Package: SpaDES.experiment (via r-universe)

October 27, 2024

Type Package

Title Simulation Experiments Within The SpaDES Ecosystem

- **Description** Tools to do simulation experiments within the SpaDES ecosystem. This includes replication, parameter sweeps, scenario analysis, pattern oriented modeling, and simulation experiments. The package introduces a new object class, the simLists, which is an environment that contains many simList class objects. This package also includes tools to do post hoc analyses of such simLists objects.
- URL https://spades-experiment.predictiveecology.org/,
 - https://github.com/PredictiveEcology/SpaDES.experiment

Date 2023-08-15

Version 0.0.2.9005

Depends R (>= 4.1), reproducible (>= 0.2.11), SpaDES.core (>= 0.2.7)

Imports data.table (>= 1.10.4), DEoptim (>= 2.2-4), future, future.apply, googledrive, methods, parallel, purrr, raster (>= 2.5-8), utils

- Suggests covr, future.callr, ggplot2, igraph, knitr, NLMR (>= 1.1.1), quickPlot, RColorBrewer, rmarkdown, sf, SpaDES.tools (>= 0.2.0), testthat (>= 3.0.0)
- Remotes ropensci/NLMR, PredictiveEcology/quickPlot@development, cran/RandomFields, cran/RandomFieldsUtils, PredictiveEcology/reproducible@development, PredictiveEcology/SpaDES.core@development, PredictiveEcology/SpaDES.tools@development

Additional_repositories https://predictiveecology.r-universe.dev/

Encoding UTF-8

Language en-CA

License GPL-3

VignetteBuilder knitr, rmarkdown

BugReports https://github.com/PredictiveEcology/SpaDES.experiment/issues

ByteCompile yes

Collate 'POM.R' 'helpers.R' 'simLists-class.R' 'as.data.table.R' 'experiment.R' 'experiment2.R' 'simInitAndExperiment.R' 'spades-experiment-package.R'

RoxygenNote 7.2.3

Roxygen list(markdown = TRUE)

Config/testthat/edition 3

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

RemoteUrl https://github.com/PredictiveEcology/SpaDES.experiment

RemoteRef development

RemoteSha bc37fb350798b3e5624bfbfb30836d0d442debf8

Contents

2
3
6
14
18
18
25
25
28
20

Index

SpaDES.experiment-package

Categorized overview of the SpaDES.experiment package

Description



This package provides additional tools to do simulation experiments within the SpaDES ecosystem. This includes replication, parameter sweeps, scenario analysis, pattern oriented modeling, and simulation experiments. The package introduces a new object class, the simLists, which is an environment that contains many simList objects. This package also includes tools to do post hoc analyses of such simLists objects.

Bug reports: https://github.com/PredictiveEcology/SpaDES.experiment/issues

Module repository: https://github.com/PredictiveEcology/SpaDES-modules Wiki: https://github.com/PredictiveEcology/SpaDES/wiki

Author(s)

Maintainer: Eliot J B McIntire <eliot.mcintire@canada.ca>(ORCID)

Authors:

• Alex M Chubaty <achubaty@for-cast.ca>(ORCID)

Other contributors:

• Her Majesty the Queen in Right of Canada, as represented by the Minister of Natural Resources Canada [copyright holder]

See Also

Useful links:

- https://spades-experiment.predictiveecology.org/
- https://github.com/PredictiveEcology/SpaDES.experiment
- Report bugs at https://github.com/PredictiveEcology/SpaDES.experiment/issues

as.data.table.simLists

Coerce elements of a simLists object to a data.table

Description

This is particularly useful to build plots using the tidyverse, e.g., ggplot2.

Usage

```
## S3 method for class 'simLists'
as.data.table(x, vals, objectsFromSim = NULL, objectsFromOutputs = NULL, ...)
```

Arguments

Х	An R object.	
vals	A (named) list of object names to extract from each simList, or a named list of quoted expressions to calculate for each simList, or a mix of character an quoted expressions.	
objectsFromSim	Character vector of objects to extract from the simLists. If omitted, it will all objects from each simList in order to calculate the vals. This may computational cost. If NA, then no objects will be accessed from the sim Objects identified here will only be as they are in the simList, i.e., at end	

objectsFromOutputs

List of (named) character vectors of objects to load from the outputs(sim) prior to evaluating vals. If there already is an object with that same name in the simList, then it will be overwritten with the object loaded from outputs(sim). If there are many objects with the same name, specifically from several saveTime values in the outputs(sim), these will all be loaded, one at a time, vals evaluated one at a time, and each of the values will be returned from each saveTime. A column, saveTime, will be part of the returned data.table. For cases where more than one object is required at a given saveTime, all should be identified here, without time specified. This function will take all identified objects from the same time period.

... Additional arguments. Currently unused.

Details

See examples.

Value

This returns a data.table class object with

Examples

```
## Not run:
 if (require("ggplot2", quietly = TRUE) &&
     require("NLMR", quietly = TRUE) &&
     require("RColorBrewer", quietly = TRUE)) {
   library(SpaDES.core)
   library(SpaDES.experiment)
   tmpdir <- file.path(tempdir(), "examples")</pre>
   # Make 3 simLists -- set up scenarios
   endTime <- 2
   # Example of changing parameter values
   # Make 3 simLists with some differences between them
   mySim <- lapply(c(10, 20, 30), function(nFires) {</pre>
     simInit(
        times = list(start = 0.0, end = endTime, timeunit = "year"),
       params = list(
          .globals = list(stackName = "landscape", burnStats = "nPixelsBurned"),
          # Turn off interactive plotting
         fireSpread = list(.plotInitialTime = NA, spreadprob = c(0.2), nFires = c(10)),
         caribouMovement = list(.plotInitialTime = NA),
         randomLandscapes = list(.plotInitialTime = NA, .useCache = "init")
       ),
       modules = list("randomLandscapes", "fireSpread", "caribouMovement"),
       paths = list(modulePath = system.file("sampleModules", package = "SpaDES.core"),
                     outputPath = tmpdir),
       # Save final state of landscape and caribou
       outputs = data.frame(
```

```
objectName = c(rep("landscape", endTime), "caribou", "caribou"),
       saveTimes = c(seq_len(endTime), unique(c(ceiling(endTime / 2), endTime))),
       stringsAsFactors = FALSE
     )
  )
 })
 planTypes <- c("sequential") # try others! ?future::plan</pre>
 sims <- experiment2(sim1 = mySim[[1]], sim2 = mySim[[2]], sim3 = mySim[[3]],</pre>
                      replicates = 3)
 # Try pulling out values from simulation experiments
 # 2 variables
df1 <- as.data.table(sims, vals = c("nPixelsBurned", NCaribou = quote(length(caribou$x1))))
 # Now use objects that were saved to disk at different times during spades call
 df1 <- as.data.table(sims,</pre>
                      vals = c("nPixelsBurned", NCaribou = quote(length(caribou$x1))),
                   objectsFromOutputs = list(nPixelsBurned = NA, NCaribou = "caribou"))
 # now calculate 4 different values, some from data saved at different times
 # Define new function -- this calculates perimeter to area ratio
 fn <- quote({</pre>
   landscape$Fires[] == 0] <- NA;</pre>
   a <- boundaries(landscape$Fires, type = "inner");</pre>
 a[landscape$Fires[] > 0 & a[] == 1] <- landscape$Fires[landscape$Fires[] > 0 & a[] == 1];
   peri <- table(a[]);</pre>
   area <- table(landscape$Fires[]);</pre>
   keep <- match(names(area),names(peri));</pre>
   mean(peri[keep]/area)
 })
 df1 <- as.data.table(sims,</pre>
                       vals = c("nPixelsBurned",
                                perimToArea = fn,
                             meanFireSize = quote(mean(table(landscape$Fires[])[-1])),
                                caribouPerHaFire = quote({
                                  NROW(caribou) /
                                    mean(table(landscape$Fires[])[-1])
                                })).
                       objectsFromOutputs = list(NA, c("landscape"), c("landscape"),
                                                  c("landscape", "caribou")),
                       objectsFromSim = "nPixelsBurned")
 if (interactive()) {
   # with an unevaluated string
   library(ggplot2)
   p <- lapply(unique(df1$vals), function(var) {</pre>
     ggplot(df1[vals == var,],
            aes(x = saveTime, y = value, group = simList, color = simList)) +
       stat_summary(geom = "point", fun.y = mean) +
       stat_summary(geom = "line", fun.y = mean) +
```

```
stat_summary(geom = "errorbar", fun.data = mean_se, width = 0.2) +
         ylab(var)
     })
     # Arrange all 4 -- could use gridExtra::grid.arrange -- easier
      pushViewport(viewport(layout = grid.layout(2, 2)))
      vplayout <- function(x, y) viewport(layout.pos.row = x, layout.pos.col = y)</pre>
     print(p[[1]], vp = vplayout(1, 1))
     print(p[[2]], vp = vplayout(1, 2))
     print(p[[3]], vp = vplayout(2, 1))
     print(p[[4]], vp = vplayout(2, 2))
   }
 }
## End(Not run)
```

