Package: mcstate (via r-universe)

September 29, 2024

Title Monte Carlo Methods for State Space Models

Version 0.9.22

Description Implements Monte Carlo methods for state-space models such
as 'SIR' models in epidemiology. Particle MCMC (pmcmc) and SMC2
methods are planned. This package is particularly designed to
work with odin/dust models, but we will see how general it
becomes.

License MIT + file LICENSE

Encoding UTF-8

Language en-GB

URL https://github.com/mrc-ide/mcstate

BugReports https://github.com/mrc-ide/mcstate/issues

Imports R6, callr (>= 3.7.0), dust (>= 0.13.12), processx, progress
(>= 1.2.0)

Suggests brio, coda, decor, fs, knitr, mockery, mvtnorm, odin.dust (>=
0.3.0), rmarkdown, testthat, withr

RoxygenNote 7.3.1

Roxygen list(markdown = TRUE)

Remotes mrc-ide/dust, mrc-ide/odin.dust

VignetteBuilder knitr

Repository https://mrc-ide.r-universe.dev

RemoteUrl https://github.com/mrc-ide/mcstate

RemoteRef master

RemoteSha 3549d64ff9b0d3aac1f9eec435093fc51f9a32cc

Contents

---

adaptive_proposal_control

Adaptive proposal control

Description

Control for adaptive proposals, used in [pmcmc_control](#) for deterministic models.

Usage

```
adaptive_proposal_control(
  initial_vcv_weight = 1000,
  initial_scaling = 1,
  initial_scaling_weight = NULL,
  min_scaling = 0,
  scaling_increment = NULL,
```

```
  log_scaling_update = TRUE,
  acceptance_target = 0.234,
  forget_rate = 0.2,
  forget_end = Inf,
  adapt_end = Inf,
  pre_diminish = 0
)
```

Arguments

initial_vcv_weight

Weight of the initial variance-covariance matrix used to build the proposal of the random-walk. Higher values translate into higher confidence of the initial variance-covariance matrix and means that update from additional samples will be slower.

initial_scaling

The initial scaling of the variance covariance matrix to be used to generate the multivariate normal proposal for the random-walk Metropolis-Hastings algorithm. To generate the proposal matrix, the weighted variance covariance matrix is multiplied by the scaling parameter squared times 2.38^2 / n_pars (where n_pars is the number of fitted parameters). Thus, in a Gaussian target parameter space, the optimal scaling will be around 1.

initial_scaling_weight

The initial weight used in the scaling update. The scaling weight will increase after the first `pre_diminish` iterations, and as the scaling weight increases the adaptation of the scaling diminishes. If `NULL` (the default) the value is 5 / (acceptance_target * (1 - acceptance_target)).

min_scaling   The minimum scaling of the variance covariance matrix to be used to generate the multivariate normal proposal for the random-walk Metropolis-Hastings algorithm.

scaling_increment

The scaling increment which is added or subtracted to the scaling factor of the variance-covariance after each adaptive step. If `NULL` (the default) then an optimal value will be calculated.

log_scaling_update

Logical, whether or not changes to the scaling parameter are made on the log-scale.

acceptance_target

The target for the fraction of proposals that should be accepted (optimally) for the adaptive part of the mixture model.

forget_rate   The rate of forgetting early parameter sets from the empirical variance-covariance matrix in the MCMC chains. For example, `forget_rate = 0.2` (the default) means that once in every 5th iterations we remove the earliest parameter set included, so would remove the 1st parameter set on the 5th update, the 2nd on the 10th update, and so on. Setting `forget_rate = 0` means early parameter sets are never forgotten.

forget_end   The final iteration at which early parameter sets can be forgotten. Setting `forget_rate = Inf` (the default) means that the forgetting mechanism continues throughout

the chains. Forgetting early parameter sets becomes less useful once the chains have settled into the posterior mode, so this parameter might be set as an estimate of how long that would take.

adapt_end   The final iteration at which we can adapt the multivariate normal proposal. Thereafter the empirical variance-covariance matrix, its scaling and its weight remain fixed. This allows the adaptation to be switched off at a certain point to help ensure convergence of the chain.

pre_diminish   The number of updates before adaptation of the scaling parameter starts to diminish. Setting `pre_diminish = 0` means there is diminishing adaptation of the scaling parameter from the offset, while `pre_diminish = Inf` would mean there is never diminishing adaptation. Diminishing adaptation should help the scaling parameter to converge better, but while the chains find the location and scale of the posterior mode it might be useful to explore with it switched off.

Details

Efficient exploration of the parameter space during an MCMC might be difficult when the target distribution is of high dimensionality, especially if the target probability distribution present a high degree of correlation. Adaptive schemes are used to "learn" on the fly the correlation structure by updating the proposal distribution by recalculating the empirical variance-covariance matrix and rescale it at each adaptive step of the MCMC.

Our implementation of an adaptive MCMC algorithm is based on an adaptation of the "accelerated shaping" algorithm in Spencer (2021). The algorithm is based on a random-walk Metropolis-Hasting algorithm where the proposal is a multi-variate Normal distribution centered on the current point.

Spencer SEF (2021) Accelerating adaptation in the adaptive Metropolis–Hastings random walk algorithm. Australian & New Zealand Journal of Statistics 63:468-484.

---

array_bind   *Bind arrays*

Description

Bind a number of arrays, usually by their last dimension. This is useful for binding together the sorts of arrays produced by dust and mcstate's simulation functions.

Usage

```
array_bind(..., arrays = list(...), dimension = NULL)
```

Arguments

...   Any number of arrays. All dimensions must the the same, except for the dimension being bound on which may vary.

arrays   As an alternative to using `...` you can provide a list directly. This is often nicer to program with.

dimension   The dimension to bind on; by default `NULL` means the last dimension.

Value

A single array object

Examples

```
# Consider two matricies; this is equivalent to rbind and is
# pretty trivial
m1 <- matrix(1, 4, 5)
m2 <- matrix(2, 4, 2)
mcstate::array_bind(m1, m2)

# For a 4d array though it's less obvious
a1 <- array(1, c(2, 3, 4, 5))
a2 <- array(2, c(2, 3, 4, 1))
a3 <- array(3, c(2, 3, 4, 3))
dim(mcstate::array_bind(a1, a2, a3))
```

---

array_drop   *Drop specific array dimensions*

Description

Drop specific array dimensions that are equal to 1. This a more explicit, safer version of drop, which requires you indicate which dimensions will be dropped and errors if dimensions can't be dropped.

Usage

```
array_drop(x, i)
```

Arguments

x	An array
i	Index or indices of dimensions to remove

Value

An array

Examples

```
# Suppose we have an array with a redundant 2nd dimension
m <- array(1:25, c(5, 1, 5))

# commonly we might drop this with
drop(m)

# in this case, array_drop is the same:
mcstate::array_drop(m, 2)
```

```
# However, suppose that our matrix had, in this case, a first
# dimension that was also 1 but we did not want to drop it:
m2 <- m[1, , , drop = FALSE]

# Here, drop(m2) returns just a vector, discarding our first dimension
drop(m2)

# However, array_drop will preserve that dimension
mcstate::array_drop(m2, 2)
```

---

array_flatten   *Flatten array dimensions*

Description

Flatten array dimensions into a single dimension. This takes a multidimensional array and converts some dimensions of it into a vector. Use this to drop out "middle" dimensions of a structured array. This is conceptually the inverse of array_reshape

Usage

```
array_flatten(x, i)
```

Arguments

x	An array
i	An integer vector of dimensions to flatten

Value

A new array with at one or more dimensions removed

See Also

array_flatten which adds structure

Examples

```
x <- array(1:12, c(2, 3, 4))
mcstate::array_flatten(x, 2:3)

# array_flatten and array_reshape are each others' conceptual
# opposites:
y <- mcstate::array_flatten(x, 2:3)
identical(mcstate::array_reshape(y, 2, c(3, 4)), x)
```

---

array_reshape   *Rehape an array dimension*

Description

Reshape one dimension of a multidimensional array. Use this to say that some dimension (say with length 20) actually represents a number of other dimensions (e.g., 2 x 10 or 2 x 2 x 5). This might be the case if you've been doing a simulation with a large number of parameter sets that are pooled over some other grouping factors (e.g., in a sensitivity analysis)

Usage

```
array_reshape(x, i, d)
```

Arguments

x	An array
i	The index of the dimension to expand
d	The new dimensions for data in the i'th dimension of x

Value

A multidimensional array

See Also

[array_flatten](#) which undoes this operation

Examples

```
# Suppose we had a 4 x 6 array of data:
m <- matrix(1:24, 4, 6)

# And suppose that the second dimension really represented a 2 x 3
# matrix; so that looking at one copy of the 2nd dimension we see
m[1, ]

# But instead we might want to see
res <- mcstate::array_reshape(m, 2, c(2, 3))
res[1, , ]
```

---

if2   *Run iterated filtering (IF2 algorithm)*

Description

Create an IF2 object for running and interacting with an IF2 inference.

Usage

```
if2(pars, filter, control)

if2_sample(obj, n_particles)
```

Arguments

pars	An if2_parameters object, describing the parameters that will be varied in the simulation, and the method of transformation into model parameters.
filter	A particle_filter object. We don't use the particle filter directly (except for sampling with mcstate::if2_sample) but this shares so much validation that it's convenient. Be sure to set things like the seed and number of threads here if you want to use anything other than the default.
control	An if2_control() object
obj	An object of class if2_fit, returned by mcstate::if2()
n_particles	The number of particles to simulate, for each IF2 parameter set

Details

See: Ionides EL, Nguyen D, Atchadé Y, Stoev S, King AA (2015). "Inference for Dynamic and Latent Variable Models via Iterated, Perturbed Bayes Maps." PNAS, 112(3), 719–724. https://doi.org/10.1073/pnas.1410597112

Value

An object of class if2_fit, which contains the sampled parameters (over time) and their log-likelihoods

Examples

```
# A basic SIR model used in the particle filter example
gen <- dust::dust_example("sir")

# Some data that we will fit to, using 1 particle:
sir <- gen$new(pars = list(), time = 0, n_particles = 1)
dt <- 1 / 4
day <- seq(1, 100)
incidence <- rep(NA, length(day))
true_history <- array(NA_real_, c(5, 1, 101))
true_history[, 1, 1] <- sir$state()
```

```
for (i in day) {
  state_start <- sir$state()
  sir$run(i / dt)
  state_end <- sir$state()
  true_history[, 1, i + 1] <- state_end
  # Reduction in S
  incidence[i] <- state_start[1, 1] - state_end[1, 1]
}

# Convert this into our required format:
data_raw <- data.frame(day = day, incidence = incidence)
data <- particle_filter_data(data_raw, "day", 4, 0)

# A comparison function
compare <- function(state, observed, pars = NULL) {
  if (is.null(pars$exp_noise)) {
    exp_noise <- 1e6
  } else {
    exp_noise <- pars$exp_noise
  }
  incidence_modelled <- state[1,]
  incidence_observed <- observed$incidence
  lambda <- incidence_modelled +
    rexp(length(incidence_modelled), exp_noise)
  dpois(incidence_observed, lambda, log = TRUE)
}

# Range and initial values for model parameters
pars <- mcstate::if2_parameters$new(
  list(mcstate::if2_parameter("beta", 0.15, min = 0, max = 1),
       mcstate::if2_parameter("gamma", 0.05, min = 0, max = 1)))

# Set up of IF2 algorithm (the iterations and n_par_sets should be
# increased here for any real use)
control <- mcstate::if2_control(
  pars_sd = list("beta" = 0.02, "gamma" = 0.02),
  iterations = 10,
  n_par_sets = 40,
  cooling_target = 0.5,
  progress = interactive())

# Create a particle filter object
filter <- mcstate::particle_filter$new(data, gen, 1L, compare)

# Then run the IF2
res <- mcstate::if2(pars, filter, control)

# Get log-likelihood estimates from running a particle filter at
# each final parameter estimate
ll_samples <- mcstate::if2_sample(res, 20)
```

`if2_control`   *Control for IF2*

Description

Control for `if2()`. This function constructs a list of options and does some basic validation. Do not manually change the values in this object. Do not refer to any argument by position as the order of the arguments may change in future.

