Package: metaRange (via r-universe)

August 22, 2024

Title Framework to Build Mechanistic and Metabolic Constrained Species Distribution Models

Version 1.2.0

Description Build spatially and temporally explicit process-based species distribution models, that can include an arbitrary number of environmental factors, species and processes including metabolic constraints and species interactions. The focus of the package is simulating populations of one or multiple species in a grid-based landscape and studying the meta-population dynamics and emergent patterns that arise from the interaction of species under complex environmental conditions. It provides functions for common ecological processes such as negative exponential, kernel-based dispersal (see Nathan et al. (2012) <doi:10.1093/acprof:oso/9780199608898.003.0015>), calculation of the environmental suitability based on cardinal values (Yin et al. (1995) <doi:10.1016/0168-1923(95)02236-Q>, simplified by Yan and Hunt (1999) <doi:10.1006/anbo.1999.0955> see eq: 4), reproduction in form of an Ricker model (see Ricker (1954) <doi:10.1139/f54-039> and Cabral and Schurr (2010) <doi:10.1111/j.1466-8238.2009.00492.x>), as well as metabolic scaling based on the metabolic theory of ecology (see Brown et al. (2004) <doi:10.1890/03-9000> and Brown, Sibly and Kodric-Brown (2012) <doi:10.1002/9781119968535.ch>).

License GPL-3

Copyright see inst/COPYRIGHTS

URL https://metaRange.github.io/metaRange/

BugReports https://github.com/metaRange/metaRange/issues

Depends R (>= 3.5.0)

Imports Rcpp (>= 1.0.10), terra, R6, checkmate, grDevices, graphics, utils

Suggests knitr, rmarkdown, tinytest

VignetteBuilder knitr

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.2.3

LinkingTo Rcpp, RcppArmadillo

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

RemoteUrl https://github.com/metarange/metarange

RemoteRef HEAD

RemoteSha d643acb92ff8bce486a698b25c8c9e284cbb4620

Contents

`calculate_dispersal_kernel`

Calculate 2D dispersal kernel.

Description

Use a probability function to create a 2D dispersal kernel matrix.

Usage

```
calculate_dispersal_kernel(max_dispersal_dist, kfun, normalize = TRUE, ...)
```

Arguments

max_dispersal_dist	<numeric> maximum dispersal distance in grid cells. The size (rows and columns) of the created dispersal kernel matrix will be 2 * max_dispersal_dist + 1.
kfun	<function> the probability function that is used to calculate the values for each cell of the dispersal kernel. Can be user-defined, in which case it needs to vectorized and accept (at least) the parameter "x" representing the distance from the source as its input and return a vector of the same size as max_dispersal_dist.
normalize	<boolean> whether to normalize the kernel (sum(kernel) == 1)).
...	additional parameters to be passed to the kernel function.

Details

This function first creates an matrix of size 2 * max_dispersal_dist + 1, where each cell contains the distance from the center of the cell to the center of the matrix. After that, the kernel function (kfun) is called with this matrix as input and expected to return a vector with the calculated probabilities for each cell. Lastly, the kernel is optionally normalized.

Value

Dispersal kernel with probabilities.

Examples

```
# a very simple uniform kernel
uniform_kernel <- calculate_dispersal_kernel(
    max_dispersal_dist = 2,
    kfun = function(x) {
        rep(1, length(x))
    },
    normalize = FALSE
)
# same as
stopifnot(
    uniform_kernel == matrix(1, nrow = 5, ncol = 5)
)

# How does the input matrix look like,
# that is used as input for `kfun`?
calculate_dispersal_kernel(
    max_dispersal_dist = 2,
    kfun = function(x) {
        return(x)
    },
    normalize = FALSE
)
```

```
# now a negative exponential kernel.
# note that `mean_dispersal_dist` is a parameter of
# the `negative_exponential_function`
# and is passed to `kfun` via the `...`.
calculate_dispersal_kernel(
    max_dispersal_dist = 2,
    kfun = negative_exponential_function,
    mean_dispersal_dist = 1
)
```

`calculate_normalization_constant`
Normalization constant calculation

Description

Calculates the normalization constant for the metabolic scaling based on a known or estimated parameter value under at a reference temperature.

