Also

[mif-class](#), [mif-methods](#), [pomp](#), [pomp-class](#), [pfilter](#). See the “intro\_to\_pomp” vignette for an example.

---

mif-class

The "mif" class

---

### Description

The `mif` class holds a fitted model and is created by a call to `mif`. See [mif](#) for usage.

## Objects from the Class

Objects can be created by calls to the `mif` method on an `pomp` object. Such a call uses the MIF algorithm to fit the model parameters.

## Slots

A `mif` object is derived from a `pomp` object and therefore has all the slots of such an object. See `pomp-class` for details. A full description of slots in a `mif` object follows.

**ivps** A character vector containing the names of initial-value parameters (IVPs). These are parameters which are to be estimated using fixed-lag smoothing.

**pars** A character vector containing the names of parameters to be estimated using MIF.

**Nmif** Number of MIF iterations that have been completed.

**particles** A function of prototype `particles(Np, center, sd, ...)` that draws particles from a distribution centered on `center` and with width proportional to `sd`. This function can be optionally specified by the user.

**alg.pars** A named list of algorithm parameters. This consists of `Np`, the number of particles to use in filtering; `var.factor`, the scaling coefficient relating the width of the initial particle distribution to `rw.sd`; `ic.lag`, the fixed lag used in the estimation of initial-value parameters (IVPs); and `cooling.factor`, the exponential cooling factor, where  $0 < \text{cooling.factor} < 1$ .

**random.walk.sd** A named vector containing the random-walk variance to be used for ordinary parameters.

**pred.mean** Matrix of prediction means. See `pfilter`.

**pred.var** Matrix of prediction variances. See `pfilter`.

**filter.mean** Matrix of filtering means. See `pfilter`.

**eff.sample.size** A vector containing the effective number of particles at each time point. See `pfilter`.

**cond.loglik** A vector containing the conditional log likelihoods at each time point. See `pfilter`.

**conv.rec** The “convergence record”: a matrix containing a record of the parameter values, log likelihoods, and other pertinent information, with one row for each MIF iteration.

**loglik** A numeric value containing the value of the log likelihood, as evaluated for the random-parameter model. Note that this will not be equal to the log likelihood for the fixed-parameter model.

**data, times, t0, rprocess, dprocess, dmeasure, rmeasure, skeleton.type, skeleton, initializer, states, params, statename**  
Inherited from the `pomp` class.

## Extends

Class `pomp`, directly. See `pomp-class`.

## Methods

See `mif`, `mif-methods`, `particles-mif`, `pfilter-mif`.

**Author(s)**

Aaron A. King (kingaa at umich dot edu)

**References**

E. L. Ionides, C. Bretó, & A. A. King, Inference for nonlinear dynamical systems, Proc. Natl. Acad. Sci. U.S.A., 103:18438–18443, 2006.

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

**See Also**

[mif](#), [mif-methods](#), [pomp](#), [pomp-class](#)

---

mif-methods

*Methods of the "mif" class*


---

**Description**

Methods of the "mif" class.

**Usage**

```
## S4 method for signature 'mif':
logLik(object, ...)
conv.rec(object, ...)
## S4 method for signature 'mif':
conv.rec(object, pars, ...)
pred.mean(object, ...)
## S4 method for signature 'mif':
pred.mean(object, pars, ...)
pred.var(object, ...)
## S4 method for signature 'mif':
pred.var(object, pars, ...)
filter.mean(object, ...)
## S4 method for signature 'mif':
filter.mean(object, pars, ...)
## S4 method for signature 'mif':
plot(x, y = NULL, ...)
compare.mif(z)
```

**Arguments**

object	The mif object.
pars	Names of parameters.
x	The mif object.

<code>y</code>	Ignored.
<code>z</code>	A <code>mif</code> object or list of <code>mif</code> objects.
<code>...</code>	Further arguments (either ignored or passed to underlying functions).

## Methods

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

**logLik** Returns the value in the `loglik` slot.

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

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

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

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

**pred.var** `pred.var(object, pars = NULL)` returns the rows of the prediction-variance matrix corresponding to the names in `pars`. By default, all rows are returned.

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

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

**print** Prints a summary of the `mif` object.

**show** Displays the `mif` object.

**pfilter** See `pfilter-mif`.

## Author(s)

Aaron A. King (kingaa at umich dot edu)

## References

E. L. Ionides, C. Bretó, & A. A. King, Inference for nonlinear dynamical systems, *Proc. Natl. Acad. Sci. U.S.A.*, 103:18438–18443, 2006.

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

## See Also

`mif`, `pomp`, `pomp-class`, `pfilter`

**Description**

Calls an optimizer to maximize the nonlinear forecasting (NLF) goodness of fit, by simulating data from a model, fitting a nonlinear autoregressive model to the simulated time series (which may be multivariate) and using the fitted model to predict some or all variables in the data time series. NLF is an ‘indirect inference’ method using a quasi-likelihood as the objective function.

**Usage**

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

**Arguments**

object	A pomp object, with the data and model to fit to it.
start	Named numeric vector with guessed parameters.
est	Vector containing the names or indices of parameters to be estimated.
lags	A vector specifying the lags to use when constructing the nonlinear autoregressive prediction model. The first lag is the prediction interval.
period	numeric; period=NA means the model is nonseasonal. period>0 is the period of seasonal forcing in ‘real time’.
tensor	logical; if FALSE, the fitted model is a generalized additive model with time mod period as one of the predictors, i.e., a gam with time-varying intercept. If TRUE, the fitted model is a gam with lagged state variables as predictors and time-periodic coefficients, constructed using tensor products of basis functions of state variables with basis functions of time.
nconverge	Number of convergence timesteps to be discarded from the model simulation.
nasymp	Number of asymptotic timesteps to be recorded from the model simulation.
seed	Integer specifying the random number seed to use. When fitting, it is usually best to always run the simulations with the same sequence of random numbers, which is accomplished by setting seed to an integer. If you want a truly random simulation, set seed=NULL.
nrbf	A scalar specifying the number of radial basis functions to be used at each lag.
method	Optimization method. Choices are <a href="#">subplex</a> and any of the methods used by <a href="#">optim</a> .
skip.se	Logical; if TRUE, skip the computation of standard errors.
verbose	Logical; if TRUE, the negative log quasilielihood and parameter values are printed at each iteration of the optimizer.

<code>gr</code>	optional; passed to <code>optim</code> if <code>optim</code> is used.
<code>bootstrap</code>	Logical; if <code>TRUE</code> the indices in <code>bootsamp</code> will determine which of the conditional likelihood values be used in computing the quasi-loglikelihood.
<code>bootsamp</code>	Vector of integers; used to have the quasi-loglikelihood evaluated using a bootstrap re-sampling of the data set.
<code>lql.frac</code>	target fractional change in log quasi-likelihood for quadratic standard error estimate
<code>se.par.frac</code>	initial parameter-change fraction for quadratic standard error estimate
<code>eval.only</code>	logical; if <code>TRUE</code> , no optimization is attempted and the quasi-loglikelihood value is evaluated at the <code>start</code> parameters.
<code>...</code>	Arguments that will be passed to <code>optim</code> or <code>subplex</code> in the <code>control</code> list.