Usage

```
if2_control(pars_sd, iterations, n_par_sets, cooling_target, progress = TRUE)
```

Arguments

`pars_sd`	The initial standard deviation of parameter walks.
`iterations`	The number of IF2 iterations to run, across which the cooling is performed
`n_par_sets`	The number of parameter sets to walk (c.f. the population size)
`cooling_target`	A factor < 1 multiplying pars_sd, which will be reached by the end of the iterations, and approached geometrically
`progress`	Logical, indicating if a progress bar should be displayed, using `progress::progress_bar`.

Value

An `if2_control` object, which should not be modified once created. Pass this into `if2()`

Examples

```
mcstate::if2_control(list(beta = 0.2, gamma = 0.2), 100, 1000, 0.5)
```

`if2_parameter`   *Describe single IF2 parameter*

Description

Describe a single parameter for use within IF2. Note that the name is not set here, but will end up being naturally defined when used with `if2_parameters`, which collects these together for use with `if2()`.

Usage

```
if2_parameter(
  name,
  initial,
  min = -Inf,
  max = Inf,
  discrete,
  integer = FALSE,
  prior = NULL
)
```

Arguments

name	Name for the parameter (a string)
initial	Initial value of the parameter
min	Optional minimum value for the parameter (otherwise `-Inf`). If given, then `initial` must be at least this value.
max	Optional max value for the parameter (otherwise `Inf`). If given, then `initial` must be at most this value.
discrete	Deprecated; use `integer` instead.
integer	Logical, indicating if this parameter is integer. If `TRUE` then the parameter will be rounded after a new parameter is proposed.
prior	A prior function (if not given an improper flat prior is used - be careful!). It must be a function that takes a single argument, being the value of this parameter. If given, then `prior(initial)` must evaluate to a finite value.

Examples

```
mcstate::if2_parameter("a", 0.1)
```

---

if2_parameters   *if2_parameters*

Description

Construct parameters for use with [if2()](). This creates a utility object that is used internally to work with parameters. Most users only need to construct this object, but see the examples for how it can be used.

Methods

Public methods:

- [if2_parameters$new()]()
- [if2_parameters$initial()]()

- if2_parameters$walk_initialise()
- if2_parameters$walk()
- if2_parameters$names()
- if2_parameters$summary()
- if2_parameters$prior()
- if2_parameters$model()

Method new(): Create the if2_parameters object

Usage:

if2_parameters$new(parameters, transform = NULL)

Arguments:

parameters A list of if2_parameter objects, each of which describe a single parameter in your model. If parameters is named, then these names must match the $name element of each parameter is used (this is verified).

transform An optional transformation function to apply to your parameter vector immediately before passing it to the model function. If not given, then as.list is used, as dust models require this. However, if you need to generate derived parameters from those being actively sampled you can do arbitrary transformations here.

Method initial(): Return the initial parameter values as a named numeric vector

Usage:

if2_parameters$initial()

Method walk_initialise(): Set up a parameter walk

Usage:

if2_parameters$walk_initialise(n_par_sets, pars_sd)

Arguments:

n_par_sets An integer number of parameter sets, which defines the size of the population being peturbed.

pars_sd A vector of standard deviations for the walk of each parameter

Method walk(): Propose a new parameter matrix given a current matrix and walk standard deviation vector.

Usage:

if2_parameters$walk(pars, pars_sd)

Arguments:

pars A parameter matrix, from this function or $walk_initialise()

pars_sd A vector of standard deviations for the walk of each parameter

Method names(): Return the names of the parameters

Usage:

if2_parameters$names()

Method summary(): Return a data.frame with information about parameters (name, min, max, and integer).

Usage:
```
if2_parameters$summary()
```

Method `prior()`: Compute the prior for a parameter vector

Usage:
```
if2_parameters$prior(pars)
```

Arguments:

pars a parameter matrix from `$walk()`

Method `model()`: Apply the model transformation function to a parameter vector. Output is a list for lists, suitable for use with a dust object with `pars_multi = TRUE`

Usage:
```
if2_parameters$model(pars)
```

Arguments:

pars a parameter matrix from `$walk()`

Examples

```
# Construct an object with two parameters:
pars <- mcstate::if2_parameters$new(
  list(mcstate::if2_parameter("a", 0.1, min = 0, max = 1,
                              prior = function(a) log(a)),
       mcstate::if2_parameter("b", 0, prior = dnorm)))

# Initial parameters
pars$initial()

# Create the initial parameter set
n_par_sets <- 5
pars_sd <- list("a" = 0.02, "b" = 0.02)
p_mat <- pars$walk_initialise(n_par_sets, pars_sd)
p_mat

# Propose a new parameter set
p_mat <- pars$walk(p_mat, pars_sd)
p_mat

# Information about parameters:
pars$names()
pars$summary()

# Compute prior
pars$prior(p_mat)

# Transform data for your model
pars$model(p_mat)
```

---

multistage_epoch   *Multistage filter epoch*

Description

Describe an epoch within a multistage_parameters object

Usage

```
multistage_epoch(start, pars = NULL, transform_state = NULL)
```

Arguments

start
: The start time, in units of time in your data set. These must correspond to time points with data. The model will complete the step to this time point, then change parameters, then continue (so start represents the time point we move from with these parameters)

pars
: Optional parameter object, replacing the model parameters at this point. If NULL then the model parameters are not changed, and it is assumed that you will be changing model state via transform_state.

transform_state
: Optional parameter transformation function. This could be used in two cases (1) arbitrary change to the model state (e.g., a one-off movement of state within particles at a given time point that would be otherwise awkward to code directly in your model), or (2) where you have provided pars and these imply a different model state size. In this case you *must* provide transform_state to fill in new model state, move things around, or delete model state depending on how the state has changed. This function will be passed three arguments: (1) the current model state, (2) the result of the $info() method from the model used to this point, (3) the result of the $info() method for the new model that was created with pars which will be run from this point. Future versions of this interface may allow passing the parameters in too.

---

multistage_parameters *Multistage filter parameters*

Description

Construct parameters for a multi-stage particle filter.

Usage

```
multistage_parameters(pars, epochs)
```

Arguments

pars	The parameters covering the period up to the first change in epoch.
epochs	A list of multistage_epoch objects corresponding to a new paramter regime starting at a new time point.

Value

An object of class multistage_parameters, suitable to pass through to the run method of particle_filter

---

particle_deterministic

Deterministic particle likelihood

Description

Create a deterministic version of the particle_filter object, which runs a single particle deterministically.

Public fields

model The dust model generator being simulated (cannot be re-bound)

has_multiple_parameters Logical, indicating if the deterministic particle requires multiple parameter sets in a list as inputs, and if it it will produce a vector of likelihoods the same length (read only). The parameter sets may or may not use the same data (see has_multiple_data).

has_multiple_data Logical, indicating if the deterministic particle simultaneously calculates the likelihood for multiple parameter sets (read only). If TRUE, has_multiple_parameters will always be TRUE.

n_parameters The number of parameter sets used by this deterministic particle (read only). The returned vector of likelihoods will be this length, and if has_multiple_parameters is FALSE this will be 1.

n_data The number of data sets used by this deterministic particle (read only). This will either be 1 or the same value as n_parameters.

Methods

Public methods:

- particle_deterministic$new()
- particle_deterministic$run()
- particle_deterministic$run_begin()
- particle_deterministic$state()
- particle_deterministic$history()
- particle_deterministic$restart_state()
- particle_deterministic$inputs()
- particle_deterministic$set_n_threads()

Method new(): Create the particle filter

Usage:
```
particle_deterministic$new(
  data,
  model,
  compare,
  index = NULL,
  initial = NULL,
  constant_log_likelihood = NULL,
  n_threads = 1L,
  n_parameters = NULL,
  stochastic_schedule = NULL,
  ode_control = NULL
)
```

Arguments:

data The data set to be used for the particle filter, created by [particle_filter_data()](). This
   is essentially a [data.frame()]() with at least columns time_start and time_end, along
   with any additional data used in the compare function, and additional information about
   how your steps relate to time.

model A stochastic model to use. Must be a dust_generator object.

compare A comparison function. Must take arguments state, observed and pars as argu-
   ments (though the arguments may have different names). state is the simulated model
   state (a matrix with as many rows as there are state variables and as many columns as there
   are particles, data is a list of observed data corresponding to the current time's row in
   the data object provided here in the constructor. pars is any additional parameters passed
   through to the comparison function (via the pars argument to $run).

index An index function. This is used to compute the "interesting" indexes of your model. It
   must be a function of one argument, which will be the result of calling the $info() method
   on your model. It should return a list with elements run (indices to return at the end of
   each run, passed through to your compare function) and state (indices to return if saving
   state). These indices can overlap but do not have to. This argument is optional but using it
   will likely speed up your simulation if you have more than a few states as it will reduce the
   amount of memory copied back and forth.

initial A function to generate initial conditions. If given, then this function must accept 3
   arguments: info (the result of calling $info() as for index), n_particles (the number of
   particles that the particle filter is using) and pars (parameters passed in in the $run method
   via the pars argument). It must return a list, which can have the elements state (initial
   model state, passed to the particle filter - either a vector or a matrix, and overriding the initial
   conditions provided by your model) and time (the initial time, overriding the first time step
   of your data - this must occur within your first epoch in your data provided to the construc-
   tor, i.e., not less than the first element of time_start and not more than time_end). Your
   function can also return a vector or matrix of state and not alter the starting time, which
   is equivalent to returning list(state = state, time = NULL). (TODO: this no longer is
   allowed, and the docs might be out of date?)

