

Package: monty (via r-universe)

October 3, 2026

Title Monte Carlo Models

Version 0.4.14

Description Experimental sources for the next generation of mcstate, now called 'monty', which will support much of the old mcstate functionality but new things like better parameter interfaces, Hamiltonian Monte Carlo, and other features.

License MIT + file LICENSE

Encoding UTF-8

Roxygen list(markdown = TRUE)

URL <https://mrc-ide.github.io/monty>, <https://github.com/mrc-ide/monty>

BugReports <https://github.com/mrc-ide/monty/issues>

Imports cli, parallel, rlang

LinkingTo cpp11

Suggests callr, cpp11, coda, decor, knitr, mockery, mvtnorm, numDeriv, pkgload, posterior, rmarkdown, testthat (>= 3.0.0), withr

Config/testthat/edition 3

Language en-GB

VignetteBuilder knitr

Config/roxygen2/version 8.0.0

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

Date/Publication 2026-07-08 09:20:13 UTC

RemoteUrl <https://github.com/mrc-ide/monty>

RemoteRef main

RemoteSha a0e76e32024cb0787420038d93f2a19af76b0657

Contents

monty_differentiation	3
monty_domain_expand	4
monty_dsl	5
monty_dsl_distributions	6
monty_dsl_error_explain	7
monty_dsl_parse_distribution	7
monty_example	8
monty_flatten_chains	10
monty_model	11
monty_model_combine	13
monty_model_density	14
monty_model_direct_sample	15
monty_model_function	16
monty_model_gradient	17
monty_model_properties	18
monty_model_split	19
monty_observer	20
monty_packer	21
monty_packer_grouped	25
monty_random_beta	28
monty_random_beta_binomial_prob	28
monty_random_binomial	29
monty_random_cauchy	30
monty_random_exponential_rate	30
monty_random_gamma_scale	31
monty_random_hypergeometric	32
monty_random_log_normal	32
monty_random_negative_binomial_prob	33
monty_random_normal	34
monty_random_poisson	34
monty_random_real	35
monty_random_truncated_normal	36
monty_random_uniform	36
monty_random_weibull	37
monty_random_zi_negative_binomial_prob	38
monty_random_zi_poisson	39
monty_rng_create	39
monty_rng_jump	41
monty_rng_state	41
monty_runner_callr	42
monty_runner_parallel	43
monty_runner_serial	43
monty_runner_simultaneous	44
monty_sample	45
monty_sample_continue	47
monty_sample_manual_cleanup	48

monty_sample_manual_collect	49
monty_sample_manual_info	50
monty_sample_manual_prepare	51
monty_sample_manual_prepare_continue	53
monty_sample_manual_run	54
monty_sampler	55
monty_sampler_adaptive	58
monty_sampler_hmc	61
monty_sampler_parallel_tempering	61
monty_sampler_properties	63
monty_sampler_random_walk	64
monty_samples_thin	65
monty_unflatten_chains	67
with_trace_random	67
Index	68

monty_differentiation *Differentiate expressions*

Description

Differentiate expressions in the monty DSL. This function is exported for advanced use, and really so that we can use it from odin. But it has the potential to be generally useful, so while we'll tweak the interface quite a lot over the next while it is fine to use if you can handle some disruption.

Usage

```
monty_differentiation()
```

Details

R already has support for differentiating expressions using [D](#), which is useful for creating derivatives of simple functions to pass into non-linear optimisation. We need something a bit more flexible for differentiating models in the monty DSL ([monty_dsl](#)) and also in the related odin DSL.

Value

A list of related objects:

- `differentiate`: A function that can differentiate an expression with respect to a variable (as a string).
- `maths`: Some mathematical utilities for constructing expressions. This will be documented later, but the most useful bits on here are the function elements `times`, `plus` and `plus_fold`.

We will expand this soon to advertise what functions we are able to differentiate to allow programs to fail fast.

Differences to `D()`

- We try a little harder to simplify expressions.
- The distribution functions in the monty DSL (e.g., Poisson) are (will be) handled specially, allowing substitution of log-densities and expectations.
- Once we support array expressions, we will be able to differentiate through these.

Roadmap

We may need to make this slightly extensible in future, but for now the set of functions that can be differentiated is closed.

Warning

The way of accessing distribution support here is peculiar and the return type unusual. This is intentional, and we expect a more conventional interface in the future once this package settles down.

Examples

```
d <- monty_differentiation()
d$differentiate(quote(sqrt(sin(x))), "x")
D(quote(sqrt(sin(x))), "x")
```

monty_domain_expand	<i>Expand (and check) domain against a packer</i>
---------------------	---

Description

Check and expand a domain, where it is used alongside a [monty_packer](#) object. This can be used to expand domains for logical parameters (e.g. a vector `b`) into its specific names (e.g., `b[1]`, `b[2]`, etc) without having to rely on the internals about how these names are constructed.

Usage

```
monty_domain_expand(domain, packer)
```

Arguments

domain	A two-column matrix as defined in monty_model , with row names corresponding to either logical names (e.g., <code>b</code>) or specific names <code>b[1]</code> that are present in your packer. <code>NULL</code> is allowed where all parameters are defined over the entire real line.
packer	A monty_packer object

Value

A two dimensional matrix representing your domain, or `NULL` if domain was given as `NULL`.

Examples

```
packer <- monty_packer(c("a", "b"), list(x = 3, y = c(2, 2)))
monty_domain_expand(NULL, packer)
monty_domain_expand(rbind(x = c(0, 1)), packer)
monty_domain_expand(rbind(x = c(0, 1), "x[2]" = c(0, Inf)), packer)
monty_domain_expand(rbind(x = c(0, 1), "y" = c(0, Inf)), packer)
```

monty_dsl

Domain Specific Language for monty

Description

Create a model using the monty DSL; this function will likely change name in future, as will its interface.

Usage

```
monty_dsl(x, type = NULL, gradient = NULL, fixed = NULL, domain = NULL)
```

Arguments

x	The model as an expression. This may be given as an expression, as a string, or as a path to a filename. Typically, we'll do a reasonable job of working out what you've provided but use the type argument to disambiguate or force a particular interpretation. The argument uses rlang's quosures to allow you to work with expressions directly; see examples for details.
type	Force interpretation of the type of expression given as x. If given, valid options are expression, text or file.
gradient	Control gradient derivation. If NULL (the default) we try and generate a gradient function for your model and warn if this is not possible. If FALSE, then we do not attempt to construct a gradient function, which prevents a warning being generated if this is not possible. If TRUE, then we will error if it is not possible to create a gradient function.
fixed	An optional list of values that can be used within the DSL code. Anything you provide here is available for your calculations. In the interest of future compatibility, we check currently that all elements are scalars. In future this may become more flexible and allow passing environments, etc. Once provided, these values cannot be changed without rebuilding the model; they are fixed data. You might use these for hyperparameters that are fixed across a set of model runs, for example.
domain	An optional domain. Normally this is not wanted, but you can use this to truncate the domain of one or more parameters. The domain is effectively applied <i>after</i> the calculations implied by the DSL. The density is not recalculated to reflect the change in the marginal density. Applying a domain will remove the ability to sample from the model, at least for now. See monty_model for details on the format. The provided parameters must match the parameters of your model.

Value

A [monty_model](#) object derived from the expressions you provide.

Examples

```
# Expressions that define models can be passed in with no quoting
monty_dsl(a ~ Normal(0, 1))
monty_dsl({
  a ~ Normal(0, 1)
  b ~ Exponential(1)
})

# You can also pass strings
monty_dsl("a ~ Normal(0, 1)")
```

monty_dsl_distributions

Information about supported distributions

Description

Report information about supported distributions in the DSL. This is primarily intended for use in packages which use [monty_dsl_parse_distribution](#), as this function reports information about which distributions and arguments would succeed there.

Usage

```
monty_dsl_distributions()
```

Value

A data.frame with columns

- name the name of the distribution; each name begins with a capital letter, and there are duplicate names where different parameterisations are supported.
- args the arguments of all parameters, *except* the random variable itself which is given as the first argument to density functions.

We may expand the output here in the future to include information on if distributions have support in C++, but we might end up supporting everything this way soon.

Examples

```
monty_dsl_distributions()
```

`monty_dsl_error_explain`*Explain monty error*

Description

Explain error codes produced by monty. This is a work in progress, and we would like feedback on what is useful as we improve it. The idea is that if you see an error you can link through to get more information on what it means and how to resolve it. The current implementation of this will send you to the rendered vignettes, but in the future we will arrange for offline rendering too.

Usage

```
monty_dsl_error_explain(code, how = "pretty")
```

Arguments

<code>code</code>	The error code, as a string, in the form Exxx (a capital "E" followed by three numbers)
<code>how</code>	How to explain the error. Options are <code>pretty</code> (render pretty text in the console), <code>plain</code> (display plain text in the console) and <code>link</code> (browse to the online help).

Value

Nothing, this is called for its side effect only

Examples

```
monty_dsl_error_explain("E201")
```

`monty_dsl_parse_distribution`*Parse distribution expression*

Description

Parse an expression as if it were a call to one of monty's distribution functions (e.g., `Normal`, `Poisson`). This will fill in any defaults, disambiguate where multiple parameterisations of the distribution are available, and provide links through to the C++ API. This function is designed for use from other packages that use monty, and is unlikely to be useful to most users.

Usage

```
monty_dsl_parse_distribution(expr, name = NULL)
```

Arguments

expr	An expression
name	Name for the expression, used in constructing messages that you can use in errors.

Value

A list; the contents of this are subject to change. However you can (to a degree) rely on the following elements:

- name: The name of the distribution (e.g., Normal). This will be the same as the name of the function called in expr
- variant: The name of the distribution variant, if more than one is supported.
- args: The arguments that you provided, in position-matched order
- cpp: The names of the C++ entrypoint to use. This is a list with elements density and sample for the log-density and sampling functions, and NULL where these do not yet exist.

Currently we also include:

- density: A function to compute the log-density. This will likely change once we support creation of differentiable models because we will want to do something with the arguments provided!
- sample: A function to sample from the distribution, given (as a first argument) a monty_rng object (see [monty_rng_create](#))

Examples

```
# A successful match
monty_dsl_parse_distribution(quote(Normal(0, 1)))

# An unsuccessful match
monty_dsl_parse_distribution(quote(Normal()))
```

monty_example	<i>Load example models from monty. These models exist so that we can create (hopefully) interesting examples in the documentation without them becoming overwhelming. You should probably not use these for anything other than exploring the package.</i>
---------------	--

Description

Load small, ready-made target distributions for demonstrations, tests, and examples. These models exist so that we can create (hopefully) interesting examples in the documentation without them becoming overwhelming. All examples contain an analytic gradient and can thus be also used to explore gradient-based samplers such as [monty_sampler_hmc](#).

Usage

```
monty_example(name, ...)
```

Arguments

name	Name of the example, as a string. See Details for supported models.
...	Optional parameters that are passed to create the model. All models can be created with no additional parameters, but you can tweak their behaviour by passing named parameters here. See Details.

Value

A [monty_model](#) object.

Supported models:

Each model has **arguments** that are passed through as ... in `monty_example` and are used to set up the example. This creates a `monty_model` object that will accept **parameters**, which are the values in the argument parameters passed to `monty_model_density`.

banana:

The banana model is a two-dimensional banana-shaped function, with strong local correlation. This example was picked because it is useful for illustrating the limitations of random-walk proposals. The model has two parameters `alpha` and `beta` and is based on two successive draws, one conditional on the other.

You can vary the argument `sigma` for this model on creation, the default is 0.5

gaussian:

A multivariate Gaussian centred at the origin. Takes a variance-covariance-matrix as its argument. Parameters are letters `a`, `b`, ... up to the number of dimensions.

ring:

A two-dimensional ring-shaped density whose log-density is proportional to $-(\|x\| - r)^2 / (2sd^2)$. Includes an analytic gradient and a direct sampler in polar coordinates. The constructor takes that arguments `r` (default 3) and `sd` (default 0.2), and the parameters of the `monty_model` are `x1` and `x2` (corresponding to the two dimensions of the polar coordinates).

Increasing `r` or decreasing `sd` will make the ridge of high probability space narrower; this is easiest to visualise in true densities (rather than log density).

mixture2d:

An equal-weight mixture of `k` isotropic 2D Gaussians with standard deviation `sd` and centres either provided by the argument (`means`) or drawn uniformly in a square of half-width spread (which must be provided if `means` is not provided). Includes an analytic gradient.

