Package ‘piecemeal’

March 11, 2026

Title Wrangle Large Simulation Studies

Date 2026-03-09

Version 0.2.0

Description An 'R6' class to set up, run, monitor, collate, and debug large simulation studies comprising many small independent replications and treatment configurations. Parallel processing, reproducibility, fault- and error-tolerance, and ability to resume an interrupted or timed-out simulation study are built in.

BugReports https://github.com/krivit/piecemeal/issues

License GPL (>= 3)

Encoding UTF-8

RoxygenNote 7.3.3.9000

Depends R (>= 4.2.0)

Imports R6, filelock, rlang, purrr, RSQLite, DBI, cli

Suggests knitr, testthat (>= 3.2.3), rmarkdown

VignetteBuilder knitr

Config/testthat/parallel false

Config/testthat/edition 3

NeedsCompilation no

Author Pavel N. Krivitsky [aut, cre] (ORCID: <https://orcid.org/0000-0002-9101-3362>)

Maintainer Pavel N. Krivitsky <pavel@statnet.org>

Repository CRAN

Date/Publication 2026-03-11 11:20:09 UTC

Contents

`piecemeal-package` *piecemeal: Wrangle Large Simulation Studies*

Description

An 'R6' class to set up, run, monitor, collate, and debug large simulation studies comprising many small independent replications and treatment configurations. Parallel processing, reproducibility, fault- and error-tolerance, and ability to resume an interrupted or timed-out simulation study are built in.

Details

This package grew out of a common problem of running a simulation with large numbers of treatment combinations and replications on a shared computing cluster. Using available tools such as **parallel** and even **foreach** can be frustrating for a number of reasons:

- If any of the function runs results in an error, all results are lost.
- Tracking down which configuration resulted in an error and reproducing it can be frustrating, and if the fix does not fix all the errors, one has to start over.
- If one underestimates the amount of time all the jobs will take, all results are lost.
- Conversely, if the cluster turns out to be freer than anticipated, it is desirable to queue another job to get things done twice as fast.

Those package with fault-tolerance capabilities typically focus on crashing worker nodes.

This can be worked around in a variety of ways. Functions can be wrapped in `try()`. Rather than returning the result to the manager process, the worker can save its results to a unique file, with the results collated at the end. `Piecemeal` automates this, and keeps careful track of inputs and random seeds, ensuring that problematic realisations can be located and debugged quickly and efficiently. A locking system even makes it possible to have multiple jobs running the same study on the same cluster without interfering with each other.

Author(s)

Maintainer: Pavel N. Krivitsky <`pavel@statnet.org`> (ORCID)

Authors:

- Pavel N. Krivitsky <`pavel@statnet.org`> (ORCID)

See Also

The `Piecemeal` `R6` class for details, the vignette (`vignette("piecemeal")`) for a worked example, and files in the '`examples/`' subdirectory of the package installation (likely '`/tmp/RtmpgG7G1t/Rinst11eb3455b1fdd4/pie`
on your system) for a typical setup on a cluster.

`init` *A convenience function for initialising `Piecemeal` objects.*

Description

There is rarely a reason to attach **piecemeal** via `library()`, so `piecemeal::init(outdir)` is provided as shorthand for `Piecemeal$new(outdir)`.

Usage

```
init(outdir)
```

Arguments

`outdir`	the directory to hold the partial simulation results.

Value

A `Piecemeal` object.

See Also

`Piecemeal`

Examples

```
outdir <- file.path(tempdir(), "piecemeal_demo")
sim <- piecemeal::init(outdir)
# a.k.a. piecemeal::Piecemeal$new(outdir)
# a.k.a.
# library(piecemeal)
# Piecemeal$new(outdir)
```

`Piecemeal` *The `Piecemeal` `R6` Class*

Description

This class exports methods for configuring a simulation, running it, debugging failed configurations, and resuming the simulation. See the vignette `vignette("piecemeal")` for long worked example.

Details

A chain of `R6` method calls is used to specify the setup and the worker functions, the treatment configurations to be passed to the worker, and parallelism and other simulation settings. Then, when `$run()` is called, the cluster is started, worker nodes are initialised, and every combination of random seed and treatment configuration is passed to `clusterApplyLB()` (if parallel processing is enabled).

On the worker nodes, the worker function is not called directly; rather, care is taken to make sure that the specified configuration and seed is not already being worked on. This makes it safe to, e.g., queue multiple jobs for the same simulation. If the configuration is available, `set.seed()` is called with the seed and then the worker function is run.

Errors in the worker function are caught and error messages saved and returned.

Methods

Public methods:

- `Piecemeal$new()`
- `Piecemeal$cluster()`
- `Piecemeal$export_vars()`
- `Piecemeal$setup()`
- `Piecemeal$worker()`
- `Piecemeal$treatments()`
- `Piecemeal$factorial()`
- `Piecemeal$nrep()`
- `Piecemeal$seeds()`
- `Piecemeal$run()`
- `Piecemeal$todo()`
- `Piecemeal$result_list()`
- `Piecemeal$result_df()`
- `Piecemeal$reset()`
- `Piecemeal$clean()`
- `Piecemeal$erred()`
- `Piecemeal$consolidate()`
- `Piecemeal$options()`
- `Piecemeal$print()`
- `Piecemeal$status()`
- `Piecemeal$eta()`
- `Piecemeal$clone()`

Method `new()`**:** Create a new `Piecemeal` instance.