constant_log_likelihood An optional function, taking the model parameters, that computes
   the constant part of the log-likelihood value (if any). You can use this where your likelihood
   depends both on the time series (via data) but also on some non-temporal data. You should

bind any non-parameter dependencies into this closure. This is applied at the beginning of the filter run, so represents the initial condition of the marginal log likelihood value propagated by the process.

n_threads Number of threads to use when running the simulation. Defaults to 1, and should not be set higher than the number of cores available to the machine. This currently has no effect as the simulation will be run in serial on a single particle for now.

n_parameters Number of parameter sets required. This, along with data, controls the interpretation of how the deterministic particle, and importantly will add an additional dimension to most outputs (scalars become vectors, vectors become matrices etc).

stochastic_schedule Vector of times to perform stochastic updates, for continuous time models. Note that despite the name, these will be applied deterministically (i.e., replacing the stochastic draw with its expectation).

ode_control Tuning control for the ODE stepper, for continuous time (ODE) models

Method run(): Run the deterministic particle filter

Usage:
```
particle_deterministic$run(
  pars = list(),
  save_history = FALSE,
  save_restart = NULL,
  min_log_likelihood = -Inf
)
```

Arguments:

pars A list representing parameters. This will be passed as the pars argument to your model, to your compare function, and (if using) to your initial function. It must be an R list (not vector or NULL) because that is what a dust model currently requires on initialisation or $reset - we may relax this later. You may want to put your observation and initial parameters under their own keys (e.g., pars$initial$whatever), but this is up to you. Extra keys are silently ignored by dust models.

save_history Logical, indicating if the history of all particles should be saved. If saving history, then it can be queried later with the $history method on the object.

save_restart An integer vector of time points to save restart infomation for. Not currently supported.

min_log_likelihood Not currently supported, exists to match the inteface with particle_filter. Providing a value larger than -Inf will cause an error.

Returns: A single numeric value representing the log-likelihood (-Inf if the model is impossible)

Method run_begin(): Begin a deterministic run. This is part of the "advanced" interface; typically you will want to use $run() which provides a user-facing wrapper around this function. Once created with $run_begin(), you should take as many steps as needed with $step().

Usage:
```
particle_deterministic$run_begin(
  pars,
  save_history = FALSE,
  save_restart = NULL,
```

```
    min_log_likelihood = -Inf
)
```

Arguments:

pars A list representing parameters. See $run_many() for details (and *not* $run())

save_history Logical, indicating if the history of all particles should be saved. See $run()
    for details.

save_restart Times to save restart state at. See $run() for details.

min_log_likelihood Not currently supported, exists to match the inteface with [particle_filter](#).
    Providing a value larger than -Inf will cause an error.

Returns: An object of class particle_deterministic_state, with methods step and end.
This interface is still subject to change.

Method state(): Extract the current model state, optionally filtering. If the model has not yet
been run, then this method will throw an error. Returns a matrix with the number of rows being
the number of model states, and the number of columns being the number of particles.

Usage:
```
particle_deterministic$state(index_state = NULL)
```

Arguments:

index_state Optional vector of states to extract

Method history(): Extract the particle trajectories. Requires that the model was run with
save_history = TRUE, which does incur a performance cost. This method will throw an error if
the model has not run, or was run without save_history = TRUE. Returns a 3d array with dimen-
sions corresponding to (1) model state, filtered by index$run if provided, (2) particle (following
index_particle if provided), (3) time point.

Usage:
```
particle_deterministic$history(index_particle = NULL)
```

Arguments:

index_particle Optional vector of particle indices to return. If NULL we return all particles'
    histories.

Method restart_state(): Return the full particle filter state at points back in time that were
saved with the save_restart argument to $run(). If available, this will return a 3d array, with
dimensions representing (1) particle state, (2) particle index, (3) time point. If multiple parameters
are used then returns a 4d array, with dimensions representing (1) particle state, (2) particle index,
(3) parameter, (4) time point. This could be quite large, especially if you are using the index
argument to create the particle filter and return a subset of all state generally. In the stochastic
version, this is different the saved trajectories returned by $history() because earlier saved state
is not filtered by later filtering, but in the deterministic model we run with a single particle so it *is*
the same.

Usage:
```
particle_deterministic$restart_state(
  index_particle = NULL,
  save_restart = NULL,
  restart_match = FALSE
)
```

Arguments:

index_particle Optional vector of particle indices to return. If NULL we return all particles' states. Practically because the only valid value of index_particle is "1", this has no effect and it is included primarily for compatibility with the stochastic interface.

Method inputs(): Return a list of inputs used to configure the deterministic particle filter. These correspond directly to the argument names for the constructor and are the same as the input arguments.

Usage:

```
particle_deterministic$inputs()
```

Method set_n_threads(): Set the number of threads used by the particle filter (and dust model) after creation. This can be used to allocate additional (or subtract excess) computing power from the deterministic filter Returns (invisibly) the previous value.

Usage:

```
particle_deterministic$set_n_threads(n_threads)
```

Arguments:

n_threads The new number of threads to use. You may want to wrap this argument in dust::dust_openmp_threads() in order to verify that you can actually use the number of threads requested (based on environment variables and OpenMP support).

---

particle_deterministic_state

Deterministic particle state

Description

Deterministic particle internal state. This object is not ordinarily constructed directly by users, but via the $run_begin method to particle_deterministic. It provides an advanced interface to the deterministic particle that allows partially running over part of the time trajectory.

This state object has a number of public fields that you can read but must not write (they are not read-only so you *could* write them, but don't).

Public fields

model The dust model being simulated

history The particle history, if created with save_history = TRUE.

restart_state Full model state at a series of points in time, if the model was created with non-NULL save_restart. This is a 3d array as described in particle_filter

log_likelihood The log-likelihood so far. This starts at 0 when initialised and accumulates value for each step taken.

current_time_index The index of the last completed step.

Methods

Public methods:

- `particle_deterministic_state$new()`
- `particle_deterministic_state$run()`
- `particle_deterministic_state$step()`
- `particle_deterministic_state$fork_multistage()`

Method `new()`: Initialise the deterministic particle state. Ordinarily this should not be called by users, and so arguments are barely documented.

Usage:

```
particle_deterministic_state$new(
  pars,
  generator,
  model,
  data,
  data_split,
  times,
  has_multiple_parameters,
  n_threads,
  initial,
  index,
  compare,
  constant_log_likelihood,
  save_history,
  save_restart,
  stochastic_schedule,
  ode_control
)
```

Arguments:

`pars` Parameters for a single phase

`generator` A dust generator object

`model` If the generator has previously been initialised

`data` A particle_filter_data data object

`data_split` The same data as `data` but split by step

`times` A matrix of time step beginning and ends

`has_multiple_parameters` Compute multiple likelihoods at once?

`n_threads` The number of threads to use

`initial` Initial condition function (or `NULL`)

`index` Index function (or `NULL`)

`compare` Compare function

`constant_log_likelihood` Constant log likelihood function

`save_history` Logical, indicating if we should save history

`save_restart` Vector of time steps to save restart at

`stochastic_schedule` Vector of times to perform stochastic updates

ode_control Tuning control for stepper

Method run(): Run the deterministic particle to the end of the data. This is a convenience function around $step() which provides the correct value of time_index

Usage:

particle_deterministic_state$run()

Method step(): Take a step with the deterministic particle. This moves the system forward one step within the *data* (which may correspond to more than one step with your model) and returns the likelihood so far.

Usage:

particle_deterministic_state$step(time_index)

Arguments:

time_index The step *index* to move to. This is not the same as the model step, nor time, so be careful (it's the index within the data provided to the filter). It is an error to provide a value here that is lower than the current step index, or past the end of the data.

Method fork_multistage(): Create a new deterministic_particle_state object based on this one (same model, position in time within the data) but with new parameters, to support the "multistage particle filter".

Usage:

particle_deterministic_state$fork_multistage(model, pars, transform_state)

Arguments:

model A model object

pars New model parameters

transform_state A function to transform the model state from the old to the new parameter set. See multistage_epoch() for details.

---

particle_filter *Particle filter*

Description

Create a particle_filter object for running and interacting with a particle filter. A higher-level interface will be implemented later.

Public fields

model The dust model generator being simulated (cannot be re-bound)

n_particles Number of particles used (read only)

has_multiple_parameters Logical, indicating if the particle filter requires multiple parameter sets in a list as inputs, and if it it will produce a vector of likelihoods the same length (read only). The parameter sets may or may not use the same data (see has_multiple_data).

has_multiple_data Logical, indicating if the particle filter simultaneously calculates the likelihood for multiple parameter sets (read only). If TRUE, has_multiple_parameters will always be TRUE.

n_parameters The number of parameter sets used by this particle filter (read only). The returned vector of likelihood will be this length, and if has_multiple_parameters is FALSE this will be 1.

n_data The number of data sets used by this particle filter (read only). This will either be 1 or the same value as n_parameters.

Methods

Public methods:

- particle_filter$new()
- particle_filter$run()
- particle_filter$run_begin()
- particle_filter$state()
- particle_filter$history()
- particle_filter$ode_statistics()
- particle_filter$restart_state()
- particle_filter$inputs()
- particle_filter$set_n_threads()

Method new(): Create the particle filter

Usage:
```
particle_filter$new(
  data,
  model,
  n_particles,
  compare,
  index = NULL,
  initial = NULL,
  constant_log_likelihood = NULL,
  n_threads = 1L,
  seed = NULL,
  n_parameters = NULL,
  gpu_config = NULL,
  stochastic_schedule = NULL,
  ode_control = NULL
)
```

Arguments:

data The data set to be used for the particle filter, created by particle_filter_data(). This is essentially a data.frame() with at least columns time_start and time_end, along with any additional data used in the compare function, and additional information about how your dust time steps relate to a more interpretable measure of model time.

model A stochastic model to use. Must be a dust_generator object.

n_particles The number of particles to simulate

compare A comparison function. Must take arguments state, observed and pars as arguments (though the arguments may have different names). state is the simulated model state (a matrix with as many rows as there are state variables and as many columns as there are particles, data is a list of observed data corresponding to the current time's row in the data object provided here in the constructor. pars is any additional parameters passed through to the comparison function (via the pars argument to $run). Alternatively, compare can be NULL if your model provides a built-in compile compare function (if model$public_methods$has_compare() is TRUE), which may be faster.

index An index function. This is used to compute the "interesting" indexes of your model. It must be a function of one argument, which will be the result of calling the $info() method on your model. It should return a list with elements run (indices to return at the end of each run, passed through to your compare function) and state (indices to return if saving state). These indices can overlap but do not have to. This argument is optional but using it will likely speed up your simulation if you have more than a few states as it will reduce the amount of memory copied back and forth.