Usage

```
calculate_normalization_constant(
  parameter_value,
  scaling_exponent,
  mass,
  reference_temperature,
  E = NULL,
  k = 8.617333e-05,
 warn_if_possibly_false_input = getOption("metaRange.verbose", default = FALSE) > 0
)
```

Arguments

`parameter_value`
: <numeric> estimated parameter value at the reference temperature.

`scaling_exponent`
: <numeric> allometric scaling exponent of the mass.

`mass`
: <numeric> mean (individual) mass.

`reference_temperature`
: <numeric> reference temperature in kelvin (K).

`E`
: <numeric> Activation energy in electronvolts (eV).

`k`
: <numeric> Boltzmann's constant (eV / K).

`warn_if_possibly_false_input`
: <boolean> Print a warning if the input is different from the known literature value combinations.

Details

Note the different scaling values for different parameter. The following is a summary from table 2 in Brown, Sibly and Kodric-Brown (2012) (see references).

Parameter	Scaling exponent	Activation energy
resource usage	3/4	-0.65
reproduction, mortality	-1/4	-0.65
carrying capacity	-3/4	0.65

Value

The calculated normalization constant.

References

Brown, J.H., Gillooly, J.F., Allen, A.P., Savage, V.M. and West, G.B. (2004) Toward a Metabolic Theory of Ecology. *Ecology*, **85** 1771–1789. doi:10.1890/039000

Brown, J.H., Sibly, R.M. and Kodric-Brown, A. (2012) Introduction: Metabolism as the Basis for a Theoretical Unification of Ecology. In *Metabolic Ecology* (eds R.M. Sibly, J.H. Brown and A. Kodric-Brown) doi:10.1002/9781119968535.ch

See Also

`metabolic_scaling()`

Examples

```
calculate_normalization_constant(
    parameter_value = 1,
    scaling_exponent = -1 / 4,
    mass = 1,
    reference_temperature = 273.15,
    E = -0.65
)
```

`calculate_suitability` *Calculate (estimate) environmental suitability*

Description

Calculate / estimate the environmental suitability for a given environmental value, based on a beta distribution, using the three "cardinal" values of the species for that environmental niche.

Usage

```
calculate_suitability(vmax, vopt, vmin, venv)
```

Arguments

vmax	<numeric> upper (i.e. maximum) tolerable value
vopt	<numeric> optimal (i.e. preferred) value
vmin	<numeric> lower (i.e. minimum) tolerable value
venv	<numeric> environmental value for which to calculate the suitability

Details

The environmental suitability is calculated based on a beta distribution after a formula provided by Yin et al. (1995) and simplified by Yan and Hunt (1999) (see references paragraph)

$$suitability = (\frac{V_{max} - V_{env}}{V_{max} - V_{opt}}) * (\frac{V_{env} - V_{min}}{V_{opt} - V_{min}})^{\frac{V_{opt} - V_{min}}{V_{max} - V_{opt}}}$$

Value

<numeric> environmental suitability

Note

The original formula by Yin et al. was only intended to calculate the relative daily growth rate of plants in relation to temperature. The abstraction to use this to A) calculate a niche suitability; and B) use it on other environmental values than temperature might not be valid. However, the assumption that the environmental suitability for one niche dimension is highest at one optimal value and decreases towards the tolerable minimum and maximum values in a nonlinear fashion seems reasonable.