Run an experiment using SpaDES.core::spades()

Description

This is essentially a wrapper around the spades call that allows for multiple calls to spades. This function will use a single processor, or multiple processors if raster::beginCluster() has been run first or a cluster object is passed in the cl argument (gives more control to user).

Usage

```
experiment(
  sim,
  replicates = 1,
 params,
 modules,
 objects = list(),
  inputs,
  dirPrefix = "simNum",
  substrLength = 3,
  saveExperiment = TRUE,
  experimentFile = "experiment.RData",
  clearSimEnv = FALSE,
  notOlderThan,
  cl,
  . . .
)
## S4 method for signature 'simList'
experiment(
  sim,
  replicates = 1,
```

```
params,
modules,
objects = list(),
inputs,
dirPrefix = "simNum",
substrLength = 3,
saveExperiment = TRUE,
experimentFile = "experiment.RData",
clearSimEnv = FALSE,
notOlderThan,
cl,
....
```

Arguments

sim	A simList simulation object, generally produced by simInit.
replicates	The number of replicates to run of the same simList. See details and examples.
params	Like for simInit(), but for each parameter, provide a list of alternative values. See details and examples.
modules	Like for simInit(), but a list of module names (as strings). See details and examples.
objects	Like for simInit(), but a list of named lists of named objects. See details and examples.
inputs	Like for simInit(), but a list of inputs data.frames. See details and examples.
dirPrefix	String vector. This will be concatenated as a prefix on the directory names. See details and examples.
substrLength	Numeric. While making outputPath for each spades call, this is the number of characters kept from each factor level. See details and examples.
saveExperiment	Logical. Should params, modules, inputs, sim, and resulting experimental de- sign be saved to a file. If TRUE are saved to a single list called experiment. Default TRUE.
experimentFile	String. Filename if saveExperiment is TRUE; saved to outputPath(sim) in .RData format. See Details.
clearSimEnv	Logical. If TRUE, then the envir(sim) of each simList in the return list is emp- tied. This is to reduce RAM load of large return object. Default FALSE.
notOlderThan	Date or time. Passed to reproducible::Cache to update the cache. Default is NULL, meaning don't update the cache. If Sys.time() is provided, then it will force a recache, i.e., remove old value and replace with new value. Ignored if cache is FALSE.
cl	A cluster object. Optional. This would generally be created using parallel::makeCluster or equivalent. This is an alternative way, instead of beginCluster(), to use parallelism for this function, allowing for more control over cluster use.
	Passed to spades. Specifically, debug, .plotInitialTime, .saveInitialTime, cache and/or notOlderThan. Caching is still experimental. It is tested to work under some conditions, but not all. See details.

Details

Generally, there are 2 reasons to do this: replication and varying simulation inputs to accomplish some sort of simulation experiment. This function deals with both of these cases. In the case of varying inputs, this function will attempt to create a fully factorial experiment among all levels of the variables passed into the function. If all combinations do not make sense, e.g., if parameters and modules are varied, and some of the parameters don't exist in all combinations of modules, then the function will do an "all meaningful combinations" factorial experiment. Likewise, fully factorial combinations of parameters and inputs may not be the desired behaviour. The function requires a simList object, acting as the basis for the experiment, plus optional inputs and/or objects and/or params and/or modules and/or replications.

This function requires a complete simList: this simList will form the basis of the modifications as passed by params, modules, inputs, and objects. All params, modules, inputs or objects passed into this function will override the corresponding params, modules, inputs, or identically named objects that are in the sim argument.

This function is parallel aware, using the same mechanism as used in the raster package. Specifically, if you start a cluster using beginCluster(), then this experiment function will automatically use that cluster. It is always a good idea to stop the cluster when finished, using endCluster().

Here are generic examples of how params, modules, objects, and inputs should be structured.

params = list(moduleName = list(paramName = list(val1, val2))).

modules = list(c("module1", "module2"), c("module1", "module3"))

objects = list(objName = list(object1=object1, object2=object2))

inputs = list(data.frame(file = pathToFile1, loadTime = 0, objectName = "landscape", stringsAsFactors = FALSE), data.frame(file = pathToFile2, loadTime = 0, objectName = "landscape", stringsAsFactors = FALSE))

Output directories are changed using this function: this is one of the dominant side effects of this function. If there are only replications, then a set of subdirectories will be created, one for each replicate. If there are varying parameters and or modules, outputPath is updated to include a subdirectory for each level of the experiment. These are not nested, i.e., even if there are nested factors, all subdirectories due to the experimental setup will be at the same level. Replicates will be one level below this. The subdirectory names will include the module(s), parameter names, the parameter values, and input index number (i.e., which row of the inputs data.frame). The default rule for naming is a concatenation of:

- 1. The experiment level (arbitrarily starting at 1). This is padded with zeros if there are many experiment levels.
- 2. The module, parameter name and parameter experiment level (not the parameter value, as values could be complex), for each parameter that is varying.
- 3. The module set.
- 4. The input index number
- 5. Individual identifiers are separated by a dash.
- 6. Module Parameter Parameter index triplets are separated by underscore.

e.g., a folder called: 01-fir_spr_1-car_N_1-inp_1 would be the first experiment level (01), the first parameter value for the spr* parameter of the fir* module, the first parameter value of the N parameter of the car* module, and the first input dataset provided.