See Also

[monty_model](#), [monty_model_properties](#)

Examples

```
# Small helper function to make the plotting easier
show_2d_example <- function(m, rx, ry = rx, n = 101) {
  x <- seq(rx[[1]], rx[[2]], length.out = n)
  y <- seq(ry[[1]], ry[[2]], length.out = n)
  xy <- unname(t(as.matrix(expand.grid(x, y))))
  z <- exp(matrix(monty_model_density(m, xy), n, n))
  image(x, y, z)
  contour(x, y, z, add = TRUE, drawlabels = FALSE,
          lwd = 0.5, col = "#00000088")
}

# Banana target
m_banana <- monty_example("banana", sigma = 0.2)
show_2d_example(m_banana, c(-0.5, 1.5), c(-1.5, 1.5))

# Ring target with radius 3 and narrow thickness
m_ring <- monty_example("ring", r = 3, sd = 0.25)
show_2d_example(m_ring, c(-4, 4))

# 2D Gaussian mixture with 20 components
m_mixture <- monty_example("mixture2d", k = 20, sd = 0.5, spread = 4)
show_2d_example(m_mixture, c(-5, 5))
```

monty_flatten_chains *Flatten chains*

Description

Flatten chains in the results of running `monty_sample()`, wherein pars, density and typically objects in observations will have the last two dimensions representing 'samples' and 'chains'. Flattening the chains results in the chains dimension being collapsed into the samples dimension.

Usage

```
monty_flatten_chains(samples)
```

Arguments

`samples` A `monty_samples` object, from running `monty_sample()`

Value

A `monty_samples` object with the chains dimension collapsed into the samples dimension.

monty_model

Create basic model

Description

Create a basic monty model. This takes a user-supplied object that minimally can compute a probability density (via a density function) and information about parameters; with this we can sample from the model using MCMC using [monty_sample](#). We don't imagine that many users will call this function directly, but that this will be glue used by packages.

Usage

```
monty_model(model, properties = NULL)
```

Arguments

model	A list or environment with elements as described in Details.
properties	Optionally, a monty_model_properties object, used to enforce or clarify properties of the model.

Details

The model argument can be a list or environment (something indexable by \$) and have elements:

- **density**: A function that will compute some probability density. It must take an argument representing a parameter vector (a numeric vector) and return a single value. This is the posterior probability density in Bayesian inference, but it could be anything really. Models can return $-\text{Inf}$ if things are impossible, and we'll try and cope gracefully with that wherever possible. If the property `allow_multiple_parameters` is `TRUE`, then this function must be able to handle the argument parameter being a matrix, and return a vector of densities.
- **parameters**: A character vector of parameter names. This vector is the source of truth for the length of the parameter vector.
- **domain**: Information on the parameter domain. This is a two column matrix with `length(parameters)` rows representing each parameter. The parameter minimum and maximum bounds are given as the first and second column. Infinite values ($-\text{Inf}$ or Inf) should be used where the parameter has infinite domain up or down. Currently used to translate from a bounded to unbounded space for HMC, but we might also use this for reflecting proposals in MCMC too, as well as a fast way of avoiding calculating densities where proposals fall out of bounds. If not present we assume that the model is valid everywhere (i.e., that all parameters are valid from $-\text{Inf}$ to Inf). If unnamed, you must provide a domain for all parameters. If named, then you can provide a subset, with parameters that are not included assumed to have a domain of $(-\text{Inf}, \text{Inf})$.
- **direct_sample**: A function to sample directly from the parameter space, given a `monty_rng` object to sample from (see [monty_rng_create](#)). In the case where a model returns a posterior (e.g., in Bayesian inference), this is assumed to be sampling from the prior. We'll use this for generating initial conditions for MCMC where those are not given, and possibly other uses. If not given then when using `monty_sample()` the user will have to provide a vector of initial states.

- **gradient**: A function to compute the gradient of density with respect to the parameter vector; takes a parameter vector and returns a vector the same length. For efficiency, the model may want to be stateful so that gradients can be efficiently calculated after a density calculation, or density after gradient, where these are called with the same parameters. This function is optional (and may not be well defined or possible to define).
- **set_rng_state**: A function to set the state (this is in contrast to the `rng` that is passed through to `direct_sample` as that is the *sampler's* rng stream, but we assume models will look after their own stream, and that they may need many streams). Models that provide this method are assumed to be stochastic; however, you can use the `is_stochastic` property (via `monty_model_properties()`) to override this (e.g., to run a stochastic model with its deterministic expectation). This function takes a raw vector of random number state from a `monty_rng` and uses it to set the random number state for your model; this is derived from the random number stream for a particular chain, jumped ahead.
- **get_rng_state**: A function to get the RNG state; must be provided if `set_rng_state` is present. Must return the random number state, which is a raw vector (potentially quite long).
- **parameter_groups**: Optionally, an integer vector indicating parameter group membership. The format here may change (especially if we explore more complex nestings) but at present parameters with group 0 affect everything (so are accepted or rejected as a whole), while parameters in groups 1 to `n` are independent (for example, changing the parameters in group 2 does not affect the density of parameters proposed in group 3).

Value

An object of class `monty_model1`. This will have elements:

- **model**: The model as provided
- **parameters**: The parameter name vector
- **parameter_groups**: The parameter groups
- **domain**: The parameter domain matrix, named with your parameters
- **direct_sample**: The `direct_sample` function, if provided by the model
- **gradient**: The gradient function, if provided by the model
- **properties**: A list of properties of the model; see `monty_model_properties()`. Currently this contains:
 - **has_gradient**: the model can compute its gradient
 - **has_direct_sample**: the model can sample from parameters space
 - **is_stochastic**: the model will behave stochastically
 - **has_parameter_groups**: The model has separable parameter groups

See Also

[monty_model_function](#), which provides a simple interface for creating models from functions.

monty_model_combine	<i>Combine two models</i>
---------------------	---------------------------

Description

Combine two models by multiplication. We'll need a better name here. In Bayesian inference we will want to create a model that represents the multiplication of a likelihood and a prior (in log space) and it will be convenient to think about these models separately. Multiplying probabilities (or adding on a log scale) is common enough that there may be other situations where we want to do this.

Usage

```
monty_model_combine(a, b, properties = NULL, name_a = "a", name_b = "b")
```

Arguments

a	The first model
b	The second model
properties	A monty_model_properties object, used to control (or enforce) properties of the combined model.
name_a	Name of the first model (defaulting to 'a'); you can use this to make error messages nicer to read, but it has no other practical effect.
name_b	Name of the first model (defaulting to 'b'); you can use this to make error messages nicer to read, but it has no other practical effect.

Details

Here we describe the impact of combining a pair of models

- **density**: this is the sum of the log densities from each model
- **parameters**: the union of parameters from each model is taken
- **domain**: The most restrictive domain is taken for each parameter. Parameters that do not appear in one model are assumed to have infinite domain there.
- **gradient**: if *both* models define a gradient, this is the sum of the gradients. If either does not define a gradient, the resulting model will not have gradient support. Set `has_gradient = TRUE` within 'properties' if you want to enforce that the combination is differentiable. If the models disagree in their parameters, parameters that are missing from a model are assumed (reasonably) to have a zero gradient.
- **direct_sample**: this one is hard to do the right thing for. If neither model can be directly sampled from that's fine, we don't directly sample. If only one model can be sampled from *and* if it can sample from the union of all parameters then we take that function (this is the case for a prior model when combined with a likelihood). Other cases will be errors, which can be avoided by setting `has_direct_gradient = FALSE` in properties.
- **is_stochastic**: a model is stochastic if *either* component is stochastic.

The properties of the model will be combined as above, reflecting the properties of the joint model. The model field will be an ordered, unnamed, list containing the two elements corresponding to the first and second model (not the `monty_model`, but the underlying model, perhaps?). This is the only part that makes a distinction between the two models here; for all components above they are equivalent.

Value

A `monty_model` object

Examples

```
# A simple example; a model that contains something of interest,
# and a simple prior from monty_dsl
likelihood <- monty_example("banana")
prior <- monty_dsl({
  alpha ~ Normal(0, 1)
  beta ~ Normal(0, 10)
})
posterior <- likelihood + prior
posterior

# The same thing, more explicitly:
monty_model_combine(likelihood, prior)

# Control properties of the combined model:
monty_model_combine(likelihood, prior,
  monty_model_properties(has_gradient = FALSE))
```

monty_model_density	<i>Compute log density</i>
---------------------	----------------------------

Description

Compute log density for a model. This is a wrapper around the `$density` property within a `monty_model` object.

Usage

```
monty_model_density(model, parameters)
```

Arguments

model	A <code>monty_model</code> object
parameters	A vector or matrix of parameters

Value

A log-density value, or vector of log-density values

See Also

[monty_model_gradient](#) for computing gradients and [monty_model_direct_sample](#) for sampling from a model.

Examples

```
m <- monty_model_function(function(a, b) dnorm(0, a, b, log = TRUE))
monty_model_density(m, c(0, 1))
monty_model_density(m, c(0, 10))
```

monty_model_direct_sample

Directly sample from a model

Description

Directly sample from a model. Not all models support this, and an error will be thrown if it is not possible.

Usage

```
monty_model_direct_sample(model, rng, named = FALSE)
```

Arguments

model	A monty_model object
rng	Random number state, created by monty_rng_create . Use of an RNG with more than one stream may or may not work as expected; this is something we need to tidy up (mrc-5292)
named	Logical, indicating if the output should be named using the parameter names.

Value

A vector or matrix of sampled parameters

Examples

```
m <- monty_example("banana")

r <- monty_rng_create()
monty_model_direct_sample(m, r)
monty_model_direct_sample(m, r, named = TRUE)

r <- monty_rng_create(n_streams = 3)
monty_model_direct_sample(m, r)
monty_model_direct_sample(m, r, named = TRUE)
```

monty_model_function *Create monty_model from a function computing density*

Description

Create a [monty_model](#) from a function that computes density. This allows use of any R function as a simple monty model. If you need advanced model features, then this interface may not suit you and you may prefer to use [monty_model](#) directly.

Usage

```
monty_model_function(
  density,
  packer = NULL,
  fixed = NULL,
  domain = NULL,
  allow_multiple_parameters = FALSE
)
```

Arguments

density	A function to compute log density. It can take any number of parameters
packer	Optionally, a monty_packer object to control how your function parameters are packed into a numeric vector. You can typically omit this if all the arguments to your functions are present in your numeric vector and if they are all scalars.
fixed	Optionally, a named list of fixed values to substitute into the call to density. This cannot be used in conjunction with packer (you should use the fixed argument to monty_packer instead).
domain	Optional domain, see monty_model 's arguments for details. You can use "logical" names for array parameters and these will be expanded as described in monty_domain_expand() .
allow_multiple_parameters	Logical, indicating if passing in vectors for all parameters will return a vector of densities. This is FALSE by default because we cannot determine this automatically. Be aware that R's recycling rules may mean that this will not always work as expected!

Details

This interface will expand in future versions of monty to support gradients, stochastic models, parameter groups and simultaneous calculation of density.

Value

A [monty_model](#) object that computes log density with the provided density function, given a numeric vector argument representing all parameters.

Examples

```

banana <- function(a, b, sd) {
  dnorm(b, log = TRUE) + dnorm((a - b^2) / sd, log = TRUE)
}
m <- monty_model_function(banana, fixed = list(sd = 0.25))
m

# Density from our new model. Note that this computes density
# using an unstructured parameter vector, which is mapped to 'a'
# and 'b':
monty_model_density(m, c(0, 0))

# Same as the built-in banana example:
monty_model_density(monty_example("banana"), c(0, 0))

# You can constrain parameters, for example:
fn <- function(a, b) {
  dbeta(a, 2, 5, log = TRUE) + dnorm(b, log = TRUE)
}

# Here 'a' must lie in [0, 1] so we pass this through as 'domain':
pr <- monty_model_function(fn, domain = rbind(a = c(0, 1)))

# Now, out-of-bounds values will be converted to -Inf:
monty_model_density(pr, c(0.5, 0.5)) # ok
monty_model_density(pr, c(0.5, 5)) # -Inf

```

monty_model_gradient *Compute gradient of log density*

Description

Compute the gradient of log density (which is returned by [monty_model_density](#)) with respect to parameters. Not all models support this, and an error will be thrown if it is not possible.

Usage

```
monty_model_gradient(model, parameters, named = FALSE)
```

Arguments

model	A monty_model object
parameters	A vector or matrix of parameters
named	Logical, indicating if the output should be named using the parameter names.

Value

A vector or matrix of gradients

See Also

[monty_model_density](#) for log density, and [monty_model_direct_sample](#) to sample from a model

Examples

```
m <- monty_example("banana")
# Global maximum at (0, 0), and gradient is zero there:
monty_model_density(m, c(0, 0))
monty_model_gradient(m, c(0, 0))

# Nonzero gradient away from the origin:
monty_model_gradient(m, c(0.4, 0.2))
```

monty_model_properties

Describe model properties

Description

Describe properties of a model. Use of this function is optional, but you can pass the return value of this as the `properties` argument of `monty_model` to enforce that your model does actually have these properties.