Usage:

```
Piecemeal$new(outdir)
```

Arguments:

`outdir` the directory to hold the partial simulation results.

Method `cluster()`**:** Cluster settings for the piecemeal run.

Usage:

```
Piecemeal$cluster(...)
```

Arguments:

`...` either arguments to `makeCluster()` or a single argument containing either an existing cluster or `NULL` to disable clustering.

Method `export_vars()`**:** Specify variables to be copied from the manager node to the worker nodes' global environment. (See `parallel::clusterExport()`.)

Usage:

```
Piecemeal$export_vars(varlist, envir = parent.frame(), .add = TRUE)
```

Arguments:

`varlist` a character vector with variable names.

`envir` the environment on the manager node from which to take the variables; defaults to the current environment.

`.add` whether the new variables should be added to the current list (if `TRUE`, the default) or replace it (if `FALSE`).

Method `setup()`**:** Specify code to be run on each worker node at the start of the simulation; if running locally, it will be evaluated in the global environment.

Usage:

```
Piecemeal$setup(
  expr = {
 }
)
```

Arguments:

`expr` an expression; if passed, replaces the previous expression; if empty, resets it to nothing.

Method `worker()`**:** Specify the function to be run for each treatment configuration; it will be run in the global environment.

Usage:

```
Piecemeal$worker(fun)
```

Arguments:

`fun` a function whose arguments are specified by `$treatments()` and `$factorial()`; if it has `.seed` as a named argument, the seed will be passed as well.

Method `treatments()`**:** Specify a list of treatment configurations to be run.

Usage:

```
Piecemeal$treatments(l, .add = TRUE)
```

Arguments:

`l` a list, typically of lists of arguments to be passed to the function specified by `worker`; it is recommended that these be as compact as possible, since they are `serialize`d and sent to the worker node for every combination of treatment configuration and random seed.

`.add` whether the new treatment configurations should be added to the current list (if `TRUE`, the default) or replace it (if `FALSE`.

Method `factorial()`**:** Specify a list of treatment configurations to be run in a factorial design.

Usage:

```
Piecemeal$factorial(..., .filter = function(...) TRUE, .add = TRUE)
```

Arguments:

`...` vectors or lists whose Cartesian product will added to the treatment list; it is recommended that these be as compact as possible, since they are `serialize`d and sent to the worker node for every combination of treatment configuration and random seed.

`.filter` a function that takes the same arguments as worker and returns `FALSE` if the treatment configuration should be skipped; defaults to accepting all configurations.

`.add` whether the new treatment configurations should be added to the current list (if `TRUE`, the default) or replace it (if `FALSE`.

Method `nrep()`**:** Specify a number of replications for each treatment configuration (starts out at 1).

Usage:

```
Piecemeal$nrep(nrep)
```

Arguments:

`nrep` a positive integer giving the number of replications; the seeds will be set to `1:nrep`.

Method `seeds()`**:** Specify the seeds to be used for each replication of each treatment configuration.

Usage:

```
Piecemeal$seeds(seeds)
```

Arguments:

`seeds` an integer vector of seeds; its length will be used to infer the number of replications.

Method `run()`**:** Run the simulation.

Usage:

```
Piecemeal$run(shuffle = TRUE)
```

Arguments:

`shuffle` Should the treatment configurations be run in a random order (`TRUE`, the default) or in the order in which they were added (`FALSE`)?

Returns: Invisibly, a character vector with an element for each seed and treatment configuration combination attempted, indicating its file name and status, including errors.

Method `todo()`**:** List the configurations still to be run.

Usage:

```
Piecemeal$todo()
```

Returns: A list of lists with arguments to the worker functions and worker-specific configuration settings; also an attribute `"done"` giving the number of configurations skipped because they are already done.

Method `result_list()`**:** Scan through the results files and collate them into a list.

Usage:

```
Piecemeal$result_list(n = Inf, trt_tf = identity, out_tf = identity)
```

Arguments:

`n` maximum number of files to load; if less than the number of results, a systematic sample is taken.

`trt_tf`, `out_tf` functions that take the treatment configuration list and the output (if not an error) respectively, and transform them; this is helpful when, for example, the output is big and so loading all the files will run out of memory.

Returns: A list of lists containing the contents of the result files.

`treatment` arguments passed to the worker

`seed` the seed set just before calling the worker

`output` value returned by the worker, or a `try-error` returned by `try()`

`OK` whether the worker succeeded or produced an error

`config` miscellaneous configuration settings such as the file name

Method `result_df()`**:** Scan through the results files and collate them into a data frame.

Usage:

```
Piecemeal$result_df(trt_tf = identity, out_tf = identity, rds = FALSE, ...)
```

Arguments:

`trt_tf`, `out_tf` functions that take the treatment configuration list and the output respectively, and return named lists that become data frame columns; a special value `I` instead creates columns `treatment` and/or `output` with the respective lists copied as is.