This subdirectory name could be long if there are many dimensions to the experiment. The parameter substrLength determines the level of truncation of the parameter, module and input names for these subdirectories. For example, the resulting directory name for changes to the spreadprob parameter in the fireSpread module and the N parameter in the caribouMovement module would be: 1_fir_spr_1-car_N_1 if substrLength is 3, the default.

Replication is treated slightly differently. outputPath is always 1 level below the experiment level for a replicate. If the call to experiment is not a factorial experiment (i.e., it is just replication), then the default is to put the replicate subdirectories at the top level of outputPath. To force this one level down, dirPrefix can be used or a manual change to outputPath before the call to experiment.

dirPrefix can be used to give custom names to directories for outputs. There is a special value, "simNum", that is used as default, which is an arbitrary number associated with the experiment. This corresponds to the row number in the attr(sims, "experiment"). This "simNum" can be used with other strings, such as dirPrefix = c("expt", "simNum").

The experiment structure is kept in two places: the return object has an attribute, and a file named experiment.RData (see argument experimentFile) located in outputPath(sim).

substrLength, if 0, will eliminate the subdirectory naming convention and use only dirPrefix.

If cache = TRUE is passed, then this will pass this to spades, with the additional argument replicate = x, where x is the replicate number. That means that if a user runs experiment with replicate = 4 and cache = TRUE, then SpaDES will run 4 replicates, caching the results, including replicate = 1, replicate = 2, replicate = 3, and replicate = 4. Thus, if a second call to experiment with the exact same simList is passed, and replicates = 6, the first 4 will be taken from the cached copies, and replicate 5 and 6 will be run (and cached) as normal. If notOlderThan used with a time that is more recent than the cached copy, then a new spades will be done, and the cached copy will be deleted from the cache repository, so there will only ever be one copy of a particular replicate for a particular simList. NOTE: caching may not work as desired on a Windows machine because the sqlite database can only be written to one at a time, so there may be collisions.

Value

Invisibly returns a list of the resulting simList objects from the fully factorial experiment. This list has an attribute, which a list with 2 elements: the experimental design provided in a wide data.frame and the experiment values in a long data.frame. There is also a file saved with these two data.frames. It is named whatever is passed into experimentFile. Since returned list of simList objects may be large, the user is not obliged to return this object (as it is returned invisibly). Clearly, there may be objects saved during simulations. This would be determined as per a normal SpaDES.core::spades() call, using outputs like, say, outputs(sims[[1]]).

Author(s)

Eliot McIntire

See Also

simInit()

Examples

```
if (interactive() &&
   require("igraph", quietly = TRUE) &&
   require("NLMR", quietly = TRUE) &&
   require("RColorBrewer", quietly = TRUE)) {
 library(raster)
 tmpdir <- file.path(tempdir(), "examples")</pre>
 # Create a default simList object for use through these examples
 mySim <- simInit(</pre>
   times = list(start = 0.0, end = 2.0, timeunit = "year"),
   params = list(
      .globals = list(stackName = "landscape", burnStats = "nPixelsBurned"),
      # Turn off interactive plotting
     fireSpread = list(.plotInitialTime = NA),
     caribouMovement = list(.plotInitialTime = NA),
     randomLandscapes = list(.plotInitialTime = NA)
   ),
   modules = list("randomLandscapes", "fireSpread", "caribouMovement"),
   paths = list(modulePath = system.file("sampleModules", package = "SpaDES.core"),
                 outputPath = tmpdir),
   # Save final state of landscape and caribou
   outputs = data.frame(objectName = c("landscape", "caribou"), stringsAsFactors = FALSE)
 )
 # Example 1 - test alternative parameter values
 # Create an experiment - here, 2 \times 2 \times 2 (2 levels of 2 params in fireSpread,
      and 2 levels of 1 param in caribouMovement)
 # Here is a list of alternative values for each parameter. They are length one
 # numerics here -- e.g., list(0.2, 0.23) for spreadprob in fireSpread module,
     but they can be anything, as long as it is a list.
 #
 experimentParams <- list(fireSpread = list(spreadprob = list(0.2, 0.23),</pre>
                                              nfires = list(20, 10)),
                           caribouMovement = list(N = list(100, 1000)))
 sims <- experiment(mySim, params = experimentParams)</pre>
 # see experiment:
 attr(sims, "experiment")
 # Read in outputs from sims object
 fireMaps <- do.call(stack, lapply(1:NROW(attr(sims, "experiment")$expDesign),</pre>
                                    function(x) sims[[x]]$landscape$Fires))
 if (interactive()) {
   library(quickPlot)
   Plot(fireMaps, new = TRUE)
 }
```