initial A function to generate initial conditions. If given, then this function must accept 3 arguments: info (the result of calling $info() as for index), n_particles (the number of particles that the particle filter is using) and pars (parameters passed in in the $run method via the pars argument). It must return a list, which can have the elements state (initial model state, passed to the particle filter - either a vector or a matrix, and overriding the initial conditions provided by your model) and time (the initial time step, overriding the first time step of your data - this must occur within your first epoch in your data provided to the constructor, i.e., not less than the first element of time_start and not more than time_end). Your function can also return a vector or matrix of state and not alter the starting time step, which is equivalent to returning list(state = state, time = NULL).

constant_log_likelihood An optional function, taking the model parameters, that computes the constant part of the log-likelihood value (if any). You can use this where your likelihood depends both on the time series (via data) but also on some non-temporal data. You should bind any non-parameter dependencies into this closure. This is applied at the beginning of the filter run, so represents the initial condition of the marginal log likelihood value propagated by the filter.

n_threads Number of threads to use when running the simulation. Defaults to 1, and should not be set higher than the number of cores available to the machine.

seed Seed for the random number generator on initial creation. Can be NULL (to initialise using R's random number generator), a positive integer, or a raw vector - see dust::dust and dust::dust_rng for more details. Note that the random number stream is unrelated from R's random number generator, except for initialisation with seed = NULL.

n_parameters Number of parameter sets required. This, along with data, controls the interpretation of how the particle filter, and importantly will add an additional dimension to most outputs (scalars become vectors, vectors become matrices etc).

gpu_config GPU configuration, typically an integer indicating the device to use, where the model has GPU support. An error is thrown if the device id given is larger than those reported to be available (note that CUDA numbers devices from 0, so that '0' is the first device, so on). See the method $gpu_info() for available device ids; this can be called before object creation as model$public_methods$gpu_info(). For additional control, provide a list with elements device_id and run_block_size. Further options (and validation) of this list will be added in a future version!

stochastic_schedule Vector of times to perform stochastic updates, for continuous time models.

ode_control Tuning control for the ODE stepper, for continuous time (ODE) models

Method run(): Run the particle filter

Usage:
```
particle_filter$run(
  pars = list(),
  save_history = FALSE,
  save_restart = NULL,
  min_log_likelihood = NULL
)
```

Arguments:

pars A list representing parameters. This will be passed as the pars argument to your model, to your compare function, and (if using) to your initial function. It must be an R list (not vector or NULL) because that is what a dust model currently requires on initialisation or $reset - we may relax this later. You may want to put your observation and initial parameters under their own keys (e.g., pars$initial$whatever), but this is up to you. Extra keys are silently ignored by dust models.

save_history Logical, indicating if the history of all particles should be saved. If saving history, then it can be queried later with the $history method on the object.

save_restart An integer vector of time points to save restart infomation for. These are in terms of your underlying time variable (the time column in [particle_filter_data()](#)) not in terms of time steps. The state will be saved after the particle filtering operation (i.e., at the end of the step).

min_log_likelihood Optionally, a numeric value representing the smallest likelihood we are interested in. If given and the particle filter drops below this number, then we terminate early and return -Inf. In this case, history and final state cannot be returned from the filter. This is primarily intended for use with [pmcmc](#) where we can avoid computing likelihoods that will certainly be rejected. Only suitable for use where log-likelihood increments (with the compare function) are always negative. This is the case if you use a normalised discrete distribution, but not necessarily otherwise. If using a multi-parameter filter this can be a single number (in which case the exit is when the sum of log-likelihoods drops below this threshold) or a vector of numbers the same length as pars (in which case exit occurs when all numbers drop below this threshold).

Returns: A single numeric value representing the log-likelihood (-Inf if the model is impossible)

Method run_begin(): Begin a particle filter run. This is part of the "advanced" interface for the particle filter; typically you will want to use $run() which provides a user-facing wrapper around this function. Once created with $run_begin(), you should take as many steps as needed with $step().

Usage:
```
particle_filter$run_begin(
  pars = list(),
  save_history = FALSE,
```

```
    save_restart = NULL,
    min_log_likelihood = NULL
)
```

Arguments:

pars A list representing parameters. See $run() for details.

save_history Logical, indicating if the history of all particles should be saved. See $run() for details.

save_restart Times to save restart state at. See $run() for details.

min_log_likelihood Optionally, a numeric value representing the smallest likelihood we are interested in. See $run() for details.

Returns: An object of class particle_filter_state, with methods step and end. This interface is still subject to change.

Method state(): Extract the current model state, optionally filtering. If the model has not yet been run, then this method will throw an error. Returns a matrix with the number of rows being the number of model states, and the number of columns being the number of particles.

Usage:
```
particle_filter$state(index_state = NULL)
```

Arguments:

index_state Optional vector of states to extract

Method history(): Extract the particle trajectories. Requires that the model was run with save_history = TRUE, which does incur a performance cost. This method will throw an error if the model has not run, or was run without save_history = TRUE. Returns a 3d array with dimensions corresponding to (1) model state, filtered by index$run if provided, (2) particle (following index_particle if provided), (3) time point. If using a multi-parameter filter then returns a 4d array with dimensions corresponding to (1) model state, (2) particle, (3) parameter, (4) time point.

Usage:
```
particle_filter$history(index_particle = NULL)
```

Arguments:

index_particle Optional vector of particle indices to return. If using a multi-parameter filter then a vector will be replicated to a matrix with number of columns equal to number of parameters, otherwise a matrix can be supplied. If NULL we return all particles' histories.

Method ode_statistics(): Fetch statistics about steps taken during the integration, by calling through to the $statistics() method of the underlying model. This is only available for continuous time (ODE) models, and will error if used with discrete time models.

Usage:
```
particle_filter$ode_statistics()
```

Method restart_state(): Return the full particle filter state at points back in time that were saved with the save_restart argument to $run(). If available, this will return a 3d array, with dimensions representing (1) particle state, (2) particle index, (3) time point. If multiple parameters are used then returns a 4d array, with dimensions representing (1) particle state, (2) particle index, (3) parameter set, (4) time point. This could be quite large, especially if you are using the index

argument to create the particle filter and return a subset of all state generally. It is also different to the saved trajectories returned by `$history()` because earlier saved state is not filtered by later filtering (in the history we return the tree of history representing the histories of the *final* particles, here we are returning all particles at the requested point, regardless if they appear in the set of particles that make it to the end of the simulation).

Usage:
```
particle_filter$restart_state(
  index_particle = NULL,
  save_restart = NULL,
  restart_match = FALSE
)
```

Arguments:

index_particle Optional vector of particle indices to return. If NULL we return all particles' states.

Method `inputs()`: Return a list of inputs used to configure the particle filter. These correspond directly to the argument names for the particle filter constructor and are the same as the input argument with the exception of seed, which is the state of the rng if it has been used (this can be used as a seed to restart the model).

Usage:
```
particle_filter$inputs()
```

Method `set_n_threads()`: Set the number of threads used by the particle filter (and dust model) after creation. This can be used to allocate additional (or subtract excess) computing power from a particle filter. Returns (invisibly) the previous value.

Usage:
```
particle_filter$set_n_threads(n_threads)
```

Arguments:

n_threads The new number of threads to use. You may want to wrap this argument in [dust::dust_openmp_threads()](#) in order to verify that you can actually use the number of threads requested (based on environment variables and OpenMP support).

Examples

```
# A basic SIR model included in the dust package
gen <- dust::dust_example("sir")

# Some data that we will fit to, using 1 particle:
sir <- gen$new(pars = list(), time = 0, n_particles = 1)
dt <- 1 / 4
day <- seq(1, 100)
incidence <- rep(NA, length(day))
true_history <- array(NA_real_, c(5, 1, 101))
true_history[, 1, 1] <- sir$state()
for (i in day) {
  state_start <- sir$state()
  sir$run(i / dt)
  state_end <- sir$state()
```

```
    true_history[, 1, i + 1] <- state_end
    # Reduction in S
    incidence[i] <- state_start[1, 1] - state_end[1, 1]
  }

  # Convert this into our required format:
  data_raw <- data.frame(day = day, incidence = incidence)
  data <- particle_filter_data(data_raw, "day", 4, 0)

  # A comparison function
  compare <- function(state, observed, pars = NULL) {
    if (is.null(pars$exp_noise)) {
      exp_noise <- 1e6
    } else {
      exp_noise <- pars$exp_noise
    }
    incidence_modelled <- state[1,]
    incidence_observed <- observed$incidence
    lambda <- incidence_modelled +
      rexp(length(incidence_modelled), exp_noise)
    dpois(incidence_observed, lambda, log = TRUE)
  }

  # Construct the particle_filter object with 100 particles
  p <- particle_filter$new(data, gen, 100, compare)
  p$run(save_history = TRUE)

  # Our simulated trajectories, with the "real" data superimposed
  history <- p$history()
  matplot(data_raw$day, t(history[1, , -1]), type = "l",
          xlab = "Time", ylab = "State",
          col = "#ff000022", lty = 1, ylim = range(history))
  matlines(data_raw$day, t(history[2, , -1]), col = "#ffff0022", lty = 1)
  matlines(data_raw$day, t(history[3, , -1]), col = "#0000ff22", lty = 1)
  matpoints(data_raw$day, t(true_history[1:3, , -1]), pch = 19,
            col = c("red", "yellow", "blue"))
```

---

particle_filter_data	*Prepare data for use with particle filter*

Description

Prepare data for use with the [particle_filter](#). This function is required to use the particle filter as helps arrange data and be explicit about the off-by-one errors that can occur. It takes as input your data to compare against a model, including some measure of "time". We need to convert this time into model time steps (see Details).

Usage

```
particle_filter_data(data, time, rate, initial_time = NULL, population = NULL)
```

Arguments

data	A [data.frame()](#) of data
time	The name of a column within data that represents your measure of time. This column must be integer-like. To avoid confusion, this cannot be called step, time, or model_time.
rate	The number of model "time steps" that occur between each time point (in model time time). This must also be integer-like for discrete time models and must be NULL for continuous time models.
initial_time	An initial time to start the model from. This should always be provided, and must be provided for continuous time models. For discrete time models, this is expressed in model time. It must be a non-negative integer and must be at most equal to the first value of the time column, minus 1 (i.e., data[[time]] - 1). For historical reasons if not given we take the first value of the time column minus one, but with a warning - this behaviour will be removed in a future version of mcstate.
population	Optionally, the name of a column within data that represents different populations. Must be a factor.

Details

We require that the time variable increments in unit steps; this may be relaxed in future to even steps, or possibly irregular steps, but for now this assumption is required. We assume that the data in the first column is recorded at the end of a period of 1 time unit. So if you have in the first column t = 10, data = 100 we assume that the model steps from t = 9 to to t = 10 and at that period the data has value 100.