### Details

This is functionally a wrapper for `nlf.objfun`, which does the statistical heavy lifting and should be consulted for details.

### Value

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

### Author(s)

Stephen P. Ellner (`<spe2 at cornell dot edu>`) and Bruce E. Kendall (`<kendall at bren dot ucsb dot edu>`)

### References

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

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

Kendall, B. E., Briggs, C. J., Murdoch, W. W., Turchin, P., Ellner, S. P., McCauley, E., Nisbet, R. M. and Wood S. N. (1999) Why do populations cycle? A synthesis of statistical and mechanistic modeling approaches. *Ecology* **80**, 1789–1805. Available online at <http://www2.bren.ucsb.edu/~kendall/pubs/1999Ecology.pdf>

Kendall, B. E., Ellner, S. P., McCauley, E., Wood, S. N., Briggs, C. J., Murdoch, W. W. and Turchin, P. (2005) Population cycles in the pine looper moth (*Bupalus piniarius*): dynamical tests of mechanistic hypotheses. *Ecological Monographs* **75**, 259–276. Available online at <http://repositories.cdlib.org/postprints/818/>

**Description**

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

**Usage**

```
data(ou2)
```

**Details**

The OU process is fully but noisily observed. The functions `rprocess`, `dprocess`, `rmeasure`, and `dmeasure` are implemented using compiled C code for computational speed: see the source code for details. The use of this object is demonstrated in the vignette.

**See Also**

[pomp-class](#) and the vignettes

**Examples**

```
data(ou2)
plot(ou2)
coef(ou2)
p <- c(
  alpha.1=0.9, alpha.2=0, alpha.3=0, alpha.4=0.99,
  sigma.1=1, sigma.2=0, sigma.3=2,
  tau=1,
  x1.0=50, x2.0=-50
)
x <- simulate(ou2, nsim=10, seed=20348585, params=p)
plot(x[[1]])
pfilter(ou2, params=p, Np=1000)$loglik
```

**Description**

Generate particles from the user-specified distribution. This is part of the low-level interface, used by `mif`. This help page does not give instruction on how to write a valid `particles` function: see the documentation for `mif` instead.

**Usage**

```
particles(object, ...)  
## S4 method for signature 'mif':  
particles(object, Np = 1, center = coef(object), sd = 0, ...)
```

**Arguments**

object	the mif object
Np	the number of particles, i.e., number of draws.
center	the central value of the distribution of particles
sd	the width of the distribution
...	additional arguments. At present, these are ignored.

**Details**

The `particles` method is used to set up the initial distribution of particles. It is an interface to the user-specified `particles` slot in the `mif` object.

**Value**

`particles` returns a list of two matrices. `states` contains the state-variable portion of the particles; `params` contains the parameter portion. Each has `Np` columns.

**Author(s)**

Aaron A. King (kingaa at umich dot edu)

**See Also**

[mif](#), [mif-methods](#), [pomp](#), [pomp-class](#)

---

pfilter

*Particle filter*

---

**Description**

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

**Usage**

```

pfilter(object, ...)
## S4 method for signature 'pomp':
pfilter(object, params, Np, tol = 1e-17,
        warn = TRUE, max.fail = 0, pred.mean = FALSE, pred.var = FALSE,
        filter.mean = FALSE, verbose = FALSE, ...)
## S4 method for signature 'mif':
pfilter(object, params, Np, tol = 1e-17, warn = TRUE,
        max.fail = 0, pred.mean = FALSE, pred.var = FALSE,
        filter.mean = FALSE, ...)

```

**Arguments**

object	An object of class <code>pomp</code> or inheriting class <code>pomp</code> .
params	A $n_{\text{pars}} \times N_{\text{p}}$ matrix containing the parameters corresponding to the initial state values in <code>xstart</code> . This must have a 'rownames' attribute. It is permissible to supply <code>params</code> as a named numeric vector, i.e., without a <code>dim</code> attribute. In this case, all particles will inherit the same parameter values.
Np	Number of particles to use. When <code>object</code> is of class <code>mif</code> , this is by default the same number of particles used in the <code>mif</code> iterations.
tol	Particles with log likelihood below <code>tol</code> are considered to be "lost". A filtering failure occurs when, at some time point, all particles are lost. When all particles are lost, the conditional log likelihood at that time point is set to be <code>log(tol)</code> .
warn	Should filtering failures generate warnings?
max.fail	The maximum number of filtering failures allowed. If the number of filtering failures exceeds this number, execution will terminate with an error.
pred.mean	If TRUE, the prediction means are calculated for the state variables and parameters.
pred.var	If TRUE, the prediction variances are calculated for the state variables and parameters.
filter.mean	If TRUE, the filtering means are calculated for the state variables and parameters.
verbose	If TRUE, progress information is reported as <code>pfilter</code> works.
...	Additional arguments unused at present.

**Value**

A list with the following elements:

pred.mean	The matrix of prediction means. The rows correspond to states and parameters (if appropriate), in that order, the columns to successive observations in the time series contained in <code>object</code> .
pred.var	The matrix of prediction variances, in the same format as <code>pred.mean</code> .
filter.mean	The matrix of filtering means, in the same format as <code>pred.mean</code> .
eff.sample.size	A vector containing the effective number of particles at each time point.

`cond.loglik` A vector containing the conditional log likelihoods at each time point.  
`nfail` The number of filtering failures encountered.  
`loglik` The estimated log-likelihood.

**Author(s)**

Aaron A. King (kingaa at umich dot edu)

**References**

M. S. Arulampalam, S. Maskell, N. Gordon, & T. Clapp. A Tutorial on Particle Filters for Online Nonlinear, Non-Gaussian Bayesian Tracking. *IEEE Trans. Sig. Proc.* 50:174–188, 2002.

**See Also**

[pomp-class](#)

**Examples**

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

---

pomp

*Partially-observed Markov process object.*

---

**Description**

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

**Usage**

```
pomp(data, times, t0, ..., rprocess, dprocess, rmeasure, dmeasure,
      measurement.model, skeleton.map, skeleton.vectorfield,
      initializer, covar, tcovar, statenames, paramnames, covarnames,
      PACKAGE)
```

**Arguments**

`data` An array holding the data. This array is of dimensions `nobs` x `ntimes`, where `nobs` is the number of observed variables and `ntimes` is the number of times at which observations were made. It is also possible to specify `data` as a data-frame, in which case the argument `times` must be the name of the column of observation times. Note that if `data` is a data-frame, it will be internally coerced to an array with storage-mode ‘double’.

