Package: dust (via r-universe)

October 11, 2026

Title Iterate Multiple Realisations of Stochastic Models

Version 0.15.3

Description An Engine for simulation of stochastic models. Includes support for running stochastic models in parallel, either with shared or varying parameters. Simulations are run efficiently in compiled code and can be run with a fraction of simulated states returned to R, allowing control over memory usage. Support is provided for building bootstrap particle filter for performing Sequential Monte Carlo (e.g., Gordon et al. 1993 <doi:10.1049/ip-f-2.1993.0015>). The core of the simulation engine is the 'xoshiro256**' algorithm (Blackman and Vigna <doi:10.48550/arXiv.1805.01407>), and the package is further described in FitzJohn et al 2021 <doi:10.12688/wellcomeopenres.16466.2>.

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

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

License MIT + file LICENSE

Encoding UTF-8

Language en-GB

Requires R (>= 4.0.0)

Imports R6, cpp11 (>= 0.4.0), glue, pkgbuild (>= 1.2.0), pkgload, withr

LinkingTo cpp11 (>= 0.4.4)

Suggests bench, brio, callr, curl, dde, decor, fs, knitr, mockery, rmarkdown, testthat (>= 3.0.0)

RoxygenNote 7.2.3

Roxygen list(markdown = TRUE)

VignetteBuilder knitr

Config/testthat/edition 3

Config/pak/sysreqs cmake make libuv1-dev

Repository https://ncov-ic.r-universe.dev

Date/Publication 2025-08-15 15:16:17 UTC

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

RemoteRef master

RemoteSha c135a2fa92b3532f4612437f16255b574ca214be

Contents

dust . . . 2
dust_cuda_configuration . . . 7
dust_cuda_options . . . 9
dust_data . . . 10
dust_example . . . 11
dust_generate . . . 12
dust_generator . . . 14
dust_ode_control . . . 23
dust_openmp_support . . . 24
dust_openmp_threads . . . 25
dust_package . . . 26
dust_repair_environment . . . 28
dust_rng . . . 28
dust_rng_distributed_state . . . 34
dust_rng_pointer . . . 35

Index **38**

`dust` *Create a dust model from a C++ input file*

Description

Create a dust model from a C++ input file. This function will compile the dust support around your model and return an object that can be used to work with the model (see the Details below, and `dust_generator`).

Usage

```
dust(
  filename,
  quiet = FALSE,
  workdir = NULL,
  gpu = FALSE,
  real_type = NULL,
  linking_to = NULL,
  cpp_std = NULL,
  compiler_options = NULL,
  optimisation_level = NULL,
```

```
  skip_cache = FALSE
)
```

Arguments

filename	The path to a single C++ file
quiet	Logical, indicating if compilation messages from pkgbuild should be displayed. Error messages will be displayed on compilation failure regardless of the value used.
workdir	Optional working directory to use. If NULL uses a temporary directory. By using a different directory of your choosing you can see the generated code.
gpu	Logical, indicating if we should generate GPU code. This requires a considerable amount of additional software installed (CUDA toolkit and drivers) as well as a CUDA-compatible GPU. If TRUE, then we call dust_cuda_options with no arguments. Alternatively, call that function and pass the value here (e.g, gpu = dust::dust_cuda_options(debug = TRUE)). Note that due to the use of the __syncwarp() primitive this may require a GPU with compute version 70 or higher.
real_type	Optionally, a string indicating a substitute type to swap in for your model's real_type declaration. If given, then we replace the string using real_type = (double|float) with the given type. This is primarily intended to be used as gpu = TRUE, real_type = "float" in order to create model for the GPU that will use 32 bit floats (rather than 64 bit doubles, which are much slower). For CPU models decreasing precision of your real type will typically just decrease precision for no additional performance.
linking_to	Optionally, a character vector of additional packages to add to the DESCRIPTION's LinkingTo field. Use this when your model pulls in C++ code that is packaged within another package's header-only library.
cpp_std	The C++ standard to use, if you need to set one explicitly. See the section "Using C++ code" in "Writing R extensions" for the details of this, and how it interacts with the R version currently being used. For R 4.0.0 and above, C++11 will be used; as dust depends on at least this version of R you will never need to specify a version this low. Sensible options are C++14, C++17, etc, depending on the features you need and what your compiler supports.
compiler_options	A character vector of additional options to pass through to the C++ compiler. These will be passed through without any shell quoting or validation, so check the generated commands and outputs carefully in case of error. Note that R will apply these *before* anything in your personal Makevars.
optimisation_level	A shorthand way of specifying common compiler options that control optimisation level. By default (NULL) no options are generated from this, and the optimisation level will depend on your user Makevars file. Valid options are none which disables optimisation (-O0), which will be faster to compile but much slower, standard which enables standard level of optimisation (-O2), useful if your Makevars/pkgload configuration is disabling optimisation, or max (-O3 and --fast-math) which enables some slower-to-compile and potentially unsafe

optimisations. These options are applied *after* `compiler_options` and may override options provided there. Note that as for `compiler_options`, R will apply these *before* anything in your personal `Makevars`

`skip_cache` Logical, indicating if the cache of previously compiled models should be skipped. If `TRUE` then your model will not be looked for in the cache, nor will it be added to the cache after compilation.

Value

A `dust_generator` object based on your source files

Input requirements

Your input dust model must satisfy a few requirements.

- Define some class that implements your model (below `model` is assumed to be the class name)
- That class must define a type `internal_type` (so `model::internal_type`) that contains its internal data that the model may change during execution (i.e., that is not shared between particles). If no such data is needed, you can do `using internal_type = dust::no_internal;` to indicate this.
- We also need a type `shared_type` that contains *constant* internal data is shared between particles (e.g., dimensions, arrays that are read but not written). If no such data is needed, you can do `using share_type = dust::no_shared;` to indicate this.
- That class must also include a type alias that describes the model's floating point type, `real_type`. Most models can include `using real_type = double;` in their public section.
- The class must also include a type alias that describes the model's *data* type. If your model does not support data, then write `using data_type = dust::no_data;`, which disables the `compare_data` and `set_data` methods. Otherwise see `vignette("data")` for more information.
- The class must have a constructor that accepts `const dust::pars_type<model>& pars` for your type `model`. This will have elements `shared` and `internal` which you can assign into your model if needed.
- The model must have a method `size()` returning `size_t` which returns the size of the system. This size may depend on values in your initialisation object but is constant within a model run.
- The model must have a method `initial` (which may not be `const`), taking a time step number (`size_t`) and returning a `std::vector<real_type>` of initial state for the model.
- The model must have a method `update` taking arguments:
  - `size_t time`: the time step number
  - `const double * state`: the state at the beginning of the time step
  - `dust::rng_state_type<real_type>& rng_state`: the dust random number generator state - this *must* be a reference, as it will be modified as random numbers are drawn
  - `double *state_next`: the end state of the model (to be written to by your function)

Your `update` function is the core here and should update the state of the system - you are expected to update all values of `state` on return.

