Package 'poems'

May 7, 2025

Type Package

Title Pattern-Oriented Ensemble Modeling System

Version 1.3.3

Description A framework of interoperable R6 classes (Chang, 2020, <https://CRAN.R-project.org/package=R6>) for building ensembles of viable models via the pattern-oriented modeling (POM) approach (Grimm et al.,2005, <doi:10.1126/science.1116681>). The package includes classes for encapsulating and generating model parameters, and managing the POM workflow. The workflow includes: model setup; generating model parameters via Latin hypercube sampling (Iman & Conover, 1980, <doi:10.1080/03610928008827996>); running multiple sampled model simulations; collating summary results; and validating and selecting an ensemble of models that best match known patterns. By default, model validation and selection utilizes an approximate Bayesian computation (ABC) approach (Beaumont et al., 2002, <doi:10.1093/genetics/162.4.2025>), although alternative user-defined functionality could be employed. The package includes a spatially explicit demographic population model simulation engine, which incorporates default functionality for density dependence, correlated environmental stochasticity, stage-based transitions, and distance-based dispersal. The user may customize the simulator by defining functionality for translocations, harvesting, mortality, and other processes, as well as defining the sequence order for the simulator processes. The framework could also be adapted for use with other model simulators by utilizing its extendable (inheritable) base classes.

Depends R (>= 4.1.0)

Language en-AU

License GPL-3

URL https://github.com/GlobalEcologyLab/poems, https://globalecologylab.github.io/poems/

BugReports https://github.com/GlobalEcologyLab/poems/issues

Encoding UTF-8

RoxygenNote 7.3.2

Imports abc (>= 2.1), doParallel (>= 1.0.16), foreach (>= 1.5.1), fossil (>= 0.4.0), lhs (>= 1.1.1), metRology (>= 0.9.28.1), R6 (>= 2.5.0), raster (>= 3.6), trend (>= 1.1.4), truncnorm (>= 1.0), gdistance, qs

Collate 'GenericClass.R' 'Region.R' 'GenericModel.R' 'SpatialModel.R'
'DispersalFriction.R' 'GenerativeTemplate.R'
'DispersalTemplate.R' 'Generator.R' 'DispersalGenerator.R'
'GenericManager.R' 'LatinHypercubeSampler.R' 'ModelSimulator.R'
'SimulationModel.R' 'PopulationModel.R' 'SimulationResults.R'
'PopulationResults.R' 'ResultsManager.R' 'SimulationManager.R'
'SimulatorReference.R' 'SpatialCorrelation.R' 'Validator.R'
'data.R' 'poems-package.R' 'population_density.R'
'population_dispersal.R' 'population_env_stoch.R'
'population_results.R' 'population_transformation.R'
'population_transitions.R' 'population_simulator.R'

Suggests knitr, rmarkdown, sf, scales, stringi, testthat

VignetteBuilder knitr, rmarkdown

NeedsCompilation no

Author Sean Haythorne [aut],
Damien Fordham [aut],
Stuart Brown [aut] (ORCID: <https://orcid.org/0000-0002-0669-1418>),
Jessie Buettel [aut],
Barry Brook [aut],
July Pilowsky [aut, cre] (ORCID:
<https://orcid.org/0000-0002-6376-2585>)

Maintainer July Pilowsky <pilowskyj@caryinstitute.org>

Repository CRAN

Date/Publication 2025-05-07 19:20:02 UTC

Contents

`DispersalFriction` *R6 class representing a dispersal friction.*

Description

R6 class functionality for modeling sea, ice and other frictional barriers to dispersal within a spatially-explicit population model. The dispersal friction model utilizes the gdistance package functionality to calculate distance multipliers to modify distance-based dispersal rates for simulated migrations in a spatio-temporal frictional landscape. The frictional landscape is defined via conductance/permeability values, the inverse of friction, which ranges from zero (barrier) to one (no friction) with values in-between representing some friction. For example, a conductance value of 1/5 = 0.2 represents a landscape in which simulated animals move 5 times slower than a non-friction landscape. In this example the resultant distance multiplier would be 5, thus reducing the effective dispersal range.

Super classes

`poems::GenericClass` -> `poems::GenericModel` -> `poems::SpatialModel` -> `DispersalFriction`

Public fields

`attached` A list of dynamically attached attributes (name-value pairs).

Active bindings

`model_attributes` A vector of model attribute names.

`region` A `Region` (or inherited class) object specifying the study region.

`coordinates` Data frame (or matrix) of X-Y population (WGS84) coordinates in longitude (degrees West) and latitude (degrees North) (get and set), or distance-based coordinates dynamically returned by region raster (get only).

`parallel_cores` Number of cores for running the simulations in parallel.

`write_to_dir` Directory path for storing distance multipliers when memory performance is an issue.

`transition_directions` Number of transition directions or neighbors in which cells are connected: usually 4, 8 (default), or 16 (see `gdistance::transition`).

`conductance` Matrix/raster of conductance (inverse friction) values (range: 0 = barrier; 0 < some friction < 1; 1 = no friction) for each grid cell (rows/cells) at each simulation time step (columns/layers).

`attribute_aliases` A list of alternative alias names for model attributes (form: `alias = "attribute"`) to be used with the set and get attributes methods.

`error_messages` A vector of error messages encountered when setting model attributes.

`warning_messages` A vector of warning messages encountered when setting model attributes.

Methods

Public methods:

- `DispersalFriction$calculate_distance_multipliers()`
- `DispersalFriction$clone()`

Method `calculate_distance_multipliers()`**:** Calculates and returns spatio-temporal dispersal distance multipliers for each in-range migration.

Usage:

```
DispersalFriction$calculate_distance_multipliers(dispersal_indices, ...)
```

Arguments:

`dispersal_indices` Two-column integer matrix, data.frame, or array representing the target and source coordinate index for each in-range migration.

`...` Parameters passed via a *params* list or individually.

Returns: Temporal list of dispersal distance multiplier arrays with values for each in-range migration.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
#' U Island example region
coordinates <- data.frame(
  x = rep(seq(177.01, 177.05, 0.01), 5),
  y = rep(seq(-18.01, -18.05, -0.01), each = 5)
)
template_raster <- Region$new(coordinates = coordinates)$region_raster #' full extent
template_raster[][-c(7, 9, 12, 14, 17:19)] <- NA #' make U Island
region <- Region$new(template_raster = template_raster)
raster::plot(region$region_raster,
  main = "Example region (indices)",
  xlab = "Longitude (degrees)", ylab = "Latitude (degrees)",
  colNA = "blue"
)

#' Dispersal distances
dispersal_gen <- DispersalGenerator$new(region = region)
dispersal_gen$set_attributes(params = list(p = 0.5, b = 700, r = 3000))
distances <- round(dispersal_gen$calculate_distance_matrix()) #' in m
dispersal_gen$calculate_distance_data()
dispersal_indices <- as.matrix(dispersal_gen$distance_data$base[, 1:2])

#' Distance multipliers with friction in cell 4
dispersal_friction <- DispersalFriction$new(
  region = region,
  conductance = c(1, 1, 1, 0.5, 1, 1, 1)
)
multipliers <- dispersal_friction$calculate_distance_multipliers(dispersal_indices)
cbind(dispersal_indices,
  distance = distances[dispersal_indices],
  multiplier = multipliers[[1]]
)

#' Note that crossing the water is avoided.
```

`DispersalGenerator` *R6 class representing a dispersal generator.*

Description

R6 class functionality for modeling dispersals within a spatially-explicit population model. The model calculates dispersal rates between population model cells using a distance-based function: *p*exp(-distance/b)* for *distance* <= *r* (otherwise zero), where *p* (proportion), *b* (breadth or average distance) and *r* (range or maximum distance) are configurable model attributes. The dispersal rates are adjusted to limit emigration from each cell to *p*. The model also generates data for constructing compacted dispersal matrices. It dynamically generates attributes defined as *outputs* (default: *dispersal_data*) given sampled *inputs* (default: *dispersal_proportion* and *dispersal_max_distance*). An optional DispersalFriction object may be utilized to modify (equivalent) distances given a

(spatio-temporal) frictional landscape. When this landscape includes temporal changes, the generated *dispersal_data* will be a temporal list of changing dispersal rates.

Super classes

poems::GenericClass -> poems::GenericModel -> poems::SpatialModel -> poems::Generator -> DispersalGenerator

Public fields

`attached` A list of dynamically attached attributes (name-value pairs).

Active bindings

`model_attributes` A vector of model attribute names.

`region` A Region (or inherited class) object specifying the study region.

`coordinates` Data frame (or matrix) of X-Y population (WGS84) coordinates in longitude (degrees West) and latitude (degrees North) (get and set), or distance-based coordinates dynamically returned by region raster (get only).

`description` A brief description of what the generator generates.

`inputs` An array of input attribute names for the generator.

`outputs` An array of output attribute names for the generator.

`file_templates` A nested list of file template attributes.

`function_templates` A nested list of function template attributes.

`distribution_templates` A list of distribution template attributes.

`uses_correlations` A boolean to indicate that a SpatialCorrelation (or inherited class) object is used for generating correlated random deviates.

`spatial_correlation` A SpatialCorrelation (or inherited class) object for generating correlated random deviates.

`temporal_correlation` Absolute correlation coefficient between simulation time steps for all grid cells (0-1; default = 1).

`time_steps` Number of simulation time steps.

`decimals` Number of decimal places applied to generated data outputs (default: NULL = no rounding).

`occupancy_mask` Optional binary mask array (matrix), data frame, or raster (stack) for generated (time-series) data outputs.

`template_attached` A list of template-nested dynamically attached model attributes that are maintained via shallow or *new* cloning.

`dispersal_friction` A DispersalFriction (or inherited class) object for dispersal distance multiplier data.

`distance_classes` Vector of distance interval boundaries for calculating discrete dispersal rates.

`max_distance_classes` The maximum number of distance classes when they are calculated automatically via the maximum distance (default: 1000).

`distance_scale` Scale of distance values in meters (default = 1). Usage: set to 1 for values in meters, or to 1000 for values in kilometers.

`distance_data` Data frame of distance classes including indices for the construction of compact matrices (columns: target_pop, source_pop, compact_row, distance_class).

`dispersal_function_data` Data frame of discrete dispersal function values. Optional first column may provide distance intervals (non-inclusive lower bounds).

`dispersal_proportion` Dispersal function: *p*exp(-distance/b) p* parameter. Represents the proportion and limit of dispersers between model cells. This represents a maximum potential proportion of dispersers; other factors such as population density and carrying capacity may limit the actual proportion of dispersers.

`dispersal_breadth` Dispersal function: *p*exp(-distance/b) b* parameter. Represents the breadth of the dispersal between model cells. Typically estimated via average migration distance.

`dispersal_max_distance` Dispersal maximum distance or range (*r*) parameter limits the use of the dispersal function: *p*exp(-distance/b)*. The function is utilized when *distance* <= *r* otherwise the dispersal rate is set to zero.

`dispersal_index` Sampled index for the dispersal function data frame (to look-up dispersal function parameters).

`dispersal_matrix` Dispersal matrix calculated via dispersal function.

`dispersal_data` Data frame of non-zero dispersal rates including indices for the construction of compact matrices (columns: target_pop, source_pop, emigrant_row, immigrant_row, dispersal_rate).

`attribute_aliases` A list of alternative alias names for model attributes (form: `alias = "attribute"`) to be used with the set and get attributes methods.

`generative_template` A nested `DispersalTemplate` (or inherited class) object for model attributes that are maintained via shallow or *new* cloning.

`generative_requirements` A list of attribute names and the template setting (*"file"*, *"function"*, or *"default"*) that is required to generate their values.

`error_messages` A vector of error messages encountered when setting model attributes.

`warning_messages` A vector of warning messages encountered when setting model attributes.

Methods

Public methods:

- `DispersalGenerator$new()`
- `DispersalGenerator$generative_requirements_satisfied()`
- `DispersalGenerator$set_distance_classes()`
- `DispersalGenerator$calculate_distance_matrix()`
- `DispersalGenerator$calculate_distance_data()`
- `DispersalGenerator$calculate_dispersals()`
- `DispersalGenerator$clone()`

Method `new()`**:** Initialization method sets the generative template and requirements, optionally the dispersal friction object, as well as any attributes passed via a *params* list or individually.

Usage:

```
DispersalGenerator$new(
  generative_template = NULL,
  generative_requirements = NULL,
  dispersal_friction = NULL,
  attribute_aliases = NULL,
  ...
)
```

Arguments:

`generative_template` Optional nested object for generative attributes that need to be maintained when a new clone object is generated for a sample simulation (usually a).

`generative_requirements` Optional list of attribute names and the template setting (*"file"* or *"function"*) that is required to generate their values (otherwise default functionality is used).

`dispersal_friction` Optional `DispersalFriction` (or inherited class) object for dispersal distance multiplier data.

`attribute_aliases` Optional list of extra alias names for model attributes (form: `alias = "attribute"`) to be used with the set and get attributes methods.

`...` Parameters passed via a *params* list or individually.

Method `generative_requirements_satisfied()`**:** Returns a boolean to indicate that all the default, file and/or function template settings that are required for attribute generation are present.

Usage:

```
DispersalGenerator$generative_requirements_satisfied()
```

Returns: Boolean to indicate that the required settings for attribute generation are present.

Method `set_distance_classes()`**:** Sets the distance classes to a sequence of values from minimum to maximum in steps of interval size.

Usage:

```
DispersalGenerator$set_distance_classes(
  minimum = 1,
  maximum = 10,
  interval = 1
)
```

Arguments:

`minimum` Minimum or first distance class sequence value (default = 1).

`maximum` Maximum or last distance class value (default = 10).

`interval` Interval or distance class sequence step size (default = 1).

Method `calculate_distance_matrix()`**:** Returns a matrix with the calculated distance (in meters by default) between each pair of region cells.

Usage:

```
DispersalGenerator$calculate_distance_matrix(use_longlat = NULL)
```

Arguments:

`use_longlat` Optional boolean indicating use of (WGS84) coordinates in longitude (degrees West) and latitude (degrees North).

Returns: Matrix with distances between region cells.

Method `calculate_distance_data()`**:** Calculates the distance class for within-range populations using the set/provided distance classes. Also calculates indices for constructing compact matrices.

Usage:

```
DispersalGenerator$calculate_distance_data(distance_matrix = NULL, ...)
```

Arguments:

`distance_matrix` Optional pre-calculated matrix with distances between population cells (population rows by population columns).

`...` Parameters passed via a *params* list or individually.

Method `calculate_dispersals()`**:** Calculates, using the conditional dispersal limiting function for a simulation sample, a dispersal matrix, or a list of data frames of non-zero dispersal rates and indices for constructing a compact dispersal matrix (default), and optional changing rates over time (via `DispersalFriction` object).

Usage:

```
DispersalGenerator$calculate_dispersals(type = "data")
```

Arguments:

`type` Optional type selector (*"data"* or *"matrix"*) to determine whether to calculate a dispersal matrix or data frame (default).

Returns: Returns character string message when calculation prerequisites are not met (for simulation logging).

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# U Island example region
coordinates <- data.frame(
  x = rep(seq(177.01, 177.05, 0.01), 5),
  y = rep(seq(-18.01, -18.05, -0.01), each = 5)
)
template_raster <- Region$new(coordinates = coordinates)$region_raster # full extent
template_raster[][-c(7, 9, 12, 14, 17:19)] <- NA # make U Island
region <- Region$new(template_raster = template_raster)
raster::plot(region$region_raster,
  main = "Example region (indices)",
  xlab = "Longitude (degrees)", ylab = "Latitude (degrees)",
  colNA = "blue"
)

# Distance-based dispersal generator
```

```
dispersal_gen <- DispersalGenerator$new(
  region = region,
  dispersal_max_distance = 3000, # in m
  inputs = c("dispersal_p", "dispersal_b"),
  decimals = 5
)
dispersal_gen$calculate_distance_data() # pre-calculate
dispersal_gen$generate(input_values = list(
  dispersal_p = 0.5,
  dispersal_b = 700
))
```

`DispersalTemplate` *R6 class representing a nested container for dispersal generator attributes*

Description

R6 class representing a nested container for `DispersalGenerator` attributes that are maintained when new model clones are created. The container maintains *input* and *output* attribute names, file, function and distribution templates, correlation parameters (for distribution generation), rounding decimals, occupancy mask, and other `DispersalGenerator` attributes that need to be maintained when cloning.

Super class

`poems::GenerativeTemplate` -> `DispersalTemplate`

Public fields

`attached` A list of dynamically attached attributes (name-value pairs).

Active bindings

`description` A brief description of what the generator generates.

`inputs` An array of input attribute names for the generator.

`outputs` An array of output attribute names for the generator.

`file_templates` A nested list of file template attributes.

`function_templates` A nested list of function template attributes.

`distribution_templates` A list of distribution template attributes.

`uses_correlations` A boolean to indicate that a `SpatialCorrelation` (or inherited class) object is used for generating correlated random deviates.

`spatial_correlation` A `SpatialCorrelation` (or inherited class) object for generating correlated random deviates.

`temporal_correlation` Absolute correlation coefficient between simulation time steps for all grid cells (0-1; default = 1).

`time_steps` Number of simulation time steps.

`decimals` Number of decimal places applied to generated data outputs (default: NULL = no rounding).

`occupancy_mask` Optional binary mask array (matrix), data frame, or raster (stack) for generated (time-series) data outputs.

`dispersal_friction` A `DispersalFriction` (or inherited class) object for dispersal distance multiplier data.

`distance_classes` Vector of distance interval boundaries (in km) for calculating discrete dispersal rates.

`max_distance_classes` The maximum number of distance classes when they are calculated automatically via the maximum distance (default: 1000).

`distance_scale` Scale of distance values in meters (default = 1). Usage: set to 1 for values in meters, or to 1000 for values in kilometers.

`distance_data` Data frame of distance classes including indices for the construction of compact matrices (columns: target_pop, source_pop, compact_row, distance_class).

`dispersal_function_data` Data frame of discrete dispersal function values. Optional first column may provide distance intervals (non-inclusive lower bounds).

`dispersal_proportion` Dispersal function: *p*exp(-distance/b)* *p* parameter. Represents the proportion and limit of dispersers between model cells. This represents a maximum potential proportion of dispersers; other factors such as population density and carrying capacity may limit the actual proportion of dispersers.

`dispersal_breadth` Dispersal function: *p*exp(-distance/b)* *b* parameter. Represents the breadth of the dispersal between model cells. Typically estimated via average migration distance.

`dispersal_max_distance` Dispersal maximum distance or range (*r*) parameter limits the use of the dispersal function: *p*exp(-distance/b)*. The function is utilized when *distance* <= *r* otherwise the dispersal rate is set to zero.