Or reload objects from files, useful if sim objects too large to store in RAM

```
caribouMaps <- lapply(sims, function(sim) {</pre>
  caribou <- readRDS(outputs(sim)$file[outputs(sim)$objectName == "caribou"])</pre>
})
names(caribouMaps) <- paste0("caribou", 1:8)</pre>
# Plot whole named list
if (interactive()) Plot(caribouMaps, size = 0.1)
# Example 2 - test alternative modules
# Example of changing modules, i.e., caribou with and without fires
# Create an experiment - here, 2 x 2 x 2 (2 levels of 2 params in fireSpread,
     and 2 levels of 1 param in caribouMovement)
experimentModules <- list(</pre>
  c("randomLandscapes", "fireSpread", "caribouMovement"),
  c("randomLandscapes", "caribouMovement"))
sims <- experiment(mySim, modules = experimentModules)</pre>
attr(sims, "experiment")$expVals # shows 2 alternative experiment levels
# Example 3 - test alternative parameter values and modules
# Note, this isn't fully factorial because all parameters are not
# defined inside smaller module list
sims <- experiment(mySim, modules = experimentModules, params = experimentParams)</pre>
attr(sims, "experiment") $ expVals # shows 10 alternative experiment levels
# Example 4 - manipulate manipulate directory names -
# "simNum" is special value for dirPrefix, it is converted to 1, 2, ...
sims <- experiment(mySim, params = experimentParams, dirPrefix = c("expt", "simNum"))</pre>
attr(sims, "experiment")$expVals # shows 8 alternative experiment levels, 24 unique
                                  # parameter values
# Example 5 - doing replicate runs -
sims <- experiment(mySim, replicates = 2)</pre>
attr(sims, "experiment")$expDesign # shows 2 replicates of same experiment
# Example 6 - doing replicate runs, but within a sub-directory
sims <- experiment(mySim, replicates = 2, dirPrefix = c("expt"))</pre>
lapply(sims, outputPath) # shows 2 replicates of same experiment, within a sub directory
# Example 7 - doing replicate runs, of a complex, non factorial experiment.
# Here we do replication, parameter variation, and module variation all together.
# This creates 20 combinations.
# The experiment function tries to make fully factorial, but won't
# if all the levels don't make sense. Here, changing parameter values
# in the fireSpread module won't affect the simulation when the fireSpread
# module is not loaded:
# library(raster)
# beginCluster(20) # if you have multiple clusters available, use them here to save time
sims <- experiment(mySim, replicates = 2, params = experimentParams,</pre>
                   modules = experimentModules,
                   dirPrefix = c("expt", "simNum"))
# endCluster() # end the clusters
attr(sims, "experiment")
```

```
# Example 8 - Use replication to build a probability map.
# For this to be meaningful, we need to provide a fixed input landscape,
# not a randomLandscape for each experiment level. So requires 2 steps.
# Step 1 - run randomLandscapes module twice to get 2 randomly
# generated landscape maps. We will use 1 right away, and we will
# use the two further below
mySimRL <- simInit(</pre>
  times = list(start = 0.0, end = 0.1, timeunit = "year"),
  params = list(
    .globals = list(stackName = "landscape"),
    # Turn off interactive plotting
    randomLandscapes = list(.plotInitialTime = NA)
  ),
  modules = list("randomLandscapes"),
  paths = list(modulePath = system.file("sampleModules", package = "SpaDES.core"),
               outputPath = file.path(tmpdir, "landscapeMaps1")),
 outputs = data.frame(objectName = "landscape", saveTime = 0, stringsAsFactors = FALSE)
)
# Run it twice to get two copies of the randomly generated landscape
mySimRLOut <- experiment(mySimRL, replicate = 2)</pre>
# extract one of the random landscapes, which will be passed into next as an object
landscape <- mySimRLOut[[1]]$landscape</pre>
# here we don't run the randomLandscapes module; instead we pass in a landscape
# as an object, i.e., a fixed input
mySimNoRL <- simInit(</pre>
  times = list(start = 0.0, end = 1, timeunit = "year"), # only 1 year to save time
  params = list(
    .globals = list(stackName = "landscape", burnStats = "nPixelsBurned"),
    # Turn off interactive plotting
    fireSpread = list(.plotInitialTime = NA),
    caribouMovement = list(.plotInitialTime = NA)
  ),
  modules = list("fireSpread", "caribouMovement"), # No randomLandscapes modules
  paths = list(modulePath = system.file("sampleModules", package = "SpaDES.core"),
               outputPath = tmpdir),
  objects = c("landscape"), # Pass in the object here
 # Save final state (the default if saveTime is not specified) of landscape and caribou
 outputs = data.frame(objectName = c("landscape", "caribou"), stringsAsFactors = FALSE)
)
# Put outputs into a specific folder to keep them easy to find
outputPath(mySimNoRL) <- file.path(tmpdir, "example8")</pre>
sims <- experiment(mySimNoRL, replicates = 8) # Run experiment</pre>
attr(sims, "experiment") # shows the experiment, which in this case is just replicates
# list all files that were saved called 'landscape'
landscapeFiles <- dir(outputPath(mySimNoRL), recursive = TRUE, pattern = "landscape",</pre>
                      full.names = TRUE)
# Can read in fires layers from disk since they were saved, or from the sims
# object
```

```
# fires <- lapply(sims, function(x) x$landscape$fires) |> stack()
fires <- lapply(landscapeFiles, function(x) readRDS(x)$fires) |> stack()
fires[fires > 0] <- 1 \# convert to 1s and 0s
fireProb <- sum(fires) / nlayers(fires) # sum them and convert to probability
if (interactive()) Plot(fireProb, new = TRUE)
# Example 9 - Pass in inputs, i.e., input data objects taken from disk
# Here, we, again, don't provide randomLandscapes module, so we need to
# provide an input stack called lanscape. We point to the 2 that we have
# saved to disk in Example 8
mySimInputs <- simInit(</pre>
  times = list(start = 0.0, end = 2.0, timeunit = "year"),
  params = list(
    .globals = list(stackName = "landscape", burnStats = "nPixelsBurned"),
    # Turn off interactive plotting
    fireSpread = list(.plotInitialTime = NA),
    caribouMovement = list(.plotInitialTime = NA)
  ),
  modules = list("fireSpread", "caribouMovement"),
  paths = list(modulePath = system.file("sampleModules", package = "SpaDES.core"),
               outputPath = tmpdir),
  # Save final state of landscape and caribou
 outputs = data.frame(objectName = c("landscape", "caribou"), stringsAsFactors = FALSE)
)
landscapeFiles <- dir(tmpdir, pattern = "landscape_year0", recursive = TRUE, full.names = TRUE)
# Varying inputs files - This could be combined with params, modules, replicates also
outputPath(mySimInputs) <- file.path(tmpdir, "example9")</pre>
sims <- experiment(mySimInputs,</pre>
                   inputs = lapply(landscapeFiles, function(filenames) {
                     data.frame(file = filenames, loadTime = 0,
                                 objectName = "landscape",
                                 stringsAsFactors = FALSE)
                   })
 )
# load in experimental design object
experiment <- load(file = file.path(tmpdir, "example9", "experiment.RData")) |>
  get()
print(experiment) # shows input files and details
# Example 10 - Use a very simple output dir name using substrLength = 0,
# i.e., just the simNum is used for outputPath of each spades call
outputPath(mySim) <- file.path(tmpdir, "example10")</pre>
sims <- experiment(mySim, modules = experimentModules, replicates = 2,</pre>
                   substrLength = 0)
lapply(sims, outputPath) # shows that the path is just the simNum
experiment <- load(file = file.path(tmpdir, "example10", "experiment.RData")) |>
  get()
print(experiment) # shows input files and details
# Example 11 - use clearSimEnv = TRUE to remove objects from simList
# This will shrink size of return object, which may be useful because the
```

```
# return from experiment function may be a large object (it is a list of
 # simLists). To see size of a simList, you have to look at the objects
 # contained in the envir(simList). These can be obtained via objs(sim)
 sapply(sims, function(x) object.size(objs(x))) |> sum() + object.size(sims)
 # around 3 MB
 # rerun with clearSimEnv = TRUE
 sims <- experiment(mySim, modules = experimentModules, replicates = 2,</pre>
                     substrLength = 0, clearSimEnv = TRUE)
 sapply(sims, function(x) object.size(objs(x))) |> sum() + object.size(sims)
 \# around 250 kB, i.e., all the simList contents except the objects.
 # Example 12 - pass in objects
 experimentObj <- list(landscape = lapply(landscapeFiles, readRDS) |>
                                    setNames(paste0("landscape", 1:2)))
 # Pass in this list of landscape objects
 sims <- experiment(mySimNoRL, objects = experimentObj)</pre>
 # Remove all temp files
 unlink(tmpdir, recursive = TRUE)
}
```

```
experiment2
```

Run experiment, algorithm 2, using SpaDES.core::spades()