It is very important that none of the functions in the class use the R API in any way as these functions will be called in parallel.

You must also provide a data/parameter-wrangling function for producing an object of type `dust::pars_type<model>` from an R list. We use cpp11 for this. Your function will look like:

```
namespace dust {
template <>
dust::pars_type<model> dust_pars<model>(cpp11::list pars) {
  // ...
  return dust::pars_type<model>(shared, internal);
}
}
```

With the body interacting with `pars` to create an object of type `model::shared_type` and `model::internal_type` before returning the `dust::pars_type` object. This function will be called in serial and may use anything in the cpp11 API. All elements of the returned object must be standard C/C++ (e.g., STL) types and *not* cpp11/R types. If your model uses only shared or internal, you may use the single-argument constructor overload to `dust::pars_type` which is equivalent to using `dust::no_shared` or `dust::no_internal` for the missing argument.

Your model *may* provided a template specialisation `dust::dust_info<model>()` returning a `cpp11::sexp` for returning arbitrary information back to the R session:

```
namespace dust {
template <>
cpp11::sexp dust_info<model>(const dust::pars_type<sir>& pars) {
  return cpp11::wrap(...);
}
}
```

What you do with this is up to you. If not present then the `info()` method on the created object will return `NULL`. Potential use cases for this are to return information about variable ordering, or any processing done while accepting the pars object used to create the pars fed into the particles.

Configuring your model

You can optionally use C++ pseudo-attributes to configure the generated code. Currently we support two attributes:

- `[[dust::class(classname)]]` will tell dust the name of your target C++ class (in this example `classname`). You will need to use this if your file uses more than a single class, as otherwise will try to detect this using extremely simple heuristics.
- `[[dust::name(modelname)]]` will tell dust the name to use for the class in R code. For technical reasons this must be alphanumeric characters only (sorry, no underscore) and must not start with a number. If not included then the C++ type name will be used (either specified with `[[dust::class()]]` or detected).

Error handling

Your model should only throw exceptions as a last resort. One such last resort exists already if `rbinom` is given invalid inputs to prevent an infinite loop. If an error is thrown, all particles will complete their current run, and then the error will be rethrown - this is required by our parallel processing design. Once this happens though the state of the system is "inconsistent" as it contains particles that have run for different lengths of time. You can extract the state of the system at the point of failure (which may help with debugging) but you will be unable to continue running the object until either you reset it (with `$update_state()`). An error will be thrown otherwise.

Things are worse on a GPU; if an error is thrown by the RNG code (happens in `rbinom` when given impossible inputs such as negative sizes, probabilities less than zero or greater than 1) then we currently use CUDA's `__trap()` function which will require a process restart to be able to use anything that uses the GPU again, covering all methods in the class. However, this is preferable to the infinite loop that would otherwise be caused.

See Also

`dust_generator` for a description of the class of created objects, and `dust_example()` for some pre-built examples. If you want to just generate the code and load it yourself with `pkgload::load_all` or some other means, see `dust_generate`)

Examples

```
# dust includes a couple of very simple examples
filename <- system.file("examples/walk.cpp", package = "dust")

# This model implements a random walk with a parameter coming from
# R representing the standard deviation of the walk
writeLines(readLines(filename))

# The model can be compiled and loaded with dust::dust(filename)
# but it's faster in this example to use the prebuilt version in
# the package
model <- dust::dust_example("walk")

# Print the object and you can see the methods that it provides
model

# Create a model with standard deviation of 1, initial time step zero
# and 30 particles
obj <- model$new(list(sd = 1), 0, 30)
obj

# Curent state is all zero
obj$state()

# Current time is also zero
obj$time()

# Run the model up to time step 100
obj$run(100)
```

```
# Reorder/resample the particles:
obj$reorder(sample(30, replace = TRUE))

# See the state again
obj$state()
```

`dust_cuda_configuration` *Detect CUDA configuration*

Description

Detect CUDA configuration. This function tries to compile a small program with `nvcc` and confirms that this can be loaded into R, then uses that program to query the presence and capabilities of your NVIDIA GPUs. If this works, then you can use the GPU-enabled dust features, and the information returned will help us. It's quite slow to execute (several seconds) so we cache the value within a session. Later versions of dust will cache this across sessions too.

Usage

```
dust_cuda_configuration(
  path_cuda_lib = NULL,
  path_cub_include = NULL,
  quiet = TRUE,
  forget = FALSE
)
```

Arguments

`path_cuda_lib`	Optional path to the CUDA libraries, if they are not on system library paths. This will be added as `-L{path_cuda_lib}` in calls to `nvcc`
`path_cub_include`	Optional path to the CUB headers, if using CUDA < 11.0.0. See Details
`quiet`	Logical, indicating if compilation of test programs should be quiet
`forget`	Logical, indicating if we should forget cached values and recompute the configuration

Details

Not all installations leave the CUDA libraries on the default paths, and you may need to provide it. Specifically, when we link the dynamic library, if the linker complains about not being able to find `libcudart` then your CUDA libraries are not in the default location. You can manually pass in the `path_cuda_lib` argument, or set the `DUST_PATH_CUDA_LIB` environment variable (in that order of precedence).

If you are using older CUDA (< 11.0.0) then you need to provide CUB headers, which we use to manage state on the device (these are included in CUDA 11.0.0 and higher). You can provide this as:

- a path to this function (`path_cub_include`)
- the environment variable DUST_PATH_CUB_INCLUDE
- CUB headers installed into the default location (R >= 4.0.0, see below).

These are checked in turn with the first found taking precedence. The default location is stored with `tools::R_user_dir("dust", "data")`, but this functionality is only available on R >= 4.0.0.

To install CUB you can do:

```
dust:::cuda_install_cub(NULL)
```

which will install CUB into the default path (provide a path on older versions of R and set this path as DUST_PATH_CUB_INCLUDE).

For editing your .Renviron file to set these environment variables, `usethis::edit_r_environ()` is very helpful.