Methods

Public methods:

- `DispersalTemplate$clone()`

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
dispersal_template <- DispersalTemplate$new()
dispersal_template$dispersal_breadth <- 130
dispersal_template$dispersal_proportion <- 0.4
coordinates <- data.frame(x = rep(1:4, 4), y = rep(1:4, each = 4))
dispersal_gen <- DispersalGenerator$new(
  coordinates = coordinates, inputs = c("dispersal_r"),
  generative_template = dispersal_template
)
dispersal_gen$dispersal_breadth
dispersal_gen$dispersal_proportion
```

`GenerativeTemplate` *R6 class representing a nested container for generator attributes*

Description

R6 class representing a nested container for Generator attributes that are maintained when new model clones are created. The container maintains *input* and *output* attribute names, file, function and distribution templates, correlation parameters (for distribution generation), rounding decimals, occupancy mask, and any inherited class model attributes that need to be maintained when cloning.

Public fields

`attached` A list of dynamically attached attributes (name-value pairs).

Active bindings

`description` A brief description of what the generator generates.

`inputs` An array of input attribute names for the generator.

`outputs` An array of output attribute names for the generator.

`file_templates` A list of file template attributes.

`function_templates` A list of function template attributes.

`distribution_templates` A list of distribution template attributes.

`uses_correlations` A boolean to indicate that a SpatialCorrelation (or inherited class) object is used for generating correlated random deviates.

`spatial_correlation` A SpatialCorrelation (or inherited class) object for generating correlated random deviates.

`temporal_correlation` Absolute correlation coefficient between simulation time steps for all grid cells (0-1; default = 1).

`time_steps` Number of simulation time steps (default = 1).

`generate_rasters` Boolean to indicate if rasters should be generated (default: NULL).

`decimals` Number of decimal places applied to the generated values (default: NULL = no rounding).

`occupancy_mask` Optional binary mask array (matrix), data frame, or raster (stack) for generated (time-series) data.

Methods

Public methods:

- GenerativeTemplate$new()
- GenerativeTemplate$clone()

Method `new()`**:** Initialization method initializes the generator templates.

Usage:

```
GenerativeTemplate$new()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
gen_template <- GenerativeTemplate$new()
gen_template$occupancy_mask <- array(c(1, 1, 0, 0, 1, 1, 1))
gen_template$decimals <- 4
gen_template$description <- "Test generator"

coordinates <- data.frame(x = c(1:4, 4:2), y = c(1, 1:4, 4:3))

generator <- Generator$new(
  region = Region$new(coordinates = coordinates), attr1 = 1,
  template_attached = gen_template
)
generator$description
generator$occupancy_mask
generator$decimals
```

`Generator` *R6 class representing a dynamic attribute generator*

Description

R6 class representing a model that dynamically generates attribute values (*outputs*) via reading data from files, running assigned functions, generating sample distributions, or built-in functions (assigned as *default* in inherited classes), using simulation sample parameters (*inputs*).

Super classes

poems::GenericClass -> poems::GenericModel -> poems::SpatialModel -> Generator

Public fields

`attached` A list of dynamically attached attributes (name-value pairs).

Active bindings

`model_attributes` A vector of model attribute names.

`region` A `Region` (or inherited class) object specifying the study region.

`coordinates` Data frame (or matrix) of X-Y population (WGS84) coordinates in longitude (degrees West) and latitude (degrees North) (get and set), or distance-based coordinates dynamically returned by region raster (get only).

`description` A brief description of what the generator generates.

`inputs` An array of input attribute names for the generator.

`outputs` An array of output attribute names for the generator.

`file_templates` A nested list of file template attributes.

`function_templates` A nested list of function template attributes.

`distribution_templates` A list of distribution template attributes.

`uses_correlations` A boolean to indicate that a `SpatialCorrelation` (or inherited class) object is used for generating correlated random deviates.

`spatial_correlation` A `SpatialCorrelation` (or inherited class) object for generating correlated random deviates.

`temporal_correlation` Absolute correlation coefficient between simulation time steps for all grid cells (0-1; default = 1).

`time_steps` Number of simulation time steps.

`generate_rasters` Boolean to indicate if rasters should be generated (defaults to TRUE when region uses rasters).

`decimals` Number of decimal places applied to generated data outputs (default: NULL = no rounding).

`occupancy_mask` Optional binary mask array (matrix), data frame, or raster (stack) for generated (time-series) data outputs.

`template_attached` A list of template-nested dynamically attached model attributes that are maintained via shallow or *new* cloning.

`attribute_aliases` A list of alternative alias names for model attributes (form: `alias = "attribute"`) to be used with the set and get attributes methods.

`generative_template` A nested `GenerativeTemplate` (or inherited class) object for model attributes that are maintained via shallow or *new* cloning.

`generative_requirements` A list of attribute names and the template setting (*"file"*, *"function"*, or *"default"*) that is required to generate their values.

`error_messages` A vector of error messages encountered when setting model attributes.

`warning_messages` A vector of warning messages encountered when setting model attributes.

Methods

Public methods:

- Generator$new()
- Generator$new_clone()
- Generator$get_attributes()
- Generator$generate()
- Generator$add_file_template()
- Generator$add_function_template()
- Generator$add_distribution_template()
- Generator$read_file()
- Generator$run_function()
- Generator$sample_distribution()
- Generator$add_generative_requirements()
- Generator$generative_requirements_satisfied()
- Generator$clone()

Method `new()`**:** Initialization method sets the generative template and requirements as well as any attributes passed via a *params* list or individually.

Usage:

```
Generator$new(generative_template = NULL, generative_requirements = NULL, ...)
```

Arguments:

`generative_template` A GenerativeTemplate (or inherited class) object containing the file, function and/or distribution templates utilized (facilitates shallow cloning).

`generative_requirements` A list of attribute names and the template setting ("file", "function", or "distribution") that is required to generate their values.

`...` Parameters passed via a *params* list or individually.

Method `new_clone()`**:** Creates a new (re-initialized) object of the current (inherited) object class with optionally passed parameters.

Usage:

```
Generator$new_clone(...)
```

Arguments:

`...` Parameters passed via the inherited class constructor (defined in initialize and run via new).

Returns: New object of the current (inherited) class.

Method `get_attributes()`**:** Returns a list of existing and template-generated values for selected attributes or attribute aliases (when array of parameter names provided), or all existing attributes (when no params).

Usage:

```
Generator$get_attributes(params = NULL)
```

Arguments:

`params` Array of attribute names to return, including those to be template-generated (all when NULL).

Returns: List of selected or all attributes values.

Method `generate()`**:** Returns a list of generated output values (attributes) corresponding to the sample input values (attributes).

Usage:

```
Generator$generate(input_values = list())
```

Arguments:

`input_values` List of sample input values for generator attributes.

Returns: List containing generated model output attributes and/or any error/warning messages.

Method `add_file_template()`**:** Adds a file template for reading raster/RData(RDS)/CSV files for a given model attribute.

Usage:

```
Generator$add_file_template(
  param,
  path_template,
  path_params = c(),
  file_type = "GRD"
)
```

Arguments:

`param` Name of model attribute to be read from a file.

`path_template` Template string for the file path with placeholders (see `sprintf`) for simulation sample parameters.

`path_params` Array of the names of the simulation sample parameters to be substituted (in order) into the path template.

`file_type` File type raster *"GRD"* (default), *"TIF"*, *"RData/RDS"*, *"QS"*, or *"CSV"* to be read.

Method `add_function_template()`**:** Adds a function template for running a user-defined function to calculate a given model attribute.

Usage:

```
Generator$add_function_template(param, function_def, call_params = c())
```

Arguments:

`param` Name of model attribute to be generated using a function.

`function_def` Function definition (or path to the file containing the function) in form: `function(params)`, where *params* is a list passed to the function.

`call_params` Array of the names of the model parameters/attributes to be passed into the function via a list: *params*.

Method `add_distribution_template()`**:** Adds a distribution template for generating a given model attribute via sampling a distribution.

Usage:

```
Generator$add_distribution_template(
  param,
  distr_type = c("uniform", "normal", "lognormal", "beta", "triangular"),
```

```
  distr_params = list(),
  sample = NULL,
  random_seed = NULL,
  normalize_threshold = NULL
)
```

Arguments:

`param` Name of model attribute to be generated via sampling a distribution.

`distr_type` Distribution type to sample from (uniform, normal, lognormal, beta or triangular).

`distr_params` List of distribution parameters and their values or associated model attributes (uniform: lower, upper; normal: mean, sd; lognormal: meanlog, sdlog (or mean, sd); beta: alpha, beta (or mean, sd); triangular: lower, mode, upper).

`sample` Model attribute(s) name(s) or values associated with single sample probabilities (0-1), or bounds as a vector (e.g. `sample = c("p_lower", "p_upper")`), or as a list (e.g. `sample = list(mid = "p", window = 0.2)` for bounds p +/- 0.1).

`random_seed` Random seed utilized when sample probability is generated internally, via bounds, and/or correlated deviates.

`normalize_threshold` Optional normalization threshold is utilized when generated values are to be normalized with a fixed upper limit/threshold.

Method `read_file()`**:** Reads and returns the value of a model attribute from a file using the corresponding file template and simulation sample parameters.

Usage:

```
Generator$read_file(param)
```

Arguments:

`param` Name of model attribute to be read from the file.

Returns: Model attribute value read from a file.

Method `run_function()`**:** Returns the calculated value of a model attribute using the corresponding function template and model simulation sample parameters.

Usage:

```
Generator$run_function(param)
```

Arguments:

`param` Name of model attribute to be calculated using a function.

Returns: Model attribute value calculated using a function.

Method `sample_distribution()`**:** Returns the calculated value of a model attribute using the corresponding distribution template and simulation sample parameters.

Usage:

```
Generator$sample_distribution(param)
```

Arguments:

`param` Name of model attribute to be calculated using a sampling distribution.

Returns: Model attribute value calculated via distribution sampling.

Method `add_generative_requirements()`**:** Adds attribute names and the template setting (*"file"*, *"function"* or *"distribution"*) that is required to generate their values (via a *params* list or individually).

Usage:

```
Generator$add_generative_requirements(params = list(), ...)
```

Arguments:

`params` Parameters passed via a list (e.g. `params = list(attr1 = "file", attr2 = "function", attr3 = "distribution")`).

`...` Parameters passed individually (e.g. `attr3 = "file"`).

Method `generative_requirements_satisfied()`**:** Returns a boolean to indicate that all the file, function and/or distribution template settings that are required for attribute generation are present.

Usage:

```
Generator$generative_requirements_satisfied()
```

Returns: Boolean to indicate that the required settings for attribute generation are present.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# U Island example region
coordinates <- data.frame(
  x = rep(seq(177.01, 177.05, 0.01), 5),
  y = rep(seq(-18.01, -18.05, -0.01), each = 5)
)
coordinates <- coordinates[c(7, 9, 12, 14, 17:19), ]
region <- Region$new(coordinates = coordinates, use_raster = FALSE)
# Spatial correlation
spatial_correlation <- SpatialCorrelation$new(
  region = region, correlation_amplitude = 0.6,
  correlation_breadth = 300
)
spatial_correlation$calculate_compact_decomposition(decimals = 4)
# Example habitat suitability in file
saveRDS(
  array(c(0.5, 0.3, 0.7, 0.9, 0.6, 0.7, 0.8), c(7, 5)),
  file.path(tempdir(), "hs_mean_1.RData")
)
# Generator
capacity_gen <- Generator$new(
  description = "capacity",
  region = region,
  time_steps = 5,
```

```
  spatial_correlation = spatial_correlation,
  temporal_correlation = 0.9,
  hs_sd = 0.1, # template attached
  inputs = c("hs_file", "density_max", "initial_n"),
  outputs = c("initial_abundance", "carrying_capacity")
)
capacity_gen$add_generative_requirements(list(
  hs_mean = "file",
  hs_sample = "distribution",
  carrying_capacity = "function",
  initial_abundance = "function"
))
# File template for mean habitat suitability
capacity_gen$add_file_template("hs_mean",
  path_template = file.path(tempdir(), "hs_mean_%s.RData"),
  path_params = c("hs_file"), file_type = "RDS"
)
# Distribution template for sampling habitat suitability
capacity_gen$add_distribution_template("hs_sample",
  distr_type = "beta",
  distr_params = list(
    mean = "hs_mean",
    sd = "hs_sd"
  )
)
# Function templates for initial abundance and carrying capacity
capacity_gen$add_function_template("initial_abundance",
  function_def = function(params) {
    stats::rmultinom(1,
      size = params$initial_n,
      prob = params$hs_sample[, 1]
    )
  },
  call_params = c("initial_n", "hs_sample")
)
capacity_gen$add_function_template("carrying_capacity",
  function_def = function(params) {
    round(params$density_max * params$hs_sample)
  },
  call_params = c("density_max", "hs_sample")
)
# Generation
capacity_gen$generate(input_values = list(
  hs_file = 1,
  initial_n = 400,
  density_max = 100
))
```

GenericClass *R6 class with generic reusable functionality*

Description

R6 class with generic (abstract) new cloning functionality.

Public fields

`object_generator` Class object generator used to create new clones, particularly for user inheritance.

`attached` A list of dynamically attached attributes (name-value pairs).

Methods

Public methods:

- `GenericClass$new()`
- `GenericClass$new_clone()`
- `GenericClass$clone()`

Method `new()`**:** Initialization method saves an object generator for new cloning.

Usage:

```
GenericClass$new(object_generator = NULL, ...)
```

Arguments:

`object_generator` Class object generator used to create new clones, particularly for user inheritance.

`...` Parameters passed individually (ignored).

Method `new_clone()`**:** Creates a new (re-initialized) object of the current (inherited) object class with optionally passed parameters.

Usage:

```
GenericClass$new_clone(...)
```

Arguments:

`...` Parameters passed via the inherited class constructor (defined in initialize and run via new).

Returns: New object of the current (inherited) class.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
object1 <- GenericClass$new()
class(object1)
# Referencing
object_ref <- object1
object_ref$attached$a <- 1
```

```
object1$attached
# Cloning
object2 <- object1$clone()
object2$attached$b <- 2
object1$attached
object2$attached
# New cloning
object3 <- object1$new_clone()
object3$attached$c <- 3
object1$attached
object3$attached
```

GenericManager *R6 class representing a generic manager.*

Description

R6 class to represent a generic (abstract) manager for generating or processing simulation results, as well as optionally generating values via generators.

Super class

poems::GenericClass -> GenericManager

Public fields

attached A list of dynamically attached attributes (name-value pairs).

Active bindings

sample_data A data frame of sampled parameters for each simulation/result.

generators A list of generators (Generator or inherited class) objects for generating simulation model values.

parallel_cores Number of cores for running the simulations in parallel.

results_dir Results directory path.

results_ext Result file extension (default is .RData).

results_filename_attributes A vector of: prefix (optional); attribute names (from the sample data frame); postfix (optional); utilized to construct results filenames.

error_messages A vector of error messages encountered.

warning_messages A vector of warning messages encountered.

Methods

Public methods:

- GenericManager$new()
- GenericManager$get_attribute()
- GenericManager$get_message_sample()
- GenericManager$get_results_filename()
- GenericManager$clone()

Method `new()`**:** Initialization method sets any included attributes (*sample_data*, *generators*, *parallel_cores*, *results_dir*, *results_filename_attributes*) and attaches other attributes individually listed.

Usage:

```
GenericManager$new(...)
```

Arguments:

`...` Parameters listed individually.

Method `get_attribute()`**:** Returns a named manager or attached attribute.

Usage:

```
GenericManager$get_attribute(param)
```

Arguments:

`param` Character string name of the attribute.

Returns: Selected attribute value.

Method `get_message_sample()`**:** Substitutes the specified sample details into a status message (using sprintf) and returns the result.

Usage:

```
GenericManager$get_message_sample(status_message, sample_index)
```

Arguments:

`status_message` Character string message with a placeholder for sample details.

`sample_index` Row index of sample data frame containing details of substitution parameters.

Returns: Status message with substituted sample details.

Method `get_results_filename()`**:** Constructs and returns the results filename based on the sample data frame index and results filename attributes.

Usage:

```
GenericManager$get_results_filename(sample_index)
```

Arguments:

`sample_index` Row index of sample data frame containing details of substitution parameters.

Returns: Results filename with substituted sample details.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
generic_manager <- GenericManager$new(
  attr1 = 22:23,
  results_filename_attributes = c("attr1", "example")
)
generic_manager$get_results_filename(1)
generic_manager$get_results_filename(2)
```

GenericModel *R6 class representing a generic model.*

Description

R6 class with generic (abstract) functionality for toolset models, including model attribute get and set methods that resolve attribute scope (*public*, *active*, *attached*), attribute aliases, attribute attachment, and error and warning message attributes.

Super class

poems::GenericClass -> GenericModel

Public fields

`attached` A list of dynamically attached attributes (name-value pairs).

Active bindings

`model_attributes` A vector of model attribute names.

`attribute_aliases` A list of alternative alias names for model attributes (form: `alias = "attribute"`) to be used with the set and get attributes methods.

`error_messages` A vector of error messages encountered when setting model attributes.

`warning_messages` A vector of warning messages encountered when setting model attributes.

Methods

Public methods:

- GenericModel$new()
- GenericModel$new_clone()
- GenericModel$get_attribute_names()
- GenericModel$get_attributes()
- GenericModel$get_attribute()
- GenericModel$get_attribute_aliases()
- GenericModel$set_attributes()
- GenericModel$clone()

Method `new()`**:** Initialization method sets given attributes individually and/or from a list.

Usage:

```
GenericModel$new(
  model_attributes = NULL,
  attribute_aliases = NULL,
  params = list(),
  ...
)
```

Arguments:

`model_attributes` A vector of model attribute names.

`attribute_aliases` A list of alternative alias names for model attributes (form: `alias = "attribute"`) to be used with the set and get attributes methods.

`params` Parameters passed via a list.

`...` Parameters passed individually.

Method `new_clone()`**:** Creates a new (re-initialized) object of the current (inherited) object class with optionally passed parameters.

Usage:

```
GenericModel$new_clone(...)
```

Arguments:

`...` Parameters passed via the inherited class constructor (defined in initialize and run via new).

Returns: New object of the current (inherited) class.

Method `get_attribute_names()`**:** Returns an array of all attribute names including public and private model attributes, as well as attached attributes, error and warning messages.

Usage:

```
GenericModel$get_attribute_names()
```

Returns: Array of all attribute names.

Method `get_attributes()`**:** Returns a list of values for selected attributes or attribute aliases (when array of parameter names provided) or all attributes (when no params).

Usage:

```
GenericModel$get_attributes(params = NULL)
```

Arguments:

`params` Array of attribute names to return (all when NULL).

Returns: List of selected or all attributes values.

Method `get_attribute()`**:** Returns the value of an attribute via character name or attribute alias.

Usage:

```
GenericModel$get_attribute(param)
```

Arguments:

`param` Character string name of the attribute.

Returns: Attribute value.

Method `get_attribute_aliases()`**:** Returns an array of attribute names and aliases for specified or all attributes.

Usage:

```
GenericModel$get_attribute_aliases(params = NULL)
```

Arguments:

`params` Array of attribute names for names/aliases to return (all when NULL).

Returns: Array of selected or all attribute names and aliases.

Method `set_attributes()`**:** Sets given attributes (optionally via alias names) individually and/or from a list.

Usage:

```
GenericModel$set_attributes(params = list(), ...)
```

Arguments:

`params` List of parameters/attributes.