`times` vector of times corresponding to the observations: `times` must be a strictly increasing numeric vector. If `data` is a data-frame, `times` should be the name of the column of observation times.

<code>t0</code>	The zero-time. This must be no later than the time of the first observation, <code>times[1]</code> .
<code>rprocess</code>	optional; a function of prototype <code>rprocess(xstart, times, params, ...)</code> which simulates from the unobserved process. See below for details.
<code>dprocess</code>	optional; a function of prototype <code>dprocess(x, times, params, log, ...)</code> which evaluates the likelihood of a sequence of consecutive state transitions. See below for details.
<code>rmeasure</code>	optional; the measurement model simulator. This can be specified in one of three ways: (1) as a function of prototype <code>rmeasure(x, t, params, ...)</code> which makes a draw from the observation process given states <code>x</code> , time <code>t</code> , and parameters <code>params</code> . (2) as the name of a native (compiled) routine with prototype “ <code>pomp_measure_model_simulator</code> ” as defined in the header file “ <code>pomp.h</code> ”. In the above cases, if the measurement model depends on covariates, the optional argument <code>covars</code> will be filled with interpolated values at each call. (3) using the formula-based <code>measurement.model</code> facility (see below).
<code>dmeasure</code>	optional; the measurement model probability density function. This can be specified in one of three ways: (1) as a function of prototype <code>dmeasure(y, x, t, params, log, ...)</code> which computes the p.d.f. of <code>y</code> given <code>x</code> , <code>t</code> , and <code>params</code> . (2) as the name of a native (compiled) routine with prototype “ <code>pomp_measure_model_density</code> ” as defined in the header file “ <code>pomp.h</code> ”. In the above cases, if the measurement model depends on covariates, the optional argument <code>covars</code> will be filled with interpolated values at each call. (3) using the formula-based <code>measurement.model</code> facility (see below). As might be expected, if <code>log=TRUE</code> , this function should return the log likelihood.
<code>measurement.model</code>	optional; a formula or list of formulae, specifying the measurement model. These formulae are parsed and used to generate <code>rmeasure</code> and <code>dmeasure</code> functions. If <code>measurement.model</code> is given it overrides any specification of <code>rmeasure</code> or <code>dmeasure</code> . See below for an example. <b>NB:</b> it will typically be possible to accelerate measurement model computations by writing <code>dmeasure</code> and/or <code>rmeasure</code> functions directly.
<code>skeleton.map</code>	optional. If we are dealing with a discrete-time Markov process, its deterministic skeleton is a map and can be specified using <code>skeleton.map</code> in one of two ways: (1) as a function of prototype <code>skeleton(x, t, params, ...)</code> which evaluates the deterministic skeleton at state <code>x</code> and time <code>t</code> given the parameters <code>params</code> , or (2) as the name of a native (compiled) routine with prototype “ <code>pomp_vectorfield_map</code> ” as defined in the header file “ <code>pomp.h</code> ”. If the deterministic skeleton depends on covariates, the optional argument <code>covars</code> will be filled with interpolated values of the covariates at the time <code>t</code> .
<code>skeleton.vectorfield</code>	optional. If we are dealing with a continuous-time Markov process, its deterministic skeleton is a vectorfield and can be specified using <code>skeleton.vectorfield</code> in either of the two ways described above, under <code>skeleton.map</code> . Note that it makes no sense to specify the deterministic skeleton both as a map and as a vectorfield: an attempt to do so will generate an error.
<code>initializer</code>	optional; a function of prototype <code>initializer(params, t0, ...)</code> which yields initial conditions for the state process when given a vector, <code>params</code> ,

of parameters. By default, i.e., if it is unspecified when `pomp` is called, the initializer assumes any names in `params` that end in “.0” correspond to initial value parameters. These are simply copied over as initial conditions when `init.state` is called (see [init.state-pomp](#)). The names of the state variables are the same as the corresponding initial value parameters, but with the “.0” dropped.

`covar`, `tcovar`

An optional table of covariates: `covar` is the table (with one column per variable) and `tcovar` the corresponding times (one entry per row of `covar`). This can be in the form of a matrix or a data frame. In either case the columns are taken to be distinct covariates. If `covar` is a data frame, `tcovar` can be either the name or the index of the time variable. If a covariate table is supplied, then the value of each of the covariates is interpolated as needed, i.e., whenever `rprocess`, `dprocess`, `rmeasure`, `dmeasure`, or `init.state` is evaluated. The resulting interpolated values are passed to the corresponding functions as a numeric vector named `covars`.

`statenames`, `paramnames`, `covarnames`

Optional character vectors specifying the names of state variables, parameters, or covariates, respectively. These are only used in the event that one or more of the basic functions (`rprocess`, `dprocess`, `rmeasure`, `dmeasure`, `skeleton`) are defined using native routines. In that case, these name vectors are matched against the corresponding names and the indices of the names are passed to the native routines.

`PACKAGE`

An optional string giving the name of the dynamically loaded library in which the native routines are to be found.

`...`

Any additional arguments are passed as arguments to each of the functions `rprocess`, `dprocess`, `rmeasure`, `dmeasure`, and `initializer` whenever they are evaluated.

## Details

**It is not typically necessary (or easy) to define all of the functions `rprocess`, `dprocess`, `rmeasure`, `dmeasure`, and `initializer` in any given problem. Each algorithm makes use of a different subset of these functions.** The specification of process-model codes `rprocess` and/or `dprocess` can be somewhat nontrivial: for this reason, *plug-ins* have been developed to streamline this process for the user. At present, if the process model evolves in discrete-time or one is willing to make such an approximation (e.g., via an Euler approximation), then the [onestep.simulate](#) and [euler.simulate](#) plug-ins for `rprocess` and [onestep.density](#) plug-in for `dprocess` are available. The following describes how each of these functions should be written. Examples are provided in the help documentation and the vignettes.

**`rprocess`** if provided, must have at least the following arguments: `xstart`, `times`, `params`, and `...`. It can also take additional arguments. It is guaranteed that these will be filled with the corresponding elements the user has included as additional arguments in the construction of the `pomp` object.

In calls to `rprocess`, `xstart` can be assumed to be a rank-2 array (matrix) with rows corresponding to state variables and columns corresponding to independent realizations of the process. `params` will similarly be a rank-2 array with rows corresponding to parameters and

columns corresponding to independent realizations. The columns of `params` correspond to those of `xstart`; in particular, they will agree in number. Both `xstart` and `params` will have rownames, which are available for use by the user.

`rprocess` must return a rank-3 array with rownames. Suppose `x` is the array returned. Then  $\dim(x) = c(nvars, nreps, ntimes)$ , where `nvars` is the number of state variables ( $=nrow(xstart)$ ), `nreps` is the number of independent realizations simulated ( $=ncol(xstart)$ ), and `ntimes` is the length of the vector `times`. `x[, j, k]` is the value of the state process in the `j`-th realization at time `times[k]`. In particular, `x[, , 1]` must be identical to `xstart`. The rownames of `x` must correspond to those of `xstart`.