Value

A list of configuration information. This includes:

- `has_cuda`: logical, indicating if it is possible to compile CUDA on this machine (not necessarily use it though)
- `cuda_version`: the version of CUDA found
- `devices`: a data.frame of device information:
  - id: the device id (integer, typically in a sequence from 0)
  - name: the human-friendly name of the device
  - memory: the memory of the device, in MB
  - version: the compute version for this device
- `path_cuda_lib`: path to CUDA libraries, if required
- `path_cub_include`: path to CUB headers, if required

If compilation of the test program fails, then `has_cuda` will be `FALSE` and all other elements will be `NULL`.

See Also

dust_cuda_options which controls additional CUDA compilation options (e.g., profiling, debug mode or custom flags)

Examples

```
# If you have your CUDA library in an unusual location, then you
# may need to add a path_cuda_lib argument:
dust::dust_cuda_configuration(
  path_cuda_lib = "/usr/local/cuda-11.1/lib64",
  forget = TRUE, quiet = FALSE)

# However, if things are installed in the default location or you
# have set the environment variables described above, then this
# may work:
dust::dust_cuda_configuration(forget = TRUE, quiet = FALSE)
```

`dust_cuda_options` *Create CUDA options*

Description

Create options for compiling for CUDA. Unless you need to change paths to libraries/headers, or change the debug level you will probably not need to directly use this. However, it's potentially useful to see what is being passed to the compiler.

Usage

```
dust_cuda_options(
  ...,
  debug = FALSE,
  profile = FALSE,
  fast_math = FALSE,
  flags = NULL
)
```

Arguments

`...`	Arguments passed to `dust_cuda_configuration()`
`debug`	Logical, indicating if we should compile for debug (adding `-g`, `-G` and `-O0`)
`profile`	Logical, indicating if we should enable profiling
`fast_math`	Logical, indicating if we should enable "fast maths", which lets the optimiser enable optimisations that break IEEE compliance and disables some error checking (see the CUDA docs for more details).
`flags`	Optional extra arguments to pass to nvcc. These options will not be passed to your normal C++ compiler, nor the linker (for that use R's user Makevars system). This can be used to do things like tune the maximum number of registers (`--maxrregcount x`). If not `NULL`, this must be a character vector, which will be concatenated with spaces between options.

Value

An object of type `cuda_options`, which can be passed into dust as argument `gpu`

See Also

dust_cuda_configuration which identifies and returns the core CUDA configuration (often used implicitly by this function).

Examples

```
tryCatch(
  dust::dust_cuda_options(),
  error = function(e) NULL)
```

`dust_data` *Process data for dust*

Description

Prepare data for use with the `$set_data()` method. This is not required for use but tries to simplify the most common use case where you have a data.frame with some column indicating "dust time step" (`name_time`), and other columns that might be use in your `data_compare` function. Each row will be turned into a named R list, which your `dust_data` function can then work with to get this time-steps values. See Details for use with multi-pars objects.

Usage

```
dust_data(object, name_time = "time", multi = NULL)
```

Arguments

`object`	An object, at this point must be a data.frame
`name_time`	The name of the data column within `object`; this column must be integer-like and every integer must be nonnegative and unique
`multi`	Control how to interpret data for multi-parameter dust object; see Details

Details

Note that here "dust time step" (`name_time`) refers to the *dust* time step (which will be a non-negative integer) and not the rescaled value of time that you probably use within the model. See dust_generator for more information.

The data object as accepted by `data_set` must be a list and each element must itself be a list with two elements; the dust `time` at which the data applies and any R object that corresponds to data at that point. We expect that most of the time this second element will be a key-value list with scalar keys, but more flexibility may be required.

For multi-data objects, the final format is a bit more awkward; each time step we have a list with elements `time`, `data_1`, `data_2`, ..., `data_n` for `n` parameters. There are two ways of creating this that might be useful: *sharing* the data across all parameters and using some column as a grouping value.

The behaviour here is driven by the `multi` argument;

- `NULL`: (the default) do nothing; this creates an object that is suitable for use with a `pars_multi = FALSE` dust object.
- `<integer>` (e.g., multi = 3); share the data across 3 sets of parameters. This number must match the number of parameter sets that your dust object is created with
- `<column_name>` (e.g., multi = "country"); the name of a column within your data to split the data at. This column must be a factor, and that factor must have levels that map to integers 1, 2, ..., n (e.g., `unique(as.integer(object[[multi]]))` returns the integers `1:n`).

Value

A list of dust time/data pairs that will be used for the compare function in a compiled model. Each element is a list of length two or more where the first element is the time step and the subsequent elements are data for that time step.

Examples

```
d <- data.frame(time = seq(0, 50, by = 10), a = runif(6), b = runif(6))
dust::dust_data(d)
```

---

dust_example *Access dust's built-in examples*

Description

Access dust's built-in examples. These are compiled into the package so that examples and tests can be run more quickly without having to compile code directly via `dust()`. These examples are all "toy" examples, being small and fast to run.

Usage

```
dust_example(name)
```

Arguments

`name`	The name of the example to use. There are five examples: `sir`, `sirs`, `variable`, `volatility`, `walk` and `logistic` (see Details).

Details

- `sir`: a basic SIR (Susceptible, Infected, Resistant) epidemiological model. Draws from the binomial distribution to update the population between each time step.
- `sirs`: an SIRS model, the SIR model with an added R->S transition. This has a non-zero steady state, so can be run indefinitely for testing.
- `volatility`: A volatility model that might be applied to currency fluctuations etc.
- `walk`: A 1D random walk, following a Gaussian distribution each time step.
- `logistic`: Logistic growth in continuous time

Value

A `dust_generator` object that can be used to create a model. See examples for usage.

Examples

```
# A SIR (Susceptible, Infected, Resistant) epidemiological model
sir <- dust::dust_example("sir")
sir

# Initialise the model at time step 0 with 50 independent trajectories
mod <- sir$new(list(), 0, 50)

# Run the model for 400 steps, collecting "infected" every 4th time step
times <- seq(0, 400, by = 4)
mod$set_index(2L)
y <- mod$simulate(times)

# A plot of our epidemic
matplot(times, t(drop(y)), type = "l", lty = 1, col = "#00000044",
        las = 1, xlab = "Time", ylab = "Number infected")
```

`dust_generate` *Generate dust code*

Description

Generate a package out of a dust model. The resulting package can be installed or loaded via `pkgload::load_all()` though it contains minimal metadata and if you want to create a persistent package you should use `dust_package()`. This function is intended for cases where you either want to inspect the code or generate it once and load multiple times (useful in some workflows with CUDA models).