`...` Parameters/attributes passed individually.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
model1 <- GenericModel$new(
  model_attributes = c("a", "b", "c"),
  attribute_aliases = list(A = "a"),
  params = list(a = 1, b = 2), c = 3
)
# Get/set attributes
model1$get_attribute_names()
model1$set_attributes(d = 4)
model1$get_attributes()
model1$get_attribute("A")
model1$get_attribute("B")
model1$get_attribute_aliases() # all attribute names
# New cloning
model2 <- model1$new_clone(e = 5)
model2$get_attributes()
model2$modelattributes
model2$attribute_aliases
```

LatinHypercubeSampler *R6 class to represent a Latin hypercube sampler.*

Description

R6 class that generates Latin hypercube samples (using randomLHS) for parameters drawn from configured distributions: uniform, Poisson, normal, lognormal, beta, truncated normal or triangular. It generates a data frame of sample values.

Super class

poems::GenericClass -> LatinHypercubeSampler

Public fields

attached A list of dynamically attached attributes (name-value pairs).

Active bindings

parameter_names A vector of sample parameter names.

parameter_distributions A list of sample distribution values (nested list with appropriate parameters).

Methods

Public methods:

- LatinHypercubeSampler$new()
- LatinHypercubeSampler$set_class_parameter()
- LatinHypercubeSampler$set_uniform_parameter()
- LatinHypercubeSampler$set_normal_parameter()
- LatinHypercubeSampler$set_poisson_parameter()
- LatinHypercubeSampler$set_lognormal_parameter()
- LatinHypercubeSampler$set_beta_parameter()
- LatinHypercubeSampler$set_truncnorm_parameter()
- LatinHypercubeSampler$set_triangular_parameter()
- LatinHypercubeSampler$generate_samples()
- LatinHypercubeSampler$clone()

Method new(): Initialization method sets parameter names when provided.

Usage:

```
LatinHypercubeSampler$new(parameter_names = NULL, ...)
```

Arguments:

parameter_names Optional vector of sample parameter names.

... Additional parameters passed individually.

Method `set_class_parameter()`: Sets a parameter to sampled from a vector of classes.

Usage:

```
LatinHypercubeSampler$set_class_parameter(parameter_name, classes)
```

Arguments:

`parameter_name` Character string name of sample parameter.

`classes` Vector of class values.

Method `set_uniform_parameter()`: Sets a parameter to be sampled from a `uniform` distribution with lower and upper bounds, optionally rounded to a specified number of decimal places.

Usage:

```
LatinHypercubeSampler$set_uniform_parameter(
  parameter_name,
  lower = 0,
  upper = 1,
  decimals = NULL
)
```

Arguments:

`parameter_name` Character string name of sample parameter.

`lower` Lower bound of the uniform distribution (default = 0).

`upper` Upper bound of the uniform distribution (default = 1).

`decimals` Optional number of decimals applied to generated samples.

Method `set_normal_parameter()`: Sets a parameter to be sampled from a `normal` distribution with mean and standard deviation, optionally rounded to a specified number of decimal places.

Usage:

```
LatinHypercubeSampler$set_normal_parameter(
  parameter_name,
  mean = 0,
  sd = 1,
  decimals = NULL
)
```

Arguments:

`parameter_name` Character string name of sample parameter.

`mean` Mean parameter for the normal distribution (default = 0).

`sd` Standard deviation parameter for the normal distribution (default = 1).

`decimals` Optional number of decimals applied to generated samples.

Method `set_poisson_parameter()`: Sets a parameter to be sampled from a `Poisson` distribution with lambda parameter. Produces integers.

Usage:

```
LatinHypercubeSampler$set_poisson_parameter(parameter_name, lambda = 1)
```

Arguments:

`parameter_name` Character string name of sample parameter.

`lambda` Lambda parameter for the Poisson distribution. Must be positive (default = 1).

Method `set_lognormal_parameter()`: Sets a parameter to be sampled from a lognormal distribution with log mean and log standard deviation, optionally expressed as regular mean and SD (overriding log mean/sd), and optionally rounded to a specified number of decimal places.

Usage:

```
LatinHypercubeSampler$set_lognormal_parameter(
  parameter_name,
  meanlog = 0,
  sdlog = 1,
  mean = NULL,
  sd = NULL,
  decimals = NULL
)
```

Arguments:

`parameter_name` Character string name of sample parameter.

`meanlog` Log mean parameter for the lognormal distribution (default = 0).

`sdlog` Log standard deviation parameter for the lognormal distribution (default = 1).

`mean` Optional (overriding) regular mean parameter for the lognormal distribution (default = NULL).

`sd` Optional (overriding) standard deviation parameter for the lognormal distribution (default = NULL).

`decimals` Optional number of decimals applied to generated samples.

Method `set_beta_parameter()`: Sets a parameter to be sampled from a beta distribution configured with alpha and beta parameters, or optionally with mean and standard deviation (overriding alpha and beta), and optionally rounded to a specified number of decimal places.

Usage:

```
LatinHypercubeSampler$set_beta_parameter(
  parameter_name,
  alpha = 1,
  beta = 1,
  mean = NULL,
  sd = NULL,
  decimals = NULL
)
```

Arguments:

`parameter_name` Character string name of sample parameter.

`alpha` Shaping (towards 1) parameter (> 0) for the beta distribution (default = 1).

`beta` Shaping (towards 0) parameter (> 0) for the beta distribution (default = 1).

`mean` Optional (overriding) mean parameter for the beta distribution (default = NULL).

`sd` Optional (overriding) standard deviation parameter for the beta distribution (default = NULL).

`decimals` Optional number of decimals applied to generated samples.

Method `set_truncnorm_parameter()`: Sets a parameter to be sampled from a truncated normal distribution with mean, standard deviation, and lower and upper bounds, optionally rounded to a specified number of decimal places.

Usage:

```
LatinHypercubeSampler$set_truncnorm_parameter(
  parameter_name,
  mean = 0,
  sd = 1,
  lower = -Inf,
  upper = Inf,
  decimals = NULL
)
```

Arguments:

`parameter_name` Character string name of sample parameter.

`mean` Mean parameter of the truncated normal distribution (default = 0).

`sd` Standard deviation of the truncated normal distribution (default = 1).

`lower` Lower bound of the truncated normal distribution (default = -Inf, meaning no lower bound).

`upper` Upper bound of the truncated normal distribution (default = Inf, meaning no upper bound).

`decimals` Optional number of decimals that generated samples are rounded to.

Method `set_triangular_parameter()`**:** Sets a parameter to be sampled from a `triangular` distribution with lower and upper bounds and mode (peak), optionally rounded to a specified number of decimal places.

Usage:

```
LatinHypercubeSampler$set_triangular_parameter(
  parameter_name,
  lower = 0,
  upper = 1,
  mode = (lower + upper)/2,
  decimals = NULL
)
```

Arguments:

`parameter_name` Character string name of sample parameter.

`lower` Lower bound of the triangular distribution (default = 0).

`upper` Upper bound of the triangular distribution (default = 1).

`mode` Mode (or peak) of the triangular distribution (default = (lower + upper)/2).

`decimals` Optional number of decimals applied to generated samples.

Method `generate_samples()`**:** Generates Latin hypercube sample data (via `randomLHS`) for the set parameters using corresponding distributions.

Usage:

```
LatinHypercubeSampler$generate_samples(number = 10, random_seed = NULL)
```

Arguments:

`number` Number of samples to generate (default = 10).

`random_seed` Optional seed for the random generation of samples.

Returns: A data frame of generated sample values.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
lhs_gen <- LatinHypercubeSampler$new(parameter_names = c("size", "age", "km", "price"))
lhs_gen$set_class_parameter("size", c("small", "medium", "large"))
lhs_gen$set_uniform_parameter("age", lower = 18, upper = 70, decimals = 0)
lhs_gen$set_poisson_parameter("offspring", lambda = 2)
lhs_gen$set_normal_parameter("km", mean = 50000, sd = 20000, decimals = 0)
lhs_gen$set_truncnorm_parameter("kg", mean = 75, sd = 20, lower = 0, upper = Inf, decimals = 2)
lhs_gen$set_lognormal_parameter("price", mean = 30000, sd = 10000, decimals = 0)
lhs_gen$set_beta_parameter("tread", mean = 0.7, sd = 0.1, decimals = 2)
lhs_gen$set_triangular_parameter("rating",
  lower = 0, upper = 10, mode = 5,
  decimals = 1
)
lhs_gen$generate_samples(number = 10, random_seed = 123)
```

`ModelSimulator` *R6 class representing a model simulator.*

Description

`R6` class for running individual model simulations via a simulation function, storing results, and generating success/error statuses.

Super class

`poems::GenericClass` -> `ModelSimulator`

Public fields

`attached` A list of dynamically attached attributes (name-value pairs).

Active bindings

`simulation_model` A SimulationModel object or an inherited class object.

`simulation_function` Name (character string) or direct assignment (assigned or loaded via source path) of the simulation function, which takes a `SimulationModel` (or inherited class) as an input and returns the simulation results.

`sample_id` An identifier for the simulation sample.

`results` A list of result structures.

Methods

Public methods:

- ModelSimulator$new()
- ModelSimulator$new_clone()
- ModelSimulator$get_attribute()
- ModelSimulator$run()
- ModelSimulator$clone()

Method `new()`**:** Initialization method sets the population model, and optionally the simulation function, the sample ID, and any attached attributes listed individually.

Usage:

```
ModelSimulator$new(
  simulation_model = NULL,
  simulation_function = NULL,
  sample_id = NULL,
  ...
)
```

Arguments:

`simulation_model` A `SimulationModel` (or inherited class) object (can be set later).

`simulation_function` Optional name (character string) or direct assignment (assigned or loaded via source path) of the simulation function, which takes a `SimulationModel` (or inherited class) as an input and returns the simulation results.

`sample_id` Optional identifier for the simulation sample.

`...` Additional parameters passed individually are attached.

Method `new_clone()`**:** Creates a new (re-initialized) object of the current (inherited) object class with optionally passed parameters.

Usage:

```
ModelSimulator$new_clone(...)
```

Arguments:

`...` Parameters passed via the inherited class constructor (defined in initialize and run via new).

Returns: New object of the current (inherited) class.

Method `get_attribute()`**:** Returns selected named simulator or attached attribute.

Usage:

```
ModelSimulator$get_attribute(param)
```

Arguments:

`param` Name of the parameter/attribute.

Returns: Selected parameter/attribute value.

Method `run()`**:** Runs a model simulator (function), stores the results, and creates a status log entry as a list.

Usage:

`ModelSimulator$run()`

Returns: A list representing a simulation log entry with a *successful* boolean and a status message template (with a placeholder for the sample identifier).

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

Examples

```
# Simulation model
model1 <- SimulationModel$new(
  time_steps = 10,
  model_attributes = c("time_steps", "a", "b"),
  params = list(a = 1:7)
)
model1$required_attributes <- model1$model_attributes
# Simulation function
test_simulator <- function(model) {
  sum(unlist(model$get_attributes(model$required_attributes)))
}
# Model simulator
simulator1 <- ModelSimulator$new(
  simulation_model = model1,
  simulation_function = test_simulator
)
simulator1$run()
model1$set_attributes(a = 1:10, b = 15)
model1$get_attributes(model1$required_attributes)
simulator1$run()
simulator1$results
```

---

poems *poems: Pattern-oriented ensemble modeling and simulation*

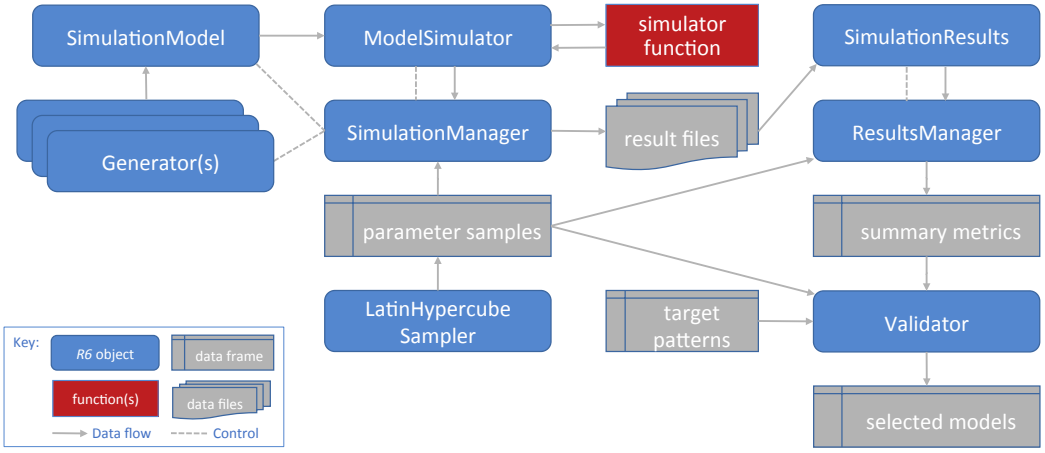
Description

The *poems* package provides a framework of interoperable R6 classes for building ensembles of viable models via the pattern-oriented modeling (POM) approach (Grimm et al., 2005). The package includes classes for encapsulating and generating model parameters, and managing the POM workflow. The workflow includes: model setup; generating model parameters via Latin hypercube sampling; running multiple sampled model simulations; collating summary results; and validating and selecting an ensemble of models that best match known patterns. By default, model validation and selection utilizes an approximate Bayesian computation (ABC) approach (Beaumont,

Zhang, & Balding, 2002), although alternative user-defined functionality could be employed. The package also includes a spatially explicit demographic population model simulation engine, which includes default functionality for density dependence, correlated environmental stochasticity, stage-based transitions, and distance-based dispersal. The user may customize the simulator by defining functionality for translocations, harvesting, mortality, and other processes, as well as defining the sequence order for the simulator processes. The framework could also be adapted for use with other model simulators by utilizing its extendable (inheritable) base classes.

Framework and workflow

The *poems* framework utilizes a hierarchy of extendable (inheritable) `R6` class objects that work together to manage a POM workflow for building an ensemble of simulation models.



The workflow is summarized in the following steps:

1. Create a simulation model template (a `SimulationModel` or inherited class object) with appropriate fixed parameters for the study domain. Also define a study region via the `Region` class if the simulations are to be spatially explicit.
2. Create generators (`Generator` or inherited class objects) for dynamically generating (non-singular) model parameters represented by data structures, such as arrays or lists.
3. Generate a data frame of sampled variable model parameters using the `LatinHypercubeSampler`. This will include singular model parameter values as well as input parameters for the generators.
4. Create a `SimulationManager` object configured with the simulation model (template), the generators, and the sample parameter data frame. Running this manager sets and runs the models via the simulator function for each set (row) of sampled parameters, utilising the generators when required. The results of each model simulation run are written to a file. A simulation log file is also created.
5. Create a `ResultsManager` object configured with the sample parameter data and result file details. Running this manager constructs a data frame of configured summary metrics, one row for each simulation result file. The manager utilizes the `SimulationResults` (or inherited) class to encapsulate, and dynamically generate additional derived, results. The metrics are generated via user-defined specifications and functions for calculating each metric from the results (objects).

6. Create a Validator object configured with the sample parameter data, summary metrics, and target (observed) pattern values for each metric. By default, the validator utilizes an approximate Bayesian computation (ABC) validation method via the abc library, although the validator (call) function can be configured to utilize other library or user-defined functions. Running the validator (with appropriate call function configuration) produces an ensemble of models (indices to sampled parameters) that were found to best match the targets. Diagnostic outputs may also be produced (depending on the call function and its configuration).
7. The selected models may then be utilized for further studies, such as alternative model scenarios or counterfactuals. This can be achieved by utilizing the selected subset of parameter samples to form inputs for further model simulations (by repeating the steps above).

Population modeling components

The spatially explicit demographic population model simulation engine and its associated classes are summarized by the following:

population_simulator function: The simulation engine's main function processes the model input parameters, controls the flow, calling other function modules as required, and returns the results of each simulation.

- population_density function: Module for configuring and performing density dependence calculations at each simulation time step. A user-defined function may be utilized.
- population_env_stoch function: Module for configuring and stochastically applying environmental variability to stage-based population transition rates at each simulation time step.
- population_transitions function: Module for configuring and performing stage-based demographic transitions of population abundances at each simulation time step.
- population_transformation function: Module for configuring and performing user-defined transformations to staged population abundances. This functionality is utilized when defining functions for `translocation`, `harvest`, `mortality`, or other custom transformative functions.
- population_dispersal function: Module for configuring and performing dispersal calculations at each simulation time step. A user-defined function may be utilized.
- population_results function: Module for configuring, initializing, and collating simulation results.
- PopulationModel class: Inherited from SimulationModel, this class encapsulates the input parameters utilized by the population_simulator.
- SimulatorReference class: This simple R6 class enables user-defined functionality to maintain persistent (attached) attributes and to write to the simulator results.
- SpatialCorrelation class: Provides functionality for generating parameters that can be utilized when optionally applying a spatial correlation within the simulator's environmental variability calculations.
- DispersalGenerator class: Inherited from Generator, this class provides functionality for generating distance-based dispersal parameters that can be utilized when performing dispersal calculations.
- DispersalFriction class: Provides functionality for adjusting the (equivalent) distance between population cells given a spatio-temporal frictional landscape. These adjustments may be utilized by the DispersalGenerator.

- PopulationResults class: Inherited from SimulationResults, this class encapsulates the results generated by the population_simulator, as well as dynamically generating additional derived results.

References

Beaumont, M. A., Zhang, W., & Balding, D. J. (2002). 'Approximate Bayesian computation in population genetics'. *Genetics*, vol. 162, no. 4, pp, 2025–2035.

Grimm, V., Revilla, E., Berger, U., Jeltsch, F., Mooij, W. M., Railsback, S. F., Thulke, H. H., Weiner, J., Wiegand, T., DeAngelis, D. L., (2005). 'Pattern-Oriented Modeling of Agent-Based Complex Systems: Lessons from Ecology'. *Science* vol. 310, no. 5750, pp. 987–991.

Examples

```
# Here we demonstrate building and running a simple population model. For a
# demonstration of the POM workflow with the model, see vignette("simple_example").

# Demonstration example region (U Island) and initial abundance
coordinates <- data.frame(
  x = rep(seq(177.01, 177.05, 0.01), 5),
  y = rep(seq(-18.01, -18.05, -0.01), each = 5)
)
template_raster <- Region$new(coordinates = coordinates)$region_raster # full extent
template_raster[][-c(7, 9, 12, 14, 17:19)] <- NA # make U Island
region <- Region$new(template_raster = template_raster)
initial_abundance <- seq(0, 300, 50)
raster::plot(region$raster_from_values(initial_abundance),
  main = "Initial abundance", xlab = "Longitude (degrees)",
  ylab = "Latitude (degrees)", zlim = c(0, 300), colNA = "blue"
)

# Set population model
pop_model <- PopulationModel$new(
  region = region,
  time_steps = 5,
  populations = 7,
  initial_abundance = initial_abundance,
  stage_matrix = matrix(c(
    0, 2.5, # Leslie/Lefkovitch matrix
    0.8, 0.5
  ), nrow = 2, ncol = 2, byrow = TRUE),
  carrying_capacity = rep(200, 7),
  density_dependence = "logistic",
  dispersal = (!diag(nrow = 7, ncol = 7)) * 0.05,
  result_stages = c(1, 2)
)

# Run single simulation
results <- population_simulator(pop_model)
results # examine
raster::plot(region$raster_from_values(results$abundance[, 5]),
  main = "Final abundance", xlab = "Longitude (degrees)",
```

```
  ylab = "Latitude (degrees)", zlim = c(0, 300), colNA = "blue"
)
```

`PopulationModel` *R6 class representing a population model*

Description

R6 class representing a spatially-explicit demographic-based population model. It extends the SimulationModel class with parameters for the population_simulator function. It inherits functionality for creating a nested model, whereby a nested template model with fixed parameters is maintained when a model is cloned for various sampled parameters. Also provided are extensions to the methods for checking the consistency and completeness of model parameters.

