Package: monty (via r-universe)

September 2, 2024

Title Monte Carlo Models

Version 0.2.3

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)

RoxygenNote 7.3.2

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

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

Imports R6, cli, parallel, rlang

LinkingTo cpp11

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

Config/testthat/edition 3

Language en-GB

VignetteBuilder knitr

Remotes mrc-ide/dust

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

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

RemoteRef main

RemoteSha c86a53b45ef7b3f7769ca54f16a770d522feb881

Contents

`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.

`monty_domain_expand` *Expand (and check) domain against a packer*

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., `b`) or specific names `b[1]` that are present in your packer. `NULL` 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)
```

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.

Value

A monty_model object derived from the expressions you provide.

Examples

```
# Expressions that correspond to 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.

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)
```

Arguments

`code`	The error code, as a string, in the form `Exxx` (a capital "E" followed by three numbers)

Value

Nothing, this is called for its side effect only

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 rng object (see monty_rng)

`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 `-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 (`-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 `-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 `(-Inf, Inf)`.
- `direct_sample`: A function to sample directly from the parameter space, given a monty_rng object to sample from. 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 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_model`. 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

`monty_model_combine` *Combine two models*

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

`monty_model_density` *Compute log density*

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 monty_model 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.

`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. 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

`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)
```

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).

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.

`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

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,
  allow_multiple_parameters = FALSE
)
```

Arguments

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

`monty_observer` *Create observer*

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 `model` (the model that you passed in to monty_model) and `rng` (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 parameter packer*

Description

Build a parameter packer, which can be used in models to translate between an unstructured vector of numbers (the vector being updated by an MCMC for example) to a structured list of named values, which is easier to program against. 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 scalar parameters. 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 array parameters and values correspond to the lengths of parameters. 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 *strings* as values for the lengths, in which case these will be looked for within `fixed`.
`fixed`	A named list of fixed parameters; these will be added into the final list directly. 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 parameter list; it may create any *additional* 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.

Details

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

- An MCMC is typically discussed as a 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 be 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 three elements:

- `parameters`: a character vector of computed parameter names; 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 parameters back into a numeric vector suitable for the statistical model. This ignores values created by a `preprocess` function.
- `index`: a function which produces a named list where each element has the name of a value in `parameters` and each value has the indices within an unstructured vector where these values can be found.

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 = `3toarray, so that our unpacked list holds` b_raw = c(b11, b12, b22)`. We would then write` pro-cess‘ 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 to be 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 `ith` 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.

We do not currently offer the ability to pack this sort of output back up, though it’s not hard. Please let us know if you would use this.

`monty_rng` *Monty Random Number Generator*

Description

Create an object that can be used to generate random numbers with the same RNG as monty uses internally. This is primarily meant for debugging and testing the underlying C++ rather than a source of random numbers from R.

Value

A `monty_rng` object, which can be used to drawn random numbers from monty’s distributions.

Running multiple streams, perhaps in parallel

The underlying random number generators are designed to work in parallel, and with random access to parameters (see `vignette("rng")` for more details). However, this is usually done within the context of running a model where each particle sees its own stream of numbers. We provide some support for running random number generators in parallel, but any speed gains from parallelisation are likely to be somewhat eroded by the overhead of copying around a large number of random numbers.

All the random distribution functions support an argument `n_threads` which controls the number of threads used. This argument will *silently* have no effect if your installation does not support OpenMP.

Parallelisation will be performed at the level of the stream, where we draw `n` numbers from *each* stream for a total of `n * n_streams` random numbers using `n_threads` threads to do this. Setting `n_threads` to be higher than `n_streams` will therefore have no effect. If running on somebody else's system (e.g., an HPC, CRAN) you must respect the various environment variables that control the maximum allowable number of threads.

With the exception of `random_real`, each random number distribution accepts parameters; the interpretations of these will depend on `n`, `n_streams` and their rank.

- If a scalar then we will use the same parameter value for every draw from every stream
- If a vector with length `n` then we will draw `n` random numbers per stream, and every stream will use the same parameter value for every stream for each draw (but a different, shared, parameter value for subsequent draws).
- If a matrix is provided with one row and `n_streams` columns then we use different parameters for each stream, but the same parameter for each draw.
- If a matrix is provided with `n` rows and `n_streams` columns then we use a parameter value `[i, j]` for the `i`th draw on the `j`th stream.

The rules are slightly different for the `prob` argument to `multinomial` as for that `prob` is a vector of values. As such we shift all dimensions by one:

- If a vector we use same `prob` every draw from every stream and there are `length(prob)` possible outcomes.
- If a matrix with `n` columns then vary over each draw (the `i`th draw using vector `prob[, i]` but shared across all streams. There are `nrow(prob)` possible outcomes.
- If a 3d array is provided with 1 column and `n_streams` "layers" (the third dimension) then we use then we use different parameters for each stream, but the same parameter for each draw.
- If a 3d array is provided with `n` columns and `n_streams` "layers" then we vary over both draws and streams so that with use vector `prob[, i, j]` for the `i`th draw on the `j`th stream.

The output will not differ based on the number of threads used, only on the number of streams.

Public fields

`info` Information about the generator (read-only)

Methods

Public methods:

- `monty_rng$new()`
- `monty_rng$size()`
- `monty_rng$jump()`
- `monty_rng$long_jump()`
- `monty_rng$random_real()`
- `monty_rng$random_normal()`
- `monty_rng$uniform()`
- `monty_rng$normal()`
- `monty_rng$binomial()`
- `monty_rng$nbinomial()`
- `monty_rng$hypergeometric()`
- `monty_rng$gamma_scale()`
- `monty_rng$gamma_rate()`
- `monty_rng$poisson()`
- `monty_rng$exponential_rate()`
- `monty_rng$exponential_mean()`
- `monty_rng$cauchy()`
- `monty_rng$multinomial()`
- `monty_rng$beta()`
- `monty_rng$state()`

Method `new()`: Create a `monty_rng` object

Usage:

```
monty_rng$new(
  seed = NULL,
  n_streams = 1L,
  real_type = "double",
  deterministic = FALSE
)
```

Arguments:

`seed` The seed, as an integer, a raw vector or `NULL`. If an integer we will create a suitable seed via the "splitmix64" algorithm, if a raw vector it must the correct length (a multiple of either 32 or 16 for `float = FALSE` or `float = TRUE` respectively). If `NULL` then we create a seed using R's random number generator.

`n_streams` The number of streams to use (see Details)

`real_type` The type of floating point number to use. Currently only `float` and `double` are supported (with `double` being the default). This will have no (or negligible) impact on speed, but exists to test the low-precision generators.

`deterministic` Logical, indicating if we should use "deterministic" mode where distributions return their expectations and the state is never changed.

Method `size()`: Number of streams available

Usage:

```
monty_rng$size()
```

Method `jump()`**:** The jump function updates the random number state for each stream by advancing it to a state equivalent to 2^128 numbers drawn from each stream.

Usage:

```
monty_rng$jump()
```

Method `long_jump()`**:** Longer than `$jump`, the `$long_jump` method is equivalent to 2^192 numbers drawn from each stream.

Usage:

```
monty_rng$long_jump()
```

Method `random_real()`**:** Generate `n` numbers from a standard uniform distribution

Usage:

```
monty_rng$random_real(n, n_threads = 1L)
```

Arguments:

`n` Number of samples to draw (per stream)

`n_threads` Number of threads to use; see Details

Method `random_normal()`**:** Generate `n` numbers from a standard normal distribution

Usage:

```
monty_rng$random_normal(n, n_threads = 1L, algorithm = "box_muller")
```

Arguments:

`n` Number of samples to draw (per stream)

`n_threads` Number of threads to use; see Details

`algorithm` Name of the algorithm to use; currently `box_muller` and `ziggurat` are supported, with the latter being considerably faster.

Method `uniform()`**:** Generate `n` numbers from a uniform distribution

Usage:

```
monty_rng$uniform(n, min, max, n_threads = 1L)
```

Arguments:

`n` Number of samples to draw (per stream)

`min` The minimum of the distribution (length 1 or n)

`max` The maximum of the distribution (length 1 or n)

`n_threads` Number of threads to use; see Details

Method `normal()`**:** Generate `n` numbers from a normal distribution

Usage:

```
monty_rng$normal(n, mean, sd, n_threads = 1L, algorithm = "box_muller")
```

Arguments:

`n` Number of samples to draw (per stream) `mean` The mean of the distribution (length 1 or n)

`sd` The standard deviation of the distribution (length 1 or n)

`n_threads` Number of threads to use; see Details

`algorithm` Name of the algorithm to use; currently `box_muller` and `ziggurat` are supported, with the latter being considerably faster.

Method `binomial()`**:** Generate `n` numbers from a binomial distribution

Usage:

```
monty_rng$binomial(n, size, prob, n_threads = 1L)
```

Arguments:

`n` Number of samples to draw (per stream)

`size` The number of trials (zero or more, length 1 or n)

`prob` The probability of success on each trial (between 0 and 1, length 1 or n)

`n_threads` Number of threads to use; see Details

Method `nbinomial()`**:** Generate `n` numbers from a negative binomial distribution

Usage:

```
monty_rng$nbinomial(n, size, prob, n_threads = 1L)
```

Arguments:

`n` Number of samples to draw (per stream)

`size` The target number of successful trials (zero or more, length 1 or n)

`prob` The probability of success on each trial (between 0 and 1, length 1 or n)

`n_threads` Number of threads to use; see Details

Method `hypergeometric()`**:** Generate `n` numbers from a hypergeometric distribution

Usage:

```
monty_rng$hypergeometric(n, n1, n2, k, n_threads = 1L)
```

Arguments:

`n` Number of samples to draw (per stream)

`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

`n_threads` Number of threads to use; see Details

Method `gamma_scale()`**:** Generate `n` numbers from a gamma distribution

Usage:

```
monty_rng$gamma_scale(n, shape, scale, n_threads = 1L)
```

Arguments:

`n` Number of samples to draw (per stream)

`shape` Shape

`scale` Scale '

`n_threads` Number of threads to use; see Details

Method `gamma_rate()`**:** Generate `n` numbers from a gamma distribution

Usage:

```
monty_rng$gamma_rate(n, shape, rate, n_threads = 1L)
```

Arguments:

`n` Number of samples to draw (per stream)

`shape` Shape

`rate` Rate '

`n_threads` Number of threads to use; see Details

Method `poisson()`**:** Generate `n` numbers from a Poisson distribution

Usage:

```
monty_rng$poisson(n, lambda, n_threads = 1L)
```

Arguments:

`n` Number of samples to draw (per stream)

`lambda` The mean (zero or more, length 1 or n). Only valid for lambda <= 10^7

`n_threads` Number of threads to use; see Details

Method `exponential_rate()`**:** Generate `n` numbers from a exponential distribution

Usage:

```
monty_rng$exponential_rate(n, rate, n_threads = 1L)
```

Arguments:

`n` Number of samples to draw (per stream)

`rate` The rate of the exponential

`n_threads` Number of threads to use; see Details

Method `exponential_mean()`**:** Generate `n` numbers from a exponential distribution

Usage:

```
monty_rng$exponential_mean(n, mean, n_threads = 1L)
```

Arguments:

`n` Number of samples to draw (per stream)

`mean` The mean of the exponential

`n_threads` Number of threads to use; see Details

Method `cauchy()`**:** Generate `n` draws from a Cauchy distribution.

Usage:

```
monty_rng$cauchy(n, location, scale, n_threads = 1L)
```

Arguments:

`n` Number of samples to draw (per stream)

`location` The location of the peak of the distribution (also its median)

`scale` A scale parameter, which specifies the distribution's "half-width at half-maximum"

`n_threads` Number of threads to use; see Details

Method `multinomial()`**:** Generate `n` draws from a multinomial distribution. In contrast with most of the distributions here, each draw is a *vector* with the same length as `prob`.

Usage:

```
monty_rng$multinomial(n, size, prob, n_threads = 1L)
```

Arguments:

`n` The number of samples to draw (per stream)

`size` The number of trials (zero or more, length 1 or n)

`prob` A vector of probabilities for the success of each trial. This does not need to sum to 1 (though all elements must be non-negative), in which case we interpret `prob` as weights and normalise so that they equal 1 before sampling.

`n_threads` Number of threads to use; see Details

Method `beta()`**:** Generate `n` numbers from a beta distribution

Usage:

```
monty_rng$beta(n, a, b, n_threads = 1L)
```

Arguments:

`n` Number of samples to draw (per stream)

`a` The first shape parameter

`b` The second shape parameter

`n_threads` Number of threads to use; see Details

Method `state()`**:** Returns the state of the random number stream. This returns a raw vector of length 32 * n_streams. It is primarily intended for debugging as one cannot (yet) initialise a monty_rng object with this state.

Usage:

```
monty_rng$state()
```

Examples

```
rng <- monty::monty_rng$new(42)

# Shorthand for Uniform(0, 1)
rng$random_real(5)

# Shorthand for Normal(0, 1)
rng$random_normal(5)

# Uniform random numbers between min and max
rng$uniform(5, -2, 6)

# Normally distributed random numbers with mean and sd
rng$normal(5, 4, 2)

# Binomially distributed random numbers with size and prob
rng$binomial(5, 10, 0.3)

# Negative binomially distributed random numbers with size and prob
```

```
rng$nbinomial(5, 10, 0.3)

# Hypergeometric distributed random numbers with parameters n1, n2 and k
rng$hypergeometric(5, 6, 10, 4)

# Gamma distributed random numbers with parameters shape and scale
rng$gamma_scale(5, 0.5, 2)

# Gamma distributed random numbers with parameters shape and rate
rng$gamma_rate(5, 0.5, 2)

# Poisson distributed random numbers with mean lambda
rng$poisson(5, 2)

# Exponentially distributed random numbers with rate
rng$exponential_rate(5, 2)

# Exponentially distributed random numbers with mean
rng$exponential_mean(5, 0.5)

# Multinomial distributed random numbers with size and vector of
# probabiltiies prob
rng$multinomial(5, 10, c(0.1, 0.3, 0.5, 0.1))
```

monty_rng_distributed_state

Create a set of distributed seeds

Description

Create a set of initial random number seeds suitable for using within a distributed context (over multiple processes or nodes) at a level higher than a single group of synchronised threads.

Usage

```
monty_rng_distributed_state(
  seed = NULL,
  n_streams = 1L,
  n_nodes = 1L,
  algorithm = "xoshiro256plus"
)

monty_rng_distributed_pointer(
  seed = NULL,
  n_streams = 1L,
  n_nodes = 1L,
  algorithm = "xoshiro256plus"
)
```

Arguments

`seed`	Initial seed to use. As for monty_rng, this can be `NULL` (create a seed using R's generators), an integer or a raw vector of appropriate length.
`n_streams`	The number of streams to create per node.
`n_nodes`	The number of separate seeds to create. Each will be separated by a "long jump" for your generator.
`algorithm`	The name of an algorithm to use.

Details

See `vignette("rng_distributed")` for a proper introduction to these functions.

Value

A list of either raw vectors (for `monty_rng_distributed_state`) or of monty_rng_pointer objects (for `monty_rng_distributed_pointer`)

Examples