Description

Given one or more simList objects, run a series of spades calls in a structured, organized way. Methods are available to deal with outputs, such as as.data.table.simLists which can pull out simple to complex values from every resulting simList or object saved by outputs in every simList run. This uses future internally, allowing for various backends and parallelism.0

Usage

```
experiment2(
    ...,
    replicates = 1,
    clearSimEnv = FALSE,
    createUniquePaths = c("outputPath"),
    useCache = FALSE,
    debug = getOption("spades.debug"),
    drive_auth_account,
    meanStaggerIntervalInSecs
)
## S4 method for signature 'simList'
experiment2(
    ...,
    replicates = 1,
    clearSimEnv = FALSE,
```

```
createUniquePaths = c("outputPath"),
useCache = FALSE,
debug = getOption("spades.debug"),
drive_auth_account = NULL,
meanStaggerIntervalInSecs = 1
)
```

Arguments

	One or more simList objects
replicates	The number of replicates to run of the same simList. See details and examples. To minimize memory overhead, currently, this must be length 1, i.e., all simList objects will receive the same number of replicates.
clearSimEnv	Logical. If TRUE, then the envir(sim) of each simList in the return list is emp- tied. This is to reduce RAM load of large return object. Default FALSE.
createUniquePat	hs
	A character vector of the paths passed to simInit, indicating which should cre- ate a new, unique path, as a sub-path to the original paths of simInit. Default, and currently only option, is "outputPath"
useCache	Logical. Passed to spades. This will be passed with the simList name and replicate number, allowing each replicate and each simList to be seen as a non-cached call to spades. This will, however, may prevent spades calls from running a second time during second call to the same experiment2 function.
debug	Optional tools for invoking debugging. Supplying a list will invoke the more powerful logging package. See details. Default is to use the value in getOption("spades.debug").
drive_auth_acco	punt
	Optional character string. If provided, it will be passed to each worker and run as googledrive::drive_auth(drive_auth_account) to allow a specific user account for googledrive
meanStaggerInte	ervalInSecs
	If used, this will use Sys.sleep(cumsum(c(0, rnorm(nbrOfWorkers() - 1, mean = meanStaggerIntersection))) and distribute these delays to the workers.

Details

This function, because of its class formalism, allows for methods to be used. For example, as.data.table.simLists() allows user to pull out specific objects (in the simList objects or on disk saved in outputPath(sim)).

The outputPath is changed so that every simulation puts outputs in a sub-directory of the original outputPath of each simList.

Value

Invisibly returns a simLists object. This class extends the environment class and contains simList objects.

A simLists object can be made manually, if, say, many manual spades calls have already been run. See example, via new("simLists")

Author(s)

Eliot McIntire

See Also

```
as.data.table.simLists(), simInit(), SpaDES.core::spades(), experiment()
```

Examples

```
## Not run:
 if (require("ggplot2", quietly = TRUE) &&
     require("NLMR", quietly = TRUE) &&
      require("RColorBrewer", quietly = TRUE)) {
   library(SpaDES.core)
   library(SpaDES.experiment)
   tmpdir <- file.path(tempdir(), "examples")</pre>
   # Make 3 simLists -- set up scenarios
   endTime <- 2
   # Example of changing parameter values
   # Make 3 simLists with some differences between them
   mySim <- lapply(c(10, 20, 30), function(nFires) {</pre>
     simInit(
       times = list(start = 0.0, end = endTime, timeunit = "year"),
       params = list(
          .globals = list(stackName = "landscape", burnStats = "nPixelsBurned"),
          # Turn off interactive plotting
         fireSpread = list(.plotInitialTime = NA, spreadprob = c(0.2), nFires = c(10)),
          caribouMovement = list(.plotInitialTime = NA),
         randomLandscapes = list(.plotInitialTime = NA, .useCache = "init")
       ),
       modules = list("randomLandscapes", "fireSpread", "caribouMovement"),
       paths = list(modulePath = system.file("sampleModules", package = "SpaDES.core"),
                     outputPath = tmpdir),
        # Save final state of landscape and caribou
       outputs = data.frame(
          objectName = c(rep("landscape", endTime), "caribou", "caribou"),
          saveTimes = c(seq_len(endTime), unique(c(ceiling(endTime / 2), endTime))),
          stringsAsFactors = FALSE
        )
     )
   })
   planTypes <- c("sequential") # try others! ?future::plan</pre>
   sims <- experiment2(sim1 = mySim[[1]], sim2 = mySim[[2]], sim3 = mySim[[3]],</pre>
                        replicates = 3)
```

```
# Try pulling out values from simulation experiments
 # 2 variables
df1 <- as.data.table(sims, vals = c("nPixelsBurned", NCaribou = quote(length(caribou$x1))))</pre>
 # Now use objects that were saved to disk at different times during spades call
 df1 <- as.data.table(sims,</pre>
                      vals = c("nPixelsBurned", NCaribou = quote(length(caribou$x1))),
                   objectsFromOutputs = list(nPixelsBurned = NA, NCaribou = "caribou"))
 # now calculate 4 different values, some from data saved at different times
 # Define new function -- this calculates perimeter to area ratio
 fn <- quote({</pre>
   landscape$Fires[] == 0] <- NA;</pre>
   a <- boundaries(landscape$Fires, type = "inner");</pre>
 a[landscape$Fires[] > 0 & a[] == 1] <- landscape$Fires[landscape$Fires[] > 0 & a[] == 1];
   peri <- table(a[]);</pre>
   area <- table(landscape$Fires[]);</pre>
   keep <- match(names(area),names(peri));</pre>
   mean(peri[keep]/area)
 })
 df1 <- as.data.table(sims,</pre>
                       vals = c("nPixelsBurned",
                                perimToArea = fn,
                             meanFireSize = quote(mean(table(landscape$Fires[])[-1])),
                                caribouPerHaFire = quote({
                                  NROW(caribou) /
                                    mean(table(landscape$Fires[])[-1])
                                })),
                       objectsFromOutputs = list(NA, c("landscape"), c("landscape"),
                                                  c("landscape", "caribou")),
                       objectsFromSim = "nPixelsBurned")
 if (interactive()) {
   # with an unevaluated string
   library(ggplot2)
   p <- lapply(unique(df1$vals), function(var) {</pre>
     ggplot(df1[vals == var,],
            aes(x = saveTime, y = value, group = simList, color = simList)) +
       stat_summary(geom = "point", fun.y = mean) +
       stat_summary(geom = "line", fun.y = mean) +
       stat_summary(geom = "errorbar", fun.data = mean_se, width = 0.2) +
       ylab(var)
   })
   # Arrange all 4 -- could use gridExtra::grid.arrange -- easier
   pushViewport(viewport(layout = grid.layout(2, 2)))
   vplayout <- function(x, y) viewport(layout.pos.row = x, layout.pos.col = y)</pre>
   print(p[[1]], vp = vplayout(1, 1))
   print(p[[2]], vp = vplayout(1, 2))
   print(p[[3]], vp = vplayout(2, 1))
```

```
print(p[[4]], vp = vplayout(2, 2))
}
## End(Not run)
```

initialize, simLists-method

Generate a simLists object

Description

Given the name or the definition of a class, plus optionally data to be included in the object, new returns an object from that class.