Super classes

poems::GenericClass -> poems::GenericModel -> poems::SpatialModel -> poems::SimulationModel -> PopulationModel

Public fields

`attached` A list of dynamically attached attributes (name-value pairs).

Active bindings

`simulation_function` Name (character string) or source path of the default simulation function, which takes a model as an input and returns the simulation results.

`model_attributes` A vector of model attribute names.

`region` A Region (or inherited class) object specifying the study region.

`coordinates` Data frame (or matrix) of X-Y population (WGS84) coordinates in longitude (degrees West) and latitude (degrees North) (get and set), or distance-based coordinates dynamically returned by region raster (get only).

`random_seed` Number to seed the random number generation for stochasticity.

`replicates` Number of replicate simulation runs.

`time_steps` Number of simulation time steps.

`years_per_step` Number of years per time step.

`populations` Number of population cells.

`stages` Number of life cycle stages.

`initial_abundance` Array (matrix) or raster (stack) of initial abundance values at each population cell (for each age/stage).

`stage_matrix` Matrix of transition (fecundity & survival) rates between stages at each time step (Leslie/Lefkovitch matrix).

`fecundity_mask` Matrix of 0-1 to indicate which (proportions) of transition rates refer to fecundity.

`fecundity_max` Maximum transition fecundity rate (in Leslie/Lefkovitch matrix).

`demographic_stochasticity` Boolean for choosing demographic stochasticity for transition, dispersal, harvest and/or other processes.

`standard_deviation` Standard deviation matrix (or single value) for applying environmental stochasticity to transition rates.

`correlation` Simulator-dependent attribute or list of attributes for describing/parameterizing the correlation strategy utilized when applying environmental stochasticity and/or other processes (see `population_simulator`).

`carrying_capacity` Array (matrix), or raster (stack) of carrying capacity values at each population cell (across time).

`density_dependence` Simulator-dependent function, attribute or list of attributes for describing/parameterizing the density dependence strategy utilized (see `population_simulator`).

`growth_rate_max` Maximum growth rate (utilized by density dependence processes).

`density_affects` Transition vital rates that are affected by density, including *"fecundity"*, *"survival"*, or a matrix of booleans or numeric (0-1) indicating vital rates affected (default is all).

`density_stages` Array of booleans or numeric (0-1) for each stage to indicate (the degree to) which stages are affected by density (default is 1 for all stages).

`translocation` Simulator-dependent function, attribute or list of attributes for describing/parameterizing translocation (management) strategies utilized (see `population_simulator`).

`harvest` Simulator-dependent function, attribute or list of attributes for describing/parameterizing a harvest (organism removal/hunting) strategy (see `population_simulator`).

`mortality` Simulator-dependent function, attribute or list of attributes to describe/parameterize a spatio-temporal mortality strategy (see `population_simulator`).

`dispersal` Simulator-dependent function, attribute or list of attributes for describing/parameterizing the dispersal (migration) strategy utilized (see `population_simulator`).

`dispersal_stages` Array of relative dispersal (0-1) for each stage to indicate the degree to which each stage participates in dispersal (default is 1 for all stages). This factor modifies dispersal proportion, not dispersal rate.

`dispersal_source_n_k` Simulator-dependent attribute for describing/parameterizing dispersal dependent on source population abundance divided by carrying capacity (see `population_simulator`).

`dispersal_target_k` Simulator-dependent attribute for describing/parameterizing dispersal dependent on target population carrying capacity (see `population_simulator`).

`dispersal_target_n` Simulator-dependent attribute (default is list with *threshold* and *cutoff*) of attributes for describing/parameterizing dispersal dependent on target population abundance (see `population_simulator`).

`dispersal_target_n_k` Simulator-dependent attribute (default is list with *threshold* and *cutoff*) of attributes for describing/parameterizing dispersal dependent on target population abundance/capacity (see `population_simulator`).

`abundance_threshold` Abundance threshold (that needs to be exceeded) for each population to persist.

`simulation_order` A vector of simulation process names in configured order of execution (default is "transition", "translocation", "harvest", "mortality", "dispersal", "results").

`results_selection` List of attributes to be included in the returned results of each simulation run, selected from: "abundance", "ema", "extirpation", "extinction_location", "harvested", "occupancy"; "summarize" or "replicate".

`result_stages` Array of booleans or numeric (0, 1, 2, ...) for each stage to indicate which stages are included/combined (each unique digit > 0; optionally named) in the results (default is 1 for all stages).

`attribute_aliases` A list of alternative alias names for model attributes (form: `alias = "attribute"`) to be used with the set and get attributes methods.

`template_model` Nested template model for fixed (non-sampled) attributes for shallow cloning.

`sample_attributes` Vector of sample attribute names (only).

`required_attributes` Vector of required attribute names (only), i.e. those needed to run a simulation.

`error_messages` A vector of error messages encountered when setting model attributes.

`warning_messages` A vector of warning messages encountered when setting model attributes.

Methods

Public methods:

- `PopulationModel$new()`
- `PopulationModel$list_consistency()`
- `PopulationModel$clone()`

Method `new()`**:** Initialization method sets default aliases and given attributes individually and/or from a list.

Usage:

```
PopulationModel$new(attribute_aliases = NULL, ...)
```

Arguments:

`attribute_aliases` A list of alternative alias names for model attributes (form: `alias = "attribute"`) to be used with the set and get attributes methods.

`...` Parameters passed via a *params* list or individually.

Method `list_consistency()`**:** Returns a boolean to indicate if (optionally selected or all) model attributes (such as dimensions) are consistent.

Usage:

```
PopulationModel$list_consistency(params = NULL)
```

Arguments:

`params` Optional array of parameter/attribute names.

Returns: List of booleans (or NAs) to indicate consistency of selected/all attributes.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# U Island example region
coordinates <- data.frame(
  x = rep(seq(177.01, 177.05, 0.01), 5),
  y = rep(seq(-18.01, -18.05, -0.01), each = 5)
)
template_raster <- Region$new(coordinates = coordinates)$region_raster # full extent
template_raster[][-c(7, 9, 12, 14, 17:19)] <- NA # make U Island
region <- Region$new(template_raster = template_raster)
# Harvest function
harvest <- list(
  rate = NA, # set later
  function(params) round(params$stage_abundance * (1 - params$rate))
)
harvest_rate_alias <- list(harvest_rate = "harvest$rate")
# Template model
stage_matrix <- matrix(c(
  0, 2.5, # Leslie/Lefkovitch matrix
  0.8, 0.5
), nrow = 2, ncol = 2, byrow = TRUE)
template_model <- PopulationModel$new(
  region = region,
  time_steps = 10, # years
  populations = region$region_cells, # 7
  stage_matrix = stage_matrix,
  harvest = harvest,
  results_selection = c("abundance", "harvested"),
  attribute_aliases = harvest_rate_alias
)
template_model$model_attributes
template_model$required_attributes
# Nested model
nested_model <- PopulationModel$new(template_model = template_model)
nested_model$incomplete_attributes()
nested_model$set_sample_attributes(
  initial_abundance = rep(10, 7),
  carrying_capacity = array(70:1, c(10, 7)),
  harvest_rate = 0.3
)
nested_model$inconsistent_attributes()
nested_model$carrying_capacity <- array(70:1, c(7, 10))
nested_model$is_consistent()
nested_model$is_complete()
nested_model$harvest$rate
```

PopulationResults *R6 class representing population simulator results.*

Description

R6 class encapsulating and dynamically generating spatially-explicit population_simulator results, as well as optional re-generated Generator outputs.

Super classes

poems::GenericClass -> poems::GenericModel -> poems::SpatialModel -> poems::SimulationResults -> PopulationResults

Public fields

attached A list of dynamically attached attributes (name-value pairs).

Active bindings

model_attributes A vector of model attribute names.

region A Region (or inherited class) object specifying the study region.

coordinates Data frame (or matrix) of X-Y population (WGS84) coordinates in longitude (degrees West) and latitude (degrees North) (get and set), or distance-based coordinates dynamically returned by region raster (get only).

time_steps Number of simulation time steps.

burn_in_steps Optional number of initial 'burn-in' time steps to be ignored.

occupancy_mask Optional binary mask array (matrix), data frame, or raster (stack) for each cell at each time-step of the simulation including burn-in.

trend_interval Optional time-step range (indices) for trend calculations (assumes indices begin after the burn-in when utilized).

abundance Population abundance across simulation time-steps (summary list or replicate array).

abundance_stages Population abundance for combined stages across simulation time-steps (list of summary lists or replicate arrays for each combined stage).

abundance_trend Trend or average Sen's slope of abundance (optionally across a time-step interval).

ema Array of population expected minimum abundance (EMA) across simulation time-steps.

extirpation Array of population extirpation times.

extinction_location The weighted centroid of cells occupied in the time-step prior to the extirpation of all populations (if it occurred).

harvested Number of animals harvested from each population across simulation time-steps (summary list or replicate array).

harvested_stages Number of animals harvested from each population for combined stages across simulation time-steps (list of summary lists or replicate arrays for each combined stage).

occupancy Array of the number of populations occupied at each time-step.

all Nested simulation results for all cells.

parent Parent simulation results for individual cells.

`default` Default value/attribute utilized when applying primitive metric functions (e.g. max) to the results.

`attribute_aliases` A list of alternative alias names for model attributes (form: `alias = "attribute"`) to be used with the set and get attributes methods.

`error_messages` A vector of error messages encountered when setting model attributes.

`warning_messages` A vector of warning messages encountered when setting model attributes.

Methods

Public methods:

- `PopulationResults$clone()`

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# U Island example region
coordinates <- data.frame(
  x = rep(seq(177.01, 177.05, 0.01), 5),
  y = rep(seq(-18.01, -18.05, -0.01), each = 5)
)
template_raster <- Region$new(coordinates = coordinates)$region_raster # full extent
template_raster[][-c(7, 9, 12, 14, 17:19)] <- NA # make U Island
region <- Region$new(template_raster = template_raster)
raster::plot(region$region_raster,
  main = "Example region (indices)",
  xlab = "Longitude (degrees)", ylab = "Latitude (degrees)",
  colNA = "blue"
)
# Sample results occupancy (ignore cell 2 in last 5 time steps)
occupancy_raster <- region$raster_from_values(array(1, c(7, 13)))
occupancy_raster[region$region_indices][2, 9:13] <- 0
occupancy_raster[region$region_indices]
# Population simulation example results
example_results <- list(abundance = t(apply(matrix(11:17), 1, function(n) {
  c(rep(n, 3), round(n * exp(-(0:9) / 2)))
})))
example_results$harvested <- round(example_results$abundance * 0.3)
example_results
# Population results object
pop_results <- PopulationResults$new(
  region = region,
  time_steps = 13,
  burn_in_steps = 3,
  occupancy_mask = occupancy_raster,
```

```
  trend_interval = 1:5
)
pop_results$get_attribute_names(all = TRUE)
# Clone (for each population simulation results)
results_clone <- pop_results$new_clone(results = example_results)
results_clone$all$get_attribute("abundance")
results_clone$get_attributes(c(
  "abundance", "all$abundance",
  "abundance_trend", "all$abundance_trend",
  "all$ema", # only defined for all
  "extirpation", "all$extirpation",
  "all$extinction_location", # only defined for all
  "harvested", "all$harvested",
  "occupancy", "all$occupancy"
))
```

`population_density` *Nested functions for population density dependence.*

Description

Modular functions for the population simulator for performing density dependent adjustments to transition rates.

Usage

```
population_density(
  populations,
  stage_matrix,
  fecundity_mask,
  fecundity_max,
  density_dependence,
  growth_rate_max,
  density_affects,
  density_stages,
  density_precision,
  simulator
)
```

Arguments

`populations`	Number of populations.
`stage_matrix`	Matrix of transition (fecundity & survival) rates between stages at each time step (Leslie/Lefkovitch matrix).
`fecundity_mask`	Matrix of 0-1 to indicate which (proportions) of transition rates refer to fecundity.
`fecundity_max`	Maximum transition fecundity rate (in Leslie/Lefkovitch matrix).

density_dependence

Density dependence can be "ceiling" (default), "logistic" (Ricker), or a user-defined function (optionally nested in a list with additional attributes) for adjusting transition rates: `function(params)`, where *params* is a list passed to the function containing:

`transition_array` 3D array of transition rates: stages by stages by populations.

`fecundity_mask` Matrix of 0-1 to indicate which (proportions) of transition rates refer to fecundity.

`fecundity_max` Maximum transition fecundity rate (in Leslie/Lefkovitch matrix).

`carrying_capacity` Array of carrying capacity values for each population.

`stage_abundance` Matrix of abundance for each stage (rows) and population (columns).

`population_abundance` Array of summed population abundances for all stages.

`density_abundance` Array of summed population abundances for stages affected by density.

`growth_rate_max` Maximum growth rate value or array for populations.

`occupied_indices` Array of indices for populations occupied at (current) time step.

`calculate_multipliers` Function (`function(growth_rates)`) for finding multipliers (when stages > 1) to apply to affected transitions that result in target growth rates (dominant eigenvalues).

`apply_multipliers` Function (`function(transition_array, multipliers)` for applying (when stages > 1) multipliers to the affected transition rates within a transition array (returns multiplied transition array).

`simulator` `SimulatorReference` object with dynamically accessible *attached* and *results* lists.

`additional attributes` Additional attributes when density dependence is optionally nested in a list.

returns an adjusted transition array for occupied populations

growth_rate_max

Maximum growth rate (utilized by density dependence processes).

density_affects

Matrix of booleans or numeric (0-1) indicating the transition vital rates affected by density (default is all).

density_stages Array of booleans or numeric (0,1) for each stage to indicate which stages are affected by density (default is all).

density_precision

Numeric precision of the calculated multipliers (used when stages > 1) applied to affected transition rates (default is 3 decimal places).

simulator `SimulatorReference` object with dynamically accessible *attached* and *results* lists.

Value

Density dependent calculation function, either:

`function(carrying_capacity, stage_abundance)` For ceiling density dependence function, OR

`function(transition_array, carrying_capacity, stage_abundance, occupied_indices)`
For user-defined density dependence function, where:

`transition_array` 3D array of transition rates: stages by stages by populations.

`carrying_capacity` Array of carrying capacity values for each population.

`stage_abundance` Matrix of abundance for each stage (rows) and population (columns).

`occupied_indices` Array of indices for populations occupied.

Examples

```
# Ceiling density dependence
stage_matrix <- array(c(0, 0.5, 0, 3, 0, 0.7, 4, 0, 0.8), c(3, 3))
fecundity_mask <- array(c(0, 0, 0, 1, 0, 0, 1, 0, 0), c(3, 3))
simulator <- SimulatorReference$new()
density_function <- population_density(
  populations = 7, stage_matrix = stage_matrix, fecundity_mask = fecundity_mask,
  fecundity_max = NULL, density_dependence = "ceiling",
  growth_rate_max = NULL, density_affects = NULL, density_stages = c(0, 1, 1),
  density_precision = NULL, simulator = simulator
)
carrying_capacity <- rep(10, 7)
stage_abundance <- matrix(c(
  7, 13, 0, 26, 0, 39, 47,
  2, 0, 6, 8, 0, 12, 13,
  0, 3, 4, 6, 0, 9, 10
), nrow = 3, ncol = 7, byrow = TRUE)

# Life cycle stages 2 and 3 (rows 2 and 3) all add up to 10 or less
density_function(carrying_capacity, stage_abundance)
```

---

population_dispersal *Nested functions for population dispersal.*

Description

Modular functions for the population simulator for performing dispersal of stage abundance at a specified time step via dispersal rates provided.

Usage

```
population_dispersal(
  replicates,
  time_steps,
```

```
  years_per_step,
  populations,
  demographic_stochasticity,
  density_stages,
  dispersal,
  dispersal_stages,
  dispersal_source_n_k = NULL,
  dispersal_target_k = NULL,
  dispersal_target_n = NULL,
  dispersal_target_n_k = NULL,
  simulator
)
```

Arguments

`replicates`	Number of replicate simulation runs.
`time_steps`	Number of simulation time steps.
`years_per_step`	Number of years per time step.
`populations`	Number of populations.
`demographic_stochasticity`	Boolean for optionally choosing demographic stochasticity for the transformation.
`density_stages`	Array of booleans or numeric (0,1) for each stage to indicate which stages are affected by density.
`dispersal`	Either a matrix of dispersal rates between populations (source columns to target rows) or a list of data frames of non-zero dispersal rates and indices for constructing a compact dispersal matrix, and optional changing rates over time (as per class `DispersalGenerator` *dispersal_data* attribute). Alternatively a user-defined function (optionally nested in a list with additional attributes) may be used: `function(params)`, where *params* is a list passed to the function containing: `replicates` Number of replicate simulation runs. `time_steps` Number of simulation time steps. `years_per_step` Number of years per time step. `populations` Number of populations. `stages` Number of life cycle stages. `demographic_stochasticity` Boolean for optionally choosing demographic stochasticity for the transformation. `density_stages` Array of booleans or numeric (0,1) for each stage to indicate which stages are affected by density. `dispersal_stages` Array of relative dispersal (0-1) for each stage to indicate the degree to which each stage participates in dispersal. This factor modifies dispersal proportion, not dispersal rate. `dispersal_source_n_k` Dispersal proportion (p) density dependence via source population abundance divided by carrying capacity (n/k), where p is reduced via a linear slope (defined by two list items) from n/k <= *cutoff* (p = 0) to n/k >= *threshold*.

`dispersal_target_k` Dispersal rate (r) density dependence via target population carrying capacity (k), where r is reduced via a linear slope (through the origin) when k <= *threshold*.

`dispersal_target_n` Dispersal rate (r) density dependence via target population abundance (n), where r is reduced via a linear slope (defined by two list items) from n >= *threshold* to n <= *cutoff* (r = 0) or vice versa.

`dispersal_target_n_k` Dispersal rate (r) density dependence via target population abundance divided by carrying capacity (n/k), where r is reduced via a linear slope (defined by two list items) from n/k >= *threshold* to n/k <= *cutoff* (r = 0) or vice versa.

`r` Simulation replicate.

`tm` Simulation time step.

`carrying_capacity` Array of carrying capacity values for each population at time step.

`stage_abundance` Matrix of abundance for each stage (rows) and population (columns) at time step.

`occupied_indices` Array of indices for populations occupied at time step.

`simulator` `SimulatorReference` object with dynamically accessible *attached* and *results* lists.

`additional attributes` Additional attributes when the transformation is optionally nested in a list.

returns the post-dispersal abundance matrix

`dispersal_stages`

Array of relative dispersal (0-1) for each stage to indicate the degree to which each stage participates in dispersal (default is 1 for all stages). This factor modifies dispersal proportion, not dispersal rate.

`dispersal_source_n_k`

Dispersal proportion (p) density dependence via source population abundance divided by carrying capacity (n/k), where p is reduced via a linear slope (defined by two list items) from n/k <= *cutoff* (p = 0) to n/k >= *threshold* or vice versa.

`dispersal_target_k`

Dispersal rate (r) density dependence via target population carrying capacity (k), where r is reduced via a linear slope (through the origin) when k <= *threshold*.