```
monty::monty_rng_distributed_state(n_nodes = 2)
monty::monty_rng_distributed_pointer(n_nodes = 2)
```

`monty_rng_pointer` *Create pointer to random number generator stream*

Description

This function exists to support use from other packages that wish to use monty's random number support, and creates an opaque pointer to a set of random number streams.

Public fields

`algorithm`	The name of the generator algorithm used (read-only)
`n_streams`	The number of streams of random numbers provided (read-only)

Methods

Public methods:

- `monty_rng_pointer$new()`
- `monty_rng_pointer$sync()`
- `monty_rng_pointer$state()`
- `monty_rng_pointer$is_current()`

Method `new()`**:** Create a new `monty_rng_pointer` object

Usage:

```
monty_rng_pointer$new(
  seed = NULL,
  n_streams = 1L,
  long_jump = 0L,
  algorithm = "xoshiro256plus"
)
```

Arguments:

`seed` The random number seed to use (see monty_rng for details)

`n_streams` The number of independent random number streams to create

`long_jump` Optionally an integer indicating how many "long jumps" should be carried out immediately on creation. This can be used to create a distributed parallel random number generator (see monty_rng_distributed_state)

`algorithm` The random number algorithm to use. The default is `xoshiro256plus` which is a good general choice

Method `sync()`**:** Synchronise the R copy of the random number state. Typically this is only needed before serialisation if you have ever used the object.

Usage:

```
monty_rng_pointer$sync()
```

Method `state()`**:** Return a raw vector of state. This can be used to create other generators with the same state.

Usage:

```
monty_rng_pointer$state()
```

Method `is_current()`**:** Return a logical, indicating if the random number state that would be returned by `state()` is "current" (i.e., the same as the copy held in the pointer) or not. This is `TRUE` on creation or immediately after calling `$sync()` or `$state()` and `FALSE` after any use of the pointer.

Usage:

```
monty_rng_pointer$is_current()
```

Examples

```
monty::monty_rng_pointer$new()
```

`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 the "worker" support in `mcstate` version 1 this is very simple and we'll improve it over time. 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

`n_workers`	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 `parallel::detectCores()` 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). The progress bar itself responds to cli's options; in particular `cli.progress_show_after` and `cli.progress_clear` will affect your experience.

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). The progress bar itself responds to cli's options; in particular `cli.progress_show_after` and `cli.progress_clear` will affect your experience.

Value

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

`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,
  observer = NULL,
  restartable = FALSE
)
```

Arguments

`model`	The model to sample from; this should be a `monty_model` 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 `monty_sampler_random_walk()`.
`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).
`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 `parallel` package to run chains in parallel. If you only run one chain then this argument is best left alone.
`observer`	An observer, created via monty_observer, which you can use to extract additional information from your model at points included in the chain (for example, trajectories from a dynamical model).
`restartable`	Logical, indicating if the chains should be restartable. This will add additional data to the chains object.

Value

A list of parameters and densities; we'll write tools for dealing with this later. Elements 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.