Usage

```
dust_generate(
  filename,
  quiet = FALSE,
  workdir = NULL,
  gpu = FALSE,
  real_type = NULL,
  linking_to = NULL,
  cpp_std = NULL,
  compiler_options = NULL,
  optimisation_level = NULL,
  mangle = FALSE
)
```

Arguments

`filename`	The path to a single C++ file

`quiet` Logical, indicating if compilation messages from `pkgbuild` should be displayed. Error messages will be displayed on compilation failure regardless of the value used.

`workdir` Optional working directory to use. If `NULL` uses a temporary directory. By using a different directory of your choosing you can see the generated code.

`gpu` Logical, indicating if we should generate GPU code. This requires a considerable amount of additional software installed (CUDA toolkit and drivers) as well as a CUDA-compatible GPU. If `TRUE`, then we call dust_cuda_options with no arguments. Alternatively, call that function and pass the value here (e.g, `gpu = dust::dust_cuda_options(debug = TRUE)`). Note that due to the use of the `__syncwarp()` primitive this may require a GPU with compute version 70 or higher.

`real_type` Optionally, a string indicating a substitute type to swap in for your model's `real_type` declaration. If given, then we replace the string `using real_type = (double|float)` with the given type. This is primarily intended to be used as `gpu = TRUE, real_type = "float"` in order to create model for the GPU that will use 32 bit `floats` (rather than 64 bit doubles, which are much slower). For CPU models decreasing precision of your real type will typically just decrease precision for no additional performance.

`linking_to` Optionally, a character vector of additional packages to add to the `DESCRIPTION`'s `LinkingTo` field. Use this when your model pulls in C++ code that is packaged within another package's header-only library.

`cpp_std` The C++ standard to use, if you need to set one explicitly. See the section "Using C++ code" in "Writing R extensions" for the details of this, and how it interacts with the R version currently being used. For R 4.0.0 and above, C++11 will be used; as dust depends on at least this version of R you will never need to specify a version this low. Sensible options are `C++14`, `C++17`, etc, depending on the features you need and what your compiler supports.

`compiler_options` A character vector of additional options to pass through to the C++ compiler. These will be passed through without any shell quoting or validation, so check the generated commands and outputs carefully in case of error. Note that R will apply these *before* anything in your personal `Makevars`.

`optimisation_level` A shorthand way of specifying common compiler options that control optimisation level. By default (`NULL`) no options are generated from this, and the optimisation level will depend on your user `Makevars` file. Valid options are `none` which disables optimisation (`-O0`), which will be faster to compile but much slower, `standard` which enables standard level of optimisation (`-O2`), useful if your Makevars/pkgload configuration is disabling optimisation, or `max` (`-O3` and `--fast-math`) which enables some slower-to-compile and potentially unsafe optimisations. These options are applied *after* `compiler_options` and may override options provided there. Note that as for `compiler_options`, R will apply these *before* anything in your personal `Makevars`

`mangle` Logical, indicating if the model name should be mangled when creating the package. This is safer if you will load multiple copies of the package into a single session, but is `FALSE` by default as the generated code is easier to read.

Value

The path to the generated package (will be `workdir` if that was provided, otherwise a temporary directory).

Examples

```
filename <- system.file("examples/walk.cpp", package = "dust")
path <- dust::dust_generate(filename)

# Simple package created:
dir(path)
dir(file.path(path, "R"))
dir(file.path(path, "src"))
```

---

dust_generator *The dust class*

Description

All "dust" dust models are R6 objects and expose a common set of "methods". To create a dust model of your own, see dust and to interact with some built-in ones see `dust_example()`

Value

A `dust_generator` object

Time

For discrete time models, dust has an internal "time", which was called `step` in version `0.11.x` and below. This must always be non-negative (i.e., zero or more) and always increases in unit increments. Typically a model will remap this internal time onto a more meaningful time in model space, e.g. by applying the transform `model_time = offset + time * dt`; with this approach you can start at any real valued time and scale the unit increments to control the model dynamics.

Methods

Public methods:

- `dust_generator$new()`
- `dust_generator$name()`
- `dust_generator$param()`
- `dust_generator$run()`
- `dust_generator$simulate()`
- `dust_generator$run_adjoint()`
- `dust_generator$set_index()`
- `dust_generator$index()`
- `dust_generator$ode_control()`

- dust_generator$ode_statistics()
- dust_generator$n_threads()
- dust_generator$n_state()
- dust_generator$n_particles()
- dust_generator$n_particles_each()
- dust_generator$shape()
- dust_generator$update_state()
- dust_generator$state()
- dust_generator$time()
- dust_generator$set_stochastic_schedule()
- dust_generator$reorder()
- dust_generator$resample()
- dust_generator$info()
- dust_generator$pars()
- dust_generator$rng_state()
- dust_generator$set_rng_state()
- dust_generator$has_openmp()
- dust_generator$has_gpu_support()
- dust_generator$has_compare()
- dust_generator$real_size()
- dust_generator$time_type()
- dust_generator$rng_algorithm()
- dust_generator$uses_gpu()
- dust_generator$n_pars()
- dust_generator$set_n_threads()
- dust_generator$set_data()
- dust_generator$compare_data()
- dust_generator$filter()
- dust_generator$gpu_info()

Method `new()`**:** Create a new model. Note that the behaviour of this object created by this function will change considerably based on whether the `pars_multi` argument is `TRUE`. If not (the default) then we create `n_particles` which all share the same parameters as specified by the `pars` argument. If `pars_multi` is `TRUE` then `pars` must be an unnamed list, and each element of it represents a different set of parameters. We will create `length(pars)` *sets* of `n_particles` particles which will be simulated together. These particles must have the same dimension - that is, they must correspond to model state that is the same size.

Usage:

```
dust_generator$new(
  pars,
  time,
  n_particles,
  n_threads = 1L,
  seed = NULL,
```

```
  pars_multi = FALSE,
  deterministic = FALSE,
  gpu_config = NULL,
  ode_control = NULL
)
```

Arguments:

`pars` Data to initialise your model with; a `list` object, but the required elements will depend on the details of your model. If `pars_multi` is `TRUE`, then this must be an *unnamed* list of `pars` objects (see Details).