`dispersal_target_n`

Dispersal rate (r) density dependence via target population abundance (n), where r is reduced via a linear slope (defined by two list items) from n >= *threshold* to n <= *cutoff* (r = 0) or visa-versa.

`dispersal_target_n_k`

Dispersal rate (r) density dependence via target population abundance divided by carrying capacity (n/k), where r is reduced via a linear slope (defined by two list items) from n/k >= *threshold* to n/k <= *cutoff* (r = 0) or vice versa.

`simulator` `SimulatorReference` object with dynamically accessible *attached* and *results* lists.

Value

Dispersal function: `function(r, tm, carrying_capacity, stage_abundance, occupied_indices)`, where:

`r` Simulation replicate.

`tm` Simulation time step.

`carrying_capacity` Array of carrying capacity values for each population at time step.

`stage_abundance` Matrix of abundance for each stage (rows) and population (columns) at time step.

`occupied_indices` Array of indices for populations occupied at time step.

`returns` New stage abundance matrix with dispersal applied.

Examples

```
# User-defined dispersal: one-quarter of dispersing stages move one population over
simulator <- SimulatorReference$new()
example_function <- function(params) {
  params$simulator$attached$params <- params # attach to reference object
  emigrants <- round(params$stage_abundance * params$dispersal_stages * 0.25)
  return(params$stage_abundance - emigrants + emigrants[, c(7, 1:6)])
}
dispersal_function <- population_dispersal(
  replicates = 4,
  time_steps = 10,
  years_per_step = 1,
  populations = 7,
  demographic_stochasticity = TRUE,
  density_stages = c(0, 1, 1),
  dispersal = example_function,
  dispersal_stages = c(0, 1, 0.5),
  dispersal_source_n_k = list(cutoff = -0.5, threshold = 1.5),
  dispersal_target_k = 5,
  dispersal_target_n = list(threshold = 10, cutoff = 15),
  simulator = simulator
)
carrying_capacity <- rep(10, 7)
stage_abundance <- matrix(
  c(
    7, 13, 0, 26, 0, 39, 47,
    2, 0, 6, 8, 0, 12, 13,
    0, 3, 4, 6, 0, 9, 10
  ),
  nrow = 3,
  ncol = 7,
  byrow = TRUE
)
occupied_indices <- (1:7)[-5]
dispersal_function(
  r = 2, tm = 6, carrying_capacity, stage_abundance,
  occupied_indices
```

```
)
```

`population_env_stoch` *Nested functions for population environmental stochasticity.*

Description

Modular functions for the population simulator for performing correlated environmentally stochastic adjustments to transition rates.

Usage

```
population_env_stoch(
  populations,
  fecundity_matrix,
  fecundity_max,
  survival_matrix,
  standard_deviation,
  correlation
)
```

Arguments

`populations`	Number of populations.
`fecundity_matrix`	Matrix of transition fecundity rates (Leslie/Lefkovitch matrix with non-zero fecundities only).
`fecundity_max`	Maximum transition fecundity rate (in Leslie/Lefkovitch matrix).
`survival_matrix`	Matrix of transition survival rates (Leslie/Lefkovitch matrix with non-zero survivals only).
`standard_deviation`	Standard deviation matrix for applying environmental stochasticity to transition rates.
`correlation`	List containing either an environmental correlation matrix (correlation_matrix), a pre-calculated transposed (Cholesky) decomposition matrix (t_decomposition_matrix), or a compact transposed (Cholesky) decomposition matrix (t_decomposition_compact_matrix) and a corresponding map of population indices (t_decomposition_compact_map), as per *SpatialCorrelation* class attributes.

Value

Environmental stochasticity calculation function: `function(fecundity_array, survival_array, occupied_indices)`, where:

`fecundity_array` 3D array of fecundity rates (*stages* rows by *stages* columns by *populations* deep).

`survival_array` 3D array of survival rates (*stages* rows by *stages* columns by *populations* deep).

`occupied_indices` Array of indices for those populations occupied.

`returns` List containing stochastically varied fecundity and survival arrays.

Examples

```
fecundity_matrix <- array(c(0, 0, 0, 3, 0, 0, 4, 0, 0), c(3, 3))
survival_matrix <- array(c(0, 0.5, 0, 0, 0, 0.7, 0, 0, 0.8), c(3, 3))
standard_deviation <- (fecundity_matrix + survival_matrix) * 0.3
variation_array <- array(rep(seq(0.85, 1.15, 0.05), each = 9), c(3, 3, 7))
fecundity_array <- array(fecundity_matrix, c(3, 3, 7)) * variation_array
survival_array <- array(survival_matrix, c(3, 3, 7)) * variation_array
stage_abundance <- matrix(c(
  7, 13, 0, 26, 0, 39, 47,
  2, 0, 6, 8, 0, 12, 13,
  0, 3, 4, 6, 0, 9, 10
), nrow = 3, ncol = 7, byrow = TRUE)
occupied_indices <- (1:7)[-5]
env_stoch_function <- population_env_stoch(
  populations = 7, fecundity_matrix, fecundity_max = NULL, survival_matrix,
  standard_deviation, correlation = NULL
)
env_stoch_function(fecundity_array, survival_array, occupied_indices)
```

---

population_results *Nested functions for initializing, calculating and collecting population simulator results.*

Description

Modular functions for the population simulator for initializing, calculating and collecting simulator results.

Usage

```
population_results(
  replicates,
  time_steps,
  coordinates,
  initial_abundance,
  results_selection = NULL,
  result_stages = NULL
)
```

Arguments

`replicates`	Number of replicate simulation runs.
`time_steps`	Number of simulation time steps.
`coordinates`	Data frame (or matrix) of X-Y population coordinates.
`initial_abundance`	Matrix of initial abundances at each stage (in rows) for each population (in columns).
`results_selection`	List of results selection from: "abundance" (default), "ema", "extirpation", "extinction_location", "harvested", "occupancy"; "summarize" (default) or "replicate".
`result_stages`	Array of booleans or numeric (0, 1, 2, ...) for each stage to indicate which stages are included/combined (each unique digit > 0; optionally named) in the results (default is 1 for all stages).

Value

List of result functions:

`initialize_attributes = function())` Constructs and returns an initialized nested list for the selected result attributes.

`initialize_replicate = function(results)` Initializes and returns nested result attributes at the start of each replicate.

`calculate_at_timestep = function(r, tm, stage_abundance, harvested, results)` Appends and calculates (non-NULL) results and returns nested result attributes at the end of each time step (tm) within replicate (r).

`finalize_attributes = function(results)` Finalizes result calculations at the end of the simulation.

Examples

```
coordinates <- array(c(1:4, 4:1), c(7, 2))
initial_abundance <- matrix(c(
  7, 13, 0, 26, 0, 39, 47,
  2, 0, 6, 8, 0, 12, 13,
  0, 3, 4, 6, 0, 9, 10
), nrow = 3, ncol = 7, byrow = TRUE)
results_selection <- c(
  "abundance", "ema", "extirpation",
  "extinction_location", "harvested", "occupancy"
)
result_functions <- population_results(
  replicates = 1, time_steps = 10, coordinates, initial_abundance,
  results_selection = results_selection, result_stages = c(0, 1, 1)
)
result_functions$initialize_attributes()
```

population_simulator *Runs a stage-based demographic population model simulation.*

Description

Simulates a stage-based demographic population model and returns simulation results across multiple replicate runs. Processes run at each simulation time-step include:

1. Density dependence calculations (ceiling, logistic, or user-defined)
2. Environmental stochasticity calculations
3. Stage transition (stochastic) calculations
4. Translocation calculations (user-defined)
5. Harvest calculations (user-defined)
6. Mortality calculations (user-defined)
7. Dispersal calculations (default or user-defined)
8. Results collection

Usage

```
population_simulator(inputs)
```

Arguments

inputs	Nested list/object with named elements:
	random_seed Number to seed the random number generation for stochasticity.
	replicates Number of replicate simulation runs (default is 1).
	time_steps Number of simulation time steps. Required input.
	years_per_step Number of years per time step (default is 1).
	populations Number of populations. Required input.
	coordinates Data frame (or matrix) of X-Y population coordinates.
	stages Number of life cycle stages.
	region A Region object defining the study region.
	initial_abundance Array (or matrix) of initial abundances (at each stage in rows) for each population (in columns). If there is only one stage and a region object is attached, then initial abundance may be provided in the form of a raster with the same specs as the region raster. A vector can be provided that will be forced to a matrix. Required input.
	stage_matrix Matrix of transition (fecundity & survival) rates between stages at each time step (Leslie/Lefkovitch matrix). Required input.
	fecundity_mask Matrix of 0-1 to indicate which (proportions) of transition rates refer to fecundity.
	fecundity_max Maximum transition fecundity rate (in Leslie/Lefkovitch matrix).

`demographic_stochasticity` Boolean for choosing demographic stochasticity for transition, dispersal, harvest and/or other processes (default is TRUE).

`standard_deviation` Standard deviation matrix (or single value) for applying environmental stochasticity to transition rates.

`correlation` List containing either an environmental correlation matrix (correlation_matrix), a pre-calculated transposed (Cholesky) decomposition matrix (t_decomposition_matrix), or a compact transposed (Cholesky) decomposition matrix (t_decomposition_compact_matrix) and a corresponding map of population indices (t_decomposition_compact_map), as per `SpatialCorrelation` class attributes.

`carrying_capacity` Array (matrix) of carrying capacity values at each population cell (*populations* rows by *time_steps* columns when across time). Required input.

`density_dependence` Density dependence can be "ceiling" (default), "logistic" (Ricker), or a user-defined function (optionally nested in a list with additional attributes) for adjusting transition rates: `function(params)`, where *params* is a list passed to the function containing:

- `transition_array` 3D array of transition rates: stages by stages by populations.
- `fecundity_mask` Matrix of 0-1 to indicate which (proportions) of transition rates refer to fecundity.
- `fecundity_max` Maximum transition fecundity rate (in Leslie/Lefkovitch matrix).
- `carrying_capacity` Array of carrying capacity values for each population.
- `stage_abundance` Matrix of abundances for each stage (rows) and population (columns).
- `population_abundance` Array of summed population abundances for all stages.
- `density_abundance` Array of summed population abundances for stages affected by density.
- `growth_rate_max` Maximum growth rate value or array for populations.
- `occupied_indices` Array of indices for populations occupied at (current) time step.
- `calculate_multipliers` Function (`function(growth_rates)`) for finding multipliers (when stages > 1) to apply to affected transitions that result in target growth rates (dominant eigenvalues).
- `apply_multipliers` Function (`function(transition_array, multipliers)`) for applying multipliers (when stages > 1) to the affected transition rates within a transition array (returns multiplied array).
- `simulator` `SimulatorReference` object with dynamically accessible *attached* and *results* lists.
- `optional attributes` Additional numeric attributes when density dependence is optionally nested in a list.

returns a transformed transition 3D array

`growth_rate_max` Maximum growth rate (utilized by density dependence processes).

`density_affects` Matrix of booleans or numeric (0-1) indicating the transition vital rates affected by density (default is all).

`density_stages` Array of booleans or numeric (0,1) for each stage to indicate which stages are affected by density (default is all).

`density_precision` Numeric precision of the calculated multipliers (used when stages > 1) applied to affected transition rates (default is 3 decimal places).

`translocation` An optional user-defined function (optionally nested in a list with additional attributes) for applying translocation or spatio-temporal management (to abundances): `function(params)`, where *params* is a list passed to the function containing:

`replicates` Number of replicate simulation runs.

`time_steps` Number of simulation time steps.

`years_per_step` Number of years per time step.

`populations` Number of populations.

`stages` Number of lifecycle stages.

`demographic_stochasticity` Boolean for optionally choosing demographic stochasticity for the transformation.

`density_stages` Array of booleans or numeric (0,1) for each stage to indicate which stages are affected by density.

`r` Simulation replicate.

`tm` Simulation time step.

`carrying_capacity` Array of carrying capacity values for each population at time step.

`stage_abundance` Matrix of (current) abundance for each stage (rows) and population (columns) at time step.

`occupied_indices` Array of indices for populations occupied at (current) time step.

`simulator` `SimulatorReference` object with dynamically accessible *attached* and *results* lists.

`additional attributes` Additional attributes when the transformation is optionally nested in a list.

returns a transformed stage abundance matrix (or a list with stage abundance and carrying capacity)

`harvest` An optional user-defined function (optionally nested in a list with additional attributes) for applying harvesting (to abundances): `function(params)` as per translocation.

`mortality` An optional user-defined function (optionally nested in a list with additional attributes) for applying mortality (to abundances): `function(params)` as per translocation.

`dispersal` Either a matrix of dispersal rates between populations (source columns to target rows) or a list of data frames of non-zero dispersal rates and indices for constructing a compact dispersal matrix, and optional changing rates

over time (as per class DispersalGenerator *dispersal_data* attribute). Alternatively a user-defined function (optionally nested in a list with additional attributes) may be used: `function(params)`, where *params* is a list passed to the function containing:

`replicates` Number of replicate simulation runs.

`time_steps` Number of simulation time steps.

`years_per_step` Number of years per time step.

`populations` Number of populations.

`demographic_stochasticity` Boolean for optionally choosing demographic stochasticity for the transformation.

`density_stages` Array of booleans or numeric (0,1) for each stage to indicate which stages are affected by density.

`dispersal_stages` Array of relative dispersal (0-1) for each stage to indicate the degree to which each stage participates in dispersal. This factor modifies dispersal proportion, not dispersal rate.

`r` Simulation replicate.

`tm` Simulation time step.

`carrying_capacity` Array of carrying capacity values for each population at time step.

`stage_abundance` Matrix of abundance for each stage (rows) and population (columns) at time step.

`occupied_indices` Array of indices for populations occupied at time step.

`simulator` SimulatorReference object with dynamically accessible *attached* and *results* lists.

`additional attributes` Additional attributes when the transformation is optionally nested in a list.

returns the post-dispersal abundance matrix

`dispersal_stages` Array of relative dispersal (0-1) for each stage to indicate the degree to which each stage participates in dispersal (default is 1 for all stages). This factor modifies dispersal proportion, not dispersal rate.

`dispersal_source_n_k` Dispersal proportion (p) density dependence via source population abundance divided by carrying capacity (n/k), where p is reduced via a linear slope (defined by two list items) from n/k <= *cutoff* (p = 0) to n/k >= *threshold* (aliases: *dispersal_n_k_cutoff* & *dispersal_n_k_threshold*).

`dispersal_target_k` Dispersal rate (r) density dependence via target population carrying capacity (k), where r is reduced via a linear slope (through the origin) when k <= *threshold* (alias: *dispersal_k_threshold*).

`dispersal_target_n` Dispersal rate (r) density dependence via target population abundance (n), where r is reduced via a linear slope (defined by two list items) from n >= *threshold* to n <= *cutoff* (r = 0) or vice versa (aliases: *dispersal_n_threshold* & *dispersal_n_cutoff*).

`dispersal_target_n_k` Dispersal rate (r) density dependence via target population abundance divided by carrying capacity (n/k), where r is reduced via a linear slope (defined by two list items) from n/k >= *threshold* to n/k <= *cutoff* (r = 0) or vica versa.

`abundance_threshold` Abundance threshold (that needs to be exceeded) for each population to persist.

`simulation_order` A vector of simulation process names in configured order of execution (default is "transition", "translocation", "harvest" (plus harvested results), "mortality", "dispersal", "results" (except harvested).

`additional transformation functions` Additional user-defined abundance transformation functions (optionally nested in lists with additional attributes) are utilised when listed in *simulation_order* (function as per translocation).

`results_selection` List of results selection from: "abundance" (default), "ema", "extirpation", "extinction_location", "harvested", "occupancy"; "summarize" (default) or "replicate".

`result_stages` Array of booleans or numeric (0, 1, 2, ...) for each stage to indicate which stages are included/combined (each unique digit > 0; optionally named) in the results (default is 1 for all stages).

Value

Selected simulation results as a nested list summarized (mean, sd, min, max) across multiple replicates (default), or 2-3D arrays including results for each replicate:

`abundance` Matrix or 3D array of simulation abundance: *populations* rows by *time_steps* columns (by *replicates* deep).

`abundance_stages` List of matrices or 3D arrays of simulation abundance for unique stage combinations when present: each *populations* rows by *time_steps* columns (by *replicates* deep).

`all$abundance` Array or matrix of total abundance across populations: *time_steps* (rows by *replicates* columns).

`all$abundance_stages` List of arrays or matrices of total abundance across populations for unique stage combinations when present: each *time_steps* (rows by *replicates* columns).

`all$ema` Array of expected minimum abundance at each time step (averaged across replicates).

`extirpation` Array or matrix of extirpation times: *populations* (rows by *replicates* columns).

`all$extirpation` Array of extirpation time across populations for each replicate.

`all$extinction_location` The weighted centroid of cells occupied in the time-step prior to the extirpation of all populations (if it occurred) for each replicate.

`harvested` Matrix or 3D array of individuals harvested: *populations* rows by *time_steps* columns (by *replicates* deep).

`harvested_stages` List of matrices or 3D arrays of individuals harvested for unique stage combinations when present: each *populations* rows by *time_steps* columns (by *replicates* deep).

`all$harvested` Array or matrix of individuals harvested across populations: *time_steps* (rows by *replicates* columns).

`all$harvested_stages` List of arrays or matrices of individuals harvested across populations for unique stage combinations when present: each *time_steps* (rows by *replicates* columns).

`all$occupancy` Array or matrix of the number of populations occupied at each time-step: *time_steps* (rows by *replicates* columns).

`additional results` Additional results may be attached via user-defined functions (using `params$simulator$results`).

Examples

```
# U Island example region
coordinates <- data.frame(
  x = rep(seq(177.01, 177.05, 0.01), 5),
  y = rep(seq(-18.01, -18.05, -0.01), each = 5)
)
template_raster <- Region$new(coordinates = coordinates)$region_raster # full extent
template_raster[][-c(7, 9, 12, 14, 17:19)] <- NA # make U Island
region <- Region$new(template_raster = template_raster)
# Harvest function
harvest <- list(
  rate = 0.3,
  function(params) round(params$stage_abundance * (1 - params$rate))
)
# Population model
stage_matrix <- matrix(c(
  0, 2.5, # Leslie/Lefkovitch matrix
  0.8, 0.5
), nrow = 2, ncol = 2, byrow = TRUE)
pop_model <- PopulationModel$new(
  region = region,
  time_steps = 10, # years
  populations = region$region_cells, # 7
  stage_matrix = stage_matrix,
  initial_abundance = rep(10, 7),
  carrying_capacity = array(70:1, c(7, 10)),
  harvest = harvest,
  results_selection = c("abundance", "harvested")
)
# Simulations
population_simulator(pop_model) # model
inputs <- pop_model$get_attributes()
population_simulator(inputs) # list
```

---

population_transformation

Nested functions for a user-defined population abundance (and capacity) transformation.

Description

Modular functions for the population simulator for performing a transformation of the stage abundance (and optionally carrying capacity) at a specified time step via a user-defined function.