As mentioned above, some plug-ins are available for `rprocess`.

**dprocess** if provided, must have at least the following arguments: `x`, `times`, `params`, `log`, and `. . .`. It may take additional arguments. It is guaranteed that these will be filled with the corresponding elements the user has included as additional arguments in the construction of the `pomp` object.

In calls to `dprocess`, `x` may be assumed to be an `nvars x nreps x ntimes` array, where these terms have the same meanings as above. `params` will be a rank-2 array with rows corresponding to individual parameters and columns corresponding to independent realizations. The columns of `params` correspond to those of `x`; in particular, they will agree in number. Both `x` and `params` will have rownames, available for use by the user.

`dprocess` must return a rank-2 array (matrix). Suppose `d` is the array returned. Then  $\dim(d) = c(nreps, ntimes-1)$ . `d[j, k]` is the probability density of the transition from state `x[, j, k-1]` at time `times[k-1]` to state `x[, j, k]` at time `times[k]`. If `log=TRUE`, then the log of the pdf is returned.

As mentioned above, some plug-ins are available for `dprocess`.

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

**rmeasure** if provided, must take at least the arguments `x`, `t`, `params`, and `. . .`. It may take additional arguments, which will be filled with user-specified data as above. `x` may be assumed to be a named vector of length `nvars`, (which has the same meanings as above). `t` is a scalar quantity, the time at which the measurement is made. `params` may be assumed to be a named vector of length `nparams`.

`rmeasure` must return a named vector. If `y` is the returned vector, then  $\text{length}(y) = \text{nobs}$ , where `nobs` is the number of observable variables.

**dmeasure** if provided, must take at least the arguments `y`, `x`, `times`, `params`, `log`, and `. . .`. `y` may be assumed to be a named vector of length `nobs` containing (actual or simulated) values of the observed variables; `x` will be a named vector of length `nvars` containing state variables `params`, a named vector containing parameters; and `t`, a scalar, the corresponding observation time. It may take additional arguments which will be filled with user-specified data as above. `dmeasure` must return a single numeric value, the pdf of `y` given `x` at time `t`. If `log=TRUE`, then the log of the pdf is returned.

**skeleton** If `skeleton` is an R function, it must have at least the arguments `x`, `t`, `params`, and `. . .`. `x` is a numeric vector containing the coordinates of a point in state space at which evaluation of the skeleton is desired. `t` is a numeric value giving the time at which evaluation of the skeleton is desired. Of course, these will be irrelevant in the case of an autonomous skeleton. `params` is a numeric vector holding the parameters. The optional argument `covars`

is a numeric vector containing the values of the covariates at the time  $t$ . `covars` will have one value for each column of the covariate table specified when the `pomp` object was created. `covars` is constructed from the covariate table (see `covar`, below) by interpolation. `skeleton` may take additional arguments, which will be filled, as above, with user-specified data. `skeleton` must return a numeric vector of the same length as  $x$ . The return value is interpreted as the vectorfield (if the dynamical system is continuous) or the value of the map (if the dynamical system is discrete), at the point  $x$  at time  $t$ .

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

**initializer** if provided, must have at least the arguments `params`, `t0`, and `...`. `params` is a named vector of parameters. `t0` will be the time at which initial conditions are desired. `initializer` must return a named vector of initial conditions.

### Value

An object of class `pomp`.

### Warning

Some error checking is done, but complete error checking is impossible. If the user-specified functions do not conform to the above specifications (see Details), then the results may be invalid. Each algorithm that uses a `pomp` object uses some subset of the four basic components (`rprocess`, `dprocess`, `rmeasure`, `dmeasure`). It is unlikely that any algorithm will use all of these.

### Author(s)

Aaron A. King (kingaa at umich dot edu)

### See Also

[pomp-class](#), [pomp-methods](#), [rprocess](#), [dprocess](#), [rmeasure](#), [dmeasure](#), [skeleton](#), [init.state](#), [euler](#)

### Examples

```
## Simple example: a 2-D Brownian motion, observed with normal error
## first, set up the pomp object:
```

```
rw2 <- pomp(
  rprocess = function (xstart, times, params, ...) {
    ## this function simulates two independent random walks with intensities s1, s2
    nreps <- ncol(params)
    ntimes <- length(times)
    dt <- diff(times)
    x <- array(0, dim=c(2, nreps, ntimes))
    rownames(x) <- rownames(xstart)
    noise.sds <- params[c('s1', 's2'), ]
    x[, , 1] <- xstart
    for (j in 2:ntimes) {
```

```

    ## we are mimicking a continuous-time process, so the increments have SD ~ s
    ## note that we do not have to assume that 'times' are equally spaced
    x[, , j] <- rnorm(n=2*nreps, mean=x[, , j-1], sd=noise.sds*dt[j-1])
  }
  x
},
dprocess = function(x, times, params, log, ...) {
  ## given a sequence of consecutive states in 'x', this function computes the p
  nreps <- ncol(params)
  ntimes <- length(times)
  dt <- diff(times)
  d <- array(0, dim=c(2, nreps, ntimes-1))
  noise.sds <- params[c('s1', 's2'), ]
  for(j in 2:ntimes)
    d[, , j-1] <- dnorm(x[, , j]-x[, , j-1], mean=0, sd=noise.sds*dt[j-1], log=TRUE)
  if(log) {
    apply(d, c(2, 3), sum)
  } else {
    exp(apply(d, c(2, 3), sum))
  }
},
measurement.model=list(
  y1 ~ norm(mean=x1, sd=tau),
  y2 ~ norm(mean=x2, sd=tau)
),
skeleton = function(x, t, params, covars, ...) {
  ## since this is a continuous-time process, the skeleton is the vectorfield
  ## since the random walk is unbiased, the vectorfield is identically zero
  rep(0, length=length(x))
},
times=1:100,
data=rbind(
  y1=rep(0, 100),
  y2=rep(0, 100)
),
t0=0
)

## specify some parameters
p <- rbind(s1=c(2, 2, 3), s2=c(0.1, 1, 2), tau=c(1, 5, 0), x1.0=c(0, 0, 5), x2.0=c(0, 0, 0))

## simulate
examples <- simulate(rw2, params=p)
plot(examples[[1]])

## construct an (almost) equivalent pomp object using a plugin:
rw2 <- pomp(
  rprocess = euler.simulate,
  step.fun = function(x, t, params, delta.t, ...) {
    y <- rnorm(n=2, mean=x[c("x1", "x2")], sd=params[c("s1", "s2")]*delta.t)
    names(y) <- c("x1", "x2")
  }
),

```

```

rmeasure = function (x, t, params, ...) {
  y <- rnorm(n=2,mean=x[c("x1","x2")],sd=params["tau"])
  names(y) <- c("y1","y2")
  y
},
dmeasure = function (y, x, t, params, log, ...) {
  d <- dnorm(x=y[c("y1","y2")],mean=x[c("x1","x2")],sd=params["tau"],log=TRUE)
  print(d)
  if (log) sum(d,na.rm=T) else exp(sum(d,na.rm=T))
},
delta.t=1,
data=data.frame(
  time=1:100,
  y1=rep(c(1,2,3,4,NA),20),
  y2=0
),
times="time",
t0=0
)
rw2 <- simulate(rw2,params=c(s1=2,s2=0.1,tau=1,x1.0=0,x2.0=0))

## For more examples, see the vignettes.