For continuous time models, time is simple to think about; time is continuous (and real-valued) and really any time is acceptable. For discrete time models there are two correlated measures of time we need to consider - (1) the dust "time step", a non-negative integer value that increases in unit steps, and (2) the "model time" which is related to the dust time step based on the rate parameter here as <model time> = <dust time> * <rate>. For a concrete example, consider a model where we want to think in terms of days, but which we take 10 steps per day. Time step 0 and model time 0 are the same, but day 1 occurs at step 10, day 15 at step 150 and so on.

Value

If population is NULL, a data.frame with new columns time_start and time_end (required by [particle_filter](#)), along side all previous data except for the time variable, which is replaced by new <time>_start and <time>_end columns. If population is not NULL then a named list of data.frames as described above where each element represents populations in the order specified in the data.

Examples

```
d <- data.frame(day = 5:20, y = runif(16))
mcstate::particle_filter_data(d, "day", rate = 4, initial_time = 4)

# If providing an initial day, then the first epoch of simulation
```

```
# will be longer (see the first row)
mcstate::particle_filter_data(d, "day", rate = 4, initial_time = 0)

# If including populations:
d <- data.frame(day = 5:20, y = runif(16),
                population = factor(rep(letters[1:2], each = 16)))
mcstate::particle_filter_data(d, "day", 4, 0, "population")
```

---

particle_filter_initial

Create restart initial state

Description

Create a suitable initial condition function from a set of restart state. This takes care of a few bookkeping and serialisation details and returns a function appropriate to pass to particle_filter as initial.

Usage

```
particle_filter_initial(state)
```

Arguments

state   A matrix of state (rows are different states, columns are different realisations). This is the form of a slice pulled from a restart.

Value

A function with arguments info, n_particles and pars that will sample, with replacement, a matrix of state suitable as a starting point for a particle filter. The info and pars arguments are ignored.

---

particle_filter_state   *Particle filter state*

Description

Particle filter internal state. This object is not ordinarily constructed directly by users, but via the $run_begin method to particle_filter. It provides an advanced interface to the particle filter that allows partially running the particle filter over part of the time trajectory.

This state object has a number of public fields that you can read but must not write (they are not read-only so you *could* write them, but don't).

Public fields

    `model` The dust model being simulated

    `history` The particle history, if created with `save_history = TRUE`. This is an internal format subject to

    `restart_state` Full model state at a series of points in time, if the model was created with non-NULL `save_restart`. This is a 3d (or greater) array as described in [particle_filter](#)

    `log_likelihood` The log-likelihood so far. This starts at 0 when initialised and accumulates value for each step taken.

    `log_likelihood_step` The log-likelihood attributable to the last step (i.e., the contribution to `log_likelihood` made on the last call to `$step()`.

    `current_time_index` The index of the last completed step.

Methods

    Public methods:

- [particle_filter_state$new()](#)
- [particle_filter_state$run()](#)
- [particle_filter_state$step()](#)
- [particle_filter_state$fork_multistage()](#)
- [particle_filter_state$fork_smc2()](#)

Method `new()`: Initialise the particle filter state. Ordinarily this should not be called by users, and so arguments are barely documented.

    Usage:

```
particle_filter_state$new(
  pars,
  generator,
  model,
  data,
  data_split,
  times,
  n_particles,
  has_multiple_parameters,
  n_threads,
  initial,
  index,
  compare,
  constant_log_likelihood,
  gpu_config,
  seed,
  min_log_likelihood,
  save_history,
  save_restart,
  stochastic_schedule,
  ode_control
)
```

Arguments:

pars Parameters for a single phase

generator A dust generator object

model If the generator has previously been initialised

data A [particle_filter_data](#) data object

data_split The same data as data but split by step

times A matrix of time step beginning and ends

n_particles Number of particles to use

has_multiple_parameters Compute multiple likelihoods at once?

n_threads The number of threads to use

initial Initial condition function (or NULL)

index Index function (or NULL)

compare Compare function

constant_log_likelihood Constant log likelihood function

gpu_config GPU configuration, passed to generator

seed Initial RNG seed

min_log_likelihood Early termination control

save_history Logical, indicating if we should save history

save_restart Vector of time steps to save restart at

stochastic_schedule Vector of times to perform stochastic updates

ode_control Tuning control for stepper

Method run(): Run the particle filter to the end of the data. This is a convenience function around $step() which provides the correct value of time_index

Usage:
```
particle_filter_state$run()
```

Method step(): Take a step with the particle filter. This moves the particle filter forward one step within the *data* (which may correspond to more than one step with your model) and returns the likelihood so far.

Usage:
```
particle_filter_state$step(time_index, partial = FALSE)
```

Arguments:

time_index The step *index* to move to. This is not the same as the model step, nor time, so be careful (it's the index within the data provided to the filter). It is an error to provide a value here that is lower than the current step index, or past the end of the data.

partial Logical, indicating if we should return the partial likelihood, due to this step, rather than the full likelihood so far.

Method fork_multistage(): Create a new particle_filter_state object based on this one (same model, position in time within the data) but with new parameters, to support the "multistage particle filter". Unlike fork_smc2, here the parameters may imply a different model shape and arbitrary transformations of the state are allowed. The model is not rerun to the current point, just transformed at that point.

Usage:

```
particle_filter_state$fork_multistage(model, pars, transform_state)
```

Arguments:

model A model object (or NULL)

pars New model parameters

transform_state A function to transform the model state from the old to the new parameter
        set. See [multistage_epoch()](#) for details.

Method `fork_smc2()`: Create a new `particle_filter_state` object based on this one (same
model, position in time within the data) but with new parameters, run up to the date, to support the
[smc2()](#) algorithm. To do this, we create a new `particle_filter_state` with new parameters at
the beginning of the simulation (corresponding to the start of your data or the `initial` argument
to [particle_filter](#)) with your new `pars`, and then run the filter foward in time until it reaches the
same step as the parent model.

Usage:

```
particle_filter_state$fork_smc2(pars)
```

Arguments:

pars New model parameters

---

pmcmc   *Run a pmcmc sampler*

Description

Run a pmcmc sampler

Usage

```
pmcmc(pars, filter, initial = NULL, control = NULL)
```

Arguments

pars	A [pmcmc_parameters](#) object containing information about parameters (ranges, priors, proposal kernel, translation functions for use with the particle filter).
filter	A [particle_filter](#) object
initial	Optional initial starting point. If given, it must be compatible with the parameters given in `pars`, and must be valid against your prior. You can use this to override the initial conditions saved in your `pars` object. You can provide either a vector of initial conditions, or a matrix with `n_chains` columns to use a different starting point for each chain.
control	A [pmcmc_control](#) object which will control how the MCMC runs, including the number of steps etc.

Details

This is a basic Metropolis-Hastings MCMC sampler. The `filter` is run with a set of parameters to evaluate the likelihood. A new set of parameters is proposed, and these likelihoods are compared, jumping with probability equal to their ratio. This is repeated for `n_steps` proposals.

While this function is called `pmcmc` and requires a particle filter object, there's nothing special about it for particle filtering. However, we may need to add things in the future that make assumptions about the particle filter, so we have named it with a "p".

Value

A `mcstate_pmcmc` object containing `pars` (sampled parameters) and `probabilities` (log prior, log likelihood and log posterior values for these probabilities). Two additional fields may be present: `state` (if `return_state` was TRUE), containing the final state of a randomly selected particle at the end of the simulation, for each step (will be a matrix with as many rows as your state has variables, and as `n_steps + 1` columns corresponding to each step). `trajectories` will include a 3d array of particle trajectories through the simulation (if `return_trajectories` was TRUE).

---

pmcmc_chains_prepare   *pMCMC with manual chain scheduling*

Description

Run a pMCMC, with sensible random number behaviour, but schedule execution of the chains yourself. Use this if you want to distribute chains over (say) the nodes of an HPC system.

Usage

```
pmcmc_chains_prepare(path, pars, filter, control, initial = NULL)

pmcmc_chains_run(chain_id, path, n_threads = NULL)

pmcmc_chains_collect(path)

pmcmc_chains_cleanup(path)
```

Arguments

path	The path to use to exchange inputs and results. You can use a temporary directory or a different path (relative or absolute). Several rds files will be created. It is strongly recommended not to use .
pars	A [pmcmc_parameters](#) object containing information about parameters (ranges, priors, proposal kernel, translation functions for use with the particle filter).
filter	A [particle_filter](#) object
control	A [pmcmc_control](#) object which will control how the MCMC runs, including the number of steps etc.

initial	Optional initial starting point. If given, it must be compatible with the parameters given in `pars`, and must be valid against your prior. You can use this to override the initial conditions saved in your `pars` object. You can provide either a vector of initial conditions, or a matrix with `n_chains` columns to use a different starting point for each chain.
chain_id	The integer identifier of the chain to run
n_threads	Optional thread count, overriding the number set in the `control`. This will be useful where preparing the threads on a machine with one level of resource and running it on another.

Details

Basic usage will look like

```
path <- mcstate::pmcmc_chains_prepare(tempfile(), pars, filter, control)
for (i in seq_len(control$n_chains)) {
  mcstate::pmcmc_chains_run(i, path)
}
samples <- mcstate::pmcmc_chains_collect(path)
mcstate::pmcmc_chains_cleanup(path)
```

You can safely parallelise (or not) however you like at the point where the loop is (even across other machines) and get the same outputs regardless.

---

pmcmc_combine   *Combine pmcmc samples*

Description

Combine multiple [pmcmc()](#) samples into one object

Usage

```
pmcmc_combine(..., samples = list(...))
```

Arguments

| ... | Arguments representing [pmcmc()](#) sample, i.e., `mcstate_pmcmc` objects. Alternatively, pass a list as the argument `samples`. Names are ignored. |
| samples | A list of `mcstate_pmcmc` objects. This is often more convenient for programming against than ... |

Description

Control for the pmcmc. This function constructs a list of options and does some basic validation to ensure that the options will work well together. Do not manually change the values in this object. Do not refer to any argument except n_steps by position as the order of the arguments may change in future.