References

Yin, X., Kropff, M.J., McLaren, G., Visperas, R.M., (1995) A nonlinear model for crop development as a function of temperature, *Agricultural and Forest Meteorology*, Volume **77**, Issues 1–2, Pages 1–16, doi:10.1016/01681923(95)02236Q

Also, see equation 4 in: Weikai Yan, L.A. Hunt, (1999) An Equation for Modelling the Temperature Response of Plants using only the Cardinal Temperatures, *Annals of Botany*, Volume **84**, Issue 5, Pages 607–614, ISSN 0305-7364, doi:10.1006/anbo.1999.0955

Examples

```
calculate_suitability(
    vmax = 30,
    vopt = 25,
    vmin = 10,
    venv = 1:40
)
calculate_suitability(
    vmax = seq(30, 32, length.out = 40),
    vopt = seq(20, 23, length.out = 40),
    vmin = seq(9, 11, length.out = 40),
    venv = 1:40
)
```

```
try(calculate_suitability(
    vmax = 1,
    vopt = seq(20, 23, length.out = 40),
    vmin = seq(9, 11, length.out = 40),
    venv = 1:40
))
```

`create_simulation` *Create a simulation*

Description

Creates a metaRangeSimulation object. A convenience wrapper for `metaRangeSimulation$new()`.

Usage

```
create_simulation(source_environment, ID = NULL, seed = NULL)
```

Arguments

`source_environment`	<SpatRasterDataset> created by `terra::sds()` that represents the environment. The individual data sets represent different environmental variables (e.g. temperature or habitat availability) and the different layer of the data sets represent the different timesteps of the simulation. The function metaRangeSimulation`$set_time_layer_mapping()` can be used to extend/ shorten the simulation timesteps and set the mapping between each time step and a corresponding environmental layer. This can be used e.g. to repeat the first (few) layer as a burn-in period. The number of layers must be the same for all data sets.
`ID`	<string> optional simulation identification string. Will be set automatically if none is specified.
`seed`	<integer> optional seed for the random number generator. Will be set automatically if none is specified.

Value

A metaRangeSimulation object

Examples

```
sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2))
names(sim_env) <- "env_01"
test_sim <- create_simulation(sim_env)
```

`dispersal` *Dispersal process*

Description

Disperse a (abundance) matrix using a dispersal kernel and optional weights.

Usage

```
dispersal(dispersal_kernel, abundance, weights)
```

Arguments

`dispersal_kernel`	<numeric matrix> dispersal kernel. A 2D matrix of uneven size, containing the weights that deciedes how the individuals from the cell in the center are going to be distributed to the sourrounding cells.
`abundance`	<numeric matrix> abundance matrix.
`weights`	<numeric matrix> optional weights in form of a matrix that has the same dimensions as the abundance and a range between `0` and `1`. Should not contain any `NA`.

Details

Each cell in the abundance matrix is dispersed using the dispersal kernel. If a matrix of weights is supplied, the individuals will redistribute within the dispersal kernel according to the weights. I.e. individuals will more likely move towards areas with a higher weight, if they are within their dispersal distance. Note:

- the abundance is modified in place, to optimize performance.
- Any `NA` or `NaN` in abundance or weights will be (in-place) replaced by `0`.

Value

<numeric matrix> Dispersed abundance matrix.

Examples

```
n <- 10
n2 <- n^2
abu <- matrix(1:n2, nrow = n, ncol = n)
suitab <- matrix(1, nrow = n, ncol = n)
kernel <- calculate_dispersal_kernel(
    max_dispersal_dist = 4,
    kfun = negative_exponential_function,
    mean_dispersal_dist = 1.2
)
res1 <- dispersal(
    dispersal_kernel = kernel,
```

```
    abundance = abu
)
res2 <- dispersal(
    dispersal_kernel = kernel,
    abundance = abu,
    weights = suitab
)
stopifnot(sum(res1) - sum(res2) < 0.01)
# Note that the abundance is modified in place, i.e:
stopifnot(sum(abu - res2) < 0.01)
```

`metabolic_scaling` *Metabolic scaling*

Description

A function to calculate the metabolic scaling of a parameter, based on the metabolic theory of ecology (Brown et al. 2004).