```

---

pomp-class

*Partially-observed Markov process*


---

## Description

The class `pomp` encodes a partially-observed Markov process. This page documents the structure of the class: see the documentation for [pomp](#) for usage instructions.

## Objects from the Class

Objects should be created by calls of the function `pomp`.

## Slots

**data** An array holding the data. This array is of dimensions `nobs` x `ntimes`, where `nobs` is the number of observed variables and `ntimes` is the number of times at which observations were made.

**times** The times corresponding to the observations. `times` must be a strictly increasing numeric vector.

**t0** The zero-time.

**rprocess** Function of prototype `rprocess(xstart,times,params,...)` which simulates from the unobserved process.

**dprocess** Function of prototype `dprocess(x,times,params,log=FALSE,...)` which evaluates the likelihood of a sequence of consecutive state transitions.

- dmeasure** an object of class “pomp.fun” which encodes the measurement model density.
- rmeasure** an object of class “pomp.fun” which encodes the measurement model simulator.
- skeleton.type** a character variable specifying whether the deterministic skeleton is a map or a vectorfield.
- skeleton** an object of class “pomp.fun” which encodes the deterministic skeleton.
- initializer** Function of prototype `initializer(params, t0, ...)` which gives a vector of initial conditions when given a vector of parameters, `params`, and a time `t0`.
- states** An array to hold a trajectory of the unobserved process.
- params** A named numeric vector to hold model parameters.
- covar, tcovar** An optional table of covariates.
- statenames, paramnames, covarnames** Names state variables, parameters, and covariates, respectively. These are used to make the interface with native routines more robust.
- PACKAGE** Character variable giving the name of the dynamically loadable library containing native routines.
- userdata** A list containing any objects the user desires. Using this mechanism, the user can store additional information necessary for the definition of the model.

## Methods

See the pomp methods documentation: [pomp-methods](#).

## Author(s)

Aaron A. King (kingaa at umich dot edu)

## See Also

[pomp](#), [pomp-methods](#), [rprocess](#), [dprocess](#), [rmeasure](#), [dmeasure](#), [init.state](#), [simulate-pomp](#)

## Description

Methods of the `pomp` class.

**Usage**

```
## S4 method for signature 'pomp':
coef(object, pars, ...)
## S4 replacement method for signature 'pomp':
coef(object, pars, ...) <- value
## S4 method for signature 'pomp':
data.array(object, vars, ...)
## S4 method for signature 'pomp':
states(object, vars, ...)
## S4 method for signature 'pomp':
time(x, t0 = FALSE, ...)
## S4 replacement method for signature 'pomp':
time(object, include.t0 = FALSE, ...) <- value
## S4 method for signature 'pomp':
show(object)
## S4 method for signature 'pomp':
as(object, class)
## S4 method for signature 'pomp, data.frame':
coerce(from, to = "data.frame", strict = TRUE)
## S4 method for signature 'pomp':
print(x, ...)
## S4 method for signature 'pomp':
plot(x, y, variables, panel = lines,
      nc = NULL, yax.flip = FALSE,
      mar = c(0, 5.1, 0, if (yax.flip) 5.1 else 2.1),
      oma = c(6, 0, 5, 0), axes = TRUE, ...)
```

**Arguments**

<code>object, x</code>	The pomp object.
<code>pars</code>	optional character; names of parameters to be retrieved or set.
<code>vars</code>	optional character; names of observed variables to be retrieved.
<code>value</code>	numeric; values to be assigned to the parameters.
<code>t0</code>	logical; if TRUE, the zero time is prepended to the time vector.
<code>include.t0</code>	logical; if TRUE, the first element in <code>value</code> is taken to be the initial time.
<code>class</code>	character; name of the class to which <code>object</code> should be coerced.
<code>from, to</code>	the classes between which coercion should be performed.
<code>strict</code>	ignored.
<code>y</code>	ignored.
<code>variables</code>	optional character; names of variables to plot.
<code>panel</code>	a function of prototype <code>panel(x, col, bg, pch, type, ...)</code> which gives the action to be carried out in each panel of the display.
<code>nc</code>	the number of columns to use. Defaults to 1 for up to 4 series, otherwise to 2.
<code>yax.flip</code>	logical; if TRUE, the y-axis (ticks and numbering) should flip from side 2 (left) to 4 (right) from series to series.

**mar, oma** the 'par' settings for 'mar' and 'oma' to use. Modify with care!  
**axes** logical; indicates if x- and y- axes should be drawn.  
**...** Further arguments (either ignored or passed to underlying functions).

## Details

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

**coef<-** Assigns values to the `params` slot of the `pomp` object. If `coef(object)` exists, then `coef(object) <- value` has the effect of replacing the parameters of `object` with `value`; the names of `value` will be ignored and the names of `coef(object)` will be unchanged. If `coef(object)` does not exist, then `coef(object) <- value` assigns `value` to the parameters of `object`; the names of `coef(object)` will be those of `value` and an error will be generated if `value` does not have names. If `coef(object)` exists, then `coef(object, pars) <- value` replaces those parameters of `object` named in `pars` with the elements of `value`; the names of `value` are ignored. If `coef(object)` does not exist, then `coef(object, pars) <- value` assigns `value` to the parameters of `object`; in this case, the names of `object` will be `pars` and the names of `value` will be ignored.

**data.array** `data.array(object)` returns the array of observations. `data.array(object, vars)` gives just the observations of variables named `vars`. `vars` may specify the variables by position or by name.

**states** `states(object)` returns the array of states. `states(object, vars)` gives just the state variables named in `vars`. `vars` may specify the variables by position or by name.

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

**time<-** `time(object) <- value` replaces the observation times slot (`times`) of `object` with `value`. `time(object, include.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.

**show** Displays the `pomp` object.

**plot** Plots the data and state trajectories (if the latter exist). Additional arguments are passed to the low-level plotting routine.

**print** Prints the `pomp` object in a nice way.

**as, coerce** The `coerce` method should typically not be used directly. It is defined by `setAs` as a method to be used by `as`. A `pomp` object can be coerced to a data frame via `as(object, "data.frame")`. The data frame contains the times, the data, and the state trajectories, if they exist.