Usage

```
monty_model_properties(
  has_gradient = NULL,
  has_direct_sample = NULL,
  is_stochastic = NULL,
  has_parameter_groups = NULL,
  has_observer = NULL,
  allow_multiple_parameters = NULL
)
```

Arguments

<code>has_gradient</code>	Logical, indicating if the model has a gradient method. Use NULL (the default) to detect this from the model.
<code>has_direct_sample</code>	Logical, indicating if the model has a <code>direct_sample</code> method. Use NULL (the default) to detect this from the model.
<code>is_stochastic</code>	Logical, indicating if the model is stochastic. Stochastic models must supply <code>set_rng_state</code> and <code>get_rng_state</code> methods.
<code>has_parameter_groups</code>	Logical, indicating that the model can be decomposed into parameter groups which are independent of each other. This is indicated by using the <code>parameter_groups</code> field within the model object passed to monty_model , and by the presence of a

	by_group argument to density and (later we may also support this in gradient). Use NULL (the default) to detect this from the model.
has_observer	Logical, indicating if the model has an "observation" function, which we will describe more fully soon. An observer is a function observe which takes no arguments and returns arbitrary data about the previously evaluated density. Use NULL (the default) to detect this from the model.
allow_multiple_parameters	Logical, indicating if the density calculation can support being passed a matrix of parameters (with each column corresponding to a different parameter set) and return a vector of densities. If FALSE, we will support some different approaches to sort this out for you if this feature is needed. This cannot be detected from the model, and the default is FALSE because it's not always straightforward to implement. However, where it is possible it may be much more efficient (via vectorisation or parallelisation) to do this yourself.

Value

A list of class `monty_model_properties` which should not be modified.

Examples

```
# Default properties:
monty_model_properties()

# Set some properties:
monty_model_properties(has_gradient = TRUE, is_stochastic = FALSE)
```

<code>monty_model_split</code>	<i>Split a combined model</i>
--------------------------------	-------------------------------

Description

Split a model that has been combined by `monty_model_combine()` into its constituent parts.

Usage

```
monty_model_split(model, prior_first = FALSE)
```

Arguments

<code>model</code>	A combined model
<code>prior_first</code>	Logical, indicating that we should require that the model component that could be the prior is listed first. If TRUE and one component model is not plausibly the prior, we will error. See Details for the heuristic used.

Details

We assume that a split model can be broken into a "prior" and a "likelihood" if exactly one model:

- can be directly sampled from
- is not stochastic
- consumes all parameters

Typically, it will be the first criterion that will separate a model into prior and likelihood (if you could sample from your likelihood, then you would not use a sampler, which is where we are typically going to perform this action).

If `prior_first` is `FALSE` we just return the parts.

Value

An unnamed list of length 2, being the component models. If one model might be the prior it will be listed first.

monty_observer	<i>Create observer</i>
----------------	------------------------

Description

Create an observer to extract additional details from your model during the sampling process.

Usage

```
monty_observer(observe, finalise = NULL, combine = NULL, append = NULL)
```

Arguments

observe	A function that will run with arguments <code>model</code> (the model that you passed in to monty_model) and <code>rng</code> (an rng object). This function should return a list. It is best if the list returned is named, with no duplicated names, and with return values that have the same exact dimensions for every iteration. If you do this, then you will not have to provide any of the following arguments, which are going to be hard to describe and worse to implement.
finalise	A function that runs after a single chain has run, and you use to simplify across all samples drawn from that chain. Takes a single argument which is the list with one set of observations per sample.
combine	A function that runs after all chains have run, and you use to simplify across chains. Takes a single argument, which is the list with one set of observations per chain.
append	A function that runs after a continuation of chain has run (via monty_sample_continue). Takes two arguments representing the fully simplified observations from the first and second chains.

Details

Sometimes you want to extract additional information from your model as your chain runs. The case we see this most is when running MCMC with a particle filter (pmcmc); in this case while the likelihood calculation is running we are computing lots of interesting quantities such as the final state of the system (required for onward simulation) and filtered trajectories through time. Because these are stochastic we can't even just rerun the model with our sampled parameter sets, because the final states that are recovered depend also on the random number generators (practically we would not want to, as it is quite expensive to compute these quantities).

The observer mechanism allows you to carry out arbitrary additional calculations with your model at the end of the step.

Value

An object with class `monty_observer` which can be passed in to `monty_sample`.

monty_packer

Build a packer

Description

Build a packer, which can be used to translate between an unstructured vector of numbers (the vector being updated by an MCMC for example) and a structured list of named values, which is easier to program against. This is useful for the bridge between model parameters and a model's implementation, but it is also useful for the state vector in a state-space model. We refer to the process of taking a named list of scalars, vectors and arrays and converting into a single vector "packing" and the inverse "unpacking".

Usage

```
monty_packer(scalar = NULL, array = NULL, fixed = NULL, process = NULL)
```

Arguments

scalar	Names of scalars. This is similar for listing elements in array with values of 1, though elements in scalar will be placed ahead of those listed in array within the final parameter vector, and elements in array will have generated names that include square brackets.
array	A list, where names correspond to the names of arrays and values correspond to their lengths. Multiple dimensions are allowed (so if you provide an element with two entries these represent dimensions of a matrix). Zero-length integer vectors or NULL values are counted as scalars, which allows you to put scalars at positions other than the front of the packing vector. In future, you may be able to use <i>strings</i> as values for the lengths, in which case these will be looked for within fixed.

fixed	A named list of fixed data to be inserted into the final unpacked list; these will be added into the final list directly. In the parameter packer context, these typically represent additional pieces of data that your model needs to run, but which you are not performing inference on.
process	An arbitrary R function that will be passed the final assembled list; it may create any <i>additional</i> entries, which will be concatenated onto the original list. If you use this you should take care not to return any values with the same names as entries listed in <code>scalar</code> , <code>array</code> or <code>fixed</code> , as this is an error (this is so that <code>pack()</code> is not broken). We will likely play around with this process in future in order to get automatic differentiation to work.

Details

There are several places where it is most convenient to work in an unstructured vector:

- An MCMC is typically discussed as the updating of some vector x to another x'
- An optimisation algorithm will try and find a set of values for a vector x that minimises (or maximises) some function $f(x)$
- An ode solver works with a vector $x(t)$ (x at time t) and considers $x(t+h)$ by computing the vector of derivatives $dx(t)/dt$

In all these cases, the algorithm that needs the vector of numbers knows nothing about what they represent. Commonly, these will be a packed vector of parameters. So our vector x might actually represent the parameters a , b and c in a vector as $[a, b, c]$ - this is a very common pattern, and you have probably implemented this yourself.

In more complex settings, we might want our vector x to collect more structured quantities. Suppose that you are fitting a model with an age-structured or sex-structured parameter. Rather than having a series of scalars packed into your vector x you might have a series of values destined to be treated as a vector:

```
| 1 2 3 4 5 6 7 |
| a b c d1 d2 d3 d4 |
```

So here we might have a vector of length 7, where the first three elements will represent the scalar values a , b and c but the next four will be a vector d .

Unpacked, this might be written as:

```
list(a = 1, b = 2, c = 3, d = 4:7)
```

The machinery here is designed to make these transformations simple and standardised within `monty`, and should be flexible enough for many situations. We will also use these from within `dust2` and `odin2` for transformations in and out of vectors of ODE state.

Value

An object of class `monty_packer`, which has elements:

- `names`: a function that returns a character vector of computed names; in the parameter packer context these are the names that your statistical model will use.

- `unpack`: a function that can unpack an unstructured vector (say, from your statistical model parameters) into a structured list (say, for your generative model)
- `pack`: a function that can pack your structured list of data back into a numeric vector, for example suitable for a statistical model. This ignores values created by a preprocess function and present in `fixed`.
- `index`: a function which produces a named list where each element has the name of a value in scalar or array and each value has the indices within an unstructured vector where these values can be found, in the shape of the data that would be unpacked. This is of limited most use to most people.
- `subset`: an experimental interface which can be used to subset a packer to a packer for a subset of contents. Documentation will be provided once the interface settles, but this is for advanced use only!
- `inputs`: inputs that could be used to reconstitute this packer.

When to use process

The `process` function is a get-out-of-jail function designed to let you do arbitrary transformations when unpacking a vector. In general, this should not be the first choice to use because it is less easy to reason about by other tooling (for example, as we develop automatic differentiation support for use with the HMC algorithm, a `process` function will be problematic because we will need to make sure we can differentiate this `process`). However, there are cases where it will be only way to achieve some results.

Imagine that you are packing a 2x2 covariance matrix into your vector in order to use within an MCMC or optimisation algorithm. Ultimately, our unpacked vector will need to hold four elements (`b11`, `b12`, `b21`, `b22`), but there are only three distinct values as the two off-diagonal elements will be the same (i.e., `b12 == b21`). So we might write this passing in `b_raw = 3` to `array`, so that our unpacked list holds `b_raw = c(b11, b12, b22)`. We would then write `process` as something like:

```
process <- function(x) {
  list(b = matrix(x$b_raw[c(1, 2, 2, 3)], 2, 2))
}
```

which creates the symmetric 2x2 matrix `b` from `b_raw`.

Unpacking matrices

If you do not use `fixed` or `process` when defining your packer, then you can use `$unpack()` with a matrix or higher-dimensional output. There are two ways that you might like to unpack this sort of output. Assume you have a matrix `m` with 3 rows and 2 columns; this means that we have two sets of parameters or state (one per column) and 3 states within each; this is the format that MCMC parameters will be in for example.

The first would be to return a list where the *i*th element is the result of unpacking the *i*th parameter/state vector. You can do this by running

```
apply(m, 2, p$unpack)
```

The second would be to return a named list with three elements where the *i*th element is the unpacked version of the *i*th state. In this case you can pass the matrix directly in to the unpacker:

```
p$unpack(m)
```

When you do this, the elements of `m` will acquire an additional dimension; scalars become vectors (one per set), vectors become matrices (one column per set) and so on.

This approach generalises to higher dimensional input, though we suspect you'll spend a bit of time head-scratching if you use it.

Packing lists into vectors and matrices

The unpacking operation is very common - an MCMC proceeds, produces an unstructured vector, and you unpack it into a list in order to be able to easily work with it. The reverse is much less common, where we take a list and convert it into a vector (or matrix, or multidimensional array). Use of this direction ("packing") may be more common where using packers to work with the output of state-space models (e.g. in `odin2` or `dust2`, which use this machinery).

The input to `pack()` will be the shape that `unpack()` returned; a named list of numerical vectors, matrices and arrays. The names must correspond to the names in your packer (i.e., `scalar` and the names of array). Each element has dimensions

```
<...object, ...residual>
```

where `...object` is the dimensions of the data itself and `...residual` is the dimensions of the hypothetical input to pack.

There is an unfortunate ambiguity in R's lack of true scalar types that we cannot avoid. It is hard to tell the difference packing a vector vs packing an array where all dimensions are 1. See the examples, and please let us know if the behaviour needs changing.

Examples

```
# Here's a really simple example
p <- monty_packer(c("a", "b", "c"))
p

p$pack(list(a = 1, b = 2, c = 3))
p$unpack(1:3)

# Sometimes we have a vector embedded in our parameters:
p <- monty_packer(c("a", "b"), list(v = 4))
p$pack(list(a = 1, b = 2, v = c(6, 7, 8, 9)))
p$unpack(c(1, 2, 6, 7, 8, 9))

# Or a higher dimensional structure such as a matrix:
p <- monty_packer(c("a", "b"), list(m = c(2, 2)))
p$unpack(c(1, 2, 6, 7, 8, 9))

# You can use a packer to set "fixed" parameters that do not vary
# with the underlying model being fit, but are required by your model.
# This is simpler than the "closure" approach used previously in our
# mcstate package and also easier to accommodate with differentiable
# models:
p <- monty_packer(
```

```

    c("a", "b"),
    fixed = list(d = data.frame(n = 1:3, m = runif(3))))
p$unpack(1:2)
p$pack(p$unpack(1:2))

# The example from above, where we create a symmetric 2 x 2 matrix
# from a 3-element vector, alongside a scalar:
p <- monty_packer(
  scalar = "a",
  array = list(b_flat = 3),
  process = function(p) list(b = matrix(p$b_flat[c(1, 2, 2, 3)], 2, 2)))

# Unpacking we see "b_flat" is still in the list, but "b" is our
# symmetric matrix:
p$unpack(1:4)

# The processed elements are ignored on the return pack:
p$pack(list(a = 1, b_flat = 2:4, b = matrix(c(2, 3, 3, 4), 2, 2)))
p$pack(list(a = 1, b_flat = 2:4))

# R lacks scalars, which means that some packers will unpack
# different inputs to the same outputs:
p <- monty_packer(c("a", "b"))
p$unpack(1:2)
p$unpack(cbind(1:2))

# This means that we can't reliably pack these inputs in a way
# that guarantees round-tripping is possible. We have chosen to
# prioritise the case where a *single vector* is round-trippable:
p$pack(list(a = 1, b = 2))

# This ambiguity goes away if unpacking matrices with more than one
# column:
p$unpack(matrix(1:6, 2, 3))

```

monty_packer_grouped *Build a nested packer*

Description

Build a grouped version of `monty_packer()` with the same basic idea; convert between a vector representation of some group of numbers to a named list of structured data, but with an extra twist: this time the unstructured vector of numbers contains values that correspond to multiple *groups* and some are shared across groups while others vary between groups. This function does a lot of bookkeeping in a relatively short amount of code, so you should be familiar with the ideas in `monty_packer()` before continuing.