`time` Initial time - must be nonnegative

`n_particles` Number of particles to create - must be at least 1

`n_threads` Number of OMP threads to use, if `dust` and your model were compiled with OMP support (details to come). `n_particles` should be a multiple of `n_threads` (e.g., if you use 8 threads, then you should have 8, 16, 24, etc particles). However, this is not compulsory.

`seed` The seed to use for the random number generator. Can be a positive integer, `NULL` (initialise with R's random number generator) or a `raw` vector of a length that is a multiple of 32 to directly initialise the generator (e..g., from the `dust` object's `$rng_state()` method).

`pars_multi` Logical, indicating if `pars` should be interpreted as a set of different initialisations, and that we should prepare `n_particles * length(pars)` particles for simulation. This has an effect on many of the other methods of the object.

`deterministic` Run random number generation deterministically, replacing a random number from some distribution with its expectation. Deterministic models are not compatible with running on a a GPU.

`gpu_config` GPU configuration, typically an integer indicating the device to use, where the model has GPU support. If not given, then the default value of `NULL` will fall back on the first found device if any are available. An error is thrown if the device id given is larger than those reported to be available (note that CUDA numbers devices from 0, so that '0' is the first device, and so on). See the method `$gpu_info()` for available device ids; this can be called before object creation as `dust_generator$public_methods$gpu_info()`. For additional control, provide a list with elements `device_id` and `run_block_size`. Further options (and validation) of this list will be added in a future version!

`ode_control` For ODE models, control over the integration; must be a `dust_ode_control` model, produced by `dust_ode_control()`. It is an error to provide a non-`NULL` value for discrete time models.

Method `name()`**:** Returns friendly model name

Usage:

```
dust_generator$name()
```

Method `param()`**:** Returns parameter information, if provided by the model. This describes the contents of pars passed to the constructor or to `$update_state()` as the `pars` argument, and the details depend on the model.

Usage:

```
dust_generator$param()
```

Method `run()`**:** Run the model up to a point in time, returning the filtered state at that point.

Usage:

```
dust_generator$run(time_end)
```

Arguments:

`time_end` Time to run to (if less than or equal to the current time(), silently nothing will happen)

Method `simulate()`**:** Iterate all particles forward in time over a series of times, collecting output as they go. This is a helper around `$run()` where you want to run to a series of points in time and save output. The returned object will be filtered by your active index, so that it has shape (`n_state` x `n_particles` x `length(time_end)`) for single-parameter objects, and (`n_state` x `n_particles` x `n_pars` x `length(time_end)`) for multiparameter objects. Note that this method is very similar to `$run()` except that the rank of the returned array is one less. For a scalar `time_end` you would ordinarily want to use `$run()` but the resulting numbers would be identical.

Usage:

```
dust_generator$simulate(time_end)
```

Arguments:

`time_end` A vector of time points that the simulation should report output at. This the first time must be at least the same as the current time, and every subsequent time must be equal or greater than those before it (ties are allowed though probably not wanted).

Method `run_adjoint()`**:** Run model with gradient information (if supported). The interface here will change, and documentation written once it stabilises.

Usage:

```
dust_generator$run_adjoint()
```

Method `set_index()`**:** Set the "index" vector that is used to return a subset of pars after using `run()`. If this is not used then `run()` returns all elements in your state vector, which may be excessive and slower than necessary.

Usage:

```
dust_generator$set_index(index)
```

Arguments:

`index` The index vector - must be an integer vector with elements between 1 and the length of the state (this will be validated, and an error thrown if an invalid index is given).

Method `index()`**:** Returns the `index` as set by `$set_index`

Usage:

```
dust_generator$index()
```

Method `ode_control()`**:** Return the ODE control set into the object on creation. For discrete-time models this always returns `NULL`.

Usage:

```
dust_generator$ode_control()
```

Method `ode_statistics()`**:** Return statistics about the integration, for ODE models. For discrete time models this makes little sense and so errors if used.

Usage:

```
dust_generator$ode_statistics()
```

Method `n_threads()`**:** Returns the number of threads that the model was constructed with

Usage:

```
dust_generator$n_threads()
```

Method `n_state()`**:** Returns the length of the per-particle state

Usage:

```
dust_generator$n_state()
```

Method `n_particles()`**:** Returns the number of particles

Usage:

```
dust_generator$n_particles()
```

Method `n_particles_each()`**:** Returns the number of particles per parameter set

Usage:

```
dust_generator$n_particles_each()
```

Method `shape()`**:** Returns the shape of the particles

Usage:

```
dust_generator$shape()
```

Method `update_state()`**:** Update one or more components of the model state. This method can be used to update any or all of `pars`, `state` and `time`. If both `pars` and `time` are given and `state` is not, then by default we will update the model internal state according to your model's initial conditions - use `set_initial_state = FALSE` to prevent this.

Usage:

```
dust_generator$update_state(
  pars = NULL,
  state = NULL,
  time = NULL,
  set_initial_state = NULL,
  index = NULL,
  reset_step_size = NULL
)
```

Arguments:

`pars` New pars for the model (see constructor)

`state` The state vector - can be either a numeric vector with the same length as the model's current state (in which case the same state is applied to all particles), or a numeric matrix with as many rows as your model's state and as many columns as you have particles (in which case you can set a number of different starting states at once).

`time` New initial time for the model. If this is a vector (with the same length as the number of particles), then particles are started from different initial times and run up to the largest time given (i.e., `max(time)`)

`set_initial_state` Control if the model initial state should be set while setting parameters. It is an error for this to be `TRUE` when either `pars` is `NULL` or when `state` is non-`NULL`.

`index` Used in conjunction with `state`, use this to set a fraction of the model state; the `index` vector provided must be the same length as the number of provided states, and indicates the index within the model state that should be updated. For example, if your model has states `[a, b, c, d]` and you provide an index of `[1, 3]` then of `state` was `[10, 20]` you would set `a` to 10 and `c` to 20.

`reset_step_size` Logical, indicating if we should reset the initial step size. This only has an effect with ode models and is silently ignored in discrete time models where the step size is constant.

Method `state()`**:** Return full model state

Usage:

```
dust_generator$state(index = NULL)
```

Arguments:

`index` Optional index to select state using

Method `time()`**:** Return current model time For ODE models, sets the schedule at which stochastic events are handled. The timing here is quite subtle - an event happens immediately *after* the time (so at `time + eps`). If your model runs up to `time` an event is not triggered, but as soon as that time is passed, by any amount of time, the event will trigger. It is an error to set this to a non-`NULL` value in a discrete time model; later we may generalise the approach here.