`rds` whether to include an `.rds` column described below.

`...` additional arguments, passed to `Piecemeal$result_list()`.

Returns: A data frame with columns corresponding to the values returned by `trt_tf` and `out_tf`, with the following additional columns:

`.seed` the random seed used.

`.rds` the path to the RDS file (if requested).

Runs that erred are filtered out.

Method `reset()`**:** Clear the simulation results so far.

Usage:

```
Piecemeal$reset(confirm = interactive())
```

Arguments:

`confirm` whether the user should be prompted to confirm deletion.

Method `clean()`**:** Delete the result files for which the worker function produced an error and/or which were somehow corrupted, or based on some other predicate.

Usage:

```
Piecemeal$clean(which = function(res) !res$OK)
```

Arguments:

`which` a function of a result list (see `Piecemeal$result_list()`) returning `TRUE` if the result file is to be deleted and `FALSE` otherwise.

Method `erred()`**:** List the configurations for which the worker function failed.

Usage:

```
Piecemeal$erred()
```

Method `consolidate()`**:** Consolidate successful run result files into a SQLite database.

Usage:

```
Piecemeal$consolidate()
```

Details: This method consolidates individual RDS result files into a single database to reduce inode usage. Only successful runs (where `OK = TRUE`) are consolidated. This function is safe to run while simulations are running and to interrupt (using `CTRL-C` or analogous) and resume, but only one consolidation may be run at the same time. Consolidated and unconsolidated results can be accessed transparently.

Returns: Invisibly, the number of files consolidated.

Method `options()`**:** Set miscellaneous options.

Usage:

```
Piecemeal$options(split = c(1L, 1L), error = c("auto", "save", "skip", "stop"))
```

Arguments:

`split` a two-element vector indicating whether the output files should be split up into subdirectories and how deeply, the first for splitting configurations and the second for splitting seeds; this can improve performance on some file systems.

`error` how to handle worker errors:

- `"save"` save the seed, the configuration, and the status, preventing future runs until the file is removed using `Piecemeal$clean()`.
- `"skip"` return the error message as a part of `run()`'s return value, but do not save the RDS file; the next `run()` will attempt to run the worker for that configuration and seed again.
- `"stop"` allow the error to propagate; can be used in conjunction with `Piecemeal$cluster(NULL)` and (global) `options(error = recover)` to debug the worker.
- `"auto"` (default) as `"save"`, but if any of the methods that change how each configuration is run (i.e., `$worker()`, `$setup()`, and `$export_vars()`) is called, `$clean()` will be called automatically before the next `$run()`.

Method `print()`**:** Print the current simulation settings, including whether there is enough information to run it.

Usage:

```
Piecemeal$print(...)
```

Arguments:

`...` additional arguments, currently unused.

Method `status()`**:** Summarise the current status of the simulation, including the number of runs succeeded, the number of runs still to be done, the number of runs currently running, the errors encountered, and, if started, the estimated time to completion at the current rate.

Usage:

`Piecemeal$status(...)`

Arguments:

`...` additional arguments, currently passed to `Piecemeal$eta()`.

Method `eta()`**:** Estimate the rate at which runs are being completed and how much more time is needed.

Usage:

`Piecemeal$eta(window = 3600)`

Arguments:

`window` initial time window to use, either a `difftime` object or the number in seconds; defaults to 1 hour.

Details: The window used is actually between the last completed run and the earliest run in the `window` before that. This allows to take an interrupted simulation and estimate how much more time (at the most recent rate) is needed.

Returns: A list with elements `window`, `recent`, `cost`, `left`, `rate`, and `eta`, containing, respectively, the time window, the number of runs completed in this time, the average time per completion, the estimated time left (all in seconds), the corresponding rate (in Hertz), and the expected time of completion.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

`Piecemeal$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

Note

If no treatment is specified, the function is called with no arguments (or just `.seed`).

If `Piecemeal$options(error = "auto")` (the default) is set, changing some configuration settings, including the worker function, the setup code, and the exported variables, will automatically set a flag to run `clean()` before the next run.

The estimation method is a simple ratio, so it may be biased under some circumstances. Also, it does not check if the runs have been completed successfully.

Examples

```
# Initialise, with the output directory.
sim <- piecemeal::init(file.path(tempdir(), "piecemeal_demo"))
# Clear the previous simulation, if present.
sim$reset()

# Set up a simulation:
sim$
  # for every combination of x = 1, 2 and y = 1, 3, 9, 27,
  factorial(x = 2^(0:1), y = 3^(0:3))$
```

```
    # each replicated 3 times,
    nrep(3)$
    # first load library 'rlang',
    setup({library(rlang)})$
    # then for each x, y, and seed, evaluate
    worker(function(x, y) {
      p <- x*y
      u <- runif(1)
      dbl(p = p, u = u)
    })$
    # on a cluster with two nodes.
    cluster(2)

  # Summarise
  sim

  # Go!
  sim$run()

  # Get a table with the results.
  sim$result_df()

  # For a more involved version of this example, see vignette("piecemeal").
```

Index