Usage

```
monty_packer_grouped(
```

```

    groups,
    scalar = NULL,
    array = NULL,
    fixed = NULL,
    process = NULL,
    shared = NULL
  )

```

Arguments

groups	A character vector of group names. These must not be present within any of your scalar or array arguments.
scalar	Names of scalars. This is similar for listing elements in array with values of 1, though elements in scalar will be placed ahead of those listed in array within the final parameter vector, and elements in array will have generated names that include square brackets.
array	A list, where names correspond to the names of arrays and values correspond to their lengths. Multiple dimensions are allowed (so if you provide an element with two entries these represent dimensions of a matrix). Zero-length integer vectors or NULL values are counted as scalars, which allows you to put scalars at positions other than the front of the packing vector. In future, you may be able to use <i>strings</i> as values for the lengths, in which case these will be looked for within fixed.
fixed	A named list of fixed data to be inserted into the final unpacked list; these will be added into the final list directly. In the parameter packer context, these typically represent additional pieces of data that your model needs to run, but which you are not performing inference on.
process	An arbitrary R function that will be passed the final assembled list for each group ; it may create any <i>additional</i> entries, which will be concatenated onto the original list. If you use this you should take care not to return any values with the same names as entries listed in scalar, array or fixed, as this is an error (this is so that pack() is not broken). We will likely play around with this process in future in order to get automatic differentiation to work.
shared	Names of the elements in scalar and array that are shared among all groups.

Details

Recall from `monty_packer()` that our original problem was to take an unstructured vector like

```

| 1 2 3 4 5 6 7 |
| a b c d1 d2 d3 d4 |

```

and **unpack it** into a structured list like

```
list(a = 1, b = 2, c = 3, d = 4:7)
```

Our aim here is to do the same but to allow some of these values (say b and c) to be shared (constant) over groups while the others (a and d) to vary by group. So for groups x and y we might try and create something like

```
list(
  list(x = list(a = 3, b = 1, c = 2, d = 4:7),
        y = list(a = 8, b = 1, c = 2, d = 9:12))
```

from a vector

```
| 1 2 3 4 5 6 7 8 9 10 11 12 |
| b c a d1 d2 d3 d4 a d1 d2 d3 d4 |
| xy xy x x x x x y y y y y |
```

Value

An object of class `monty_packer_grouped`, which has the same elements as `monty_packer`, though with slightly different effects.

- `names`: a function that returns a character vector of computed names; in the parameter packer context these are the names that your statistical model will use.
- `groups`: A function that returns your group names (the `groups` argument as supplied)
- `unpack`: A function for converting from an unstructured vector into a nested list. Each element of this list is conceptually the same as the result of `unpack()` from [monty_packer\(\)](#).
- `pack`: The inverse to `unpack()` but less commonly performed. Convert a nested list into an unstructured vector. Quite a lot of validation is required to make sure that the input has not been tampered with, and errors thrown while doing this validation may not be very interpretable.
- `index`: The nested version of the `index()` function in [monty_packer\(\)](#). The outer list is over groups, and the inner list contains the position within the original unstructured vector where this value can be found. It is not clear to us if this is a useful list.
- `subset`: A function that might eventually allow subsetting a grouped packer. Currently it just errors.

Examples

```
p <- monty_packer_grouped(c("x", "y"), c("a", "b", "c", "d", "e"),
                          shared = c("b", "c"))
p$names()
p$unpack(1:8)
```

monty_random_beta	<i>Sample from beta distribution</i>
-------------------	--------------------------------------

Description

Sample from the beta distribution

Usage

```
monty_random_beta(a, b, state)
```

```
monty_random_n_beta(n_samples, a, b, state)
```

Arguments

a, b	The shape parameters
state	The random number state, from monty_rng_create
n_samples	The number of samples to take, per stream . When using the multiple-sample interface, all other parameters are held constant (per stream).

Value

A vector of random numbers, the same length as the number of streams in state.

monty_random_beta_binomial_prob	<i>Sample from beta-binomial distribution</i>
---------------------------------	---

Description

Sample from a beta-binomial distribution. There are two parameterisations available: one in terms of probability and dispersion and the other in terms of two shape parameters.

Usage

```
monty_random_beta_binomial_prob(size, prob, rho, state)
```

```
monty_random_n_beta_binomial_prob(n_samples, size, prob, rho, state)
```

```
monty_random_beta_binomial_ab(size, a, b, state)
```

```
monty_random_n_beta_binomial_ab(n_samples, size, a, b, state)
```

Arguments

size	The number of trials (zero or more)
prob	The mean probability of success on each trial (between 0 and 1)
rho	The dispersion parameter (between 0 and 1)
state	The random number state, from monty_rng_create
n_samples	The number of samples to take, per stream . When using the multiple-sample interface, all other parameters are held constant (per stream).
a	The first shape parameter (zero or more)
b	The second shape parameter (zero or more)

Value

A vector of random numbers, the same length as the number of streams in state.

monty_random_binomial *Sample from binomial distribution*

Description

Sample from the binomial distribution

Usage

```
monty_random_binomial(size, prob, state)
```

```
monty_random_n_binomial(n_samples, size, prob, state)
```

Arguments

size	The number of trials
prob	The probability of success on each trial
state	The random number state, from monty_rng_create
n_samples	The number of samples to take, per stream . When using the multiple-sample interface, all other parameters are held constant (per stream).

Value

A vector of random numbers, the same length as the number of streams in state.

Examples

```
state <- monty_rng_create()
monty_random_binomial(10, 0.3, state)
table(monty_random_n_binomial(2000, 10, 0.3, state))
```

monty_random_cauchy *Sample from Cauchy distribution*

Description

Sample from the Cauchy distribution

Usage

```
monty_random_cauchy(location, scale, state)
```

```
monty_random_n_cauchy(n_samples, location, scale, state)
```

Arguments

location	Location of the distribution (the same as the median and mode)
scale	A scale parameter which specifies the half-width at half-maximum (HWHM)
state	The random number state, from monty_rng_create
n_samples	The number of samples to take, per stream . When using the multiple-sample interface, all other parameters are held constant (per stream).

Value

A vector of random numbers, the same length as the number of streams in state.

monty_random_exponential_rate
 Sample from exponential distribution

Description

Sample from an exponential distribution. There are two parameterisations here, one in terms of the rate of the exponential, and one in terms of the mean (or scale).

Usage

```
monty_random_exponential_rate(rate, state)
```

```
monty_random_n_exponential_rate(n_samples, rate, state)
```

```
monty_random_exponential_mean(mean, state)
```

```
monty_random_n_exponential_mean(n_samples, mean, state)
```

Arguments

rate	The rate of the exponential
state	The random number state, from monty_rng_create
n_samples	The number of samples to take, per stream . When using the multiple-sample interface, all other parameters are held constant (per stream).
mean	The mean of the exponential distribution (i.e., $1 / \text{rate}$)

Value

A vector of random numbers, the same length as the number of streams in state.

Examples

```
state <- monty_rng_create()
monty_random_exponential_rate(0.2, state)
summary(monty_random_n_exponential_rate(2000, 0.2, state))
```

monty_random_gamma_scale

Sample from a gamma distribution. There are two parameterisations here, one in terms of rate, and one in terms of scale.

Description

Sample from gamma distribution

Usage

```
monty_random_gamma_scale(shape, scale, state)

monty_random_n_gamma_scale(n_samples, shape, scale, state)

monty_random_gamma_rate(shape, rate, state)

monty_random_n_gamma_rate(n_samples, shape, rate, state)
```

Arguments

shape	Shape
scale	Scale '
state	The random number state, from monty_rng_create
n_samples	The number of samples to take, per stream . When using the multiple-sample interface, all other parameters are held constant (per stream).
rate	Rate

Value

A vector of random numbers, the same length as the number of streams in state.

monty_random_hypergeometric
Sample from hypergeometric distribution

Description

Sample from a hypergeometric distribution.

Usage

```
monty_random_hypergeometric(n1, n2, k, state)
```

```
monty_random_n_hypergeometric(n_samples, n1, n2, k, state)
```

Arguments

n1	The number of white balls in the urn (called n in R's rhyper)
n2	The number of black balls in the urn (called m in R's rhyper)
k	The number of balls to draw
state	The random number state, from monty_rng_create
n_samples	The number of samples to take, per stream . When using the multiple-sample interface, all other parameters are held constant (per stream).

Value

A vector of random numbers, the same length as the number of streams in state.

monty_random_log_normal
Sample from log-normal

Description

Sample from a log-normal distribution

Usage

```
monty_random_log_normal(meanlog, sdlog, state)
```

```
monty_random_n_log_normal(n_samples, meanlog, sdlog, state)
```

Arguments

meanlog	The mean of the distribution on the log scale
sdlog	The standard deviation of the distribution on the log scale
state	The random number state, from monty_rng_create
n_samples	The number of samples to take, per stream . When using the multiple-sample interface, all other parameters are held constant (per stream).

Value

A vector of random numbers, the same length as the number of streams in state.

monty_random_negative_binomial_prob

Sample from negative binomial distribution

Description

Sample from a negative binomial distribution. There are two parameterisations available: one in based on the success probability and the other based on the mean.

Usage

```
monty_random_negative_binomial_prob(size, prob, state)
```

```
monty_random_n_negative_binomial_prob(n_samples, size, prob, state)
```

```
monty_random_negative_binomial_mu(size, mu, state)
```

```
monty_random_n_negative_binomial_mu(n_samples, size, mu, state)
```

Arguments

size	The target number of successful trials (zero or more)
prob	The probability of success on each trial (between 0 and 1)
state	The random number state, from monty_rng_create
n_samples	The number of samples to take, per stream . When using the multiple-sample interface, all other parameters are held constant (per stream).
mu	The mean (zero or more)

Value

A vector of random numbers, the same length as the number of streams in state.

monty_random_normal *Sample from normal distribution*

Description

Sample from a normal distribution

Usage

```
monty_random_normal(mean, sd, state, algorithm = "box_muller")
```

```
monty_random_n_normal(n_samples, mean, sd, state, algorithm = "box_muller")
```

Arguments

mean	The mean of the normal distribution
sd	The standard deviation of the normal distribution
state	The random number state, from monty_rng_create
algorithm	The algorithm to use for the normal samples; currently box_muller, polar and and ziggurat are supported, with the latter being considerably faster. The default may change in a future version.
n_samples	The number of samples to take, per stream . When using the multiple-sample interface, all other parameters are held constant (per stream).

Value

A vector of random numbers, the same length as the number of streams in state.

monty_random_poisson *Sample from Poisson distribution*

Description

Sample from the Poisson distribution

Usage

```
monty_random_poisson(lambda, state)
```

```
monty_random_n_poisson(n_samples, lambda, state)
```

Arguments

lambda	The mean (zero or more, length 1 or n). Only valid for lambda <= 10^7
state	The random number state, from monty_rng_create
n_samples	The number of samples to take, per stream . When using the multiple-sample interface, all other parameters are held constant (per stream).

Value

A vector of random numbers, the same length as the number of streams in state.

monty_random_real	<i>Sample from Uniform(0, 1)</i>
-------------------	----------------------------------

Description

Generate a random number uniformly sampled on the range 0 to 1; this is the most basic of all random number functions in monty and all other algorithms are composed from this. Quite often, you will want a number on $[0, 1]$ (e.g., for a Bernoulli trial), and this function is the most efficient way of generating one.

Usage

```
monty_random_real(state)

monty_random_n_real(n_samples, state)
```

Arguments

state	The random number state, from monty_rng_create
n_samples	The number of samples to take, per stream . When using the multiple-sample interface, all other parameters are held constant (per stream).

Value

A vector of random numbers, the same length as the number of streams in state.

Examples

```
state <- monty_rng_create()
monty_random_real(state)
monty_random_n_real(5, state)
```

monty_random_truncated_normal
Sample from truncated normal

Description

Sample from a truncated normal distribution

Usage

monty_random_truncated_normal(mean, sd, min, max, state)

monty_random_n_truncated_normal(n_samples, mean, sd, min, max, state)

Arguments

mean	The mean of the parent (untruncated) normal distribution (this is not necessarily the mean of the truncated distribution, unless the min and max are symmetrically placed around mean)
sd	The standard deviation of the parent distribution (this is not the same as the standard deviation of the truncated distribution with finite bounds).
min	The lower bound (can be $-\text{Inf}$).
max	The upper bound (can be Inf).
state	The random number state, from monty_rng_create
n_samples	The number of samples to take, per stream . When using the multiple-sample interface, all other parameters are held constant (per stream).

Value

A vector of random numbers, the same length as the number of streams in state.

monty_random_uniform *Sample from uniform distribution*

Description

Sample from a uniform distribution

Usage

monty_random_uniform(min, max, state)

monty_random_n_uniform(n_samples, min, max, state)

Arguments

min	The minimum value of the uniform distribution
max	The maximum value of the uniform distribution
state	The random number state, from monty_rng_create
n_samples	The number of samples to take, per stream . When using the multiple-sample interface, all other parameters are held constant (per stream).

Value

A vector of random numbers, the same length as the number of streams in state.

monty_random_weibull	<i>Sample from Weibull</i>
----------------------	----------------------------

Description

Sample from a Weibull distribution

Usage

```
monty_random_weibull(shape, scale, state)
```

```
monty_random_n_weibull(n_samples, shape, scale, state)
```

Arguments

shape	Shape
scale	Scale
state	The random number state, from monty_rng_create
n_samples	The number of samples to take, per stream . When using the multiple-sample interface, all other parameters are held constant (per stream).