Usage

```
pmcmc_control(
  n_steps,
  n_chains = 1L,
  n_threads_total = NULL,
  n_workers = 1L,
  rerun_every = Inf,
  rerun_random = FALSE,
  use_parallel_seed = FALSE,
  save_state = TRUE,
  save_restart = NULL,
  save_trajectories = FALSE,
  progress = FALSE,
  nested_step_ratio = 1,
  nested_update_both = FALSE,
  filter_early_exit = FALSE,
  restart_match = FALSE,
  n_burnin = NULL,
  n_steps_retain = NULL,
  adaptive_proposal = NULL,
  path = NULL
)
```

Arguments

n_steps   Number of MCMC steps to run. This is the only required argument.

n_chains   Optional integer, indicating the number of chains to run. If more than one then we run a series of chains and merge them with [pmcmc_combine()](). Chains are run in series, with the same filter if n_workers is 1, or run in parallel otherwise.

n_threads_total

The total number of threads (i.e., cores) the total number of threads/cores to use. If n_workers is greater than 1 then these threads will be divided evenly across your workers at first and so n_threads_total must be an even multiple of n_workers. If n_chains is not a clean multiple of n_workers we will try and allocate the leftover threads evenly across the last wave of chains. This

	value must be provided if n_workers is given, but is optional otherwise - if given it overrides the value in the particle filter.
n_workers	Number of "worker" processes to use to run chains in parallel. This must be at most n_chains and is recommended to be a divisor of n_chains. If n_workers is 1, then chains are run in series (i.e., one chain after the other). See the parallel vignette (vignette("parallelisation", package = "mcstate")) for more details about this approach.
rerun_every	Optional integer giving the frequency at which we should rerun the particle filter on the current "accepted" state. The default for this (Inf) will never rerun this point, but if you set to 100, then every 100 steps we run the particle filter on both the proposed *and* previously accepted point before doing the comparison. This may help "unstick" chains, at the cost of some bias in the results.
rerun_random	Logical, controlling the behaviour of rerunning (when rerun_every is finite). The default value of FALSE will rerun the filter deterministically at a fixed number of iterations (given by rerun_every). If TRUE, then we stochastically rerun each step with probability of 1 / rerun_every. This gives the same expected number of MCMC steps between reruns but a different pattern.
use_parallel_seed	
	Logical, indicating if seeds should be configured in the same way as when running workers in parallel (with n_workers > 1). Set this to TRUE to ensure reproducibility if you use this option sometimes (but not always). This option only has an effect if n_workers is 1.
save_state	Logical, indicating if the state should be saved at the end of the simulation. If TRUE, then a single randomly selected particle's state will be collected at the end of each MCMC step. This is the full state (i.e., unaffected by and index used in the particle filter) so that the process may be restarted from this point for projections. If save_trajectories is TRUE the same particle will be selected for each. The default is TRUE, which will cause n_state * n_steps of data to be output alongside your results. Set this argument to FALSE to save space, or use [pmcmc_thin()](#) after running the MCMC.
save_restart	An integer vector of time points to save restart information for; this is in addition to save_state (which saves the final model state) and saves the full model state. It will use the same trajectory as save_state and save_trajectories. Note that if you use this option you will end up with lots of model states and will need to process them in order to actually restart the pmcmc or the particle filter from this state. The integers correspond to the *time* variable in your filter (see [particle_filter](#) for more information).
save_trajectories	
	Logical, indicating if the particle trajectories should be saved during the simulation. If TRUE, then a single randomly selected particle's trajectory will be collected at the end of each MCMC step. This is the filtered state (i.e., using the state component of index provided to the particle filter). If save_state is TRUE the same particle will be selected for each.
progress	Logical, indicating if a progress bar should be displayed, using [progress::progress_bar](#).
nested_step_ratio	
	Either integer or 1/integer, which specifies the ratio of fixed:varied steps in a nested pMCMC. For example 3 would run 3 steps proposing fixed parameters

only and then 1 step proposing varied parameters only; whereas 1/3 would run 3 varied steps for every 1 fixed step. The default value of 1 runs an equal number of iterations updating the fixed and varied parameters. Sensible choices of this parameter may depend on the true ratio of fixed:varied parameters or on desired run-time, for example updating fixed parameters is quicker so more varied steps could be more efficient.

nested_update_both

If FALSE (default) then alternates between proposing fixed and varied parameter updates according to the ratio in nested_step_ratio. If TRUE then proposes fixed and varied parameters simultaneously and collectively accepts/rejects them, nested_step_ratio is ignored.

filter_early_exit

Logical, indicating if we should allow the particle filter to exit early for points that will not be accepted. Only use this if your log-likelihood never increases between steps. This will the the case where your likelihood calculation is a sum of discrete normalised probability distributions, but may not be for continuous distributions!

restart_match   Logical, indicating whether the restart state saved from the particle filter should match the trajectory saved, otherwise the restart state will be randomly drawn from the states of the particle filter after filtering to the restart time point.

n_burnin   Optionally, the number of points to discard as burnin. This happens separately to the burnin in [pmcmc_thin](#) or [pmcmc_sample](#). See Details.

n_steps_retain   Optionally, the number of samples to retain from the n_steps − n_burnin steps. See Details.

adaptive_proposal

Optionally, control over an adaptive proposal ([adaptive_proposal_control](#)). Alternatively FALSE to disable, TRUE to enable defaults. This is only valid for single-population deterministic models.

path   Optional path to save partial pmcmc results in, when using workers. If not given (or NULL) then a temporary directory is used.

Details

pMCMC is slow and you will want to parallelise it if you possibly can. There are two ways of doing this which are discussed in some detail in vignette("parallelisation", package = "mcstate").

Value

A pmcmc_control object, which should not be modified once created.

Thinning the chain at generation

Generally it may be preferable to thin the chains after generation using [pmcmc_thin](#) or [pmcmc_sample](#). However, waiting that long can create memory consumption issues because the size of the trajectories can be very large. To avoid this, you can thin the chains at generation - this will avoid creating large trajectory arrays, but will discard some information irretrivably.

If either of the options n_burnin or n_steps_retain are provided, then we will subsample the chain at generation.

- If n_burnin is provided, then the first n_burnin (of n_steps) samples is discarded. This must be at most n_steps

- If n_steps_retain is provided, then we *evenly* sample out of the remaining samples. The algorithm will try and generate a sensible set here, and will always include the last sample of n_steps but may not always include the first post-burnin sample. An error will be thrown if a suitable sampling is not possible (e.g., if n_steps_retain is larger than n_steps - n_burnin

If either of n_burnin or n_steps_retain is provided, the resulting samples object will include the full set of parameters and probabilities sampled, along with an index showing how they relate to the filtered samples.

Examples

```
mcstate::pmcmc_control(1000)

# Suppose we have a fairly large node with 16 cores and we want to
# run 8 chains. We can use all cores for a single chain and run
# the chains sequentially like this:
mcstate::pmcmc_control(1000, n_chains = 8, n_threads_total = 16)

# However, on some platforms (e.g., Windows) this may only realise
# a 50% total CPU use, in which case you might benefit from
# splitting these chains over different worker processes (2-4
# workers is likely the largest useful number).
mcstate::pmcmc_control(1000, n_chains = 8, n_threads_total = 16,
                       n_workers = 4)
```

---

pmcmc_parameter   *Describe single pmcmc parameter*

Description

Describe a single parameter for use within the pmcmc. Note that the name is not set here, but will end up being naturally defined when used with pmcmc_parameters, which collects these together for use with pmcmc().

Usage

```
pmcmc_parameter(
  name,
  initial,
  min = -Inf,
  max = Inf,
  discrete,
  integer = FALSE,
  prior = NULL,
```

```
    mean = NULL
  )
```

Arguments

name	Name for the parameter (a string)
initial	Initial value for the parameter
min	Optional minimum value for the parameter (otherwise `-Inf`). If given, then `initial` must be at least this value.
max	Optional max value for the parameter (otherwise `Inf`). If given, then `initial` must be at most this value.
discrete	Deprecated; use `integer` instead.
integer	Logical, indicating if this parameter is integer. If `TRUE` then the parameter will be rounded after a new parameter is proposed.
prior	A prior function (if not given an improper flat prior is used - be careful!). It must be a function that takes a single argument, being the value of this parameter. If given, then `prior(initial)` must evaluate to a finite value.
mean	Optionally, an estimate of the mean of the parameter. If not given, then we assume that `initial` is a reasonable estimate. This is used only in adaptive mcmc.

Examples

```
pmcmc_parameter("a", 0.1)
```

---

pmcmc_parameters   *pmcmc_parameters*

Description

Construct parameters for use with [pmcmc()](). This creates a utility object that is used internally to work with parameters. Most users only need to construct this object, but see the examples for how it can be used.

Parameter transformations

Unless you have a very simple model, it is highly unlikely that the parameters that you are interested in performing inference on are the same as the parameters that you might need to initialise your model.

Due to the nature of mcmc and other inference algorithms, the general assumption is that the inference parameters will be a simple vector of real values; here each of the `parameters` elements corresponds to one of these. The proposal matrix maps one vector to another via a simple multivariate-gaussian kernel.

On the other hand, dust models can take a named list of arbitrary data as their input parameters (see [dust::dust_generator]()). These might include:

- things that are not parameters at all from the perspective of the inference - for example some quantity that you might vary depending on the region/species/etc you're running the model for but that you are not fitting.

- non-scalar quantities that are directly derived from some parameters that you are fitting. As an example of this, in sircovid, a transmission model of COVID, we take a number of "contact rates" which apply at different points in time, and generate from this an interpolated series of contact rates per time step (a very long vector). Other users have needed to generate equilibrium solutions to parts of their model and used these at initialisation.

- arbitrary complex inputs to the model, for example weather data, demographic matrices, population contact rate matrices etc. These are all "parameters" from the perspective of a dust model but not at all from the perspective of the inference process.

To allow for this in a flexible way, mcstate allows a "transform" function, the `transform` argument to the constructor. This function maps a named numeric vector of inference parameters to whatever you need for your dust model. The default value for this function is as.list which just converts the named vector to a named list, which works well in the example cases here.

When providing a transformation function, you may want to provide a "closure" rather than a top-level function. This way you can bind additional data into your function. For example, suppose that you want to use some demographic matrix `m` in your model, and perform inference on parameters `a` and `b` you might write

```
make_transform <- function(m) {
  function(theta) {
    c(list(m = m), as.list(theta))
  }
}
```

and pass this into `mcstate::pmcmc_parameters$new`, providing parameter definitions only for `a` and `b`. See the examples for full working of this.

Methods

Public methods:

- `pmcmc_parameters$new()`
- `pmcmc_parameters$initial()`
- `pmcmc_parameters$mean()`
- `pmcmc_parameters$vcv()`
- `pmcmc_parameters$names()`
- `pmcmc_parameters$summary()`
- `pmcmc_parameters$prior()`
- `pmcmc_parameters$propose()`
- `pmcmc_parameters$model()`
- `pmcmc_parameters$fix()`

Method `new()`: Create the pmcmc_parameters object

Usage:

```
pmcmc_parameters$new(parameters, proposal, transform = NULL)
```

Arguments:

parameters A list of [pmcmc_parameter](#) objects, each of which describe a single parameter in your model. If parameters is named, then these names must match the $name element of each parameter is used (this is verified).

proposal A square proposal distribution corresponding to the variance-covariance matrix of a multivariate gaussian distribution used to generate new parameters. It must have the same number of rows and columns as there are elements in parameters, and if named the names must correspond exactly to the names in parameters. Because it corresponds to a variance-covariance matrix it must be symmetric and positive definite.

transform An optional transformation function to apply to your parameter vector immediately before passing it to the model function. If not given, then [as.list](#) is used, as dust models require this. However, if t you need to generate derived parameters from those being actively sampled you can do arbitrary transformations here.