Usage:

```
dust_generator$time()
```

Method `set_stochastic_schedule()`**:**

Usage:

```
dust_generator$set_stochastic_schedule(time)
```

Arguments:

`time` A vector of times to run the stochastic update at

Method `reorder()`**:** Reorder particles.

Usage:

```
dust_generator$reorder(index)
```

Arguments:

`index` An integer vector, with values between 1 and n_particles, indicating the index of the current particles that new particles should take.

Method `resample()`**:** Resample particles according to some weight.

Usage:

```
dust_generator$resample(weights)
```

Arguments:

`weights` A numeric vector representing particle weights. For a "multi-parameter" dust object this should be be a matrix with the number of rows being the number of particles per parameter set and the number of columns being the number of parameter sets. long as all particles or be a matrix.

Method `info()`**:** Returns information about the pars that your model was created with. Only returns non-NULL if the model provides a `dust_info` template specialisation.

Usage:

```
dust_generator$info()
```

Method `pars()`**:** Returns the `pars` object that your model was constructed with.

Usage:

```
dust_generator$pars()
```

Method `rng_state()`**:** Returns the state of the random number generator. This returns a raw vector of length 32 * n_particles. This can be useful for debugging or for initialising other dust objects. The arguments `first_only` and `last_only` are mutually exclusive. If neither is given then all all particles states are returned, being 32 bytes per particle. The full returned state or `first_only` are most suitable for reseeding a new dust object.

Usage:

```
dust_generator$rng_state(first_only = FALSE, last_only = FALSE)
```

Arguments:

`first_only` Logical, indicating if we should return only the first random number state

`last_only` Logical, indicating if we should return only the *last* random number state, which does not belong to a particle.

Method `set_rng_state()`**:** Set the random number state for this model. This replaces the RNG state that the model is using with a state of your choosing, saved out from a different model object. This method is designed to support advanced use cases where it is easier to manipulate the state of the random number generator than the internal state of the dust object.

Usage:

```
dust_generator$set_rng_state(rng_state)
```

Arguments:

`rng_state` A random number state, as saved out by the `$rng_state()` method. Note that unlike `seed` as passed to the constructor, this *must* be a raw vector of the expected length.

Method `has_openmp()`**:** Returns a logical, indicating if this model was compiled with "OpenMP" support, in which case it will react to the `n_threads` argument passed to the constructor. This method can also be used as a static method by running it directly as `dust_generator$public_methods$has_openmp()`

Usage:

```
dust_generator$has_openmp()
```

Method `has_gpu_support()`**:** Returns a logical, indicating if this model was compiled with "CUDA" support, in which case it will react to the `device` argument passed to the run method. This method can also be used as a static method by running it directly as `dust_generator$public_methods$has_gpu_sup`

Usage:

```
dust_generator$has_gpu_support(fake_gpu = FALSE)
```

Arguments:

`fake_gpu` Logical, indicating if we count as `TRUE` models that run on the "fake" GPU (i.e., using the GPU version of the model but running on the CPU)

Method `has_compare()`**:** Returns a logical, indicating if this model was compiled with "compare" support, in which case the `set_data` and `compare_data` methods are available (otherwise these methods will error). This method can also be used as a static method by running it directly as `dust_generator$public_methods$has_compare()`

Usage:

```
dust_generator$has_compare()
```

Method `real_size()`**:** Return the size of real numbers (in bits). Typically this will be 64 for double precision and 32 for `float`. This method can also be used as a static method by running it directly as `dust_generator$public_methods$real_size()`

Usage:

```
dust_generator$real_size()
```

Method `time_type()`**:** Return the type of time this model uses; will be one of `discrete` (for discrete time models) or `continuous` (for ODE models). This method can also be used as a static method by running it directly as `dust_generator$public_methods$time_type()`

Usage:

```
dust_generator$time_type()
```

Method `rng_algorithm()`**:** Return the random number algorithm used. Typically this will be `xoshiro256plus` for models using double precision reals and `xoshiro128plus` for single precision (`float`). This method can also be used as a static method by running it directly as `dust_generator$public_methods$rng_algorithm()`

Usage:

```
dust_generator$rng_algorithm()
```

Method `uses_gpu()`**:** Check if the model is running on a GPU

Usage:

```
dust_generator$uses_gpu(fake_gpu = FALSE)
```

Arguments:

`fake_gpu` Logical, indicating if we count as `TRUE` models that run on the "fake" GPU (i.e., using the GPU version of the model but running on the CPU)

Method `n_pars()`**:** Returns the number of distinct pars elements required. This is `0` where the object was initialised with `pars_multi = FALSE` and an integer otherwise. For multi-pars dust objects, Where `pars` is accepted, you must provide an unnamed list of length `$n_pars()`.

Usage:

```
dust_generator$n_pars()
```

Method `set_n_threads()`**:** Change the number of threads that the dust object will use. Your model must be compiled with "OpenMP" support for this to have an effect. Returns (invisibly) the previous value.

Usage:

```
dust_generator$set_n_threads(n_threads)
```

Arguments:

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

Method `set_data()`**:** Set "data" into the model for use with the `$compare_data()` method. This is not supported by all models, depending on if they define a `data_type` type. See `dust_data()` for a helper function to construct suitable data and a description of the required format. You will probably want to use that here, and definitely if using multiple parameter sets.

Usage:

```
dust_generator$set_data(data, shared = FALSE)
```

Arguments:

`data` A list of data to set.

`shared` Logical, indicating if the data should be shared across all parameter sets, if your model is initialised to use more than one parameter set (`pars_multi = TRUE`).

Method `compare_data()`**:** Compare the current model state against the data as set by `set_data`. If there is no data set, or no data corresponding to the current time then `NULL` is returned. Otherwise a numeric vector the same length as the number of particles is returned. If model's underlying `compare_data` function is stochastic, then each call to this function may be result in a different answer.

Usage:

```
dust_generator$compare_data()
```

Method `filter()`**:** Run a particle filter. The interface here will change a lot over the next few versions. You *must* reset the dust object using `$update_state(pars = ..., time = ...)` before using this method to get sensible values.

Usage:

```
dust_generator$filter(
  time_end = NULL,
  save_trajectories = FALSE,
  time_snapshot = NULL,
  min_log_likelihood = NULL
)
```

Arguments:

`time_end` The time to run to. If `NULL`, run to the end of the last data. This value must be larger than the current model time (`$time()`) and must exactly appear in the data.