Usage

```
## S4 method for signature 'simLists'
initialize(.Object, ...)
```

Arguments

.Object	A simList object.
	Optional Values passed to any or all slot

POM

Use Pattern Oriented Modeling to fit unknown parameters

Description

This is very much in alpha condition. It has been tested on simple problems, as shown in the examples, with up to 2 parameters. It appears that DEoptim is the superior package for the stochastic problems. This should be used with caution as with all optimization routines. This function can nevertheless take optim as optimizer, using stats::optim. However, these latter approaches do not seem appropriate for stochastic problems, and have not been widely tested and are not supported within POM.

Usage

```
POM(
   sim,
   params,
   objects = NULL,
   objFn,
   cl,
   optimizer = "DEoptim",
```

POM

```
sterr = FALSE,
  ...,
  objFnCompare = "MAD",
  optimControl = NULL,
 NaNRetries = NA,
 logObjFnVals = FALSE,
 weights,
 useLog = FALSE
)
## S4 method for signature 'simList,character'
POM(
  sim,
 params,
 objects = NULL,
 objFn,
  cl,
  optimizer = "DEoptim",
  sterr = FALSE,
  ...,
 objFnCompare = "MAD",
  optimControl = NULL,
 NaNRetries = NA,
  logObjFnVals = FALSE,
 weights,
 useLog = FALSE
```

```
)
```

Arguments

sim	A simList simulation object, generally produced by simInit.
params	Character vector of parameter names that can be changed by the optimizer. These must be accessible with params(sim) internally.
objects	A optional named list (must be specified if objFn is not). The names of each list element must correspond to an object in the .GlobalEnv and the list elements must be objects or functions of objects that can be accessed in the ls(sim) internally. These will be used to create the objective function passed to the optimizer. See details and examples.
objFn	An optional objective function to be passed into optimizer. If missing, then POM will use objFnCompare and objects instead. If using POM with a SpaDES simulation, this objFn must contain a spades call internally, followed by a deriva- tion of a value that can be minimized but the optimizer. It must have, as first argument, the values for the parameters. See example.
cl	A cluster object. Optional. This would generally be created using parallel::makeCluster or equivalent. This is an alternative way, instead of beginCluster(), to use parallelism for this function, allowing for more control over cluster use.
optimizer	The function to use to optimize. Default is "DEoptim". Currently it can also

be "optim", which uses stats::optim. The latter two do not seem optimal for stochastic problems and have not been widely tested.
Logical. If using optimizer = "optim", the hessian can be calculated. If this is TRUE, then the standard errors can be estimated using that hessian, assuming normality.
All objects needed in objFn
Character string. Either, "MAD" (default) or "RMSE" indicating that inside the objective function, data and prediction will be compared by Mean Absolute Deviation or Root Mean Squared Error.
List of control arguments passed into the control of each optimization routine. Currently, only passed to DEoptim.control() when optimizer is "DEoptim"
Numeric. If greater than 1, then the function will retry the objective function for a total of that number of times if it results in an NaN. In general this should not be used as the objective function should be made so that it doesn't produce NaN. But, sometimes it is difficult to diagnose stochastic results.
Logical or Character string indicating a filename to log the outputs. Ignored if objFn is supplied. If TRUE (and there is no objFn supplied), then the value of the individual patterns will be output the console if being run interactively or to a tab delimited text file named ObjectiveFnValues.txt (or that passed by the user here) at each evaluation of the POM created objective function. See details.
Numeric. If provided, this vector will be multiplied by the standardized devi- ations (possibly MAD or RMSE) as described in objects. This has the effect of weighing each standardized deviation (pattern-data pair) to a user specified amount in the objective function.
Logical. Should the data patterns and output patterns be logged (log) before calculating the objFnCompare. I.e., mean(abs(log(output) - log(data))). This should be length 1 or length objects. It will be recycled if length >1, less than objects.

Details

There are two ways to use this function: via 1) objFn or 2) objects.

- 1. The user can pass the entire objective function to the objFn argument that will be passed directly to the optimizer. For this, the user will likely need to pass named objects as part of the
- 2. The slightly simpler approach is to pass a list of 'actual data-simulated data' pairs as a named list in objects and specify how these objects should be compared via objFnCompare (whose default is Mean Absolute Deviation or "MAD").

Option 1 offers more control to the user, but may require more knowledge. Option 1 should likely contain a call to simInit(Copy(simList)) and spades internally. See examples that show simple examples of each type, option 1 and option 2. In both cases, params is required to indicate which parameters can be varied in order to achieve the fit.

Currently, option 1 only exists when optimizer is "DEoptim", the default.

POM

The upper and lower limits for parameter values are taken from the metadata in the module. Thus, if the module metadata does not define the upper and lower limits, or these are very wide, then the optimization may have troubles. Currently, there is no way to override these upper and lower limits; the module metadata should be changed if there needs to be different parameter limits for optimization.

objects is a named list of data-pattern pairs. Each of these pairs will be assessed against one another using the objFnCompare, after standardizing each independently. The standardization, which only occurs if the abs(data value < 1), is: mean(abs(derived value - data value))/mean(data value). If the data value is between -1 and 1, then there is no standardization. If there is more than one datapattern pair, then they will simply be added together in the objective function. This gives equal weight to each pair. If the user wishes to put different weight on each pattern, a weights vector can be provided. This will be used to multiply the standardized values described above. Alternatively, the user may wish to weight them differently, in which case, their relative scales can be adjusted.

There are many options that can be passed to DEoptim::DEoptim(), (the details of which are in the help), using optimControl. The defaults sent from POM to DEoptim are: steptol = 3 (meaning it will start assessing convergence after 3 iterations (*which may not be sufficient for your problem*), NP = 10 * length(params) (meaning the population size is 10 x the number of parameters) and itermax = 200 (meaning it won't go past 200 iterations). These and others may need to be adjusted to obtain good values. **NOTE:** DEoptim does not provide a direct estimate of confidence intervals. Also, convergence may be unreliable, and may occur because itermax is reached. Even when convergence is indicated, the estimates are not guaranteed to be global optima. This is different than other optimizers that will normally indicate if convergence was not achieved at termination of the optimization.

Using this function with a parallel cluster currently requires that you pass optimControl = list(parallelType = 1), and possibly package and variable names (and does not yet accept the cl argument). See examples. This setting will use all available threads on your computer. Future versions of this will allow passing of a custom cluster object via cl argument. POM will automatically determine packages to load in the spawned cluster (via packages()) and it will load all objects in the cluster that are necessary, by sending names(objects) to parVar in DEoptim.control.