Value

A vector of random numbers, the same length as the number of streams in state.

monty_random_zi_negative_binomial_prob

Sample from zero-inflated negative binomial distribution

Description

Sample from a zero-inflated negative binomial distribution. With probability π_0 a 0 is drawn, otherwise it draws from the corresponding standard negative binomial distribution (which may also result in a 0 being drawn). As with the standard negative binomial distribution, there are two parameterisations available, based on either the success probability or mean of the negative binomial part of the distribution.

Usage

```
monty_random_zi_negative_binomial_prob(pi0, size, prob, state)
```

```
monty_random_n_zi_negative_binomial_prob(n_samples, pi0, size, prob, state)
```

```
monty_random_zi_negative_binomial_mu(pi0, size, mu, state)
```

```
monty_random_n_zi_negative_binomial_mu(n_samples, pi0, size, mu, state)
```

Arguments

pi0	The probability of excess zeros
size	The target number of successful trials (zero or more)
prob	The probability of success on each trial (between 0 and 1)
state	The random number state, from monty_rng_create
n_samples	The number of samples to take, per stream . When using the multiple-sample interface, all other parameters are held constant (per stream).
mu	The mean (zero or more) of the negative binomial part of the distribution. Note the mean of the zero-inflated negative binomial distribution will actually be $(1 - \pi_0) * \mu$

Value

A vector of random numbers, the same length as the number of streams in state.

monty_random_zi_poisson

Sample from zero-inflated Poisson distribution

Description

Sample from the zero-inflated Poisson distribution. With probability `pi0` a 0 is drawn, otherwise it draws from the corresponding standard Poisson distribution (which may also result in a 0 being drawn).

Usage

```
monty_random_zi_poisson(pi0, lambda, state)
```

```
monty_random_n_zi_poisson(n_samples, pi0, lambda, state)
```

Arguments

<code>pi0</code>	The probability of extra zeros
<code>lambda</code>	The mean (zero or more, length 1 or n) of the Poisson part of the distribution. Only valid for $\lambda \leq 10^7$. Note that the mean of the zero-inflated Poisson distribution will actually be $(1 - \text{pi0}) * \lambda$
<code>state</code>	The random number state, from monty_rng_create
<code>n_samples</code>	The number of samples to take, per stream . When using the multiple-sample interface, all other parameters are held constant (per stream).

Value

A vector of random numbers, the same length as the number of streams in `state`.

monty_rng_create

Create a monty random number generator

Description

Create a monty random number generator. This allows you to sample random numbers from the same random number algorithms as monty provides via C++ to [dust2](#), and which it uses within its samplers and filters. This function creates the internal state, and will be passed to actual generation functions `monty_random_*`, such as [monty_random_real\(\)](#).

Usage

```
monty_rng_create(
  n_streams = 1L,
  seed = NULL,
  n_threads = 1L,
  deterministic = FALSE,
  preserve_stream_dimension = FALSE
)
```

Arguments

<code>n_streams</code>	The number of streams to create (see Details)
<code>seed</code>	The initial seed of the random number generator. This can be <code>NULL</code> , in which case we seed the generator from R's random number state (meaning that we respond to <code>set.seed</code> as one would expect). Alternatively, you can provide an integer here, but this should be used sparingly and primarily for testing.
<code>n_threads</code>	The number of threads to use, if OpenMP is enabled.
<code>deterministic</code>	Logical, indicating if we should use "deterministic" mode where distributions return their expectations and the state is never changed.
<code>preserve_stream_dimension</code>	Logical, indicating if the stream dimension should be preserved in the case where <code>n_streams</code> is 1 and the multiple-sample functions are used. Set this to <code>TRUE</code> to ensure that the rank of the result does not change with the number of streams (see Details).

Details

Monty's random number generation is very different to that in base R. We have the concept of "streams" of random numbers, with a generator having 1 or many streams. Each stream is statistically independent, and can be sampled from simultaneously. When you use any of the random number functions from R, you will draw one number *per stream*.

Because the random number state can have multiple streams, and we return a vector, we have a separate set of functions where multiple numbers are requested *per stream*; these are all prefixed `monty_random_n_` (e.g., `monty_random_n_real()`). These will return a matrix where you have used multiple streams, with each column representing a stream. If you have a single stream and you set `preserve_stream_dimension` to `FALSE` then we will drop this dimension and return a matrix.

Value

An object of class `monty_rng_state`, which can be passed as the `state` argument to random-number producing functions, such as `monty_random_real`

Examples

```
state <- monty_rng_create()
state

monty_random_real(state)
```

monty_rng_jump	<i>Jump random number state</i>
----------------	---------------------------------

Description

Jump random number state. There are two "lengths" of jumps; a normal jump and a long jump. The normal jump is the distance between streams within a random number state, so if you have a multi-stream rng this shifts states left. The long jump is used to create distributed states. We will properly explain all this once the interface stabilises.

Usage

```
monty_rng_jump(state, n = 1)
```

```
monty_rng_long_jump(state, n = 1)
```

Arguments

state	Either a <code>monty_rng_state</code> object (created via monty_rng_create) or a raw vector suitable for creating one.
n	The number of jumps to take (integer, 1 or more)

Value

The `monty_rng_state` object (invisibly, modified in place) or a raw vector, matching the input argument state (visibly).

monty_rng_state	<i>Get and set random number state</i>
-----------------	--

Description

Get and set internal random number state

Usage

```
monty_rng_state(state)
```

```
monty_rng_set_state(value, state)
```

Arguments

state	The random number state, from monty_rng_create
value	A vector of raw values, typically the result of exporting a random state with <code>monty_rng_state()</code>

Value

A vector of raws

Examples

```
s1 <- monty_rng_create()
r1 <- monty_rng_state(s1)

s2 <- monty_rng_create(seed = r1)
identical(r1, monty_rng_state(s2))
monty_random_real(s1)
monty_random_real(s2)

monty_rng_set_state(r1, s1)
monty_random_real(s1)
monty_random_real(s1)
monty_random_real(s2)
```

monty_runner_callr	<i>Run MCMC chains in parallel with callr</i>
--------------------	---

Description

Run MCMC chains in parallel (at the same time). This runner uses the `callr` package to distribute your chains over a number of worker processes on the same machine. If you have used `mcstate`, this is the same as "worker" processes. Unless your chains take a few seconds to run, this will be slower than running with the default serial runner ([monty_runner_serial](#)), however for long running chains, the speed-up will typically scale with workers added, so long as your chains divide neatly over workers.

Usage

```
monty_runner_callr(n_workers, progress = NULL)
```

Arguments

<code>n_workers</code>	The number of workers to use. This should be no larger than the number of chains (though this is harmless) and no larger than the total number of cores available on your computer. Ideally the number of chains you want to run is a multiple of this number (for example, if you had 8 chains, then 1, 2, 4, and 8 are good choices of <code>n_workers</code> , and 7 workers will likely be no faster than 4).
<code>progress</code>	Optional logical, indicating if we should print a progress bar while running. If <code>NULL</code> , we use the value of the option <code>monty.progress</code> if set, otherwise we show the progress bar (as it is typically wanted). Alternatively, you can provide a string indicating the progress bar type. Options are <code>fancy</code> (equivalent to <code>TRUE</code>), <code>none</code> (equivalent to <code>FALSE</code>) and <code>simple</code> (a very simple text-mode progress indicator designed play nicely with logging; it does not use special codes to clear the line).

Value

A runner of class `monty_runner` that can be passed to `monty_sample()`

`monty_runner_parallel` *Run MCMC chain in parallel*

Description

Run MCMC chains in parallel (at the same time). This runner uses the `parallel` package to distribute your chains over a number of worker processes on the same machine. Compared with `monty_runner_callr` (Which is similar to the "worker" support in `mcstate` version 1), this is very simple. In particular we do not report back any information about progress while a chain is running on a worker or even across chains. There's also no support to warn you if your number of chains do not neatly divide through by the number of workers. Mostly this exists as a proof of concept for us to think about the different interfaces. Unless your chains are quite slow, the parallel runner will be slower than the serial runner (`monty_runner_serial`) due to the overhead cost of starting the cluster.

Usage

```
monty_runner_parallel(n_workers)
```

Arguments

<code>n_workers</code>	Number of workers to create a cluster from. In a multi-user setting be careful not to set this to more cores than you are allowed to use. You can use <code>parallel::detectCores()</code> to get an estimate of the number of cores you have on a single user system (but this is often an overestimate as it returns the number of logical cores, including those from "hyperthreading"). Fewer cores than this will be used if you run fewer chains than you have workers.
------------------------	---

Value

A runner of class `monty_runner` that can be passed to `monty_sample()`

`monty_runner_serial` *Run MCMC chain in series*

Description

Run MCMC chains in series (one after another). This is the simplest chain runner, and the default used by `monty_sample()`. It has nothing that can be configured (yet).

Usage

```
monty_runner_serial(progress = NULL)
```

Arguments

`progress` Optional logical, indicating if we should print a progress bar while running. If `NULL`, we use the value of the option `monty.progress` if set, otherwise we show the progress bar (as it is typically wanted). Alternatively, you can provide a string indicating the progress bar type. Options are `fancy` (equivalent to `TRUE`), `none` (equivalent to `FALSE`) and `simple` (a very simple text-mode progress indicator designed play nicely with logging; it does not use special codes to clear the line).

Value

A runner of class `monty_runner` that can be passed to `monty_sample()`

`monty_runner_simultaneous`

Run MCMC chains simultaneously

Description

Run chains *simultaneously*. This differs from `monty_runner_parallel`, which runs chains individually in parallel by working with models that can evaluate multiple densities at the same time. There are situations where this might be faster than running in parallel, but primarily this exists so that we can see that samplers can work with multiple samples at once.

Usage

```
monty_runner_simultaneous(progress = NULL)
```

Arguments

`progress` Optional logical, indicating if we should print a progress bar while running. If `NULL`, we use the value of the option `monty.progress` if set, otherwise we show the progress bar (as it is typically wanted). Alternatively, you can provide a string indicating the progress bar type. Options are `fancy` (equivalent to `TRUE`), `none` (equivalent to `FALSE`) and `simple` (a very simple text-mode progress indicator designed play nicely with logging; it does not use special codes to clear the line).

Value

A runner of class `monty_runner` that can be passed to `monty_sample()`

Examples

```
m <- monty_example("banana")
s <- monty_sampler_random_walk(vcv = diag(2) * 0.01)
r <- monty_runner_simultaneous()
samples <- monty_sample(m, s, 200, runner = r)
```

monty_sample

Sample from a model

Description

Sample from a model. Uses a Monte Carlo method (or possibly something else in future) to generate samples from your distribution. This is going to change a lot in future, as we add support for distributing over workers, and for things like parallel reproducible streams of random numbers. For now it just runs a single chain as a proof of concept.

Usage

```
monty_sample(
  model,
  sampler,
  n_steps,
  initial = NULL,
  n_chains = 1L,
  runner = NULL,
  restartable = FALSE,
  burnin = NULL,
  thinning_factor = NULL,
  save_full_chains = FALSE,
  flatten_chains = FALSE
)
```

Arguments

model	The model to sample from; this should be a <code>monty_model</code> for now, but we might change this in future to test to see if things match an interface rather than a particular class attribute.
sampler	A sampler to use. These will be described later, but we hope to make these reasonably easy to implement so that we can try out different sampling ideas. For now, the only sampler implemented is <code>monty_sampler_random_walk()</code> .
n_steps	The number of steps to run the sampler for.
initial	Optionally, initial parameter values for the sampling. If not given, we sample from the model (or its prior). Alternatively, you can provide a <code>monty_samples</code> object here – the result of a previous call to this function – and we will sample some starting points from the final portion of the chains (the exact details here are subject to change, but we'll sample from the last 20 points or 5% of the chain, whichever is smaller, with replacement, pooled across all chains in the previous sample).
n_chains	Number of chains to run. The default is to run a single chain, but you will likely want to run more.

runner	A runner for your chains. The default option is to run chains in series (via monty_runner_serial). The only other current option is monty_runner_parallel which uses the <code>parallel</code> package to run chains in parallel. If you only run one chain then this argument is best left alone.
restartable	Logical, indicating if the chains should be restartable. This will add additional data to the chains object.
burnin	Number of steps to discard as burnin. This affects only the recording of steps as your chains run; we don't record the first burnin steps. Generally you would want to do this in post-processing with monty_samples_thin() as this data is discarded with no chance of getting it back. However, if your observation process creates a large amount of data, then you may prefer to apply a burnin here to reduce how much memory is used.
thinning_factor	A thinning factor to apply while the chain is running. If given, then we save every <code>thinning_factor</code> 'th step. So if <code>thinning_factor = 2</code> we save every second step, and if 10, we'd save every 10th. Like burnin above, it is preferable to apply this in post processing with monty_samples_thin() . However, for slow-mixing chains that have a large observer output you can use this to reduce the memory usage. Use of <code>thinning_factor</code> requires that <code>n_steps</code> is an even multiple of <code>thinning_factor</code> ; so if <code>thinning_factor</code> is 10, then <code>n_steps</code> must be a multiple of 10. This ensures that the last step is in the sample. The thinning factor cannot be changed when continuing a chain.
save_full_chains	Logical, indicating whether or not the full chains of the parameters (without discarding the burnin and thinning) should additionally be saved. This is useful if you are thinning to reduce the size of observations but still want to retain the full chains of the parameters for evaluation purposes.
flatten_chains	Logical, indicating whether or not the chains dimension is collapsed into the samples dimension in <code>pars</code> , <code>density</code> and (typically) objects in <code>observations</code> . This can be reversed using monty_unflatten_chains() .