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
)
```

Arguments

`initial_vcv`	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.
`initial_vcv_weight`	Weight of the initial variance-covariance matrix used to build the proposal of the random-walk. Higher values translate into higher confidence of the initial variance-covariance matrix and means that update from additional samples will be slower.
`initial_scaling`	The initial scaling of the variance covariance matrix to be used to generate the multivariate normal proposal for the random-walk Metropolis-Hastings algorithm. To generate the proposal matrix, the weighted variance covariance matrix is multiplied by the scaling parameter squared times 2.38^2 / n_pars (where n_pars is the number of fitted parameters). Thus, in a Gaussian target parameter space, the optimal scaling will be around 1.
`initial_scaling_weight`	The initial weight used in the scaling update. The scaling weight will increase after the first `pre_diminish` iterations, and as the scaling weight increases the

	adaptation of the scaling diminishes. If `NULL` (the default) the value is 5 / (acceptance_target * (1 - acceptance_target)).
`min_scaling`	The minimum scaling of the variance covariance matrix to be used to generate the multivariate normal proposal for the random-walk Metropolis-Hastings algorithm.
`scaling_increment`	The scaling increment which is added or subtracted to the scaling factor of the variance-covariance after each adaptive step. If `NULL` (the default) then an optimal value will be calculated.
`log_scaling_update`	Logical, whether or not changes to the scaling parameter are made on the log-scale.
`acceptance_target`	The target for the fraction of proposals that should be accepted (optimally) for the adaptive part of the chain.
`forget_rate`	The rate of forgetting early parameter sets from the empirical variance-covariance matrix in the MCMC chains. For example, `forget_rate = 0.2` (the default) means that once in every 5th iterations we remove the earliest parameter set included, so would remove the 1st parameter set on the 5th update, the 2nd on the 10th update, and so on. Setting `forget_rate = 0` means early parameter sets are never forgotten.
`forget_end`	The final iteration at which early parameter sets can be forgotten. Setting `forget_rate = Inf` (the default) means that the forgetting mechanism continues throughout the chains. Forgetting early parameter sets becomes less useful once the chains have settled into the posterior mode, so this parameter might be set as an estimate of how long that would take.
`adapt_end`	The final iteration at which we can adapt the multivariate normal proposal. Thereafter the empirical variance-covariance matrix, its scaling and its weight remain fixed. This allows the adaptation to be switched off at a certain point to help ensure convergence of the chain.
`pre_diminish`	The number of updates before adaptation of the scaling parameter starts to diminish. Setting `pre_diminish = 0` means there is diminishing adaptation of the scaling parameter from the offset, while `pre_diminish = Inf` would mean there is never diminishing adaptation. Diminishing adaptation should help the scaling parameter to converge better, but while the chains find the location and scale of the posterior mode it might be useful to explore with it switched off.

Details

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

Our implementation of an adaptive MCMC algorithm is based on an adaptation of the "accelerated shaping" algorithm in Spencer (2021). The algorithm is based on a random-walk Metropolis-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

`monty_sampler_hmc` *Create HMC*

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 *will* slow things down though as we accumulate the history inefficiently.

Value

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

monty_sampler_nested_adaptive

Nested Adaptive Metropolis-Hastings Sampler

Description

Create a nested adaptive Metropolis-Hastings sampler, which extends the adaptive sampler monty_sampler_adaptive, tuning the variance covariance matrices for proposal for the separable sections of a nested model (vs the simple nested random walk sampler monty_sampler_random_walk). This sampler requires that models support the `has_parameter_groups` property.

Usage