Usage

```
population_transformation(
  replicates,
```

```
  time_steps,
  years_per_step,
  populations,
  demographic_stochasticity,
  density_stages,
  transformation,
  simulator,
  name = "transformation"
)
```

Arguments

`replicates`	Number of replicate simulation runs.
`time_steps`	Number of simulation time steps.
`years_per_step`	Number of years per time step.
`populations`	Number of populations.
`demographic_stochasticity`	Boolean for optionally choosing demographic stochasticity for the transformation.
`density_stages`	Array of booleans or numeric (0,1) for each stage to indicate which stages are affected by density.
`transformation`	A user-defined function (optionally nested in a list with additional attributes) for performing transformation: `function(params)`, where *params* is a list passed to the function containing: `replicates` Number of replicate simulation runs. `time_steps` Number of simulation time steps. `years_per_step` Number of years per time step. `populations` Number of populations. `stages` Number of life cycle stages. `demographic_stochasticity` Boolean for optionally choosing demographic stochasticity for the transformation. `density_stages` Array of booleans or numeric (0,1) for each stage to indicate which stages are affected by density. `r` Simulation replicate. `tm` Simulation time step. `carrying_capacity` Array of carrying capacity values for each population at time step. `stage_abundance` Matrix of (current) abundance for each stage (rows) and population (columns) at time step. `occupied_indices` Array of indices for populations occupied at (current) time step. `simulator` `SimulatorReference` object with dynamically accessible *attached* and *results* lists. `additional attributes` Additional attributes when the transformation is optionally nested in a list.

	returns a transformed stage abundance matrix (or a list with stage abundance and carrying capacity)
`simulator`	`SimulatorReference` object with dynamically accessible *attached* and *results* lists.
`name`	Optional name for the transformation (default is "transformation").

Value

Abundance (and capacity) transformation function: `function(r, tm, carrying_capacity, stage_abundance, occupied_indices)`, where:

`r` Simulation replicate.

`tm` Simulation time step.

`carrying_capacity` Array of carrying capacity values for each population at time step.

`stage_abundance` Matrix of abundance for each stage (rows) and population (columns) at time step.

`occupied_indices` Array of indices for populations occupied at time step.

`returns` List with transformed stage abundance matrix (and optionally carrying capacity).

Examples

```
simulator <- SimulatorReference$new()
# Example transformation: a random population is chosen for a severe disturbance event
# (only life cycle stage 3 individuals survive)
disturbance_function <- function(params) {
  params$simulator$attached$params <- params # attach to reference object
  random_population <- sample(params$occupied_indices, 1)
  new_stage_abundance <- params$stage_abundance
  new_stage_abundance[1:2, random_population] <- 0
  return(new_stage_abundance)
}
transformation_function <- population_transformation(
  replicates = 4, time_steps = 10, years_per_step = 1,
  populations = 7, demographic_stochasticity = TRUE,
  density_stages = c(0, 1, 1), transformation = disturbance_function,
  simulator
)
carrying_capacity <- rep(10, 7)
carrying_capacity <- rep(10, 7)
stage_abundance <- matrix(c(
  7, 13, 0, 26, 0, 39, 47,
  2, 0, 6, 8, 0, 12, 13,
  0, 3, 4, 6, 0, 9, 10
), nrow = 3, ncol = 7, byrow = TRUE)
occupied_indices <- (1:7)[-5]
transformation_function(
  r = 2, tm = 6, carrying_capacity, stage_abundance,
  occupied_indices
)
```

population_transitions *Nested functions for stage-based population transitions.*

Description

Modular functions for the population simulator for performing staged-based (Leslie/Lefkovitch matrix) transitions via 3D survival and fecundity arrays.

Usage

```
population_transitions(
  populations,
  demographic_stochasticity,
  fecundity_matrix,
  fecundity_max,
  survival_matrix
)
```

Arguments

`populations`	Number of populations.
`demographic_stochasticity`	Boolean for choosing demographic stochasticity for transitions.
`fecundity_matrix`	Matrix of transition fecundity rates (Leslie/Lefkovitch matrix with non-zero fecundities only).
`fecundity_max`	Maximum transition fecundity rate (in Leslie/Lefkovitch matrix).
`survival_matrix`	Matrix of transition survival rates (Leslie/Lefkovitch matrix with non-zero survivals only).

Value

Transition calculation function: `function(fecundity_array, survival_array, stage_abundance, occupied_indices)`, where:

`fecundity_array` 3D array of fecundity rates (*stages* rows by *stages* columns by *populations* deep).

`survival_array` 3D array of survival rates (*stages* rows by *stages* columns by *populations* deep).

`stage_abundance` Matrix of stage abundances for each population at time step (*stages* rows by *populations* columns).

`occupied_indices` Array of indices for those populations occupied.

`returns` Transitioned stage abundances.

Examples

```
# Deterministic transition (no stochasticity)
fecundity_matrix <- array(c(0, 0, 0, 3, 0, 0, 4, 0, 0), c(3, 3))
survival_matrix <- array(c(0, 0.5, 0, 0, 0, 0.7, 0, 0, 0.8), c(3, 3))
variation_array <- array(rep(seq(0.85, 1.15, 0.05), each = 9), c(3, 3, 7))
fecundity_array <- array(fecundity_matrix, c(3, 3, 7)) * variation_array
survival_array <- array(survival_matrix, c(3, 3, 7)) * variation_array
stage_abundance <- matrix(c(
  7, 13, 0, 26, 0, 39, 47,
  2, 0, 6, 8, 0, 12, 13,
  0, 3, 4, 6, 0, 9, 10
), nrow = 3, ncol = 7, byrow = TRUE)
occupied_indices <- (1:7)[-5]
transition_function <- population_transitions(
  populations = 7, demographic_stochasticity = FALSE,
  fecundity_matrix = fecundity_matrix, fecundity_max = NULL,
  survival_matrix = survival_matrix
)
transition_function(
  fecundity_array, survival_array, stage_abundance,
  occupied_indices
)
```

`Region` *R6 class representing a study region.*

Description

R6 class representing a study region of spatial grid cells defined via a list of longitude/latitude cell coordinates (WGS84), or a *RasterLayer* object (see raster).

Super class

poems::GenericClass -> Region

Public fields

`attached` A list of dynamically attached attributes (name-value pairs).

Active bindings

`coordinates` Data frame (or matrix) of X-Y population (WGS84) coordinates in longitude (degrees West) and latitude (degrees North) (get and set), or distance-based coordinates dynamically returned by region raster (get only).

`region_raster` A *RasterLayer* object (see raster) defining the region with finite values (NAs elsewhere).

`use_raster` Boolean to indicate that a raster is to be used to define the region (default TRUE).

`strict_consistency` Boolean to indicate that, as well as resolution, extent and CRS, consistency checks also ensure that a raster's finite/occupiable cells are the same or a subset of those defined by the region (default TRUE).

`region_cells` Dynamically calculated number of region coordinates or raster cells with finite/non-NA values.

`region_indices` Dynamically calculated region indices for raster cells with finite/non-NA values (all if not a raster).

Methods

Public methods:

- `Region$new()`
- `Region$raster_is_consistent()`
- `Region$raster_from_values()`
- `Region$clone()`

Method `new()`**:** Initialization method sets coordinates or raster for region.

Usage:

```
Region$new(
  coordinates = NULL,
  template_raster = NULL,
  region_raster = NULL,
  use_raster = TRUE,
  ...
)
```

Arguments:

`coordinates` Data frame (or matrix) of X-Y coordinates (WGS84) in longitude (degrees West) and latitude (degrees North).

`template_raster` A *RasterLayer* object (see `raster`) defining the region with example finite values (NAs elsewhere)

`region_raster` A *RasterLayer* object (see `raster`) defining the region with finite cell indices (NAs elsewhere).

`use_raster` Boolean to indicate that a raster is to be used to define the region (default TRUE).

`...` Additional parameters passed individually.

Method `raster_is_consistent()`**:** Returns a boolean to indicate if a raster is consistent with the region raster (matching extent, resolution, and finite/NA cells).

Usage:

```
Region$raster_is_consistent(check_raster)
```

Arguments:

`check_raster` A *RasterLayer*, *RasterStack* or *RasterBrick* object (see `raster`) to check for consistency with the region raster.

Returns: Boolean to indicate if the raster is consistent with the region raster.

Method `raster_from_values()`**:** Converts an array (or matrix) of values into a raster (or stack) consistent with the region raster (matching extent, resolution, and finite/NA cells).

Usage:

```
Region$raster_from_values(values)
```

Arguments:

`values` An array (or matrix) of values to be placed in the raster (or stack) having dimensions consistent with the region cell number.

Returns: A *RasterLayer* (or *RasterStack/Brick*) object consistent with the region raster.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# U Island example region
coordinates <- data.frame(
  x = rep(seq(177.01, 177.05, 0.01), 5),
  y = rep(seq(-18.01, -18.05, -0.01), each = 5)
)
template_raster <- Region$new(coordinates = coordinates)$region_raster # full extent
template_raster[][-c(7, 9, 12, 14, 17:19)] <- NA # make U Island
region <- Region$new(template_raster = template_raster)
raster::plot(region$region_raster,
  main = "Example region (cell indices)",
  xlab = "Longitude (degrees)", ylab = "Latitude (degrees)",
  colNA = "blue"
)
region$region_cells
region$coordinates
# Generate value layers
value_brick <- region$raster_from_values(array(8:28, c(7, 3)))
raster::plot(value_brick,
  main = "Example value layers",
  xlab = "Longitude (degrees)", ylab = "Latitude (degrees)",
  colNA = "blue"
)
value_brick[region$region_indices]
```

`ResultsManager` *R6 class representing a results manager.*

Description

R6 class to represent a manager for generating summary metrics and/or matrices from simulation results, as well as optionally regenerating values via generators.

Super classes

poems::GenericClass -> poems::GenericManager -> ResultsManager

Public fields

attached A list of dynamically attached attributes (name-value pairs).

Active bindings

sample_data A data frame of sampled parameters for each simulation/result.

simulation_results An object of a class inherited from the SimulationResults class for encapsulating and dynamically generating simulation results.

generators A list of generators (Generator or inherited class) objects for (optionally) regenerating simulation model values.

result_attachment_functions A list of functions for attaching intermediate values to the simulation results prior to generation.

summary_metrics An array of names for summary metrics, each of which are calculated as single values for each simulation. These should refer to list names for the summary functions.

summary_matrices An array of names for summary matrices, each of which are calculated as a single matrix row for each simulation. These should refer to list names for the summary functions.

summary_functions A list of functions, result attributes, or constants for transforming individual simulation results to single summary metric values stored in the metric data frame, or to matrix rows stored in the summary matrix list.

summary_metric_data A data frame of generated summary metrics (one row per simulation).

summary_matrix_list A list of generated matrices of summary results (each having one row per simulation).

summary_matrix_weighted_averages A list of calculated weighted averages for each of the summary matrices (using the sample data *weight* column).

parallel_cores Number of cores for running the simulations in parallel.

results_dir Results directory path.

results_ext Result file extension (default is .RData).

results_filename_attributes A vector of: prefix (optional); attribute names (from the sample data frame); postfix (optional); utilized to construct results filenames.

error_messages A vector of error messages encountered when setting model attributes.

warning_messages A vector of warning messages encountered when setting model attributes.

Methods

Public methods:

- ResultsManager$new()
- ResultsManager$generate()
- ResultsManager$calculate_result_attachments()

- ResultsManager$calculate_summaries()
- ResultsManager$log_generation()
- ResultsManager$calculate_summary_weighted_averages()
- ResultsManager$clone()

Method `new()`**:** Initialization method optionally copies attributes from a simulation (results) manager, sets any included attributes (*sample_data*, *simulation_results*, *generators*, *result_attachment_functions*, *summary_metrics*, *summary_functions*, *parallel_cores*, *results_dir*, *results_ext*, *results_filename_attributes*), and attaches other attributes individually listed.

Usage:

```
ResultsManager$new(simulation_manager = NULL, ...)
```

Arguments:

`simulation_manager` Optional SimulationManager object (or an object inherited from the GenericManager class), from which simulation attributes can be copied.

`...` Parameters listed individually.

Method `generate()`**:** Generates the summary metric data and/or matrix list via the summary functions for each simulation sample, and creates/writes a generation log.

Usage:

```
ResultsManager$generate(results_dir = NULL)
```

Arguments:

`results_dir` Results directory path (must be present if not already set within manager class object).

Returns: Generation log as a list.

Method `calculate_result_attachments()`**:** Calculates and attaches intermediate values to the sample result model (via the result attachment functions).

Usage:

```
ResultsManager$calculate_result_attachments(simulation_results)
```

Arguments:

`simulation_results` The sample simulation results, an object of a class inherited from SimulationResults, to which the intermediate results are attached.

Method `calculate_summaries()`**:** Calculates the summary metrics and/or matrices for the results of a sample simulation (via the summary functions).

Usage:

```
ResultsManager$calculate_summaries(simulation_results, sample_index)
```

Arguments:

`simulation_results` The sample simulation results, an object of a class inherited from SimulationResults.

`sample_index` Index of sample from data frame.

Returns: Generation log entry as a (nested) list, including generated summary metric data and (optionally) matrices.

Method `log_generation()`**:** Summarizes the log generated within the generate method and writes it to a text file in the results directory.

Usage:

```
ResultsManager$log_generation(generation_log)
```

Arguments:

`generation_log` Nested list of log entries generated via the generate method.

Returns: Extended generation log as a nested with added summary and failure/warning indices.

Method `calculate_summary_weighted_averages()`**:** Calculates the weighted averages for each of the summary matrices (providing the sample data has a *weight* column).

Usage:

```
ResultsManager$calculate_summary_weighted_averages(na_replacements = NULL)
```

Arguments:

`na_replacements` List of values or functions (form: `modified_matrix <- function(matrix)`) for dealing with NA values in each summary matrix (default NULL will ignore NAs).

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# U Island example region
coordinates <- data.frame(
  x = rep(seq(177.01, 177.05, 0.01), 5),
  y = rep(seq(-18.01, -18.05, -0.01), each = 5)
)
template_raster <- Region$new(coordinates = coordinates)$region_raster # full extent
template_raster[][-c(7, 9, 12, 14, 17:19)] <- NA # make U Island
region <- Region$new(template_raster = template_raster)
raster::plot(region$region_raster,
  main = "Example region (indices)",
  xlab = "Longitude (degrees)", ylab = "Latitude (degrees)",
  colNA = "blue"
)
# Results manager
results_manager <- ResultsManager$new(
  sample_data = data.frame(index = 1:3),
  simulation_results = PopulationResults$new(region = region),
  summary_metrics = c("trend_n", "total_h"),
  summary_matrices = c("n", "h"),
  summary_functions = list(
    trend_n = function(results) {
      round(results$all$abundance_trend, 2)
    },
```

```
    total_h = function(results) {
      sum(results$harvested)
    },
    n = "all$abundance", # string
    h = "all$harvested"
  ),
  parallel_cores = 2,
  results_dir = tempdir()
)
# Write example result files
results <- list()
for (i in 1:3) {
  results[[i]] <- list(abundance = t(apply(
    matrix(11:17), 1,
    function(n) round(n * exp(-(0:9) / i))
  )))
  results[[i]]$harvested <- round(results[[i]]$abundance * i / 7)
  file_name <- paste0(results_manager$get_results_filename(i), ".RData")
  saveRDS(results[[i]], file.path(tempdir(), file_name))
}
# Generate result metrics and matrices
gen_output <- results_manager$generate()
gen_output$summary
dir(tempdir(), "*.txt") # plus generation log
results_manager$summary_metric_data
results_manager$summary_matrix_list
```

`SimulationManager` *R6 class representing a simulation manager.*

Description

R6 class to represent a manager for running multiple model simulations and saving results.

Super classes

`poems::GenericClass` -> `poems::GenericManager` -> `SimulationManager`

Public fields

`attached` A list of dynamically attached attributes (name-value pairs).

Active bindings

`sample_data` A data frame of sampled parameters for each simulation/result.

`model_template` A `SimulationModel` (or inherited class) object with parameters common to all simulations.

`nested_model` A `SimulationModel` (or inherited class) object with empty sample parameters and a nested model template common to all simulations.

`generators` A list of generators (`Generator` or inherited class) objects for generating simulation model values.

`model_simulator` A `ModelSimulator` (or inherited class) object for running the simulations.

`parallel_cores` Number of cores for running the simulations in parallel.

`results_dir` Results directory path.

`results_ext` Result file extension (default is .RData).

`results_filename_attributes` A vector of: prefix (optional); attribute names (from the sample data frame); postfix (optional); utilized to construct results filenames.

`error_messages` A vector of error messages encountered when setting model attributes.

`warning_messages` A vector of warning messages encountered when setting model attributes.

Methods

Public methods:

- `SimulationManager$new()`
- `SimulationManager$run()`
- `SimulationManager$set_model_sample()`
- `SimulationManager$log_simulation()`
- `SimulationManager$clone()`

Method `new()`**:** Initialization method sets any included attributes (*sample_data*, *model_template*, *generators*, *model_simulator*, *parallel_cores*, *results_dir*, *results_filename_attributes*) and attaches other attributes individually listed.

Usage:

```
SimulationManager$new(model_template = NULL, ...)
```

Arguments:

`model_template` A SimulationModel (or inherited class) object with parameters common to all simulations.

`...` Parameters listed individually.

Method `run()`**:** Runs the multiple population simulations (via the set function), stores the results, and creates/writes a simulation log.

Usage:

```
SimulationManager$run(results_dir = NULL)
```

Arguments:

`results_dir` Results directory path (must be present if not already set within manager class object).

Returns: Simulator log as a list.

Method `set_model_sample()`**:** Sets the model sample attributes via the sample data frame and the generators.

Usage:

```
SimulationManager$set_model_sample(model, sample_index)
```

Arguments:

`model` SimulationModel (or inherited class) object (clone) to receive sample attributes.

`sample_index` Index of sample from data frame.

Method `log_simulation()`**:** Summarizes the simulation log generated within the run method and writes it to a text file in the results directory.