`save_trajectories` Logical, indicating if the filtered particle trajectories should be saved. If `TRUE` then the `trajectories` element will be a multidimensional array (`state x <shape> x time`) containing the state values, selected according to the index set with `$set_index()`.

`time_snapshot` Optional integer vector indicating times that we should record a snapshot of the full particle filter state. If given it must be strictly increasing vector whose elements match times given in the `data` object. The return value with be a multidimensional array (`state x <shape> x time_snapshot`) containing full state values at the requested times.

`min_log_likelihood` Optionally, a numeric value representing the smallest likelihood we are interested in. If non-`NULL` either a scalar value or vector the same length as the number of parameter sets. Not yet supported, and included for future compatibility.

Method `gpu_info()`**:** Return information about GPU devices, if the model has been compiled with CUDA/GPU support. This can be called as a static method by running `dust_generator$public_methods$gpu_info` If run from a GPU enabled object, it will also have an element `config` containing the computed device configuration: the device id, shared memory and the block size for the `run` method on the device.

Usage:

```
dust_generator$gpu_info()
```

Examples

```
# An example dust object from the package:
walk <- dust::dust_example("walk")

# The generator object has class "dust_generator"
class(walk)

# The methods below are are described in the documentation
walk
```

`dust_ode_control` *Create a dust_ode_control object.*

Description

Create a control object for controlling the adaptive stepper for systems of ordinary differential equations (ODEs). The returned object can be passed into a continuous-time dust model on initialisation.

Usage

```
dust_ode_control(
  max_steps = NULL,
  atol = NULL,
  rtol = NULL,
  step_size_min = NULL,
  step_size_max = NULL,
  debug_record_step_times = NULL
)
```

Arguments

`max_steps`	Maxmimum number of steps to take. If the integration attempts to take more steps that this, it will throw an error, stopping the integration.
`atol`	The per-step absolute tolerance.
`rtol`	The per-step relative tolerance. The total accuracy will be less than this.
`step_size_min`	The minimum step size. The actual minimum used will be the largest of the absolute value of this `step_size_min` or `.Machine$double.eps`. If the integration attempts to make a step smaller than this, it will throw an error, stopping the integration.

`step_size_max` The largest step size. By default there is no maximum step size (Inf) so the solver can take as large a step as it wants to. If you have short-lived fluctuations in your rhs that the solver may skip over by accident, then specify a smaller maximum step size here.

`debug_record_step_times`
Logical, indicating if we should record the steps taken. This information will be available as part of the `statistics()` output

Value

A named list of class "dust_ode_control"

Examples

```
# We include an example of logistic growth with the package
gen <- dust::dust_example("logistic")

# Create a control object, then pass it through as the ode_control
# parameter to the constructor:
ctrl <- dust::dust_ode_control(atol = 1e-3, rtol = 1e-3)
mod <- gen$new(list(r = 0.1, K = 100), 0, 1, ode_control = ctrl)

# When the model runs, the control parameters passed in to the
# constructor are used in the solution
mod$run(10)

# The full set of control parameters can be extracted:
mod$ode_control()
```

`dust_openmp_support` *Information about OpenMP support*

Description

Return information about OpenMP support for this system. For individual models look at the `$has_openmp()` method.

Usage

```
dust_openmp_support(check_compile = FALSE)
```

Arguments

`check_compile` Logical, indicating if we should check if we can compile an openmp program - this is slow the first time.

Value

A list with information about the openmp support on your system.

- The first few elements come from the openmp library directly: `num_proc`, `max_threads`, `thread_limit`; these correspond to a call to the function `omp_get_<name>()` in C and `openmp_version` which is the value of the `_OPENMP` macro.
- A logical `has_openmp` which is `TRUE` if it looks like runtime OpenMP support is available
- The next elements tell you about different sources that might control the number of threads allowed to run: `mc.cores` (from the R option with the same name), `OMP_THREAD_LIMIT`, `OMP_NUM_THREADS`, `MC_CORES` (from environment variables), `limit_r` (limit computed against R-related control variables), `limit_openmp` (limit computed against OpenMP-related variables) and `limit` the smaller of `limit_r` and `limit_openmp`

See Also

`dust_openmp_threads()` for setting a polite number of threads.

Examples

```
# System wide support
dust::dust_openmp_support()

# Support compiled into a generator
walk <- dust::dust_example("walk")
walk$public_methods$has_openmp()

# Support from an instance of that model
model <- walk$new(list(sd = 1), 0, 1)
model$has_openmp()
```

---

dust_openmp_threads *Select number of threads*

Description

Politely select a number of threads to use. See Details for the algorithm

Usage

```
dust_openmp_threads(n = NULL, action = "error")
```

Arguments

`n`	Either `NULL` (select automatically) or an integer as your proposed number of threads.
`action`	An action to perform if `n` exceeds the maximum number of threads you can use. Options are "error" (the default, throw an error), "fix" (print a message and reduce `n` down to the limit) or "message" (print a message and continue anyway)

Details

There are two limits and we will take the smaller of the two.

The first limit comes from piggy-backing off of R's normal parallel configuration; we will use the `MC_CORES` environment variable and `mc.cores` option as a guide to how many cores you are happy to use. We take `mc.cores` first, then `MC_CORES`, which is the same behaviour as `parallel::mclapply` and friends.

The second limit comes from openmp. If you do not have OpenMP support, then we use one thread (higher numbers have no effect at all in this case). If you do have OpenMP support, we take the smallest of the number of "processors" (reported by `omp_get_num_procs()`) the "max threads" (reported by `omp_get_max_threads()` and "thread_limit" (reported by `omp_get_thread_limit()`.

See `dust_openmp_support()` for the values of all the values that go into this calculation.

Value

An integer, indicating the number of threads that you can use

Examples

```
# Default number of threads; tries to pick something friendly,
# erring on the conservative side.
dust::dust_openmp_threads(NULL)

# Try to pick something silly and it will be reduced for you
dust::dust_openmp_threads(1000, action = "fix")
```

---

dust_package *Create dust model in package*

Description

Updates a dust model in a package. The user-provided code is assumed to the in `inst/dust` as a series of C++ files; a file `inst/dust/model.cpp` will be transformed into a file `src/model.cpp`.

Usage

```
dust_package(path, quiet = FALSE)
```

Arguments

`path`	Path to the package
`quiet`	Passed to `cpp11::cpp_register`, if `TRUE` suppresses informational notices about updates to the cpp11 files

Details

If your code provides a class `model` then dust will create C++ functions such as `dust_model_alloc` - if your code also includes names such as this, compilation will fail due to duplicate symbols.