```
monty_sampler_nested_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
)
```

Arguments

`initial_vcv`	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.
`initial_vcv_weight`	Weight of the initial variance-covariance matrix used to build the proposal of the random-walk. Higher values translate into higher confidence of the initial variance-covariance matrix and means that update from additional samples will be slower.
`initial_scaling`	The initial scaling of the variance covariance matrix to be used to generate the multivariate normal proposal for the random-walk Metropolis-Hastings algorithm. To generate the proposal matrix, the weighted variance covariance matrix is multiplied by the scaling parameter squared times 2.38^2 / n_pars (where n_pars is the number of fitted parameters). Thus, in a Gaussian target parameter space, the optimal scaling will be around 1.

`initial_scaling_weight`	The initial weight used in the scaling update. The scaling weight will increase after the first `pre_diminish` iterations, and as the scaling weight increases the adaptation of the scaling diminishes. If `NULL` (the default) the value is 5 / (acceptance_target * (1 - acceptance_target)).
`min_scaling`	The minimum scaling of the variance covariance matrix to be used to generate the multivariate normal proposal for the random-walk Metropolis-Hastings algorithm.
`scaling_increment`	The scaling increment which is added or subtracted to the scaling factor of the variance-covariance after each adaptive step. If `NULL` (the default) then an optimal value will be calculated.
`log_scaling_update`	Logical, whether or not changes to the scaling parameter are made on the log-scale.
`acceptance_target`	The target for the fraction of proposals that should be accepted (optimally) for the adaptive part of the chain.
`forget_rate`	The rate of forgetting early parameter sets from the empirical variance-covariance matrix in the MCMC chains. For example, `forget_rate = 0.2` (the default) means that once in every 5th iterations we remove the earliest parameter set included, so would remove the 1st parameter set on the 5th update, the 2nd on the 10th update, and so on. Setting `forget_rate = 0` means early parameter sets are never forgotten.
`forget_end`	The final iteration at which early parameter sets can be forgotten. Setting `forget_rate = Inf` (the default) means that the forgetting mechanism continues throughout the chains. Forgetting early parameter sets becomes less useful once the chains have settled into the posterior mode, so this parameter might be set as an estimate of how long that would take.
`adapt_end`	The final iteration at which we can adapt the multivariate normal proposal. Thereafter the empirical variance-covariance matrix, its scaling and its weight remain fixed. This allows the adaptation to be switched off at a certain point to help ensure convergence of the chain.
`pre_diminish`	The number of updates before adaptation of the scaling parameter starts to diminish. Setting `pre_diminish = 0` means there is diminishing adaptation of the scaling parameter from the offset, while `pre_diminish = Inf` would mean there is never diminishing adaptation. Diminishing adaptation should help the scaling parameter to converge better, but while the chains find the location and scale of the posterior mode it might be useful to explore with it switched off.

Details

Much like the simple nested random walk sampler monty_sampler_random_walk, the strategy is to propose all the shared parameters as a deviation from the current point in parameter space as a single move and accept or reject as a block. Then we generate points for all the region-specific parameters, compute the density and then accept or reject these updates independently. This is possible because the change in likelihood in region A is independent from region B.

The adaptive proposal algorithm of the non-nested adaptive sampler monty_sampler_adaptive is extended here to adaptively tune the variance covariance matrix of each of these parameter chunks.

Value

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

`monty_sampler_nested_random_walk` *Nested Random Walk Sampler*

Description

Create a nested random walk sampler, which uses a symmetric proposal for separable sections of a model to move around in parameter space. This sampler supports sampling from models where the likelihood is only computable randomly (e.g., for pmcmc), and requires that models support the `has_parameter_groups` property.

Usage

```
monty_sampler_nested_random_walk(vcv, boundaries = "reflect")
```

Arguments

`vcv`	A list of variance covariance matrices. We expect this to be a list with elements `base` and `groups` corresponding to the covariance matrix for base parameters (if any) and groups.
`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 `-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

The intended use case for this sampler is for models where the density can be decomposed at least partially into chunks that are independent from each other. Our motivating example for this is a model of COVID-19 transmission where some parameters region-specific (e.g., patterns and rates of contact between individuals), and some parameters are shared across all regions (e.g., intrinsic properties of the disease such as incubation period).

The strategy is to propose all the shared parameters as a deviation from the current point in parameter space as a single move and accept or reject as a block. Then we generate points for all the region-specific parameters, compute the density and then accept or reject these updates independently. This is possible because the change in likelihood in region A is independent from region B.

We expect that this approach will be beneficial in limited situations, but where it is beneficial it is likely to result in fairly large speed-ups:

- You probably need more than three regions; as the number of regions increases the benefit of independently accepting or rejecting densities increases (with 1000 separate regions your chains will mix very slowly for example).
- Your model is fairly computationally heavy so that the density calculation completely dominates the sampling process.
- You do not have access to gradient information for your model; we suspect that HMC will outperform this approach by some margin because it already includes this independence via the gradients.
- You can compute your independent calculations in parallel, which help this method reduce your walk time.

Value

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

`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 = NULL, boundaries = "reflect")
```

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 `-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.

Value

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

`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)
```

Arguments

`samples`	A `monty_samples` object created by `monty_sample()`
`n_steps`	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 `monty_runner` 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).

Value

A list of parameters and densities

`with_trace_random` *Trace random number calls*

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

Value

The result of evaluating `code`

Index