Method initial(): Return the initial parameter values as a named numeric vector

Usage:
```
pmcmc_parameters$initial()
```

Method mean(): Return the estimate of the mean of the parameters, as set when created (this is not updated by any fitting!)

Usage:
```
pmcmc_parameters$mean()
```

Method vcv(): Return the variance-covariance matrix used for the proposal.

Usage:
```
pmcmc_parameters$vcv()
```

Method names(): Return the names of the parameters

Usage:
```
pmcmc_parameters$names()
```

Method summary(): Return a data.frame with information about parameters (name, min, max, and integer).

Usage:
```
pmcmc_parameters$summary()
```

Method prior(): Compute the prior for a parameter vector

Usage:
```
pmcmc_parameters$prior(theta)
```

Arguments:

theta a parameter vector in the same order as your parameters were defined in (see $names() for that order).

Method `propose()`: Propose a new parameter vector given a current parameter vector. This proposes a new parameter vector given your current vector and the variance-covariance matrix of your proposal kernel, rounds any integer values, and reflects bounded parameters until they lie within `min:max`.

Usage:

`pmcmc_parameters$propose(theta, scale = 1, vcv = NULL)`

Arguments:

`theta` a parameter vector in the same order as your parameters were defined in (see `$names()` for that order.

`scale` an optional scaling factor to apply to the proposal distribution. This may be useful in sampling starting points. The parameter is equivalent to a multiplicative factor applied to the variance covariance matrix.

`vcv` A variance covariance matrix of the correct size, overriding the proposal matrix built into the parameters object. This will be slightly less efficient but allow a different proposal matrix to be used (e.g., during an adaptive MCMC)

Method `model()`: Apply the model transformation function to a parameter vector.

Usage:

`pmcmc_parameters$model(theta)`

Arguments:

`theta` a parameter vector in the same order as your parameters were defined in (see `$names()` for that order.

Method `fix()`: Set some parameters to fixed values. Use this to reduce the dimensionality of your system.

Usage:

`pmcmc_parameters$fix(fixed)`

Arguments:

`fixed` a named vector of parameters to fix

Examples

```
# Construct an object with two parameters:
pars <- mcstate::pmcmc_parameters$new(
  list(mcstate::pmcmc_parameter("a", 0.1, min = 0, max = 1,
                                prior = function(a) log(a)),
       mcstate::pmcmc_parameter("b", 0, prior = dnorm)),
  matrix(c(1, 0.5, 0.5, 2), 2, 2))

# Initial parameters
p <- pars$initial()
p

# Propose a new parameter point
pars$propose(p)

# Information about parameters:
```

```
pars$names()
pars$summary()

# Compute prior
pars$prior(p)

# Transform data for your model
pars$model(p)

# Above we describe a nontrivial transformation function using a closure
make_transform <- function(m) {
  function(theta) {
    c(list(m = m), as.list(theta))
  }
}

# Suppose this is our demographic matrix (note here that the name
# need not match that used in the transform)
demographic_matrix <- diag(4)

# Construct the parameters as above, but this time passing in the
# function that make_transform returns
pars <- mcstate::pmcmc_parameters$new(
  list(mcstate::pmcmc_parameter("a", 0.1, min = 0, max = 1,
                                prior = function(a) log(a)),
       mcstate::pmcmc_parameter("b", 0, prior = dnorm)),
  matrix(c(1, 0.5, 0.5, 2), 2, 2),
  make_transform(demographic_matrix))

# Now, as above we start from a position in terms of a and b only:
pars$initial()

# But when prepared for the model, our matrix will be set up
pars$model(pars$initial())
```

---

pmcmc_parameters_nested

pmcmc_parameters_nested

Description

Construct nested parameters for use with [pmcmc()](). This creates a utility object that is used internally to work with parameters that may be fixed and the same for all given populations, or varied and possibly-different between populations. Most users only need to construct this object, but see the examples for how it can be used.

Methods

Public methods:

- [pmcmc_parameters_nested$new()](#)
- [pmcmc_parameters_nested$names()](#)
- [pmcmc_parameters_nested$populations()](#)
- [pmcmc_parameters_nested$validate()](#)
- [pmcmc_parameters_nested$summary()](#)
- [pmcmc_parameters_nested$initial()](#)
- [pmcmc_parameters_nested$mean()](#)
- [pmcmc_parameters_nested$vcv()](#)
- [pmcmc_parameters_nested$prior()](#)
- [pmcmc_parameters_nested$propose()](#)
- [pmcmc_parameters_nested$model()](#)
- [pmcmc_parameters_nested$fix()](#)

Method new(): Create the pmcmc_parameters object

Usage:
```
pmcmc_parameters_nested$new(
  parameters,
  proposal_varied = NULL,
  proposal_fixed = NULL,
  populations = NULL,
  transform = NULL
)
```

Arguments:

parameters A list of [pmcmc_parameter](#) or [pmcmc_varied_parameter](#) objects, each of which describe a single (possibly-varying) parameter in your model. If parameters is named, then these names must match the $name element of each parameter that is used (this is verified).

proposal_varied, proposal_fixed Square proposal matrices corresponding to the variance-covariance matrix of a multivariate gaussian distribution used to generate new varied and fixed parameters respectively.'. They must have the same number of rows and columns as there are varied and fixed parameters respectively. The names must correspond exactly to the names in parameters. Because it corresponds to a variance-covariance matrix it must be symmetric and positive definite.

populations Specifies the names of the different populations that the varying parameters change according to. Only required if no [pmcmc_varied_parameter](#) objects are included in parameters. Otherwise population names are taken from those objects.

transform An optional transformation function to apply to your parameter vector immediately before passing it to the model function. If not given, then [as.list](#) is used, as dust models require this. However, if you need to generate derived parameters from those being actively sampled you can do arbitrary transformations here.

Method names(): Return the names of the parameters

Usage:
```
pmcmc_parameters_nested$names(type = "both")
```

Arguments:

type One of "both" (the default, all parameters), "fixed" (parameters that are shared across populations) or "varied" (parameters that vary over populations).

Method populations(): Return the names of the populations

Usage:
pmcmc_parameters_nested$populations()

Method validate(): Validate a parameter matrix. This method checks that your matrix has the expected size (rows according to parameters, columns to populations) and if named that the names are exactly what is expected. It also verifies that the fixed parameters are same across all populations.

Usage:
pmcmc_parameters_nested$validate(theta)

Arguments:
theta a parameter matrix

Method summary(): Return a data.frame with information about parameters (name, min, max, integer, type (fixed or varied) and population)

Usage:
pmcmc_parameters_nested$summary()

Method initial(): Return the initial parameter values as a named matrix with rows corresponding to parameters and columns to populations.

Usage:
pmcmc_parameters_nested$initial()

Method mean(): Return the estimate of the mean of the parameters, as set when created (this is not updated by any fitting!)

Usage:
pmcmc_parameters_nested$mean(type)

Method vcv(): Return the variance-covariance matrix used for the proposal.

Usage:
pmcmc_parameters_nested$vcv(type)

Method prior(): Compute the prior(s) for a parameter matrix. Returns a named vector with names corresponding to populations.

Usage:
pmcmc_parameters_nested$prior(theta)

Arguments:
theta a parameter matrix with columns in the same order as $names() and rows in the same order as $populations().

Method propose(): This proposes a new parameter matrix given your current matrix and the variance-covariance matrices of the proposal kernels, rounds any integer values, and reflects bounded parameters until they lie within min:max. Returns matrix with rows corresponding to parameters and columns to populations (i.e., the same orientation as theta).

Usage:

```
pmcmc_parameters_nested$propose(theta, type, scale = 1, vcv = NULL)
```

Arguments:

theta a parameter matrix with rows in the same order as $names() and columns in the same order as $populations().

type specifies which type of parameters should be proposed, either fixed parameters only ("fixed"), varied only ("varied"), or both ("both") types. For 'fixed' and 'varied', parameters of the other type are left unchanged.

scale an optional scaling factor to apply to the proposal distribution. This may be useful in sampling starting points. The parameter is equivalent to a multiplicative factor applied to the variance covariance matrix.

Method model(): Apply the model transformation function to a parameter matrix.

Usage:

```
pmcmc_parameters_nested$model(theta)
```

Arguments:

theta a parameter matrix with rows in the same order as $names() and columns in the same order as $populations().

Method fix(): Set some parameters to fixed values. Use this to reduce the dimensionality of your system. Note that this function has an unfortunate name collision - we use "fixed" and "varied" parameters generally to refer to ones that are fixed across populations or which vary among populations. However, in the context of this method "fixed" refers to parameters which will be set to a single value and no longer used in inference.

Usage:

```
pmcmc_parameters_nested$fix(fixed)
```

Arguments:

fixed a named vector of parameters to fix

Examples

```
# Construct an object with two varied parameters ('a' and 'b'),
# two fixed parameters ('c' and 'd') and two populations ('p1' and 'p2')
parameters <- list(mcstate::pmcmc_varied_parameter("a", c("p1", "p2"), 2),
                   mcstate::pmcmc_varied_parameter("b", c("p1", "p2"), 2),
                   mcstate::pmcmc_parameter("c", 3),
                   mcstate::pmcmc_parameter("d", 4))
proposal_fixed <- diag(2)
proposal_varied <- diag(2) + 1
pars <- mcstate::pmcmc_parameters_nested$new(parameters, proposal_varied,
                                             proposal_fixed)

# Initial parameters
p <- pars$initial()
p

# Propose a new parameter point
```

```
pars$propose(p, type = "both")
pars$propose(p, type = "fixed")
pars$propose(p, type = "varied")

# Information about parameters:
pars$names()
pars$names("fixed")
pars$names("varied")
pars$summary()

# Compute log prior probability, per population
pars$prior(p)

# Transform data for your model
pars$model(p)
```

---

pmcmc_predict   *Run predictions from PMCMC*

Description

Run predictions from the results of [pmcmc()](). This function can also be called by running [predict()]()
on the object, using R's S3 dispatch.

Usage

```
pmcmc_predict(
  object,
  times,
  prepend_trajectories = FALSE,
  n_threads = NULL,
  seed = NULL
)
```

Arguments

object	The results of running [pmcmc()]() with return_state = TRUE (without this extra information, prediction is not possible)
times	A vector of time times to return predictions for. The first value must be the final value run in your simulation. An error will be thrown if you get this value wrong, look in object$predict$time (or the error message) for the correct value.
prepend_trajectories	
	Prepend trajectories from the particle filter to the predictions created here.
n_threads	The number of threads used in the simulation. If not given, we default to the value used in the particle filter that was used in the pmcmc.

seed   The random number seed (see [particle_filter](#)). The default value of NULL
                    will seed the dust random number generator from R's random number generator.
                    However, you can pick up from the same RNG stream used in the simulation if
                    you pass in seed = object$predict$seed. However, do not do this if you are
                    gong to run pmcmc_predict() multiple times the result will be identical. If
                    you do want to call predict with this state multiple times you should create a
                    persistant rng state object (e.g., with [dust::dust_rng](#) and perform a "long jump"
                    between each call.