Usage:

```
SimulationManager$log_simulation(simulation_log)
```

Arguments:

`simulation_log` Nested list of simulation log entries generated via the run method.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# U Island example region
coordinates <- data.frame(
  x = rep(seq(177.01, 177.05, 0.01), 5),
  y = rep(seq(-18.01, -18.05, -0.01), each = 5)
)
template_raster <- Region$new(coordinates = coordinates)$region_raster # full extent
template_raster[][-c(7, 9, 12, 14, 17:19)] <- NA # make U Island
region <- Region$new(template_raster = template_raster)
raster::plot(region$region_raster,
  main = "Example region (indices)",
  xlab = "Longitude (degrees)", ylab = "Latitude (degrees)",
  colNA = "blue"
)
# Example population model template
model_template <- PopulationModel$new(
  region = region,
  time_steps = 10, # years
  populations = region$region_cells, # 7
  stage_matrix = 1
)
# Example generators for initial abundance and carrying capacity
hs_matrix <- c(0.5, 0.3, 0.7, 0.9, 0.6, 0.7, 0.8)
initial_gen <- Generator$new(
  description = "initial abundance",
  region = region,
  hs_matrix = hs_matrix, # template attached
  inputs = c("initial_n"),
```

```
  outputs = c("initial_abundance")
)
initial_gen$add_generative_requirements(list(initial_abundance = "function"))
initial_gen$add_function_template("initial_abundance",
  function_def = function(params) {
    stats::rmultinom(1,
      size = params$initial_n,
      prob = params$hs_matrix
    )[, 1]
  },
  call_params = c("initial_n", "hs_matrix")
)
capacity_gen <- Generator$new(
  description = "carrying capacity",
  region = region,
  hs_matrix = hs_matrix, # template attached
  inputs = c("density_max"),
  outputs = c("carrying_capacity")
)
capacity_gen$add_generative_requirements(list(carrying_capacity = "function"))
capacity_gen$add_function_template("carrying_capacity",
  function_def = function(params) {
    round(params$density_max * params$hs_matrix)
  },
  call_params = c("density_max", "hs_matrix")
)
# Sample input parameters
sample_data <- data.frame(initial_n = c(40, 60, 80), density_max = c(15, 20, 25))
# Simulation manager
sim_manager <- SimulationManager$new(
  sample_data = sample_data,
  model_template = model_template,
  generators = list(initial_gen, capacity_gen),
  parallel_cores = 2,
  results_dir = tempdir()
)
run_output <- sim_manager$run()
run_output$summary
dir(tempdir(), "*.RData") # includes 3 result files
for (i in 1:3) {
  print(paste("Run", i, "results:"))
  file_name <- paste0(sim_manager$get_results_filename(i), ".RData")
  print(readRDS(file.path(tempdir(), file_name)))
}
dir(tempdir(), "*.txt") # plus simulation log
```

SimulationModel *R6 class representing a simulation model*

Description

R6 class representing a spatially-explicit simulation model. It extends the SpatialModel class with a range of common simulation parameters and functionality for creating a nested model, whereby a nested template model with fixed parameters is maintained when a model is cloned for various sampled parameters. Also provided are methods for checking the consistency and completeness of model parameters.

Super classes

poems::GenericClass -> poems::GenericModel -> poems::SpatialModel -> SimulationModel

Public fields

attached A list of dynamically attached attributes (name-value pairs).

Active bindings

simulation_function Name (character string) or source path of the default simulation function, which takes a model as an input and returns the simulation results.

model_attributes A vector of model attribute names.

region A Region (or inherited class) object specifying the study region.

coordinates Data frame (or matrix) of X-Y population (WGS84) coordinates in longitude (degrees West) and latitude (degrees North) (get and set), or distance-based coordinates dynamically returned by region raster (get only).

random_seed Number to seed the random number generation for stochasticity.

replicates Number of replicate simulation runs.

time_steps Number of simulation time steps.

years_per_step Number of years per time step.

results_selection List of simulator-dependent attributes to be included in the returned results of each simulation run.

attribute_aliases A list of alternative alias names for model attributes (form: alias = "attribute") to be used with the set and get attributes methods.

template_model Nested template model for fixed (non-sampled) attributes for shallow cloning.

sample_attributes Vector of sample attribute names (only).

required_attributes Vector of required attribute names (only), i.e. those needed to run a simulation.

error_messages A vector of error messages encountered when setting model attributes.

warning_messages A vector of warning messages encountered when setting model attributes.

Methods

Public methods:

- SimulationModel$new()
- SimulationModel$new_clone()

- SimulationModel$get_attribute_names()
- SimulationModel$get_attributes()
- SimulationModel$set_attributes()
- SimulationModel$set_sample_attributes()
- SimulationModel$is_consistent()
- SimulationModel$list_consistency()
- SimulationModel$inconsistent_attributes()
- SimulationModel$is_complete()
- SimulationModel$list_completeness()
- SimulationModel$incomplete_attributes()
- SimulationModel$clone()

Method `new()`**:** Initialization method sets template model and sets given attributes individually and/or from a list.

Usage:

```
SimulationModel$new(template = NULL, required_attributes = NULL, ...)
```

Arguments:

`template` Template simulation model (nested) containing fixed (non-sampled) attributes.

`required_attributes` Vector of required attribute names (only), i.e. those needed to run a simulation.

`...` Parameters passed via a *params* list or individually.

Method `new_clone()`**:** Creates a new (re-initialized) object of the current (inherited) object class with optionally passed parameters.

Usage:

```
SimulationModel$new_clone(...)
```

Arguments:

`...` Parameters passed via the inherited class constructor (defined in initialize and run via new).

Returns: New object of the current (inherited) class.

Method `get_attribute_names()`**:** Returns a list of all attribute names including public and private model attributes, as well as attached attributes (including those from the template model).

Usage:

```
SimulationModel$get_attribute_names()
```

Returns: List of all attribute names.

Method `get_attributes()`**:** Returns a list of values for selected attributes or attribute aliases (when array of parameter names provided) or all attributes (when no params).

Usage:

```
SimulationModel$get_attributes(params = NULL)
```

Arguments:

`params` Array of attribute names to return (all when NULL).

Returns: List of selected or all attributes values.

Method `set_attributes()`**:** Sets given attributes (optionally via alias names) individually and/or from a list.

Usage:

```
SimulationModel$set_attributes(params = list(), ...)
```

Arguments:

`params` List of parameters/attributes.

`...` Parameters/attributes passed individually.

Method `set_sample_attributes()`**:** Sets the names (only - when *params* is a vector) and values (when *params* is a list and/or when name-value pairs are provided) of the sample attributes for the model.

Usage:

```
SimulationModel$set_sample_attributes(params = list(), ...)
```

Arguments:

`params` List of parameters/attributes (names and values) or array of names only.

`...` Parameters/attributes passed individually.

Method `is_consistent()`**:** Returns a boolean to indicate if (optionally selected or all) model attributes (such as dimensions) are consistent/valid.

Usage:

```
SimulationModel$is_consistent(params = NULL)
```

Arguments:

`params` Optional array of parameter/attribute names.

Returns: Boolean to indicate consistency of selected/all attributes.

Method `list_consistency()`**:** Returns a boolean to indicate if (optionally selected or all) model attributes (such as dimensions) are consistent/valid.

Usage:

```
SimulationModel$list_consistency(params = NULL)
```

Arguments:

`params` Optional array of parameter/attribute names.

Returns: List of booleans (or NAs) to indicate consistency of selected/all attributes.

Method `inconsistent_attributes()`**:** Returns a list of attributes necessary to simulate the model that are inconsistent/invalid.

Usage:

```
SimulationModel$inconsistent_attributes(include_nas = FALSE)
```

Arguments:

`include_nas` Optional boolean indicating whether of not to include attributes with unknown consistency (NA).

Returns: List of inconsistent attributes which prevent the model simulation (and optionally those where consistency is not available).

Method `is_complete()`**:** Returns a boolean to indicate if all attributes necessary to simulate the model have been set and are consistent/valid.

Usage:

```
SimulationModel$is_complete()
```

Returns: Boolean to indicate model completeness (and consistency).

Method `list_completeness()`**:** Returns a list of booleans (or NAs) for each parameter to indicate attributes that are necessary to simulate the model have been set and are consistent/valid.

Usage:

```
SimulationModel$list_completeness()
```

Returns: List of booleans (or NAs) for each parameter to indicate to indicate completeness (and consistency).

Method `incomplete_attributes()`**:** Returns a list of attributes necessary to simulate the model that are incomplete/inconsistent/invalid.

Usage:

```
SimulationModel$incomplete_attributes(include_nas = FALSE)
```

Arguments:

`include_nas` Optional boolean indicating whether of not to include attributes with unknown completeness (NA).

Returns: List of incomplete attributes which prevent the model simulation (and optionally those where completeness is not available).

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# U Island example region
coordinates <- data.frame(
  x = rep(seq(177.01, 177.05, 0.01), 5),
  y = rep(seq(-18.01, -18.05, -0.01), each = 5)
)
template_raster <- Region$new(coordinates = coordinates)$region_raster # full extent
template_raster[][-c(7, 9, 12, 14, 17:19)] <- NA # make U Island
region <- Region$new(template_raster = template_raster)
# Model template
template_model <- SimulationModel$new(
  simulation_function = "test_simulator",
  region = region, time_steps = 10
)
template_model$model_attributes <- c(
  template_model$model_attributes,
```

```
  "a", "b", "c", "d"
)
template_model$model_attributes
template_model$required_attributes <- c(
  template_model$required_attributes[1:2],
  "a", "b", "c", "d"
)
template_model$required_attributes
template_model$get_attributes(template_model$required_attributes)
template_model$simulation_function
# Nested model
nested_model <- SimulationModel$new(template_model = template_model)
nested_model$region$region_cells
nested_model$set_sample_attributes(a = 1:7, b = 1:10, c = 1:15)
nested_model$sample_attributes
nested_model$get_attributes(c("a", "b", "c", "d"))
# Completeness and consistency
nested_model$is_complete()
nested_model$incomplete_attributes()
nested_model$is_consistent()
nested_model$inconsistent_attributes()
nested_model$set_attributes(c = array(1:70, c(7, 10)), d = 15)
nested_model$is_complete()
nested_model$is_consistent()
# Attached attributes
nested_model$attached
template_model$attached
```

`SimulationResults` *R6 class representing simulation results.*

Description

R6 class for encapsulating and dynamically generating spatially-explicit simulation results, as well as optional re-generated `Generator` outputs.

Super classes

`poems::GenericClass` -> `poems::GenericModel` -> `poems::SpatialModel` -> `SimulationResults`

Public fields

`attached` A list of dynamically attached attributes (name-value pairs).

Active bindings

`model_attributes` A vector of model attribute names.

`region` A `Region` (or inherited class) object specifying the study region.

`coordinates` Data frame (or matrix) of X-Y population (WGS84) coordinates in longitude (degrees West) and latitude (degrees North) (get and set), or distance-based coordinates dynamically returned by region raster (get only).

`time_steps` Number of simulation time steps.

`burn_in_steps` Optional number of initial 'burn-in' time steps to be ignored.

`occupancy_mask` Optional binary mask array (matrix), data frame, or raster (stack) for each cell at each time-step of the simulation including burn-in.

`all` Nested simulation results for all cells.

`parent` Parent simulation results for individual cells.

`default` Default value/attribute utilized when applying primitive metric functions (e.g. max) to the results.

`attribute_aliases` A list of alternative alias names for model attributes (form: `alias = "attribute"`) to be used with the set and get attributes methods.

`error_messages` A vector of error messages encountered when setting model attributes.

`warning_messages` A vector of warning messages encountered when setting model attributes.

Methods

Public methods:

- `SimulationResults$new()`
- `SimulationResults$new_clone()`
- `SimulationResults$get_attribute_names()`
- `SimulationResults$get_attributes()`
- `SimulationResults$set_attributes()`
- `SimulationResults$clone()`

Method `new()`**:** Initialization method sets attributes from a results list or file, and sets object attributes individually and/or from a list.

Usage:

```
SimulationResults$new(results = NULL, parent = NULL, ...)
```

Arguments:

`results` A list containing results or a file path to simulation results.

`parent` Parent simulation results for individual cells (used when nesting a simulation results clone for all cells).

`...` Parameters passed via a *params* list or individually.

Method `new_clone()`**:** Creates a new (re-initialized) object of the current (inherited) object class with optionally passed parameters.

Usage:

```
SimulationResults$new_clone(...)
```

Arguments:

`...` Parameters passed via the inherited class constructor (defined in initialize and run via new).

Returns: New object of the current (inherited) class.

Method `get_attribute_names()`**:** Returns an array of all attribute names including public and private model attributes, as well as attached attributes, error and warning messages.

Usage:

```
SimulationResults$get_attribute_names(all = FALSE)
```

Arguments:

`all` Boolean to indicate if a nested list for all cells (when present) should be also listed (default is FALSE).

Returns: Array of all attribute names with optional inclusion of attribute names of nested results for all cells.

Method `get_attributes()`**:** Returns a list of values for selected attributes or attribute aliases (when array of parameter names provided) or all attributes (when no params).

Usage:

```
SimulationResults$get_attributes(params = NULL, remove_burn_in = TRUE)
```

Arguments:

`params` Array of attribute names to return (all when NULL).

`remove_burn_in` Boolean to indicate whether or not to remove burn-in steps from the attribute values (default = TRUE; mostly for internal use).

Returns: List of selected or all attributes values.

Method `set_attributes()`**:** Sets given attributes (optionally via alias names) individually and/or from a list.

Usage:

```
SimulationResults$set_attributes(params = list(), ...)
```

Arguments:

`params` List of parameters/attributes.

`...` Parameters/attributes passed individually.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# U Island example region
coordinates <- data.frame(
  x = rep(seq(177.01, 177.05, 0.01), 5),
  y = rep(seq(-18.01, -18.05, -0.01), each = 5)
)
template_raster <- Region$new(coordinates = coordinates)$region_raster # full extent
template_raster[][-c(7, 9, 12, 14, 17:19)] <- NA # make U Island
```

```
region <- Region$new(template_raster = template_raster)
raster::plot(region$region_raster,
  main = "Example region (indices)",
  xlab = "Longitude (degrees)", ylab = "Latitude (degrees)",
  colNA = "blue"
)
# Sample results occupancy (ignore cell 2 in last 3 time steps)
occupancy_raster <- region$raster_from_values(array(1, c(7, 13)))
occupancy_raster[region$region_indices][2, 11:13] <- 0
occupancy_raster[region$region_indices]
# Simulation example results
example_results <- list(abundance = region$raster_from_values(
  t(apply(
    matrix(11:17), 1,
    function(n) c(rep(n, 3), round(n * exp(-(0:9) / log(n))))
  ))
))
example_results$abundance[region$region_indices]
# Simulation results object
sim_results <- SimulationResults$new(
  region = region,
  time_steps = 13,
  burn_in_steps = 3,
  occupancy_mask = occupancy_raster
)
# Clone (for each simulation results)
results_clone <- sim_results$new_clone(results = example_results)
results_clone$get_attribute("abundance")
results_clone$get_attribute("abundance")[region$region_indices]
results_clone$all$get_attribute("abundance")
results_clone$get_attribute("all$abundance")
```

`SimulatorReference` *R6 class for a simulator reference*

Description

R6 class for dynamically attaching simulator attributes and results (passed by reference).

Public fields

`attached` A list of dynamically attached simulator attributes (name-value pairs).

`results` A list of dynamically accessed simulator results (name-value pairs).

Methods

Public methods:

- `SimulatorReference$clone()`

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
test_class <- SimulatorReference$new()
test_class$attached$attr1 <- "example1"
test_class$results$attr1 <- "example2"
str(test_class)
```

`SpatialCorrelation` *R6 class representing a spatial correlation.*

Description

`R6` class functionality for modeling spatial correlations within a spatially-explicit model. It provides functionality for calculating correlations between region cells using a distance-based function: *a*exp(-distance/b)*, where *a* (amplitude) and *b* (breadth) are configurable model attributes. It then calculates the Cholesky decomposition of the correlation matrix (via `chol`), which is utilized to generate (optionally temporal) correlated normal deviates. A compacted version of the decomposed matrix can also generated for computational efficiency.

Super classes

`poems::GenericClass` -> `poems::GenericModel` -> `poems::SpatialModel` -> `SpatialCorrelation`

Public fields

`attached` A list of dynamically attached attributes (name-value pairs).

Active bindings

`model_attributes` A vector of model attribute names.

`region` A `Region` (or inherited class) object specifying the study region.

`coordinates` Data frame (or matrix) of X-Y population (WGS84) coordinates in longitude (degrees West) and latitude (degrees North) (get and set), or distance-based coordinates dynamically returned by region raster (get only).

`distance_scale` Scale of distance values in meters (default = 1). Usage: set to 1 for values in meters, or to 1000 for values in kilometers.

`correlation_amplitude` Correlation function: *a*exp(-distance/b) a* parameter. Represents the amplitude or maximum magnitude of correlation values between model cells.

`correlation_breadth` Correlation function: *a*exp(-distance/b) b* parameter. Represents the breadth of the correlation between region cells. Typically estimated via average distance between correlated region cells.

`correlation_matrix` Correlation matrix calculated via correlation function: *a*exp(-distance/b)*.

`t_decomposition_matrix` The transposed Cholesky decomposition of the correlation matrix (see `chol`).

`compact_only` Boolean to indicate that only the compact versions of matrices will be maintained once calculated.

`t_decomposition_compact_matrix` A compact (rows) version of the transposed Cholesky decomposition of the correlation matrix.

`t_decomposition_compact_map` A map of the original region cell rows for the compact transposed decomposition matrix.

`attribute_aliases` A list of alternative alias names for model attributes (form: `alias = "attribute"`) to be used with the set and get attributes methods.

`error_messages` A vector of error messages encountered when setting model attributes.

`warning_messages` A vector of warning messages encountered when setting model attributes.

Methods

Public methods:

- `SpatialCorrelation$new()`
- `SpatialCorrelation$calculate_distance_matrix()`
- `SpatialCorrelation$calculate_correlations()`
- `SpatialCorrelation$calculate_cholesky_decomposition()`
- `SpatialCorrelation$calculate_compact_decomposition()`
- `SpatialCorrelation$get_compact_decomposition()`
- `SpatialCorrelation$generate_correlated_normal_deviates()`
- `SpatialCorrelation$clone()`

Method `new()`**:** Initialization method sets given attributes individually and/or from a list.

Usage:

```
SpatialCorrelation$new(compact_only = TRUE, attribute_aliases = NULL, ...)
```

Arguments:

`compact_only` Boolean to indicate that only the compact versions of matrices will be maintained once calculated.

`attribute_aliases` Optional list of extra alias names for model attributes (form: `alias = "attribute"`) to be used with the set and get attributes methods.

`...` Parameters passed via a *params* list or individually.

Method `calculate_distance_matrix()`**:** Returns a matrix with the calculated distance (in meters by default) between each pair of region cells.

Usage:

```
SpatialCorrelation$calculate_distance_matrix(use_longlat = NULL)
```

Arguments:

`use_longlat` Optional boolean indicating use of (WGS84) coordinates in longitude (degrees West) and latitude (degrees North).

Returns: Matrix with distances between region cells.

Method `calculate_correlations()`**:** Calculates the correlation matrix by applying the distance-based correlation function.

Usage:

```
SpatialCorrelation$calculate_correlations(
  distance_matrix = NULL,
  decimals = NULL,
  threshold = 1e-07,
  ...
)
```

Arguments:

`distance_matrix` Optional pre-calculated matrix with distances between region cells.

`decimals` Optional number of decimal places for correlation values.

`threshold` Optional threshold (minimum value) for correlation values (default 0.0000001).

`...` Parameters passed via a *params* list or individually.

Method `calculate_cholesky_decomposition()`**:** Calculates the transposed Cholesky decomposition of the correlation matrix (via `chol`).

Usage:

```
SpatialCorrelation$calculate_cholesky_decomposition(
  distance_matrix = NULL,
  decimals = NULL,
  threshold = 1e-07,
  ...
)
```

Arguments:

`distance_matrix` Optional pre-calculated matrix with distances between region cells.

`decimals` Optional number of decimal places for correlation values.

`threshold` Optional threshold (minimum value) for correlation values (default 0.0000001).

`...` Parameters passed via a *params* list or individually.

Method `calculate_compact_decomposition()`**:** Compacts the transposed Cholesky decomposition of the correlation matrix into the minimal number of rows, which are mapped to the original matrix.

Usage:

```
SpatialCorrelation$calculate_compact_decomposition(distance_matrix = NULL, ...)
```

Arguments:

`distance_matrix` Optional pre-calculated matrix with distances between region cells.

`...` Parameters passed via a *params* list or individually.

