Package: piecemeal (via r-universe)

October 23, 2026

Title Wrangle Large Simulation Studies

Date 2026-10-23

Version 0.3.0-209

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

Roxygen list(markdown = TRUE)

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

Config/roxygen2/version 8.1.1

Repository https://krivit.r-universe.dev

Date/Publication 2026-10-23 10:25:08 UTC

RemoteUrl https://github.com/krivit/piecemeal

RemoteRef HEAD

RemoteSha 36ecb2473afb94eb22f30c4e97c300048eb0da47

Contents

piecemeal-package 2
init 3
Piecemeal 3

Index **13**

`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 '`/github/workspace/pkglib/piecemeal/exam`
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 a long worked example. Examples of R scripts suitable for non-interactive use on a computing cluster can be found in your package installation's 'examples' directory, which can be located by running `system.file("examples", package = "piecemeal")`. (This appears to currently be '/github/workspace/pkglib/piecemeal/examples'.)

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 (if not already running), 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.

When running with parallel processing disabled, Piecemeal tries to mimic the environment of a cluster as closely as possible while still allowing interactive debugging via $test() and $debug(). Specifically:

- The worker function's own environment is ignored. That is, if the worker function is defined in an environment that contains a certain variable, the worker function will not be able to "see" it unless it was explicitly exported.
- The random number generator state (.Random.seed) is saved before the configuration's seed is set and run and restored after.
- However, the search() path (packages attached by library()) is global to an R session, so the mimicry is imperfect: any packages attached on the manager session will be visible to the worker function run locally but not to one run on a cluster node, unless a part of $setup().

Methods

Public methods:

- Piecemeal$new()
- Piecemeal$cluster()
- Piecemeal$export_vars()
- Piecemeal$setup()
- Piecemeal$worker()
- Piecemeal$treatments()
- Piecemeal$factorial()
- Piecemeal$nrep()
- Piecemeal$seeds()
- Piecemeal$test()
- Piecemeal$run()
- Piecemeal$autorun()
- Piecemeal$todo()
- Piecemeal$result_list()
- Piecemeal$result_df()
- Piecemeal$reset()
- Piecemeal$clean()

- Piecemeal$forget()
- Piecemeal$erred()
- Piecemeal$debug()
- Piecemeal$consolidate()
- Piecemeal$options()
- Piecemeal$print()
- Piecemeal$status()
- Piecemeal$eta()
- Piecemeal$last_OK()
- Piecemeal$last_consolidated()
- Piecemeal$clone()

`Piecemeal$new()`: Create a new `Piecemeal` instance.

Usage:

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

Arguments:

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

`Piecemeal$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 parallel computing.

`Piecemeal$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`).

`Piecemeal$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.

`Piecemeal$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.

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

`Piecemeal$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`).

`Piecemeal$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`.

`Piecemeal$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`.

`Piecemeal$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.

`Piecemeal$test()`**:** Test-run some treatment configurations on the local system and return their results without saving.

Usage:

```
Piecemeal$test(config = 1, shuffle = TRUE, error = getOption("error"))
```

Arguments:

`config, shuffle` which configurations to run; the configurations can be specified as follows:

- `config` a `list` of treatment configurations in the same format as that of `Piecemeal$todo()`. If only passing one configuration, remember to wrap it in `list()`.
- `config` a character vector of the form `c(treatment_hash, seed)` (or multiple such concatenated). (The seed will be converted back to an integer.)
- `config` a number and `shuffle == TRUE` (the default): run `config` configurations, chosen at random from those left to do.
- `config` a numeric vector and `shuffle == FALSE`: `config` is treated as a vector of indices from the list returned by `Piecemeal$todo()`.

`error` sets the `options()` `error=` option before calling the worker; special values `"debug"` or `"debugonce"` (with or without quotes) will instead debug the worker function from the start, i.e., as if `debugonce()` were called on it first. See the the vignette `vignette("piecemeal")` for an illustration.

Details: This function ignores cluster settings and bypasses the usual bookkeeping: the given configuration is run even if the result file already exists, and the results are returned and not saved. If the run results in an error, it is handled according to the `error` argument.

Returns: A list containing the results of the runs, with each sublist's element `$output` containing the value returned by the worker. They are not saved.

`Piecemeal$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.

`Piecemeal$autorun()`**:** Run the simulation if in a non-interactive session and at top level (that is, not `source()`d); otherwise print a message and do nothing.

Usage:

```
Piecemeal$autorun(..., call_depth = 0L)
```

Arguments:

`...` arguments passed to `Piecemeal$run()`.

`call_depth` how many call frames deep is the `autorun()` call allowed to be? Set to `+Inf` or a large number to disable the check.