**rprocess** simulates the process model. See [rprocess-pomp](#).

**dprocess** evaluates the process model density. See [dprocess-pomp](#).

**rmeasure** simulates the measurement model. See [rmeasure-pomp](#).

**dmeasure** evaluates the measurement-model density. See [dmeasure-pomp](#).

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

**init.state** returns a vector of initialial conditions. See [init.state-pomp](#).

**simulate** `simulate` can be used to simulate state and observation trajectories. See documentation under [simulate-pomp](#).

**Author(s)**

Aaron A. King (kingaa at umich dot edu)

**See Also**

[pomp](#), [pomp-class](#), [rprocess](#), [dprocess](#), [rmeasure](#), [dmeasure](#), [init.state](#), [simulate-pomp](#)

---

rmeasure-pomp

*Simulate the measurement model of a partially-observed Markov process*

---

**Description**

The method `rmeasure` draws from the distribution of measurements given the state of the system. This function is part of the low-level interface to `pomp` objects. This help page does not give instructions on the implementation of models: see [pomp](#) for instructions.

**Usage**

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

**Arguments**

<code>object</code>	an object of class <code>pomp</code> .
<code>x</code>	a rank-3 array containing the states of the unobserved process. The dimensions of <code>x</code> are <code>nvars</code> x <code>nreps</code> x <code>ntimes</code> , where <code>nvars</code> is the number of state variables, <code>nreps</code> is the number of simulations, and <code>ntimes</code> is the number of distinct times.
<code>times</code>	a numerical vector containing the times at which the measurements are to be made.
<code>params</code>	a rank-2 array (dimensions <code>nparams</code> x <code>nreps</code> ) of parameters with the parameters corresponding to the columns of <code>x</code> .
<code>...</code>	at present, these are ignored.

**Details**

This function is essentially a wrapper around the user-supplied `rmeasure` slot of the `pomp` object. For specifications on writing such a function, see [pomp](#).

**Value**

Returns a rank-3 array of dimensions `nobs` x `nreps` x `ntimes`, where `nobs` is the number of observed variables.

**Author(s)**

Aaron A. King (kingaa at umich dot edu)

**See Also**

[pomp-class](#), [pomp](#)

---

rprocess-pomp

*Simulate the process model of a partially-observed Markov process*

---

**Description**

The method `rprocess` runs simulations of the the process-model portion of partially-observed Markov process. This function is part of the low-level interface to `pomp` objects. This help page does not give instructions on the implementation of models: see [pomp](#) for instructions.

**Usage**

```
rprocess(object, xstart, times, params, ...)
## S4 method for signature 'pomp':
rprocess(object, xstart, times, params, ...)
```

**Arguments**

<code>object</code>	an object of class <code>pomp</code> .
<code>xstart</code>	a rank-2 array containing the starting state of the system. Columns of <code>xstart</code> correspond to states; rows to state variables. If there is more than one column of <code>xstart</code> , multiple independent simulations will be performed, one corresponding to each column. Note that in this case, <code>params</code> must have the same number of columns as <code>xstart</code> .
<code>times</code>	a numerical vector containing times. The first entry of <code>times</code> is the initial time (corresponding to <code>xstart</code> ). Subsequent times are the additional times at which the state of the simulated processes are required.
<code>params</code>	a rank-2 array of parameters with the parameters corresponding to the columns of <code>xstart</code> .
<code>...</code>	at present, these are ignored.

**Details**

This function is essentially a wrapper around the user-supplied `rprocess` slot of the `pomp` object. For specifications on writing such a function, see [pomp](#).

**Value**

`rprocess` returns a rank-3 array with rownames. Suppose `x` is the array returned. Then `dim(x) = c(nvars, nreps, ntimes)` where `nvars` is the number of state variables (`=nrow(xstart)`), `nreps` is the number of independent realizations simulated (`=ncol(xstart)`), and `ntimes` is the length of the vector `times`. `x[, j, k]` is the value of the state process in the `j`-th realization at time `times[k]`. In particular, `x[, , 1]` must be identical to `xstart`. The rownames of `x` must correspond to those of `xstart`.

**Author(s)**

Aaron A. King (kingaa at umich dot edu)

**See Also**

[pomp-class](#), [pomp](#)

---

rw2

*Two-dimensional random-walk process*

---

**Description**

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

**Usage**

```
data(rw2)
```

**Details**

The random-walk process is fully but noisily observed.

**See Also**

[pomp-class](#) and the vignettes

**Examples**

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

---

simulate-pomp      *Running simulations of a partially-observed Markov process*

---

## Description

`simulate` can be used to generate simulated data sets and/or to simulate the state process.

## Usage

```
## S4 method for signature 'pomp':
simulate(object, nsim = 1, seed = NULL, params,
         states = FALSE, obs = FALSE, times = time(object, t0=TRUE),
         ...)
```

## Arguments

<code>object</code>	An object of class <code>pomp</code> .
<code>nsim</code>	The number of simulations to perform. Note that the number of replicates will be <code>nsim times ncol(xstart)</code> .
<code>seed</code>	optional; the random seed to use.
<code>params</code>	The parameters to use in simulating the model. If <code>params</code> is not given, then the contents of the <code>params</code> slot of <code>object</code> will be used, if they exist.
<code>states</code>	Do we want the state trajectories?
<code>obs</code>	Do we want data-frames of the simulated observations?
<code>times</code>	The times for which observations are required. Note that the first element in <code>times</code> is the start time.
<code>...</code>	further arguments that are at present ignored.

## Details

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

## Value

If `states=FALSE` and `obs=FALSE` (the default), a list of `nsim pomp` objects is returned. Each has a simulated data set, together with the parameters used (in slot `params`) and the state trajectories also (in slot `states`). If `times` is specified, then the `t0` slot of the created ‘pomp’ objects will be filled with `times[1]` and the simulated observations will be at `times times[-1]`.

If `nsim=1`, then a single `pomp` object is returned (and not a singleton list).

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

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

If both `states=TRUE` and `obs=TRUE`, then a named list is returned. It contains the state trajectories and simulated observations as above.

### Author(s)

Aaron A. King (kingaa at umich dot edu)

### See Also

[pomp-class](#)

### Examples

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

---

skeleton-pomp

*Evaluate the deterministic skeleton at the given points in state space.*

---

### Description

The method `skeleton` computes the deterministic skeleton. In the case of a discrete-time system, this is the one-step map. In the case of a continuous-time system, this is the vector-field. NB: `skeleton` just evaluates the deterministic skeleton; it does not iterate or integrate. This function is part of the low-level interface to `pomp` objects. This help page does not give instructions on the implementation of models: see [pomp](#) for instructions.