Method `get_compact_decomposition()`**:** Returns a compact transposed Cholesky decomposition of the correlation matrix and a corresponding map of region cell indices in a list with names: matrix, map.

Usage:

```
SpatialCorrelation$get_compact_decomposition(distance_matrix = NULL, ...)
```

Arguments:

`distance_matrix` Optional pre-calculated matrix with distances between region cells.

`...` Parameters passed via a *params* list or individually.

Returns: List containing a compact Cholesky decomposition matrix and a corresponding map of region cell indices (for the compacted rows).

Method `generate_correlated_normal_deviates()`**:** Generates correlated normal deviates using the spatial correlation, utilizing the optional random seed and optional temporal correlation across time steps.

Usage:

```
SpatialCorrelation$generate_correlated_normal_deviates(
  random_seed = NULL,
  temporal_correlation = 1,
  time_steps = 1
)
```

Arguments:

`random_seed` Optional seed for the random generation of correlated deviates.

`temporal_correlation` Optional temporal correlation coefficient (0-1; default = 1).

`time_steps` Optional number of time steps for temporal correlation (default = 1 or none).

Returns: Array (non-temporal) or matrix (temporal) of correlated normal deviates.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# U Island example region
coordinates <- data.frame(
  x = rep(seq(177.01, 177.05, 0.01), 5),
  y = rep(seq(-18.01, -18.05, -0.01), each = 5)
)
template_raster <- Region$new(coordinates = coordinates)$region_raster # full extent
template_raster[][-c(7, 9, 12, 14, 17:19)] <- NA # make U Island
region <- Region$new(template_raster = template_raster)
# Spatial correlation
env_corr <- SpatialCorrelation$new(region = region, amplitude = 0.4, breadth = 500)
env_corr$calculate_distance_matrix() # m
env_corr$calculate_correlations(decimals = 5)
```

```
env_corr$correlation_matrix
env_corr$calculate_cholesky_decomposition(decimals = 2)
env_corr$t_decomposition_matrix
env_corr$get_compact_decomposition()
# Scale to km
env_corr$distance_scale <- 1000
env_corr$calculate_distance_matrix() # km
```

`SpatialModel` *R6 class representing a spatial model*

Description

R6 class representing a generic (abstract) spatially-explicit model. It extends `GenericModel` with the addition of a study region specification.

Super classes

`poems::GenericClass` -> `poems::GenericModel` -> `SpatialModel`

Public fields

`attached` A list of dynamically attached attributes (name-value pairs).

Active bindings

`model_attributes` A vector of model attribute names.

`region` A `Region` (or inherited class) object specifying the study region.

`coordinates` Data frame (or matrix) of X-Y population (WGS84) coordinates in longitude (degrees West) and latitude (degrees North) (get and set), or distance-based coordinates dynamically returned by region raster (get only).

`attribute_aliases` A list of alternative alias names for model attributes (form: `alias = "attribute"`) to be used with the set and get attributes methods.

`error_messages` A vector of error messages encountered when setting model attributes.

`warning_messages` A vector of warning messages encountered when setting model attributes.

Methods

Public methods:

- `SpatialModel$new()`
- `SpatialModel$new_clone()`
- `SpatialModel$clone()`

Method `new()`**:** Initialization method sets given attributes individually and/or from a list.

Usage:

```
SpatialModel$new(region = NULL, ...)
```

Arguments:

`region` A `Region` (or inherited class) object specifying the study region.

`...` Parameters passed individually.

Method `new_clone()`**:** Creates a new (re-initialized) object of the current (inherited) object class with optionally passed parameters.

Usage:

```
SpatialModel$new_clone(...)
```

Arguments:

`...` Parameters passed via the inherited class constructor (defined in initialize and run via new).

Returns: New object of the current (inherited) class.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# U Island example region
coordinates <- data.frame(
  x = rep(seq(177.01, 177.05, 0.01), 5),
  y = rep(seq(-18.01, -18.05, -0.01), each = 5)
)
template_raster <- Region$new(coordinates = coordinates)$region_raster # full extent
template_raster[][-c(7, 9, 12, 14, 17:19)] <- NA # make U Island
region <- Region$new(template_raster = template_raster)
# Example spatial model
model1 <- SpatialModel$new(region = region, a_layers = 3)
model1$coordinates
model1$set_attributes(a_values = array(8:28, c(7, 3)))
model1$region$raster_from_values(model1$get_attribute("a_values"))
```

---

tasmania_ibra_data *Thylacine vignette Tasmania IBRA data*

Description

A dataset describing the nine Interim Bioregionalisation of Australia (IBRA) bioregions for the Tasmanian study region of the Thylacine example vignette.

Format

A data frame with 9 rows and 4 variables:

index Cross-reference index for each bioregion

key Additional alphabetical cross-reference for each bioregion

abbr Abbreviated name for each bioregion

name Full name for each bioregion

Source

https://doi.org/10.1111/2041-210X.13720

Examples

```
data(tasmania_ibra_data)
data(tasmania_ibra_raster)
raster::values(tasmania_ibra_raster)[!is.na(raster::values(tasmania_ibra_raster))] |>
  table() |>
  as.data.frame() |>
  merge(tasmania_ibra_data, by.x = "Var1", by.y = "index")
```

---

tasmania_ibra_raster *Thylacine vignette Tasmania IBRA raster*

Description

A *raster* dataset defining the grid cells of the nine Interim Bioregionalisation of Australia (IBRA) bioregions for the Tasmanian study region of the Thylacine example vignette.

Format

A *raster::RasterLayer* object:

dimensions 32 rows by 40 columns grid

resolution 0.1 by 0.1 degree grid cells

extent longitude 144.5 to 148.5 degrees; latitude -43.8025 to -40.6025 degrees

CRS WGS84 longitude-latitude

values IBRA bioregions defined by cells with values 1 to 9 (as per `index` in `tasmania_ibra_data`)

Source

https://doi.org/10.1111/2041-210X.13720

Examples

```
data(tasmania_ibra_raster)
data(tasmania_raster)
tasmania_region <- Region$new(
  template_raster = tasmania_raster
)
tasmania_region$raster_is_consistent(tasmania_ibra_raster)
raster::plot(tasmania_ibra_raster)
```

---

tasmania_modifier *Tasmania land-use modifier raster*

Description

A *raster* dataset (11 timesteps) defining the intensity land-use cover for each grid-cell in the Tasmania study region. NB. This dataset is projected and will not natively overlay the other *raster* datasets contained in *poems*.

Format

A *raster::RasterBrick* object:

dimensions 36 rows, 34 columns, 11 layers

resolution 10km by 10km grid cells

extent -211571.8, 128428.2, -182583.2, 177416.8 (xmin, xmax, ymin, ymax)

CRS +proj=laea +lat_0=-42.2 +lon_0=147 +x_0=0 +y_0=0 +datum=WGS84 +units=m +no_defs

values region defined by 1224 cells with values between 0-1. Values of 1 indicate extensive land use modification)

Source

https://doi.org/10.1111/2041-210X.13720

Examples

```
data(tasmania_raster)
data(tasmania_modifier)
tasmania_region <- Region$new(
  template_raster = tasmania_modifier[[1]]
)
tasmania_region$raster_is_consistent(tasmania_raster)
raster::plot(tasmania_modifier)
```

tasmania_raster *Thylacine vignette Tasmania raster*

Description

A *raster* dataset defining the grid cells of the Tasmanian study region for the Thylacine example vignette.

Format

A *raster::RasterLayer* object:

dimensions 32 rows by 40 columns grid

resolution 0.1 by 0.1 degree grid cells

extent longitude 144.5 to 148.5 degrees; latitude -43.8025 to -40.6025 degrees

CRS WGS84 longitude-latitude

values region defined by 795 cells with value of 1 (surrounded by non-region NA values)

Source

https://doi.org/10.1111/2041-210X.13720

Examples

```
data(tasmania_raster)
tasmania_region <- Region$new(
  template_raster = tasmania_raster
)
raster::plot(tasmania_region$region_raster)
```

thylacine_bounty_record *Thylacine vignette bounty record*

Description

A dataset containing the historical record of the Thylacine bounty numbers submitted across the Tasmanian study region, and for each of the nine Interim Bioregionalisation of Australia (IBRA) bioregions for Thylacine example vignette.

Format

A data frame with 22 rows and 11 variables:

Year Year during bounty period from 1888 to 1909

Total Total Tasmania-wide bounty submitted

FUR Bounty submitted in IBRA bioregion: Furneaux

BEN Bounty submitted in IBRA bioregion: Ben Lomond

TNM Bounty submitted in IBRA bioregion: Tasmanian Northern Midlands

TSE Bounty submitted in IBRA bioregion: Tasmanian South East

TW Bounty submitted in IBRA bioregion: Tasmanian West

TNS Bounty submitted in IBRA bioregion: Tasmanian Northern Slopes

TSR Bounty submitted in IBRA bioregion: Tasmanian Southern Ranges

TCH Bounty submitted in IBRA bioregion: Tasmanian Central Highlands

KIN Bounty submitted in IBRA bioregion: King

Source

https://doi.org/10.1111/2041-210X.13720

Examples

```
data(thylacine_bounty_record)
summary(thylacine_bounty_record)
# Assuming your data frame is named thylacine_bounty_record
plot(thylacine_bounty_record$Year, thylacine_bounty_record$Total,
  type = "l",
  main = "Change in Total Bounties Over Time",
  xlab = "Year",
  ylab = "Total Bounties"
)
```

---

thylacine_example_matrices

Thylacine vignette demonstration example matrices

Description

A dataset containing precalculated summary matrices for use when running the Thylacine example vignette in demonstration mode. The values were obtained by running the vignette code for 20,000 model simulations with `DEMONSTRATION = FALSE`. Note that some matrices were only stored for the selected 'best' 200 models.

Format

A list containing the following matrices:

extirpation 200 row by 795 column matrix of cell extirpation dates for the 'best' 200 models

total_bounty 200 row by 80 column matrix of bounty submitted each year for the 'best' 200 models

ibra_bounty 200 row by 9 column matrix of total bounty submitted each IBRA bioregion for the 'best' 200 models

bounty_slope 20,000 row by 3 column matrix of calculated slope of total bounty submitted across 3 intervals for each sample simulation

ibra_extirpation 20,000 row by 9 column matrix of extirpation dates for each IBRA bioregion for each sample simulation

Source

Precalculated demonstration via example simulation runs.

Examples

```
data(thylacine_example_matrices)
data(tasmania_raster)
region <- Region$new(template_raster = tasmania_raster)
region$raster_from_values(thylacine_example_matrices$extirpation[1, ]) |>
  raster::plot(colNA = "blue")
```

---

thylacine_example_matrices_rerun

Thylacine vignette demonstration example (re-run) matrices

Description

A dataset containing precalculated (re-run) summary matrices for use when running the Thylacine example vignette in demonstration mode. The values were obtained by running the vignette code for 10 replicate re-runs of the selected 'best' 200 model simulations with `DEMONSTRATION = FALSE`.

Format

A list containing the following matrices:

bounty_slope 2,000 row by 3 column matrix of calculated slope of total bounty submitted across 3 intervals for each sample simulation

ibra_extirpation 2,000 row by 9 column matrix of extirpation dates for each IBRA bioregion for each sample simulation

Source

Precalculated demonstration via example simulation re-runs.

Examples

```
data(thylacine_example_matrices_rerun)
rowMeans(thylacine_example_matrices_rerun$bounty_slope)
rowMeans(thylacine_example_matrices_rerun$ibra_extirpation)
```

---

thylacine_example_metrics

Thylacine vignette demonstration example metrics

Description

A dataset containing precalculated summary metrics for use when running the Thylacine example vignette in demonstration mode. The values were obtained by running the vignette code for 20,000 model simulations with `DEMONSTRATION = FALSE`.

Format

A data frame with 20,000 rows and 4 variables:

index Example simulation number from 1 to 20,000

bounty_slope_error Root mean squared error (RMSE) from estimated total bounty submitted across three intervals (see vignette)

ibra_extirpation_error RMSE from estimated extirpation date for each IBRA bioregion (see vignette)

total_extinction Total extinction date for each example simulation (`NA` when persistent beyond 1967)

Source

Precalculated demonstration via example simulation runs.

Examples

```
data(thylacine_example_metrics)
hist(thylacine_example_metrics$bounty_slope_error)
hist(thylacine_example_metrics$ibra_extirpation_error)
hist(thylacine_example_metrics$total_extinction)
```

thylacine_example_metrics_rerun *Thylacine vignette demonstration example (re-run) metrics*

Description

A dataset containing precalculated (re-run) summary metrics for use when running the Thylacine example vignette in demonstration mode. The values were obtained by running the vignette code for 10 replicate re-runs of the selected 'best' 200 model simulations with `DEMONSTRATION = FALSE`.

Format

A data frame with 2,000 rows and 4 variables:

index Example simulation number from 1 to 2,000

bounty_slope_error Root mean squared error (RMSE) from estimated total bounty submitted across three intervals (see vignette)

ibra_extirpation_error RMSE from estimated extirpation date for each IBRA bioregion (see vignette)

total_extinction Total extinction date for each example simulation (`NA` when persistent beyond 1967)

Source

Precalculated demonstration via example simulation re-runs.

Examples

```
data(thylacine_example_metrics_rerun)
hist(thylacine_example_metrics_rerun$bounty_slope_error)
hist(thylacine_example_metrics_rerun$ibra_extirpation_error)
hist(thylacine_example_metrics_rerun$total_extinction)
```

thylacine_hs_raster *Thylacine vignette habitat suitability raster*

Description

A *raster* dataset defining estimated habitat suitability values for each grid cells of the Tasmanian study region of the Thylacine example vignette.

Format

A *raster::RasterLayer* object:

dimensions 32 rows by 40 columns grid

resolution 0.1 by 0.1 degree grid cells

extent longitude 144.5 to 148.5 degrees; latitude -43.8025 to -40.6025 degrees

CRS WGS84 longitude-latitude

values Estimated habitat suitability values of 0 to 1

Source

https://doi.org/10.1111/2041-210X.13720

Examples

```
data(thylacine_hs_raster)
raster::plot(thylacine_hs_raster, colNA = "blue")
```

`Validator` *R6 class representing a pattern-oriented validator.*

Description

R6 class for pattern-oriented validation and simulation model ensemble selection. Pattern-oriented validation is a statistical approach to compare patterns generated in simulations against observed empirical patterns.

The class wraps functionality for the validation approach, typically utilizing an external library, the default being the approximate Bayesian computation (ABC) abc library, and includes methods for resolving non-finite metrics, centering and scaling the validator inputs, running the validator analysis, and generating diagnostics (see abc).

Super classes

`poems::GenericClass` -> `poems::GenericModel` -> `Validator`

Public fields

`attached` A list of dynamically attached attributes (name-value pairs).

Active bindings

`model_attributes` A vector of model attribute names.

`simulation_parameters` A data frame of sample model parameters for each simulation.

`simulation_summary_metrics` A data frame of result summary metrics for each simulation.

`observed_metric_targets` A vector of observed targets for each summary metric.

`random_seed` A seed for randomizing the order of the simulation samples (no randomization is utilized when left NULL).

`random_indices` Randomized simulation sample indices for the validator inputs and consequently the validator results when random seed is used.

`non_finite_replacements` A list of numeric values or function names (character strings) or direct assignments (assigned or loaded via source paths) for replacing NAs in specified (list names) summary metrics.

`input_center_scale_values` A nested list of center and scale values for validator input parameters/metrics.

`output_dir` Directory path for validator (default: `abc`) regression diagnostic and other outputs.

`validation_call_function` Dynamically assigned function: `function(observed_metric_targets, simulation_parameters, simulation_summary_metrics, tolerance, method, ...)` for calling the validation function (default calls `abc` library function).

`validator_return_object` Object returned by the validator function (see `abc` documentation if using default).

`selected_simulations` A data frame of simulation sample indices and weights selected/assigned by the validation function (`abc` by default).

`attribute_aliases` A list of alternative alias names for model attributes (form: `alias = "attribute"`) to be used with the set and get attributes methods.

`error_messages` A vector of error messages encountered when setting model attributes.

`warning_messages` A vector of warning messages encountered when setting model attributes.

Methods

Public methods:

- `Validator$new()`
- `Validator$run()`
- `Validator$resolve_nonfinite_metrics()`
- `Validator$center_scale_inputs()`
- `Validator$generate_diagnostics()`
- `Validator$clone()`

Method `new()`**:** Initialization method sets given attributes individually and/or from a list.

Usage:

```
Validator$new(...)
```

Arguments:

`...` Parameters passed via a *params* list or individually.

`template` Template population model containing fixed (non-sampled) attributes.

Method `run()`**:** Pre-processes inputs, runs validator function for input parameters, and stores the function (and optionally diagnostic) outputs (see abc documentation if using the default).

Usage:

```
Validator$run(
  simulation_parameters = NULL,
  simulation_summary_metrics = NULL,
  observed_metric_targets = NULL,
  tolerance = 0.01,
  method = "neuralnet",
  output_diagnostics = FALSE,
  ...
)
```

Arguments:

`simulation_parameters` A data frame of sample model parameters for each simulation.

`simulation_summary_metrics` A data frame of result summary metrics for each simulation.

`observed_metric_targets` A vector of observed targets for each summary metric.

`tolerance` Tolerance or proportion of models to select.

`method` Validator algorithm to be applied (default is a neural network algorithm - see abc documentation) .

`output_diagnostics` Boolean to indicate whether or not to output diagnostics (PDF file - default is FALSE).

`...` Additional validator parameters passed individually (see abc documentation if using default).

Method `resolve_nonfinite_metrics()`**:** Attempts to resolve any non-finite simulation summary metric values (and optionally changing them to NAs) via the non finite replacements parameter (a list of values/functions for replacing non-finite values).

Usage:

```
Validator$resolve_nonfinite_metrics(use_nas = TRUE)
```

Arguments:

`use_nas` Boolean to indicate whether or not to replace all non-finite values with NAs (default is TRUE).

Method `center_scale_inputs()`**:** Centers and scales the model parameters, result summary metrics and observed targets.

Usage:

```
Validator$center_scale_inputs()
```

Method `generate_diagnostics()`**:** Generates the validation diagnostics (see abc documentation if using default) as a PDF file in the output directory.

Usage:

```
Validator$generate_diagnostics(output_dir = NULL)
```

Arguments:

`output_dir` Output directory path for the diagnostics PDF file (must be present if not already set within validator class object).

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Example parameter sample data
sample_data <- data.frame(
  growth_rate_max = round(log(seq(1.11, 1.30, 0.01)), 3),
  harvest_rate = seq(0.11, 0.30, 0.01),
  initial_n = seq(105, 200, 5),
  density_max = seq(132, 170, 2)
)
# Example simulation result summary metrics
summary_metric_data <- data.frame(
  trend_n = seq(10, -9, -1),
  total_h = seq(70, 355, 15)
)
# Create a validator for selecting the 'best' example models
validator <- Validator$new(
  simulation_parameters = sample_data,
  simulation_summary_metrics = summary_metric_data,
  observed_metric_targets = c(trend_n = 0, total_h = 250),
  output_dir = tempdir()
)
suppressWarnings(validator$run(tolerance = 0.25, output_diagnostics = TRUE))
dir(tempdir(), "*.pdf") # plus validation diagnostics (see abc library documentation)
validator$selected_simulations # top 5 models
```

Index