Value

A list of parameters and densities. We provide conversion to formats used by other packages, notably [posterior::as_draws_array](#), [posterior::as_draws_df](#) and [coda::as.mcmc.list](#); please let us know if you need conversion to something else. If you want to work directly with the output, the elements in the list include:

- `pars`: An array with three dimensions representing (in turn) parameter, sample and chain, so that `pars[i, j, k]` is the *i*th parameter from the *j*th sample from the *k*th chain. The rows will be named with the names of the parameters, from your model.
- `density`: A matrix of model log densities, with `n_steps` rows and `n_chains` columns.
- `initial`: A record of the initial conditions, a matrix with as many rows as you have parameters and `n_chains` columns (this is the same format as the matrix form of the initial input parameter)
- `details`: Additional details reported by the sampler; this will be a list of length `n_chains` (or `NULL`) and the details depend on the sampler. This one is subject to change.

- `observations`: Additional details reported by the model. This one is also subject to change.
- `full_chains`: If `save_full_chains = TRUE`, a list including `pars` and `density` without discarding the burnin and thinning. If `save_full_chains = FALSE`, this will be `NULL`.

Examples

```
m <- monty_example("banana")
s <- monty_sampler_hmc(epsilon = 0.1, n_integration_steps = 10)
samples <- monty_sample(m, s, 2000)

# Quick conversion of parameters into something plottable:
pars <- t(drop(samples$pars))
plot(pars, pch = 19, cex = 0.75, col = "#0000ff55")

# If you have the posterior package you might prefer converting to
# its format for performing diagnoses:

res <- posterior::as_draws_df(samples)
posterior::summarise_draws(res)

# At this point you could also use the 'bayesplot' package to plot
# diagnostics.
```

monty_sample_continue *Continue sampling*

Description

Continue (restart) chains started by [monty_sample](#). Requires that the original chains were run with `restartable = TRUE`. Running chains this way will result in the final state being exactly the same as running for the total (original + continued) number of steps in a single push.

Usage

```
monty_sample_continue(
  samples,
  n_steps,
  restartable = FALSE,
  runner = NULL,
  append = TRUE
)
```

Arguments

<code>samples</code>	A <code>monty_samples</code> object created by monty_sample()
<code>n_steps</code>	The number of new steps to run

restartable	Logical, indicating if the chains should be restartable. This will add additional data to the chains object.
runner	Optionally, a runner for your chains. The default is to continue with the backend that you used to start the chains via monty_sample (or on the previous restart with this function). You can use this argument to change the runner, which might be useful if transferring a pilot run from a high-resource environment to a lower-resource environment. If given, must be a <code>monty_runner</code> object such as monty_runner_serial or monty_runner_parallel . You can use this argument to change the configuration of a runner, as well as the type of runner (e.g., changing the number of allocated cores).
append	Logical, indicating if we should append the results of the resumed chain together with the original chain.

Value

A list of parameters and densities

`monty_sample_manual_cleanup`
Clean up samples

Description

Clean up after manual sampling. This is essentially a safe version of deleting the directory (e.g, `unlink(path, recursive = TRUE)`) which checks that the directory really was used for sampling and that it does not contain anything else unexpected.

Usage

```
monty_sample_manual_cleanup(path)
```

Arguments

`path` The path used in the call to [monty_sample_manual_prepare](#)

Value

Nothing, called for side effects only.

Examples

```
model <- monty_example("banana")
sampler <- monty_sampler_random_walk(vcv = diag(2) * 0.05)
path <- tempfile()
monty_sample_manual_prepare(model, sampler, 100, path)
monty_sample_manual_info(path)

# Run the (single) chain
```

```

monty_sample_manual_run(1, path)
monty_sample_manual_info(path)

# Collect the results
monty_sample_manual_collect(path)

# Clean up samples
monty_sample_manual_cleanup(path)

```

```

monty_sample_manual_collect
      Collect manually run samples

```

Description

Collect samples from chains that have been run with [monty_sample_manual_prepare](#) and [monty_sample_manual_run](#). If any chain has not completed, we will error.

Usage

```

monty_sample_manual_collect(
  path,
  samples = NULL,
  restartable = FALSE,
  append = TRUE,
  flatten_chains = FALSE
)

```

Arguments

path	The path used in the call to monty_sample_manual_prepare
samples	Samples from the parent run. You need to provide these where save_samples was set to anything other than "value"
restartable	Logical, indicating if the chains should be restartable. This will add additional data to the chains object. Note that this is controlled at chain collection and not creation.
append	Logical, indicating if we should append the results of the resumed chain together with the original chain.
flatten_chains	Logical, indicating whether or not the chains dimension is collapsed into the samples dimension in pars, density and (typically) objects in observations. This can be reversed using monty_unflatten_chains() .

Value

A monty_samples object.

Examples

```
model <- monty_example("banana")
sampler <- monty_sampler_random_walk(vcv = diag(2) * 0.05)
path <- tempfile()
monty_sample_manual_prepare(model, sampler, 100, path)
monty_sample_manual_info(path)

# Run the (single) chain
monty_sample_manual_run(1, path)
monty_sample_manual_info(path)

# Collect the results
monty_sample_manual_collect(path)

# Clean up samples
monty_sample_manual_cleanup(path)
```

monty_sample_manual_info

Get information about manually scheduled samples

Description

Get information about the status of manually scheduled samples.

Usage

```
monty_sample_manual_info(path)
```

Arguments

path The path used in the call to [monty_sample_manual_prepare](#)

Value

Invisibly, a logical vector, TRUE for completed chains and FALSE for incomplete chains.

Examples

```
model <- monty_example("banana")
sampler <- monty_sampler_random_walk(vcv = diag(2) * 0.05)
path <- tempfile()
monty_sample_manual_prepare(model, sampler, 100, path)
monty_sample_manual_info(path)

# Run the (single) chain
monty_sample_manual_run(1, path)
monty_sample_manual_info(path)
```

```
# Collect the results
monty_sample_manual_collect(path)

# Clean up samples
monty_sample_manual_cleanup(path)
```

```
monty_sample_manual_prepare
```

Prepare to sample with manual scheduling

Description

Run an MCMC, but schedule execution of the chains yourself. Use this if you want to distribute chains over (say) the nodes of an HPC system. The arguments are the same as for [monty_sample](#), except that the `runner` argument is missing as you will be looking after that yourself. After using this function, you will generally be wanting to run [monty_sample_manual_run](#) and [monty_sample_manual_collect](#).

Usage

```
monty_sample_manual_prepare(
  model,
  sampler,
  n_steps,
  path,
  initial = NULL,
  n_chains = 1L,
  burnin = NULL,
  thinning_factor = NULL,
  save_full_chains = FALSE
)
```

Arguments

<code>model</code>	The model to sample from; this should be a <code>monty_model</code> for now, but we might change this in future to test to see if things match an interface rather than a particular class attribute.
<code>sampler</code>	A sampler to use. These will be described later, but we hope to make these reasonably easy to implement so that we can try out different sampling ideas. For now, the only sampler implemented is monty_sampler_random_walk() .
<code>n_steps</code>	The number of steps to run the sampler for.
<code>path</code>	The path to write inputs and outputs to. This should be a path to a directory which does not yet exist, or which is empty; we will create one here. The contents of this directory are managed by monty and the names and contents of files here are an implementation detail and should not be relied on. Calling <code>monty_sample_manual_cleanup()</code> will delete the directory in its entirety. Be aware that if you use tempfile() here (which can be a reasonable choice!) that this path will be deleted when your R process ends, so if using these the process calling <code>monty_sample_manual_prepare</code> should outlive running all sampling.

initial	Optionally, initial parameter values for the sampling. If not given, we sample from the model (or its prior). Alternatively, you can provide a <code>monty_samples</code> object here – the result of a previous call to this function – and we will sample some starting points from the final portion of the chains (the exact details here are subject to change, but we'll sample from the last 20 points or 5% of the chain, whichever is smaller, with replacement, pooled across all chains in the previous sample).
n_chains	Number of chains to run. The default is to run a single chain, but you will likely want to run more.
burnin	Number of steps to discard as burnin. This affects only the recording of steps as your chains run; we don't record the first burnin steps. Generally you would want to do this in post-processing with <code>monty_samples_thin()</code> as this data is discarded with no chance of getting it back. However, if your observation process creates a large amount of data, then you may prefer to apply a burnin here to reduce how much memory is used.
thinning_factor	A thinning factor to apply while the chain is running. If given, then we save every <code>thinning_factor</code> 'th step. So if <code>thinning_factor = 2</code> we save every second step, and if 10, we'd save every 10th. Like burnin above, it is preferable to apply this in post processing with <code>monty_samples_thin()</code> . However, for slow-mixing chains that have a large observer output you can use this to reduce the memory usage. Use of <code>thinning_factor</code> requires that <code>n_steps</code> is an even multiple of <code>thinning_factor</code> ; so if <code>thinning_factor</code> is 10, then <code>n_steps</code> must be a multiple of 10. This ensures that the last step is in the sample. The thinning factor cannot be changed when continuing a chain.
save_full_chains	Logical, indicating whether or not the full chains of the parameters (without discarding the burnin and thinning) should additionally be saved. This is useful if you are thinning to reduce the size of observations but still want to retain the full chains of the parameters for evaluation purposes.

Details

In contrast to `monty_sample` there is no runner argument here, because by using this function directly you are taking responsibility for being your own runner.

As with the ways of running a set of chains in `monty`, it is expected that using `monty_sample_manual_*` will result in the same samples being generated as if you had used `monty_sample` with a runner of your choice.

Value

Invisibly, the path used to store files (the same as the value of the `path` argument)

See Also

`monty_sample_manual_run` to run the chains and `monty_sample_manual_collect` / `monty_sample_manual_cleanup` to collect results and clean up. `monty_sample_manual_info` can print human-readable information about the state of a manual run.

Examples

```

model <- monty_example("banana")
sampler <- monty_sampler_random_walk(vcv = diag(2) * 0.05)
path <- tempfile()
monty_sample_manual_prepare(model, sampler, 100, path)
monty_sample_manual_info(path)

# Run the (single) chain
monty_sample_manual_run(1, path)
monty_sample_manual_info(path)

# Collect the results
monty_sample_manual_collect(path)

# Clean up samples
monty_sample_manual_cleanup(path)

```

monty_sample_manual_prepare_continue

Prepare to continue sampling with manual scheduling

Description

Prepare to continue sampling from a model, with manual chain orchestration. This function is to [monty_sample_continue](#) what [monty_sample_manual_prepare](#) is to [monty_sample](#). The original set of samples do not need to have been run manually.

Usage

```

monty_sample_manual_prepare_continue(
  samples,
  n_steps,
  path,
  save_samples = "hash"
)

```

Arguments

<code>samples</code>	A <code>monty_samples</code> object created by monty_sample()
<code>n_steps</code>	The number of steps to run the sampler for.
<code>path</code>	The path to write inputs and outputs to. This should be a path to a directory which does not yet exist, or which is empty; we will create one here. The contents of this directory are managed by monty and the names and contents of files here are an implementation detail and should not be relied on. Calling <code>monty_sample_manual_cleanup()</code> will delete the directory in its entirety. Be aware that if you use tempfile() here (which can be a reasonable choice!) that this path will be deleted when your R process ends, so if using these the process calling <code>monty_sample_manual_prepare</code> should outlive running all sampling.

`save_samples` Control over saving samples into the inputs. The choices here are `hash` (the default) where we save a hash and validate that in `monty_sample_manual_collect`, `value` where the samples are themselves saved and you can omit the `samples` argument to `monty_sample_manual_collect`, or `nothing`, in which we save nothing, and you just have to get it right.

Value

Invisibly, the path used to store files (the same as the value of the `path` argument)

`monty_sample_manual_run`

Run sample with manual scheduling

Description

Run a chain that was prepared using `monty_sample_manual_prepare`.

Usage

```
monty_sample_manual_run(chain_id, path, progress = NULL)
```

Arguments

<code>chain_id</code>	The id for the chain to run, an integer. If you provide an integer that does not correspond to a chain in 1 to <code>n_chains</code> (where <code>n_chains</code> was the argument passed to <code>monty_sample_manual_prepare</code> it is an error.
<code>path</code>	The path used in the call to <code>monty_sample_manual_prepare</code>
<code>progress</code>	Optional logical, indicating if we should print a progress bar while running. If <code>NULL</code> , we use the value of the option <code>monty.progress</code> if set, otherwise we show the progress bar (as it is typically wanted). Alternatively, you can provide a string indicating the progress bar type. Options are <code>fancy</code> (equivalent to <code>TRUE</code>), <code>none</code> (equivalent to <code>FALSE</code>) and <code>simple</code> (a very simple text-mode progress indicator designed play nicely with logging; it does not use special codes to clear the line).