We add "cpp11 attributes" to the created functions, and will run `cpp11::cpp_register()` on them once the generated code has been created.

Your package needs a `src/Makevars` file to enable openmp (if your system supports it). If it is not present then a suitable Makevars will be written, containing

```
PKG_CXXFLAGS=$(SHLIB_OPENMP_CXXFLAGS)
PKG_LIBS=$(SHLIB_OPENMP_CXXFLAGS)
```

following "Writing R Extensions" (see section "OpenMP support"). If your package does contain a `src/Makevars` file we do not attempt to edit it but will error if it looks like it does not contain these lines or similar.

You also need to make sure that your package loads the dynamic library; if you are using roxygen, then you might create a file (say, `R/zzz.R`) containing

```
#' @useDynLib packagename, .registration = TRUE
NULL
```

substituting `packagename` for your package name as appropriate. This will create an entry in `NAMESPACE`.

Value

Nothing, this function is called for its side effects

See Also

`vignette("dust")` which contains more discussion of this process

Examples

```
# This is explained in more detail in the package vignette
path <- system.file("examples/sir.cpp", package = "dust", mustWork = TRUE)
dest <- tempfile()
dir.create(dest)
dir.create(file.path(dest, "inst/dust"), FALSE, TRUE)
writeLines(c("Package: example",
             "Version: 0.0.1",
             "LinkingTo: cpp11, dust"),
           file.path(dest, "DESCRIPTION"))
writeLines("useDynLib('example', .registration = TRUE)",
           file.path(dest, "NAMESPACE"))
file.copy(path, file.path(dest, "inst/dust"))

# An absolutely minimal skeleton contains a DESCRIPTION, NAMESPACE
# and one or more dust model files to compile:
dir(dest, recursive = TRUE)
```

```
# Running dust_package will fill in the rest
dust::dust_package(dest)

# More files here now
dir(dest, recursive = TRUE)
```

`dust_repair_environment` *Repair dust environment*

Description

Repair the environment of a dust object created by [dust] and then saved and reloaded by [saveRDS] and [readRDS]. Because we use a fake temporary package to hold the generated code, it will not ordinarily be loaded properly without using this.

Usage

```
dust_repair_environment(generator, quiet = FALSE)
```

Arguments

`generator`	The dust generator
`quiet`	Logical, indicating if we should be quiet (default prints some progress information)

Value

Nothing, called for its side effects

`dust_rng` *Dust Random Number Generator*

Description

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

Value

A `dust_rng` object, which can be used to drawn random numbers from dust'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 (see dust_openmp_support).

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; consider using dust_openmp_threads to select a safe number.

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:

- dust_rng$new()
- dust_rng$size()
- dust_rng$jump()
- dust_rng$long_jump()
- dust_rng$random_real()
- dust_rng$random_normal()
- dust_rng$uniform()
- dust_rng$normal()
- dust_rng$binomial()
- dust_rng$nbinomial()
- dust_rng$hypergeometric()
- dust_rng$gamma()
- dust_rng$poisson()
- dust_rng$exponential()
- dust_rng$cauchy()
- dust_rng$multinomial()
- dust_rng$state()

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

Usage:

```
dust_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:

```
dust_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:

```
dust_rng$jump()
```

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

Usage:

```
dust_rng$long_jump()
```

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

Usage:

```
dust_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:

```
dust_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:

```
dust_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:

```
dust_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:

```
dust_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:

```
dust_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:

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

Usage:

```
dust_rng$gamma(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 `poisson()`**:** Generate `n` numbers from a Poisson distribution

Usage:

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

Usage:

```
dust_rng$exponential(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 `cauchy()`**:** Generate `n` draws from a Cauchy distribution.

Usage:

```
dust_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:

```
dust_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 `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 dust_rng object with this state.

Usage:

```
dust_rng$state()
```

Examples

```
rng <- dust::dust_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 a and b
rng$gamma(5, 0.5, 2)

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

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

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

---

dust_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

```
dust_rng_distributed_state(
```

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

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

Arguments

seed	Initial seed to use. As for dust_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. If passing the results of this seed to a dust object's initialiser (see dust_generator) you can safely leave this at 1, but if using in a standalone setting, and especially if using dust_rng_distributed_pointer, you may need to set this to the appropriate length.
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. Alternatively pass a dust_generator or dust object here to select the algorithm used by that object automatically.

Details

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

Value

A list of either raw vectors (for dust_rng_distributed_state) or of dust_rng_pointer objects (for dust_rng_distributed_pointer)

Examples

```
dust::dust_rng_distributed_state(n_nodes = 2)
dust::dust_rng_distributed_pointer(n_nodes = 2)
```

---

dust_rng_pointer *Create pointer to random number generator stream*

Description

This function exists to support use from other packages that wish to use dust's random number support, and creates an opaque pointer to a set of random number streams. It is described more fully in vignette("rng_package.Rmd")

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:

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

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

Usage:

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

Arguments:

`seed` The random number seed to use (see dust_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 dust_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:

```
dust_rng_pointer$sync()
```

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

Usage:

```
dust_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:

```
dust_rng_pointer$is_current()
```

Examples

```
dust::dust_rng_pointer$new()
```

Index

cpp11::cpp_register(), *27*

data.frame, *10*
dust, 2, *9*, *14*, *16*, *28*
dust(), *11*
dust_cuda_configuration, 7, *9*
dust_cuda_configuration(), *9*
dust_cuda_options, *3*, *8*, 9, *13*
dust_data, 10
dust_data(), *22*
dust_example, 11
dust_example(), *6*, *14*
dust_generate, *6*, 12
dust_generator, *2*, *4*, *6*, *10, 11*, 14, *35*
dust_ode_control, 23
dust_ode_control(), *16*
dust_openmp_support, 24, *29*
dust_openmp_support(), *26*
dust_openmp_threads, 25, *29*
dust_openmp_threads(), *22*, *25*
dust_package, 26
dust_package(), *12*
dust_repair_environment, 28
dust_rng, 28, *35, 36*
dust_rng_distributed_pointer
        *(*dust_rng_distributed_state*)*, 34
dust_rng_distributed_state, 34, *36*
dust_rng_pointer, 35, *35*

list, *10*

R6, *14*
readRDS, *28*
rhyper, *32*

saveRDS, *28*