---

pmcmc_thin   *Thin a pmcmc chain*

Description

Thin results of running [pmcmc()](#). This function may be useful before using [pmcmc_predict()](#), or
before saving pmcmc output to disk. pmcmc_thin takes every thin'th sample, while pmcmc_sample
randomly selects a total of n_sample samples.

Usage

```
pmcmc_thin(object, burnin = NULL, thin = NULL)

pmcmc_sample(object, n_sample, burnin = NULL)
```

Arguments

object   Results of running [pmcmc()](#)

burnin   Optional integer number of iterations to discard as "burn-in". If given then sam-
                    ples 1:burnin will be excluded from your results. It is an error if this is not a
                    positive integer or is greater than or equal to the number of samples (i.e., there
                    must be at least one sample remaining after discarding burnin).

thin   Optional integer thinning factor. If given, then every thin'th sample is retained
                    (e.g., if thin is 10 then we keep samples 1, 11, 21, ...). Note that this can produce
                    surprising results as it will always select the first sample but not necessarily
                    always the last.

n_sample   The number of samples to draw from object *with replacement*. This means
                    that n_sample can be larger than the total number of samples taken (though it
                    probably should not)

pmcmc_varied_parameter

Describe varying pmcmc parameter

Description

Describe a varying parameter for use within the nested pmcmc. Note that the name is not set here, but will end up being naturally defined when used with pmcmc_parameters_nested, which collects these together for use with pmcmc().

Usage

```
pmcmc_varied_parameter(
  name,
  populations,
  initial,
  min = -Inf,
  max = Inf,
  discrete,
  integer = FALSE,
  prior = NULL
)
```

Arguments

name	Name for the parameter (a string)
populations	The name of the populations for which different values of the parameter are being estimated for, length n_pop.
initial	Initial value(s) for the parameter. Must be either length n_pop or 1, in which case the same value is assumed for all populations.
min	Optional minimum value(s) for the parameter (otherwise -Inf). If given, then initial must be at least this value. Must be either length n_pop or 1, in which case the same value is assumed for all populations.
max	Optional max value for the parameter (otherwise Inf). If given, then initial must be at most this value. Must be either length n_pop or 1, in which case the same value is assumed for all populations.
discrete	Deprecated; use integer instead.
integer	Logical, indicating if this parameter is integer. If TRUE then the parameter will be rounded after a new parameter is proposed.
prior	A prior function (if not given an improper flat prior is used - be careful!). It must be a function that takes a single argument, being the value of this parameter. If given, then prior(initial) must evaluate to a finite value. Must be either length n_pop or 1, in which case the same value is assumed for all populations.

Examples

```
mcstate::pmcmc_varied_parameter(
  name = "size",
  populations = c("Europe", "America"),
  initial = c(100, 200),
  min = 0,
  max = Inf,
  integer = TRUE,
  prior = list(dnorm, dexp))
```

---

smc2   *Run SMC^2*

Description

Run a SMC^2. This is experimental and subject to change. Use at your own risk.

Usage

```
smc2(pars, filter, control)
```

Arguments

pars	A [smc2_parameters](#) object containing information about parameters (ranges, priors, proposal kernel, translation functions for use with the particle filter).
filter	A [particle_filter](#) object
control	A [smc2_control](#) object to control the behaviour of the algorithm

Value

A smc2_result object, with elements

- pars: a matrix of sampled parameters (n_parameter_set long)

- probabilities: a matrix of probabilities (log_prior, log_likelihood, log_posterior and weight). The latter is the log posterior normalised over all samples

- statistics: interesting or useful statistics about your sample, including the ess (effective sample size, over time), acceptance_rate (where a regeneration step was done, the acceptance rate), n_particles, n_parameter_sets and n_steps (inputs to the simulation). The effort field is a rough calculation of the number of particle-filter runs that this run was worth.

Examples

```
# We use an example from dust which implements an epidemiological SIR
# (Susceptible-Infected-Recovered) model; see vignette("sir_models")
# for more background, as this example follows from the pMCMC example
# there
```

```
# The key tuning here is the number of particles per filter and number
# of parameter sets to consider simultaneously. Ordinarily these would
# be set (much) higher with an increase in computing time
n_particles <- 42
n_parameter_sets <- 20

# Basic epidemiological (Susceptible-Infected-Recovered) model
sir <- dust::dust_example("sir")

# Pre-computed incidence data
incidence <- read.csv(system.file("sir_incidence.csv", package = "mcstate"))

# Annotate the data so that it is suitable for the particle filter to use
dat <- mcstate::particle_filter_data(incidence, "day", 4, 0)

# Subset the output during run
index <- function(info) {
  list(run = 5L)
}

# The comparison function, used to compare simulated data with observe
# data, given the above subset
compare <- function(state, observed, pars) {
  exp_noise <- 1e6
  incidence_modelled <- state[1L, , drop = TRUE]
  incidence_observed <- observed$cases
  lambda <- incidence_modelled +
    rexp(n = length(incidence_modelled), rate = exp_noise)
  dpois(x = incidence_observed, lambda = lambda, log = TRUE)
}

# Finally, construct the particle filter:
filter <- mcstate::particle_filter$new(dat, sir, n_particles, compare,
                                       index = index)

# To control the smc2 we need to specify the parameters to consider
pars <- mcstate::smc2_parameters$new(
  list(
    mcstate::smc2_parameter("beta",
                            function(n) runif(n, 0, 1),
                            function(x) dunif(x, 0, 1, log = TRUE),
                            min = 0, max = 1),
    mcstate::smc2_parameter("gamma",
                            function(n) runif(n, 0, 1),
                            function(x) dunif(x, 0, 1, log = TRUE),
                            min = 0, max = 1)))
control <- mcstate::smc2_control(n_parameter_sets, progress = TRUE)

# Then we run the particle filter
res <- mcstate::smc2(pars, filter, control)

# This returns quite a lot of information about the fit, and this will
# change in future versions
```

```
res

# Most useful is likely the predict method:
predict(res)
```

---

smc2_control   *Control for SMC2*

Description

Control for [smc2](#). This function constructs a list of options and does some basic validation to ensure
that the options will work well together. Do not manually change the values in this object. Do not
refer to any argument except n_parameter_sets by position as the order of the arguments may
change in future.

Usage

```
smc2_control(
  n_parameter_sets,
  degeneracy_threshold = 0.5,
  covariance_scaling = 0.5,
  progress = TRUE,
  save_trajectories = FALSE
)
```

Arguments

n_parameter_sets

        The number of replicate parameter sets to simulate at once.

degeneracy_threshold

        The degeneracy threshold. Once the effective sample size drops below degeneracy_threshold
        * n_parameter_sets the algorithm will rerun simulations from the beginning
        of the data and use these to replenish the particles.

covariance_scaling

        A scaling factor to update variance covariance matrix of sampled parameters by

progress   Logical, indicating if a progress bar should be displayed, using [progress::progress_bar](#).

save_trajectories

        Logical, indicating if particle trajectories should be saved during the simulation.

Value

A smc2_control object, which should not be modified once created.

Examples

```
mcstate::smc2_control(100)
```

smc2_parameter	*Describe single pmcmc parameter*

Description

Describe a single parameter for use within the SMC^2. Note that the name is not set here, but will end up being naturally defined when used with `smc2_parameters`, which collects these together for use with `smc2()`.

Usage

```
smc2_parameter(
  name,
  sample,
  prior,
  min = -Inf,
  max = Inf,
  discrete,
  integer = FALSE
)
```

Arguments

name	Name for the parameter (a string)
sample	A sampling function; it must take a single argument representing the number of sampled to be returned. Typically this will be a r probability function corresponding to the sampling version of your prior (e.g., you might use `runif` and `dunif` for `sample` and `prior`). If you provide `min`, `max` or `integer` you *must* ensure that your function returns values that satisfy these constraints, as this is not (yet) checked.
prior	A prior function. It must be a function that takes a single argument, being the value of this parameter.
min	Optional minimum value for the parameter (otherwise `-Inf`). If given, then `initial` must be at least this value.
max	Optional max value for the parameter (otherwise `Inf`). If given, then `initial` must be at most this value.
discrete	Deprecated; use `integer` instead.
integer	Logical, indicating if this parameter is an integer. If `TRUE` then the parameter will be rounded after a new parameter is proposed.

Examples

```
mcstate::smc2_parameter("a",
                        function(n) rnorm(n),
                        function(x) dnorm(n, log = TRUE))
```

---

smc2_parameters *smc2_parameters*

Description

Construct parameters for use with smc2(). This creates a utility object that is used internally to work with parameters. Most users only need to construct this object, but see the examples for how it can be used.

Methods

Public methods:

- smc2_parameters$new()
- smc2_parameters$sample()
- smc2_parameters$names()
- smc2_parameters$summary()
- smc2_parameters$prior()
- smc2_parameters$propose()
- smc2_parameters$model()

Method new(): Create the smc2_parameters object

Usage:

smc2_parameters$new(parameters, transform = NULL)

Arguments:

parameters A list of smc2_parameter objects, each of which describe a single parameter in your model. If parameters is named, then these names must match the $name element of each parameter is used (this is verified).

transform An optional transformation function to apply to your parameter vector immediately before passing it to the model function. If not given, then as.list is used, as dust models require this. However, if t you need to generate derived parameters from those being actively sampled you can do arbitrary transformations here.

Method sample(): Create n independent random parameter vectors (as a matrix with n rows)

Usage:

smc2_parameters$sample(n)

Arguments:

n Number of replicate parameter sets to draw

Method names(): Return the names of the parameters

Usage:

smc2_parameters$names()

Method summary(): Return a data.frame with information about parameters (name, min, max, and integer).

Usage:
```
smc2_parameters$summary()
```

Method `prior()`: Compute the prior for a parameter vector

Usage:
```
smc2_parameters$prior(theta)
```

Arguments:

`theta` a parameter vector in the same order as your parameters were defined in (see `$names()` for that order.

Method `propose()`: Propose a new parameter vector given a current parameter vector and variance covariance matrix. After proposal, this rounds any integer values, and reflects bounded parameters until they lie within `min:max`.

Usage:
```
smc2_parameters$propose(theta, vcv)
```

Arguments:

`theta` a parameter vector in the same order as your parameters were defined in (see `$names()` for that order).

`vcv` the variance covariance matrix for the proposal; must be square and have a number of rows and columns equal to the number of parameters, in the same order as `theta`.

Method `model()`: Apply the model transformation function to a parameter vector.

Usage:
```
smc2_parameters$model(theta)
```

Arguments:

`theta` a parameter vector in the same order as your parameters were defined in (see `$names()` for that order.

Index