Setting logObjFnVals to TRUE may help diagnosing some problems. Using the POM derived objective function, essentially all patterns are treated equally. This may not give the correct behaviour for the objective function. Because POM weighs the patterns equally, it may be useful to use the log files to examine the behaviour of the pattern-data pairs. The first file, ObjectiveFnValues.txt, shows the result of each of the (possibly logged), pattern-data deviations, standardized, and weighted. The second file, 'ObjectiveFnValues_RawPatterns.txt', shows the actual value of the pattern (unstandardized, unweighted, unlogged). If weights is passed, then these weighted values will be reflected in the 'ObjectiveFnValues.txt' file.

Value

A list with at least 2 elements. The first (or first several) will be the returned object from the optimizer. The second (or last if there are more than 2), named args is the set of arguments that were passed into the control of the optimizer.

Author(s)

Eliot McIntire

See Also

```
SpaDES.core::spades(), parallel::makeCluster(), simInit()
```

Examples

```
if (interactive() &&
    require("NLMR", quietly = TRUE) &&
   require("RColorBrewer", quietly = TRUE)) {
 set.seed(89462)
 library(parallel)
 library(raster)
 mySim <- simInit(</pre>
   times = list(start = 0.0, end = 2.0, timeunit = "year"),
   params = list(
      .globals = list(stackName = "landscape", burnStats = "nPixelsBurned"),
     fireSpread = list(nfires = 5),
     randomLandscapes = list(nx = 300, ny = 300)
   ),
   modules = list("randomLandscapes", "fireSpread", "caribouMovement"),
   paths = list(modulePath = system.file("sampleModules", package = "SpaDES.core"))
 )
 # Since this is a made up example, we don't have real data
 # to run POM against. Instead, we will run the model once,
 # take the values at the end of the simulation as if they
 # are real data, then rerun the POM function next,
 # comparing these "data" with the simulated values
 # using Mean Absolute Deviation
 outData <- spades(reproducible::Copy(mySim), .plotInitialTime = NA)</pre>
 # Extract the "true" data, in this case, the "proportion of cells burned"
 # Function defined that will use landscape$Fires map from simList,
 # i.e., sim$landscape$Fires
 # the return value being compared via MAD with propCellBurnedData
 propCellBurnedFn <- function(landscape) {</pre>
   sum(getValues(landscape$Fires) > 0) / ncell(landscape$Fires)
 }
 # visualize the burned maps of true "data"
 propCellBurnedData <- propCellBurnedFn(outData$landscape)</pre>
 clearPlot()
 if (interactive()) {
   library(quickPlot)
   fires <- outData$landscape$Fires # Plot doesn't do well with many nested layers
   Plot(fires)
 }
 # Example 1 - 1 parameter
 # In words, this says, "find the best value of spreadprob such that
 # the proportion of the area burned in the simulation
 # is as close as possible to the proportion area burned in
 # the "data", using DEoptim().
```

```
# Can use cluster if computer is multi-threaded.
# This example can use parallelType = 1 in DEoptim. For this, you must manually
# pass all packages and variables as character strings.
# cl <- makeCluster(detectCores() - 1) # not implemented yet in DEoptim</pre>
out1 <- POM(mySim, "spreadprob",</pre>
            list(propCellBurnedData = propCellBurnedFn), # data = pattern pair
            #optimControl = list(parallelType = 1),
            logObjFnVals = TRUE)
## Once cl arg is available from DEoptim, this will work:
# out1 <- POM(mySim, "spreadprob", cl = cl,</pre>
#
             list(propCellBurnedData = propCellBurnedFn)) # data = pattern pair
# Example 2 - 2 parameters
# Function defined that will use caribou from sim$caribou, with
\ensuremath{\texttt{\#}} the return value being compared via MAD with <code>nPattern</code>
# module, parameter N, is from 10 to 1000)
caribouFn <- function(caribou) length(caribou)</pre>
# Extract "data" from simList object (normally, this would be actual data)
nPattern <- caribouFn(outData$caribou)</pre>
aTime <- Sys.time()
parsToVary <- c("spreadprob", "N")</pre>
out2 <- POM(mySim, parsToVary,</pre>
            list(propCellBurnedData = propCellBurnedFn,
                 nPattern = caribouFn), logObjFnVals = TRUE)
                 #optimControl = list(parallelType = 1))
                 #cl = cl) # not yet implemented, waiting for DEoptim
bTime <- Sys.time()</pre>
# check that population overlaps known values (0.225 and 100)
apply(out2$member$pop, 2, quantile, c(0.025, 0.975))
hists <- apply(out2$member$pop, 2, hist, plot = FALSE)</pre>
clearPlot()
for (i in seq_along(hists)) Plot(hists[[i]], addTo = parsToVary[i],
                                  title = parsToVary[i], axes = TRUE)
print(paste("DEoptim", format(bTime - aTime)))
#stopCluster(cl) # not yet implemented, waiting for DEoptim
# Example 3 - using objFn instead of objects
# list all the parameters in the simList, from these, we select to vary
params(mySim)
# Objective Function Example:
   objective function must have several elements
   - first argument must be parameter vector, passed to and used by DEoptim
#
   - likely needs to take sim object, likely needs a copy
#
       because of pass-by-reference semantics of sim objects
#
  - pass data that will be used internally for objective function
#
objFnEx <- function(pars, # param values</pre>
```

```
sim, # simList object
                    nPattern, propCellBurnedData, caribouFn, propCellBurnedFn) {
  ### data
  # make a copy of simList because it will possibly be altered by spades call
  sim1 <- reproducible::Copy(sim)</pre>
  # take the parameters and assign them to simList
  params(sim1)$fireSpread$spreadprob <- pars[1]</pre>
  params(sim1)$caribouMovement$N <- pars[2]</pre>
  # run spades, without plotting
  out <- spades(sim1, .plotInitialTime = NA)</pre>
  # calculate outputs
  propCellBurnedOut <- propCellBurnedFn(out$landscape)</pre>
  nPattern_Out <- caribouFn(out$caribou)</pre>
  minimizeFn <- abs(nPattern_Out - nPattern) +</pre>
                abs(propCellBurnedOut - propCellBurnedData)
  # have more info reported to console, if desired
  # cat(minimizeFn)
  # cat(" ")
  # cat(pars)
  return(minimizeFn)
}
# Run DEoptim with custom objFn, identifying 2 parameters to allow
# to vary, and pass all necessary objects required for the
# objFn
# choose 2 of them to vary. Need to identify them in params & inside objFn
# Change optimization parameters to alter how convergence is achieved
out5 <- POM(mySim, params = c("spreadprob", "N"),</pre>
            objFn = objFnEx,
            nPattern = nPattern,
            propCellBurnedData = propCellBurnedData,
            caribouFn = caribouFn,
            propCellBurnedFn = propCellBurnedFn,
           #cl = cl, # uncomment for cluster # not yet implemented, waiting for DEoptim
            # see ?DEoptim.control for explanation of these options
            optimControl = list(
              NP = 100, # run 100 populations, allowing quantiles to be calculated
           initialpop = matrix(c(runif(100, 0.2, 0.24), runif(100, 80, 120)), ncol = 2),
              parallelType = 1
            )
          )
# Can also use an optimizer directly -- miss automatic parameter bounds,
# and automatic objective function using option 2
```

```
library(DEoptim)
```

```
out7 <- DEoptim(fn = objFnEx,</pre>
               sim = mySim,
               nPattern = nPattern,
               propCellBurnedData = propCellBurnedData,
               caribouFn = caribouFn,
               propCellBurnedFn = propCellBurnedFn,
               # cl = cl, # uncomment for cluster
               # see ?DEoptim.control for explanation of these options
               control = DEoptim.control(
                 steptol = 3,
                 parallelType = 1, # parallelType = 3,
                 packages = list("raster", "SpaDES.core", "RColorBrewer"),
                 parVar = list("objFnEx"),
                 initialpop = matrix(c(runif(40, 0.2, 0.24),
                                        runif(40, 80, 120)), ncol = 2)),
               lower = c(0.2, 80), upper = c(0.24, 120))
```

show,simLists-method Show method for simLists

Description

}

Show method for simLists

Usage

S4 method for signature 'simLists'
show(object)