Details: This method can be used in place of `Piecemeal$run()` to allow the same '`.R`' file to be run in a batch job to run the simulation or in an interactive session or from another script to facilitate monitoring, debugging, consolidation, and exporting results. By default, its behaviour is based on `base::interactive()` and `base::sys.nframe()`, but it can be overridden by setting `options(piecemeal.autorun = TRUE/FALSE)` to force it to always run (if `TRUE`) never run (if `FALSE`). Setting to `NA` or unsetting reverts to the default behaviour. See the package installation's '`examples`' directory for usage examples.

`Piecemeal$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 (particularly with elements `$seed` with the random seed and `$treatment` with the arguments to the worker); also an attribute `"done"` giving the number of configurations skipped because they are already done.

`Piecemeal$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

`Piecemeal$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.

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

Usage:

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

Arguments:

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

`Piecemeal$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` deprecated.

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

`Piecemeal$forget()`**:** Delete the result files *and* consolidated results that match a predicate or by hash and seed.

Usage:

```
Piecemeal$forget(.p, ...)
```

Arguments:

`.p, ...` a specification of which results to forget. The following formats are supported:

- **a function that returns a logical** `.p(treatment = treatment, seed = seed, ...)` will be called for each treatment-seed combination; results of configurations for which `TRUE` is returned will be forgotten.
- **a character vector of result file names** Used directly.
- **a character vector alternating hashes and seeds** These will be concatenated two at a time.
- **a character vector of hashes** All seeds for this hash.

Details: If a list of file names is provided, their directory names will be stripped and recomputed, so the corresponding runs will be found whether they are in the database or in the file system.

Returns: The total number of configurations actually deleted invisibly.

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

Usage:

```
Piecemeal$erred(n = Inf)
```

Arguments:

`n` return up to this many errors.

`Piecemeal$debug()`**:** Debug the worker for a particular configuration.

Usage:

```
Piecemeal$debug(result = 1, error = recover)
```

Arguments:

`result` either a list in the `Piecemeal$todo()` format or a number indexing the list returned by `Piecemeal$erred()`.

`error` sets the `options()` `error=` option before calling the worker; special values `"debug"` or `"debugonce"` (with or without quotes) will instead debug the worker function from the start, i.e., as if `debugonce()` were called on it first. See the the vignette `vignette("piecemeal")` for an illustration.

Returns: The result list, with element `$output` containing the value returned by the worker.

`Piecemeal$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.

`Piecemeal$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 by configuration and the second for splitting by seed; 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$clean()`.
- `"skip"` return the error message as a part of `Piecemeal$run()`'s return value, but do not save the RDS file; the next `Piecemeal$run()` will attempt to run the worker for that configuration and seed again.
- `"stop"` allow the error to propagate; can be used in conjunction with `Piecemeal$cluster(NULL)` and (global) `options(error = recover)` to debug the worker, though `Piecemeal$debug()` method is probably more convenient.
- `"auto"` (default) as `"save"`, but if the `Piecemeal` is reinitialised or any of the methods that change how each configuration is run (i.e., `Piecemeal$worker()`, `Piecemeal$setup()`, and `Piecemeal$export_vars()`) is called, `Piecemeal$clean()` will be called automatically at the start of the next `Piecemeal$run()`.

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

Usage:

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

Arguments:

`...` additional arguments, currently unused.

`Piecemeal$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()`.

Returns: An object of class `Piecemeal_status` containing a frequency `table` summarising simulation outcomes including frequencies of different error messages, attribute `"eta"` with a `Piecemeal_eta` as well as a number of other attributes useful for printing the simulation status.

`Piecemeal$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.
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.

Returns: An object of class `Piecemeal_eta` containing 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.

`Piecemeal$last_OK()`: Return the last time a run has finished successfully.

Usage:

```
Piecemeal$last_OK()
```

`Piecemeal$last_consolidated()`: Return the last time the consolidated database had been updated.

Usage:

```
Piecemeal$last_consolidated()
```

`Piecemeal$clone()`: The objects of this class are cloneable with this method.

Usage:

```
Piecemeal$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

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', once per node,
  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

.Random.seed, *4*

base::interactive(), *8*
base::sys.nframe(), *8*

clusterApplyLB(), *4*

debugonce(), *7*, *10*
difftime, *11*

init, 3

library(), *3*, *4*
list, *7*
list(), *7*

makeCluster(), *5*

options(), *7*, *10*

parallel::clusterExport(), *5*
Piecemeal, *2*, 3, *3*
piecemeal *(*piecemeal-package*)*, 2
piecemeal-package, 2

R6, *2*, *3*

search(), *4*
serialize, *6*
source(), *7*

table, *11*
try(), *2*, *8*