Warning:

There is no lock mechanism; you can start a single chain many times. Don't do this.

Examples

```
model <- monty_example("banana")
sampler <- monty_sampler_random_walk(vcv = diag(2) * 0.05)
path <- tempfile()
monty_sample_manual_prepare(model, sampler, 100, path)
monty_sample_manual_info(path)
```

```

# Run the (single) chain
monty_sample_manual_run(1, path)
monty_sample_manual_info(path)

# Collect the results
monty_sample_manual_collect(path)

# Clean up samples
monty_sample_manual_cleanup(path)

```

monty_sampler

Create a monty sampler

Description

A `monty_sampler` object can be passed into `monty_sample` in order to draw samples from a distribution. The primary role of a sampler is to advance the state of a Markov chain one step; in doing so they may mutate some internal state (outside the knowledge of the problem being advanced). Ordinarily users will not call this function, but authors of samplers will call it from the constructor of their sampler.

Usage

```

monty_sampler(
  name,
  help,
  control,
  initialise,
  step,
  state_dump = NULL,
  state_combine = NULL,
  state_restore = NULL,
  state_details = NULL,
  properties = NULL
)

```

Arguments

name	Name of the sampler. Usually this is the name of the algorithm and can include spaces and punctuation if desired.
help	Name of the function to direct users to find help. Usually that is the name of the constructor function.
control	A list of control parameters passed to the sampler. These are immutable (i.e., once created they cannot be changed).
initialise	A function to initialise the sampler. This is called once at the start of the chain to set up any internal state, though in some cases it will not need to do very much. It must take arguments:

- `state_chain`: the state of the MCMC chain, containing elements `pars` (a vector or matrix of parameters), `density` and possibly `observation`
- `control`: the control parameters, as originally passed to `monty_sampler`
- `model`: the model being sampled from
- the random number generator state, which the sampler may draw from.

Return `NULL` if your sampler is stateless, otherwise return an environment (e.g., created with `new.env(parent = emptyenv())`) of state that will be updated at each iteration. You can store whatever is convenient in this, for example a random walk sampler might store the eigendecomposition of a variance covariance matrix used in the proposal here.

step The workhorse function of a sampler, propagating state (the pair (parameters, density)) forward one step in the chain. Typically, but not always, this will include a proposal, evaluation of a density, and an acceptance.

It must take arguments:

- `state_chain`: The state of the MCMC chain (as above)
- `state_sampler`: The state of the sampler, as passed back from `init`. If your sampler is stateless this is `NULL`, otherwise it is an environment that you will modify by reference.
- `control`: Sampler control parameters (as above)
- `model`: The model (as above)
- `rng`: The random number state, which you can use in the step (as above)

Return `state_chain`, updated after acceptance.

state_dump Optionally, a function to prepare the chain state for serialisation. If not given, we assume that nothing needs to be saved and that your sampler can be restarted from just the state of the chain, the model and control. If provided then typically you will need to provide `state_restore`, too.

state_combine Optionally, a function to combine the output of several chains (a list, where each element has come from `state_dump`) into a single object that is consistent with what the simultaneous runner would have produced.

state_restore Optionally, a function to take a dumped chain state and convert it back into an environment. If not given, we assume that `list2env(x, parent = emptyenv())` is sufficient and use that (unless your state is `NULL`, in which case we use `identity`). If provided then typically you will need to provide `state_dump`, too. The arguments here, if provided, must be

- `chain_id`
- `state_chain`
- `state_sampler`
- `control`
- `model`

state_details Optionally, a function to tidy internal state to be saved at the end of the run. If you provide this you almost certainly need to provide `state_dump` and `state_restore`. This takes the combined state as its only argument.

properties Optionally, a [monty_sampler_properties](#) object that advertises what your sampler can do and what it requires from models that it draws samples from.

Details

See `vignette("writing-samplers")` for an introduction to writing samplers.

Control parameters are used to build the sampler. These are immutable after creation. The format is unspecified by `monty_sampler` but typically this will be a named list. The sampler designer will construct this list and should take care not to include anything mutable (e.g. environments) or hard to serialise and transfer to another process here.

Value

A `monty_sampler` object

Sampler state

Your sampler can have internal state. This is not needed for simple samplers; a random walk Metropolis-Hastings sampler does not need this for example as its state is entirely defined by the `(pars, density)` pair that forms the chain state. Similarly, simple implementations of HMC or Gibbs samplers would not need this functionality. If you wish to record debug information, or if your sampler updates some internal state as it runs – as our adaptive Metropolis-Hastings sampler does – then you will need to configure your sampler to initialise, store, combine and restore this state.

There are four state handling functions. If you provide one, you probably will need to provide them all, though details can be omitted if you do not want to render out a user-facing summary of your sampler state at the end of a chain.

- `state_dump`: this takes the sampler state (which is often an environment) and returns a list. In most cases, this is for a single chain, but if your sampler is used with `monty_runner_simultaneous()` this will correspond to the state for a number of chains at once, in which case your dumped state should look like the output from combining chains.
- `state_combine`: this takes a list of sampler states, each of which was dumped with `state_dump` and combines them into a single state object. You need to aim for the case where the output of this function is the same as running `state_dump` after running with `monty_runner_simultaneous()`. Hopefully we can write some things to help with this, or at least example tests that will probably satisfy this.
- `state_restore`: this takes the output of `state_split` (or `state_combine` in the case of `monty_runner_simultaneous()`) and prepares the state for use with the sampler. This function takes arguments:
 - `chain_id`: one or more chain ids
 - `state_chain`: the state of the chain(s) at the point of restoration
 - `state_sampler`: the state from `state_combine`
 - `control`: the sampler control
 - `model`: the model
- `state_details`: this takes the output of `state_combine` and returns a cleaned version of the state back to the user, as the `$details` element of the final samples. Use this to extract a fraction of the total state where only some should be user-visible. You do not need to provide this, the default is to return nothing.

State initialisation is handled by `initialise`; however, a non-trivial return value from this function does not imply that your sampler needs to worry very much about state. If you can entirely construct this from the current state of the chain and the control parameters, then no state management is required. However, a non-trivial return value from `initialise` requires a `state_restore` argument, even if `state_dump` is not present.

The state manoeuvres may feel tedious, but it will form part of the core of how the Parallel Tempering algorithm works, where we need to be able to run multiple chains through a sampler at the same time.

We will set things up soon so that if you do not provide these functions (but if you do provide state), then your sampler will work, but it will fail informatively when you try and continue it.

monty_sampler_adaptive

Adaptive Metropolis-Hastings Sampler

Description

Create an adaptive Metropolis-Hastings sampler, which will tune its variance covariance matrix (vs the simple random walk sampler [monty_sampler_random_walk](#)).

Usage

```
monty_sampler_adaptive(
  initial_vcv,
  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,
  boundaries = "reflect"
)
```

Arguments

<code>initial_vcv</code>	An initial variance covariance matrix; we'll start using this in the proposal, which will gradually become more weighted towards the empirical covariance matrix calculated from the chain.
<code>initial_vcv_weight</code>	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.

<code>initial_scaling</code>	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 <code>n_pars</code> is the number of fitted parameters). Thus, in a Gaussian target parameter space, the optimal scaling will be around 1.
<code>initial_scaling_weight</code>	The initial weight used in the scaling update. The scaling weight will increase after the first <code>pre_diminish</code> iterations, and as the scaling weight increases the adaptation of the scaling diminishes. If NULL (the default) the value is $5 / (acceptance_target * (1 - acceptance_target))$.
<code>min_scaling</code>	The minimum scaling of the variance covariance matrix to be used to generate the multivariate normal proposal for the random-walk Metropolis-Hastings algorithm.
<code>scaling_increment</code>	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.
<code>log_scaling_update</code>	Logical, whether or not changes to the scaling parameter are made on the log-scale.
<code>acceptance_target</code>	The target for the fraction of proposals that should be accepted (optimally) for the adaptive part of the chain.
<code>forget_rate</code>	The rate of forgetting early parameter sets from the empirical variance-covariance matrix in the MCMC chains. For example, <code>forget_rate = 0.2</code> (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 <code>forget_rate = 0</code> means early parameter sets are never forgotten.
<code>forget_end</code>	The final iteration at which early parameter sets can be forgotten. Setting <code>forget_rate = Inf</code> (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.
<code>adapt_end</code>	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.
<code>pre_diminish</code>	The number of updates before adaptation of the scaling parameter starts to diminish. Setting <code>pre_diminish = 0</code> means there is diminishing adaptation of the scaling parameter from the offset, while <code>pre_diminish = Inf</code> 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.

boundaries Control the behaviour of proposals that are outside the model domain. The supported options are:

- "reflect" (the default): we reflect proposed parameters that lie outside the domain back into the domain (as many times as needed)
- "reject": we do not evaluate the density function, and return $-\text{Inf}$ for its density instead.
- "ignore": evaluate the point anyway, even if it lies outside the domain.

The initial point selected will lie within the domain, as this is enforced by [monty_sample](#).

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-Hastings algorithm where the proposal is a multi-variate Normal distribution centred 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.

Value

A `monty_sampler` object, which can be used with [monty_sample](#)

Examples

```
m <- monty_example("gaussian", matrix(c(1, 0.5, 0.5, 2), 2, 2))
vcv <- diag(2) * 0.1

# Sampling with a random walk
s_rw <- monty_sampler_random_walk(vcv)
res_rw <- monty_sample(m, s_rw, 1000)

s_adapt <- monty_sampler_adaptive(vcv)
res_adapt <- monty_sample(m, s_adapt, 1000)

plot(drop(res_adapt$density), type = "l", col = 4)
lines(drop(res_rw$density), type = "l", col = 2)

# Estimated vcv from the sampler at the end of the simulation
res_adapt$details$vcv[, 1]

coda::effectiveSize(coda::as.mcmc.list(res_rw))
coda::effectiveSize(coda::as.mcmc.list(res_adapt))
```

monty_sampler_hmc	<i>Create HMC</i>
-------------------	-------------------

Description

Create a Hamiltonian Monte Carlo sampler, implemented using the leapfrog algorithm.

Usage

```
monty_sampler_hmc(
  epsilon = 0.015,
  n_integration_steps = 10,
  vcv = NULL,
  debug = FALSE
)
```

Arguments

epsilon	The step size of the HMC steps
n_integration_steps	The number of HMC steps per step
vcv	A variance-covariance matrix for the momentum vector. The default uses an identity matrix.
debug	Logical, indicating if we should save all intermediate points and their gradients. This will add a vector "history" to the details after the integration. This <i>will</i> slow things down though as we accumulate the history inefficiently.

Value

A monty_sampler object, which can be used with [monty_sample](#)

monty_sampler_parallel_tempering	<i>Parallel Tempering Sampler</i>
----------------------------------	-----------------------------------

Description

Create a "parallel tempering" sampler, which runs multiple chains at once to try and improve mixing, or takes advantage of vectorisation/parallelisation if your underlying model supports it. We have tested the implementation with the random walk sampler ([monty_sampler_random_walk](#)) but this may work with other samplers.

Usage

```
monty_sampler_parallel_tempering(
  sampler,
  n_rungs = NULL,
  beta = NULL,
  base = NULL
)
```

Arguments

sampler	A sampler to use for the underlying chains. You might use something like <code>monty_sampler_random_walk(vcv)</code> for a random walk sampler embedded in this parallel tempering scheme.
n_rungs	The number of extra chains to run, must be at least 1. We will run a total of <code>n_rungs + 1</code> chains, with one of these being your target distribution and one being a direct sample from your base model (often your prior).
beta	A vector of beta values. If provided, then <code>n_rungs</code> should not be provided, and beta should be a vector of at least length 2, where the first value is 1, the last value is 0 and the values in between form a strictly decreasing sequence (i.e., no increases, no ties).
base	An optional base model, which must be provided if your model cannot be automatically decomposed into prior + posterior using <code>monty_model_split</code> , or if you are not using this within a Bayesian context and you want to use an alternative easy-to-sample-from reference distribution. We require that this model can be directly sampled from, that it accepts multiple parameters (as a matrix), that it is deterministic and we assume that it is cheap to compute (relative to the target).

Details

We implement the sampler based on <https://doi.org/10.1111/rssb.12464>

Value

A `monty_sampler` object, which can be used with `monty_sample`

Efficiency of the sampler

A parallel tempering sampler runs a series of chains at the same time, so is doing much more work than a simpler sampler. If you run with `n_rungs = 10` you are doing 11x more work than the underlying base sampler, so you want to make sure that this is paid back somewhere. There are a few places where this efficiency may come from:

1. **Your model is parallelisable.** If your underlying model can run very efficiently in parallel then it may not take much longer in "wall time" to run the extra copies of the calculations. In this case, you'll still be using much more CPU time but will be able to take advantage of extra cores to get more effective sampling if the parallel tempering sampler mixes better than the underlying sampler.