### Usage

```
skeleton(object, x, t, params, ...)
## S4 method for signature 'pomp':
skeleton(object, x, t, params, ...)
```

### Arguments

<code>object</code>	an object of class <code>pomp</code> .
<code>x</code>	a rank-3 array containing the states of the unobserved process at which the deterministic skeleton is to be evaluated. The dimensions of <code>x</code> are <code>nvars x nreps x ntimes</code> , where <code>nvars</code> is the number of state variables, <code>nreps</code> is the number of replicates, and <code>ntimes</code> is the length of <code>times</code> .
<code>t</code>	a numeric vector containing the times at which the deterministic skeleton is to be evaluated.
<code>params</code>	a rank-2 array of parameters with columns corresponding to the columns of <code>x</code> . Note that the <code>x</code> and <code>params</code> must agree in the number of their columns.
<code>...</code>	at present, these are ignored.

**Details**

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

The function `trajectory` iterates the skeleton (in case it is a map) or integrates it using an ODE solver (in case it is a vectorfield).

**Value**

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

**Author(s)**

Aaron A. King (kingaa at umich dot edu)

**See Also**

[pomp-class](#), [pomp](#), [trajectory](#)

---

slice.design

*Slices through multidimensional parameter space*

---

**Description**

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

**Usage**

```
slice.design(vars, n)
```

**Arguments**

<code>vars</code>	Named list of numeric vectors, each of which has length either 1 or 3. Variables along which slices are to be taken should have length 3, corresponding to the minimum of the range, central point, and maximum of the range. For fixed variables, specify just the value.
<code>n</code>	Number of points per slice.

**Value**

`slice.design` returns a data frame with `n` points per slice. The column `slice` is a factor that tells which slice each point belongs to.

**Author(s)**

Aaron A. King (kingaa at umich dot edu)

**Examples**

```
## A single 11-point slice through the point c(A=3,B=8,C=0) along the B direction.
slice.design(list(A=3,B=c(0,8,10),C=0),n=11)
## Two slices through the same point along the A and C directions.
slice.design(list(A=c(0,3,5),B=8,C=c(0,0,5)),n=11)
```

---

sobol

*Sobol' low-discrepancy sequence*

---

**Description**

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

**Usage**

```
sobol(vars, n)
```

**Arguments**

vars	Named list of ranges of variables.
n	Number of vectors requested.

**Value**

sobol Returns a data frame with n 'observations' of the variables in vars.

**Author(s)**

Aaron A. King (kingaa at umich dot edu)

**References**

W. H. Press, S. A. Teukolsky, W. T. Vetterling, & B. P. Flannery, Numerical Recipes in C, Cambridge University Press, 1992

**Examples**

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

---

traj.match	<i>Trajectory matching</i>
------------	----------------------------

---

**Description**

Match trajectories.

**Usage**

```
traj.match(object, start, est, method = "Nelder-Mead", gr = NULL, ...)
```

**Arguments**

object	A pomp object.
start	Initial guess for parameters.
est	Character vector containing the names of parameters to be estimated.
method	One of the optimization methods recognized by <a href="#">optim</a> .
gr	Passed to <a href="#">optim</a> .
...	Arguments that will be passed to <a href="#">optim</a> in the <code>control</code> list.

**Details**

Trajectory matching is accomplished using [optim](#). It is assumed that the process model is deterministic.

**See Also**

[optim](#), [pomp](#), [pomp-class](#)

**Examples**

```
data(ou2)
true.p <- c(
  alpha.1=0.9, alpha.2=0, alpha.3=0, alpha.4=0.99,
  sigma.1=1, sigma.2=0, sigma.3=2,
  tau=1,
  x1.0=50, x2.0=-50
)
simdata <- simulate(ou2, nsim=1, params=true.p, seed=43553)
guess.p <- true.p
guess.p[grep('sigma', names(guess.p))] <- 0 ## make the process model deterministic
res <- traj.match(
  simdata,
  start=guess.p,
  est=c('alpha.1', 'alpha.4', 'x1.0', 'x2.0', 'tau'),
  maxit=2000,
  reltol=1e-8
)
```

---

trajectory-pomp      *Compute trajectories of the deterministic skeleton.*

---

### Description

The method `trajectory` computes a trajectory of the deterministic skeleton of a Markov process. In the case of a discrete-time system, the deterministic skeleton is a map and a trajectory is obtained by iterating the map. In the case of a continuous-time system, the deterministic skeleton is a vectorfield; `trajectory` integrates the vectorfield to obtain a trajectory.

### Usage

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

### Arguments

<code>object</code>	an object of class <code>pomp</code> .
<code>times</code>	a numeric vector containing the times at which a trajectory is desired. The first of these will be the initial time. By default, <code>times=time(object, t0=TRUE)</code> .
<code>params</code>	a rank-2 array of parameters. Each column of <code>params</code> is a distinct parameter vector.
<code>...</code>	at present, these are ignored.

### Details

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

When the skeleton is a vectorfield, `trajectory` integrates it using [lsoda](#).

Unresolved issue: What is the behavior if `type="map"` and `times` are non-integral?

### Value

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

### Author(s)

Aaron A. King (kingaa at umich dot edu)

### See Also

[pomp-class](#), [pomp](#), [lsoda](#)

**Examples**

```

data(euler.sir)
x <- trajectory(euler.sir)
plot(time(euler.sir,t0=TRUE),x["I",1,],type='l',xlab='time',ylab='I')
lines(time(euler.sir,t0=FALSE),diff(x["cases",1,]),col='red')

coef(euler.sir,c("gamma")) <- log(12)
x <- trajectory(euler.sir)
plot(time(euler.sir,t0=TRUE),x["I",1,],type='l',xlab='time',ylab='I')
lines(time(euler.sir,t0=FALSE),diff(x["cases",1,]),col='red')

```

---

 verhulst

*Simple Verhulst-Pearl (logistic) model.*


---

**Description**

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

**Usage**

```
data(verhulst)
```

**Details**

The model is written as an Ito diffusion,  $dn = rn(1 - n/K)dt + \sigma ndW$ , where  $W$  is a Wiener process. It is implemented using the [euler.simulate](#) plug-in.