Arguments

object simLists

Author(s)

Eliot McIntire

simInitAndExperiment Run simInit and Experiment in one step

Description

Run simInit and Experiment in one step

Usage

```
simInitAndExperiment(
  times,
 params,
 modules,
 objects,
 paths,
 inputs,
 outputs,
 loadOrder,
 notOlderThan,
 replicates,
 dirPrefix,
  substrLength,
  saveExperiment,
  experimentFile,
 clearSimEnv,
 cl,
  . . .
```

```
)
```

Arguments

times	A named list of numeric simulation start and end times (e.g., times = list(start = 0.0, end = 10.0, timeunit = "year")), with the final optional element, timeunit, overriding the default time unit used in the simulation which is the "smallest time unit" across all modules. See examples.
params	A list of lists of the form list(moduleName=list(param1=value, param2=value)). See details.
modules	A named list of character strings specifying the names of modules to be loaded for the simulation. Note: the module name should correspond to the R source file from which the module is loaded. Example: a module named "caribou" will be sourced form the file 'caribou.R', located at the specified modulePath(simList) (see below).
objects	(optional) A vector of object names (naming objects that are in the calling envi- ronment of the simInit, which is often the .GlobalEnv unless used program- matically. NOTE: this mechanism will fail if object name is in a package de- pendency), or a named list of data objects to be passed into the simList (more reliable). These objects will be accessible from the simList as a normal list, e.g., mySim\$obj.
paths	An optional named list with up to 4 named elements, modulePath, inputPath, outputPath, and cachePath. See details. NOTE: Experimental feature now allows for multiple modulePaths to be specified in a character vector. The modules will be searched for sequentially in the first modulePath, then if it doesn't find it, in the second etc.
inputs	A data.frame. Can specify from 1 to 6 columns with following column names: objectName (character, required), file (character), fun (character), package

(character), interval (numeric), loadTime (numeric). See inputs() and vignette("ii-modules") section about inputs.

- outputsA data.frame. Can specify from 1 to 5 columns with following column names:
objectName (character, required), file (character), fun (character), package
(character), saveTime (numeric) and eventPriority (numeric). If eventPriority
is not set, it defaults to .last(). If eventPriority is set to a low value, e.g.,
0, 1, 2 and saveTime is start(sim), it should give "initial conditions".
See outputs() and vignette("ii-modules") section about outputs.
- loadOrder An optional character vector of module names specifying the order in which to load the modules. If not specified, the module load order will be determined automatically.
- notOlderThan A time, as in from Sys.time(). This is passed into the Cache function that wraps .inputObjects. If the module uses the .useCache parameter and it is set to TRUE or ".inputObjects", then the .inputObjects will be cached. Setting notOlderThan = Sys.time() will cause the cached versions of .inputObjects to be refreshed, i.e., rerun.
- replicates The number of replicates to run of the same simList. See details and examples.
- dirPrefix String vector. This will be concatenated as a prefix on the directory names. See details and examples.
- substrLength Numeric. While making outputPath for each spades call, this is the number of characters kept from each factor level. See details and examples.
- saveExperiment Logical. Should params, modules, inputs, sim, and resulting experimental design be saved to a file. If TRUE are saved to a single list called experiment. Default TRUE.
- experimentFile String. Filename if saveExperiment is TRUE; saved to outputPath(sim) in .RData format. See Details.
- clearSimEnv Logical. If TRUE, then the envir(sim) of each simList in the return list is emptied. This is to reduce RAM load of large return object. Default FALSE.
- cl A cluster object. Optional. This would generally be created using parallel::makeCluster or equivalent. This is an alternative way, instead of beginCluster(), to use parallelism for this function, allowing for more control over cluster use.
- An alternative way to pass objects, i.e., they can just be named arguments rather than in a objects = list(...). It can also be any options that begins with spades, reproducible or Require, i.e., those identified in spadesOptions(), reproducibleOptions() or RequireOptions(). These will be assigned to the equivalent option *during* the simInit and spades calls only, i.e., they will revert after the simInit or spades calls are complete. NOTE: these are not passed to the simList per se, i.e., they are not be available in the simList during either the simInit or spades calls via sim\$xxx, though they will be returned to the simList at the end of each of these calls (so that the next call to e.g., spades can see them). For convenience, these can be supplied without their package prefix, e.g., lowMemory can be specified instead of spades.lowMemory. In cases that share option name (reproducible.verbose and Require.verbose both exist), passing verbose = FALSE will set both. Obviously this may cause unexpected problems if a module is also expecting a value.

Details

simInitAndExperiment cannot pass modules or params to experiment because these are also in simInit. If the experiment is being used to vary these arguments, it must be done separately (i.e., simInit then experiment).

simLists-class The simLists class

Description

This is a grouping of simList objects. Normally this class will be made using experiment2, but can be made manually if there are existing simList objects.

Slots

paths Named list of modulePath, inputPath, and outputPath paths. Partial matching is performed. These will be prepended to the relative paths of each simList

.xData Environment holding the simLists.

Accessor Methods

None yet defined:

simList-accessors-envir() Simulation environment.

Author(s)

Eliot McIntire

Index

```
as.data.table.simLists,3
as.data.table.simLists(), 15, 16
beginCluster(), 8
DEoptim.control(), 20
DEoptim::DEoptim(), 21
endCluster(), 8
experiment, 6
experiment(), 16
experiment,simList-method(experiment),
        6
experiment2, 14
experiment2,simList-method
        (experiment2), 14
\texttt{initialize, simLists-method, 18}
inputs(), 27
outputs(), 27
packages(), 21
parallel::makeCluster(), 22
POM, 18
POM, simList, character-method (POM), 18
raster::beginCluster(), 6
show,simLists-method, 25
simInit(), 7, 9, 16, 22
simInitAndExperiment, 25
simList-accessors-envir(), 28
simLists (simLists-class), 28
simLists-class, 28
SpaDES.core::spades(), 6, 9, 14, 16, 22
SpaDES.experiment
        (SpaDES.experiment-package), 2
SpaDES.experiment-package, 2
```