2. **Your model is vectorised.** If your model is implemented in R and vectorises the density calculations then it will generally not take much longer to compute many densities at once than a single one.
3. **Your density is multimodal.** If your density has distinct peaks, then most samplers will struggle to explore it well, and even with a non-parallelised, non-vectorised sampler the parallel tempering sampler will explore the space more efficiently. In the limit, a normal sampler may only explore a single peak in a model with many such peaks and will never mix properly.

Tuning the beta values

The argument `beta` controls the spacing of temperature among chains. Ideally, this is set so that adjacent chains all have the same probability of swapping, which generally means that the beta values themselves will *not* be equidistant. Creating the appropriate vector of betas (or "annealing schedule") requires running a pilot run, then computing new beta values, then running another (new) chain with the new beta values, repeating this iteration a few times until the beta values stabilise. Rumour has it, this should only take a few iterations.

The updated values of `beta` are stored in the details of your samples after running. So if you have a set of samples called `s`, then `s$details$beta` will contain new beta values that you can use on the next run.

It is important not to concatenate chains that are computed with different beta values as they will not generally represent samples from the target distribution (as an extreme case, consider one set of beta values where we never accept swaps onto the target distribution and another where we always do; we obviously cannot concatenate these). Therefore, every time that you change beta values you should start a new chain. You can do this with [monty_sample_continue](#) so long as you pass `append = FALSE`.

monty_sampler_properties

Describe sampler properties

Description

Describe properties of a sampler. This is used from [monty_sampler](#) to advertise what your sampler does about state, what it requires from the runner and from the model, so that monty can ensure that it is only used where it is appropriate.

Usage

```
monty_sampler_properties(
  has_state = NULL,
  restartable = NULL,
  allow_multiple_parameters = FALSE,
  requires_gradient = FALSE,
  requires_allow_multiple_parameters = FALSE,
  requires_deterministic = FALSE
)
```

Arguments

<code>has_state</code>	Optional logical, indicating if the sampler has state. This is optional because presence of the state function <code>state_dump</code> implies this.
<code>restartable</code>	Optional logical, indicating if your sampler can be restarted. If <code>FALSE</code> , then users cannot use <code>restartable = TRUE</code> from <code>monty_sample()</code> (and therefore cannot use <code>monty_continue</code>). This is optional because the presence of the state function <code>state_restore</code> implies this.
<code>allow_multiple_parameters</code>	Logical, indicating if your sampler can accept a matrix of parameters in order to run multiple chains at once (e.g., with the monty_runner_simultaneous runner, or as part of a parallel tempering scheme with monty_sampler_parallel_tempering).
<code>requires_gradient</code>	Logical, indicating if the model must provide a gradient in order to be used with this sampler.
<code>requires_allow_multiple_parameters</code>	Logical, indicating if the model must be able to accept multiple parameters. This is different to <code>allow_multiple_parameters</code> , which concerns if the <i>sampler</i> is able to process multiple parameter sets at once. For example, monty_sampler_parallel_tempering sets <code>allow_multiple_parameters</code> to <code>FALSE</code> but <code>requires_allow_multiple_parameters</code> to <code>TRUE</code> , while monty_sampler_random_walk sets the opposite!
<code>requires_deterministic</code>	Logical, indicating if the model must be deterministic in order to be used with this sampler.

Value

A `monty_sampler_properties` object, which should not be modified.

Examples

```
monty_sampler_properties()
```

```
monty_sampler_random_walk
```

Random Walk Sampler

Description

Create a simple random walk sampler, which uses a symmetric proposal to move around parameter space. This sampler supports sampling from models where the likelihood is only computable randomly (e.g., for pmcmc).

Usage

```
monty_sampler_random_walk(
    vcv,
    boundaries = "reflect",
    rerun_every = Inf,
    rerun_random = TRUE
)
```

Arguments

vcv	A variance covariance matrix for the proposal.
boundaries	Control the behaviour of proposals that are outside the model domain. The supported options are: • "reflect" (the default): we reflect proposed parameters that lie outside the domain back into the domain (as many times as needed) • "reject": we do not evaluate the density function, and return $-\text{Inf}$ for its density instead. • "ignore": evaluate the point anyway, even if it lies outside the domain. The initial point selected will lie within the domain, as this is enforced by monty_sample.
rerun_every	Optional integer giving the frequency at which we should rerun the model on the current "accepted" parameters to obtain a new density for stochastic models. The default for this (Inf) will never trigger a rerun, but if you set to 100, then every 100 steps we run the model on both the proposed <i>and</i> previously accepted parameters 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). With the default value of TRUE, we stochastically rerun at each step with probability of $1 / \text{rerun_every}$. If FALSE we rerun the model at fixed intervals of iterations (given by rerun_every). The two methods give the same expected number of MCMC steps between reruns but a different pattern.

Value

A `monty_sampler` object, which can be used with [monty_sample](#)

monty_samples_thin	<i>Thin samples</i>
--------------------	---------------------

Description

Thin results of running [monty_sample\(\)](#), reducing autocorrelation between samples and saving space. This function may be useful before running onward simulations, or before saving output to disk.

Usage

```
monty_samples_thin(
  samples,
  thinning_factor = NULL,
  burnin = NULL,
  save_full_chains = FALSE
)
```

Arguments

<code>samples</code>	A <code>monty_samples</code> object, from running <code>monty_sample()</code>
<code>thinning_factor</code>	Optional integer thinning factor. If given, then we save every <code>thinning_factor</code> 'th step. So if <code>thinning_factor = 2</code> we save every second step, and if 10, we'd save every 10th. We will always include the last point in the chain, and exclude points counting backwards.
<code>burnin</code>	Number of steps to discard as burnin from the start of the chain.
<code>save_full_chains</code>	Logical, indicating whether or not the full chains of the parameters (without discarding the burnin and thinning) should additionally be saved. This is useful if you are thinning to reduce the size of observations but still want to retain the full chains of the parameters for evaluation purposes.

Value

A `monty_samples` object (as for `monty_sample()`), typically with fewer samples.

Limitations

Subsetting parameters (`$pars`) and density (`$density`) is easy enough, but the main use of this function is subsetting chains that have observations, otherwise you could simply cast to `samples_df` and use functions from the `posterior` package.

We can only subset observations where the observer was able to tidy them up into a nice array for us. This will typically be the case (for example when using `odin/dust`, trajectories will be in a nice array).

More specifically, an array is "nice" if the last two dimensions represent samples and chains; in that case we subset along the samples dimension and leave everything else alone. For each element of `$observations` that cannot be subsetting, we will issue a warning.

We cannot generally subset "details", and will pass that along unmodified.

monty_unflatten_chains	<i>Unflatten chains</i>
------------------------	-------------------------

Description

Unflatten chains in `monty_samples` object that has previously had the chains flattened with `monty_flatten_chains()`, reversing the effects of that function.

Usage

```
monty_unflatten_chains(samples)
```

Arguments

<code>samples</code>	A <code>monty_samples</code> object, that has been run through <code>monty_flatten_chains()</code>
----------------------	--

Value

A `monty_samples` object with the chains dimension restored.

<code>with_trace_random</code>	<i>Trace random number calls</i>
--------------------------------	----------------------------------

Description

Trace calls to R's random-number-generating functions, to detect unexpected use of random number generation outside of monty's control.

Usage

```
with_trace_random(code, max_calls = 5, show_stack = FALSE)
```

Arguments

<code>code</code>	Code to run with tracing on
<code>max_calls</code>	Maximum number of calls to report. The default is 5
<code>show_stack</code>	Logical, indicating if we should show the stack at the point of the call

Value

The result of evaluating code

Index

`coda::as.mcmc.list`, 46

`D`, 3

`D()`, 4

`monty_differentiation`, 3

`monty_domain_expand`, 4

`monty_domain_expand()`, 16

`monty_dsl`, 3, 5

`monty_dsl_distributions`, 6

`monty_dsl_error_explain`, 7

`monty_dsl_parse_distribution`, 6, 7

`monty_example`, 8

`monty_flatten_chains`, 10

`monty_flatten_chains()`, 67

`monty_model`, 4–6, 9, 11, 14–18, 20

`monty_model_combine`, 13

`monty_model_combine()`, 19

`monty_model_density`, 14, 17, 18

`monty_model_direct_sample`, 15, 15, 18

`monty_model_function`, 12, 16

`monty_model_gradient`, 15, 17

`monty_model_properties`, 9, 11, 13, 18

`monty_model_properties()`, 12

`monty_model_split`, 19, 62

`monty_observer`, 20

`monty_packer`, 4, 16, 21

`monty_packer()`, 25–27

`monty_packer_grouped`, 25

`monty_random_beta`, 28

`monty_random_beta_binomial_ab`
(`monty_random_beta_binomial_prob`),
28

`monty_random_beta_binomial_prob`, 28

`monty_random_binomial`, 29

`monty_random_cauchy`, 30

`monty_random_exponential_mean`
(`monty_random_exponential_rate`),
30

`monty_random_exponential_rate`, 30

`monty_random_gamma_rate`

(`monty_random_gamma_scale`), 31

`monty_random_gamma_scale`, 31

`monty_random_hypergeometric`, 32

`monty_random_log_normal`, 32

`monty_random_n_beta`

(`monty_random_beta`), 28

`monty_random_n_beta_binomial_ab`

(`monty_random_beta_binomial_prob`),
28

`monty_random_n_beta_binomial_prob`

(`monty_random_beta_binomial_prob`),
28

`monty_random_n_binomial`

(`monty_random_binomial`), 29

`monty_random_n_cauchy`

(`monty_random_cauchy`), 30

`monty_random_n_exponential_mean`

(`monty_random_exponential_rate`),
30

`monty_random_n_exponential_rate`

(`monty_random_exponential_rate`),
30

`monty_random_n_gamma_rate`

(`monty_random_gamma_scale`), 31

`monty_random_n_gamma_scale`

(`monty_random_gamma_scale`), 31

`monty_random_n_hypergeometric`

(`monty_random_hypergeometric`),
32

`monty_random_n_log_normal`

(`monty_random_log_normal`), 32

`monty_random_n_negative_binomial_mu`

(`monty_random_negative_binomial_prob`),
33

`monty_random_n_negative_binomial_prob`

(`monty_random_negative_binomial_prob`),
33

`monty_random_n_normal`

- (monty_random_normal), 34
- monty_random_n_poisson
 - (monty_random_poisson), 34
- monty_random_n_real
 - (monty_random_real), 35
- monty_random_n_real(), 40
- monty_random_n_truncated_normal
 - (monty_random_truncated_normal), 36
- monty_random_n_uniform
 - (monty_random_uniform), 36
- monty_random_n_weibull
 - (monty_random_weibull), 37
- monty_random_n_zi_negative_binomial_mu
 - (monty_random_zi_negative_binomial_prob), 38
- monty_random_n_zi_negative_binomial_prob
 - (monty_random_zi_negative_binomial_prob), 38
- monty_random_n_zi_poisson
 - (monty_random_zi_poisson), 39
- monty_random_negative_binomial_mu
 - (monty_random_negative_binomial_prob), 33
- monty_random_negative_binomial_prob, 33
- monty_random_normal, 34
- monty_random_poisson, 34
- monty_random_real, 35, 40
- monty_random_real(), 39
- monty_random_truncated_normal, 36
- monty_random_uniform, 36
- monty_random_weibull, 37
- monty_random_zi_negative_binomial_mu
 - (monty_random_zi_negative_binomial_prob), 38
- monty_random_zi_negative_binomial_prob, 38
- monty_random_zi_poisson, 39
- monty_rng_create, 8, 11, 15, 28–38, 39, 39, 41
- monty_rng_jump, 41
- monty_rng_long_jump(monty_rng_jump), 41
- monty_rng_set_state(monty_rng_state), 41
- monty_rng_state, 41
- monty_runner_callr, 42, 43
- monty_runner_parallel, 43, 44, 46, 48
- monty_runner_serial, 42, 43, 43, 46, 48
- monty_runner_simultaneous, 44, 64
- monty_runner_simultaneous(), 57
- monty_sample, 11, 45, 47, 48, 51–53, 60–62, 65
- monty_sample(), 10, 11, 43, 44, 47, 53, 65, 66
- monty_sample_continue, 20, 47, 53, 63
- monty_sample_manual_cleanup, 48, 52
- monty_sample_manual_collect, 49, 51, 52, 54
- monty_sample_manual_info, 50, 52
- monty_sample_manual_prepare, 48–50, 51, 53, 54
- monty_sample_manual_prepare_continue, 53
- monty_sample_manual_run, 49, 51, 52, 54
- monty_sampler, 55, 63
- monty_sampler_adaptive, 58
- monty_sampler_hmc, 8, 61
- monty_sampler_parallel_tempering, 61, 64
- monty_sampler_properties, 56, 63
- monty_sampler_random_walk, 58, 61, 64, 64
- monty_sampler_random_walk(), 45, 51
- monty_samples_thin, 65
- monty_samples_thin(), 46, 52
- monty_unflatten_chains, 67
- monty_unflatten_chains(), 46, 49
- posterior::as_draws_array, 46
- posterior::as_draws_df, 46
- rhyper, 32
- tempfile(), 51, 53
- with_trace_random, 67