**See Also**

[pomp-class](#) and the vignettes

**Examples**

```

data(verhulst)
plot(verhulst)
coef(verhulst)
params <- cbind(
  c(n.0=100,K=10000,r=0.2,sigma=0.4,tau=0.1),
  c(n.0=1000,K=11000,r=0.1,sigma=0.4,tau=0.1)
)
x <- simulate(verhulst,params=params,states=TRUE)
matplot(time(verhulst,t0=TRUE),t(x['n',,]),type='l')
y <- trajectory(verhulst,params=params)
matlines(time(verhulst,t0=TRUE),t(y['n',,]),type='l',lwd=2)

```

# Index

- \*Topic **datasets**
  - euler.sir, 9
  - ou2, 18
  - rw2, 34
  - verhulst, 41
- \*Topic **design**
  - slice.design, 37
  - sobol, 38
- \*Topic **distribution**
  - Euler-multinomial models, 8
- \*Topic **models**
  - dmeasure-pomp, 4
  - dprocess-pomp, 5
  - euler, 6
  - init.state-pomp, 10
  - mif, 11
  - mif-class, 13
  - mif-methods, 15
  - nlf, 16
  - particles-mif, 19
  - pfilter, 20
  - pomp, 22
  - pomp-class, 28
  - pomp-methods, 29
  - pomp-package, 2
  - rmeasure-pomp, 31
  - rprocess-pomp, 32
  - simulate-pomp, 34
  - skeleton-pomp, 36
  - traj.match, 38
  - trajectory-pomp, 39
- \*Topic **smooth**
  - B-splines, 3
- \*Topic **ts**
  - dmeasure-pomp, 4
  - dprocess-pomp, 5
  - init.state-pomp, 10
  - mif, 11
  - mif-class, 13
  - mif-methods, 15
  - nlf, 16
  - particles-mif, 19
  - pfilter, 20
  - pomp, 22
  - pomp-class, 28
  - pomp-methods, 29
  - pomp-package, 2
  - rmeasure-pomp, 31
  - rprocess-pomp, 32
  - simulate-pomp, 34
  - skeleton-pomp, 36
  - traj.match, 38
  - trajectory-pomp, 39
- as, pomp-method (*pomp-methods*), 29
- B-splines, 3
- bspline.basis (*B-splines*), 3
- coef, pomp-method (*pomp-methods*), 29
- coef-pomp (*pomp-methods*), 29
- coef<- (*pomp-methods*), 29
- coef<-, pomp-method (*pomp-methods*), 29
- coef<-pomp (*pomp-methods*), 29
- coerce, pomp, data.frame-method (*pomp-methods*), 29
- compare.mif (*mif-methods*), 15
- continue (*mif*), 11
- continue, mif-method (*mif*), 11
- continue-mif (*mif*), 11
- conv.rec (*mif-methods*), 15
- conv.rec, mif-method (*mif-methods*), 15
- conv.rec-mif (*mif-methods*), 15
- data.array (*pomp-methods*), 29
- data.array, pomp-method (*pomp-methods*), 29

- data.array-pomp (*pomp-methods*), 29
- deulermultinom
  - (*Euler-multinomial models*), 8
- dmeasure, 26, 29, 31
- dmeasure (*dmeasure-pomp*), 4
- dmeasure, pomp-method
  - (*dmeasure-pomp*), 4
- dmeasure-pomp, 4, 31
- dprocess, 26, 29, 31
- dprocess (*dprocess-pomp*), 5
- dprocess, pomp-method
  - (*dprocess-pomp*), 5
- dprocess-pomp, 5, 31
- euler, 6, 9, 26
- Euler-multinomial models, 8
- euler.simulate, 24, 41
- euler.sir, 9
- eulermultinom, 8
- eulermultinom (*Euler-multinomial models*), 8
- filter.mean (*mif-methods*), 15
- filter.mean, mif-method
  - (*mif-methods*), 15
- filter.mean-mif (*mif-methods*), 15
- init.state, 26, 29, 31
- init.state (*init.state-pomp*), 10
- init.state, pomp-method
  - (*init.state-pomp*), 10
- init.state-pomp, 23
- init.state-pomp, 10, 31
- logLik, mif-method (*mif-methods*), 15
- logLik-mif (*mif-methods*), 15
- lsoda, 40
- mif, 2, 11, 13–16, 19, 20
- mif, mif-method (*mif*), 11
- mif, pomp-method (*mif*), 11
- mif-class, 13
- mif-methods, 13
- mif-class, 13
- mif-methods, 14, 15, 20
- mif-mif (*mif*), 11
- mif-pomp (*mif*), 11
- nlf, 2, 16
- onestep.density, 24
- onestep.density (*euler*), 6
- onestep.simulate, 24
- onestep.simulate (*euler*), 6
- optim, 17, 39
- ou2, 18
- particles (*particles-mif*), 19
- particles, mif-method
  - (*particles-mif*), 19
- particles-mif, 14, 19
- periodic.bspline.basis
  - (*B-splines*), 3
- pfilter, 2, 13, 14, 16, 20
- pfilter, mif-method (*pfilter*), 20
- pfilter, pomp-method (*pfilter*), 20
- pfilter-mif, 14, 16
- pfilter-mif (*pfilter*), 20
- pfilter-pomp (*pfilter*), 20
- plot, mif-method (*mif-methods*), 15
- plot, pomp-method (*pomp-methods*), 29
- plot-mif (*mif-methods*), 15
- plot-pomp (*pomp-methods*), 29
- pomp, 2, 4–8, 10, 13, 14, 16, 20, 22, 28, 29, 31–33, 36, 37, 39, 40
- pomp-class, 5, 6, 9, 10, 12–14, 16, 19, 32–34, 37, 39–41
- pomp-class, 14, 20, 21, 26, 28, 31, 35
- pomp-methods, 26, 29, 29
- pomp-package, 2
- pred.mean (*mif-methods*), 15
- pred.mean, mif-method
  - (*mif-methods*), 15
- pred.mean-mif (*mif-methods*), 15
- pred.var (*mif-methods*), 15
- pred.var, mif-method
  - (*mif-methods*), 15
- pred.var-mif (*mif-methods*), 15
- print, pomp-method (*pomp-methods*), 29
- print-pomp (*pomp-methods*), 29
- reulermultinom
  - (*Euler-multinomial models*), 8
- rmeasure, 26, 29, 31

rmeasure (*rmeasure-pomp*), 31  
rmeasure, pomp-method  
    (*rmeasure-pomp*), 31  
rmeasure-pomp, 31, 31  
rprocess, 26, 29, 31  
rprocess (*rprocess-pomp*), 32  
rprocess, pomp-method  
    (*rprocess-pomp*), 32  
rprocess-pomp, 31, 32  
rw2, 34

show, pomp-method (*pomp-methods*),  
    29  
show-pomp (*pomp-methods*), 29  
simulate, pomp-method  
    (*simulate-pomp*), 34  
simulate-pomp, 2  
simulate-pomp, 29, 31, 34  
skeleton, 26  
skeleton (*skeleton-pomp*), 36  
skeleton, pomp-method  
    (*skeleton-pomp*), 36  
skeleton-pomp, 31, 36  
slice.design, 37  
sobol, 38  
states (*pomp-methods*), 29  
states, pomp-method  
    (*pomp-methods*), 29  
states-pomp (*pomp-methods*), 29  
subplex, 17

time, pomp-method (*pomp-methods*),  
    29  
time-pomp (*pomp-methods*), 29  
time<- (*pomp-methods*), 29  
time<-, pomp-method  
    (*pomp-methods*), 29  
time<-pomp (*pomp-methods*), 29  
traj.match, 38  
trajectory, 36, 37  
trajectory (*trajectory-pomp*), 39  
trajectory, pomp-method  
    (*trajectory-pomp*), 39  
trajectory-pomp, 39

verhulst, 41