Usage

```
metabolic_scaling(
  normalization_constant,
  scaling_exponent,
  mass,
  temperature,
  E,
  k = 8.617333e-05
)
```

Arguments

`normalization_constant`	<numeric> normalization constant.
`scaling_exponent`	<numeric> allometric scaling exponent of the mass.
`mass`	<numeric matrix> mean (individual) mass.
`temperature`	<numeric matrix> temperature in kelvin (K).
`E`	<numeric> activation energy in electronvolts (eV).
`k`	<numeric> Boltzmann's constant (eV / K).

Details

Equation::

The function uses the equation in the form of:

$$parameter = normalization\_constant \cdot mass^{scaling\_exponent} \cdot e^{\frac{Activation\_energy}{k \cdot temperature}}$$

Parameter::

Note the different scaling values for different parameter. The following is a summary from table 2 in Brown, Sibly and Kodric-Brown (2012) (see references).

Parameter	Scaling exponent	Activation energy
resource usage	3/4	-0.65
reproduction, mortality	-1/4	-0.65
carrying capacity	-3/4	0.65

Units::

$$1\ electronvolt = 1.602176634 \cdot 10^{-19} Joule$$

$$Boltzmann\ constant = 1.380649 \cdot 10^{-23} \frac{Joule}{Kelvin}$$

$$Boltzmann\ constant\ in \frac{eV}{K} = 8.617333e-05 = \frac{1.380649 \cdot 10^{-23}}{1.602176634 \cdot 10^{-19}}$$

Value

<numeric> The scaled parameter.

References

Brown, J.H., Gillooly, J.F., Allen, A.P., Savage, V.M. and West, G.B. (2004) Toward a Metabolic Theory of Ecology. *Ecology*, **85** 1771–1789. doi:10.1890/039000

Brown, J.H., Sibly, R.M. and Kodric-Brown, A. (2012) Introduction: Metabolism as the Basis for a Theoretical Unification of Ecology. In *Metabolic Ecology* (eds R.M. Sibly, J.H. Brown and A. Kodric-Brown) doi:10.1002/9781119968535.ch

See Also

`calculate_normalization_constant()`

Examples

```
reproduction_rate <- 0.25
E_reproduction_rate <- -0.65
estimated_normalization_constant <-
    calculate_normalization_constant(
        parameter_value = reproduction_rate,
        scaling_exponent = -1/4,
        mass = 100,
        reference_temperature = 273.15 + 10,
        E = E_reproduction_rate
    )
metabolic_scaling(
    normalization_constant = estimated_normalization_constant,
```

```
    scaling_exponent = -1/4,
    mass = 100,
    temperature = 273.15 + 20,
    E = E_reproduction_rate
)

carrying_capacity <- 100
E_carrying_capacity <- 0.65
estimated_normalization_constant <-
    calculate_normalization_constant(
        parameter_value = carrying_capacity,
        scaling_exponent = -3/4,
        mass = 100,
        reference_temperature = 273.15 + 10,
        E = E_carrying_capacity
    )
metabolic_scaling(
    normalization_constant = estimated_normalization_constant,
    scaling_exponent = -3/4,
    mass = 100,
    temperature = 273.15 + 20,
    E = E_carrying_capacity
)
```

`metaRangeEnvironment` *metaRangeEnvironment object*

Description

Creates an metaRangeEnvironment object in form of an R6 class that stores and handles the environmental values that influence the species in the simulation.

Value

An `<metaRangeEnvironment>` object

Public fields

`sourceSDS` A *SpatRasterDataset* created by `terra::sds()` that holds all the environmental values influencing the simulation. Note that the individual data sets should be sensibly named as their names will used throughout the simulation to refer to them.

`current` an R environment that holds all the environmental values influencing the present / current time step of the simulation. These values are copies of the current layers of the respective individual data sets in the sourceSDS and they are stored as regular 2D R matrices under the same name given to the corresponding sub data set in the sourceSDS. These matrices are updated automatically at the beginning of each time step.

Methods

Public methods:

- metaRangeEnvironment$new()
- metaRangeEnvironment$set_current()
- metaRangeEnvironment$print()

Method `new()`**:** Creates a new metaRangeEnvironment object. This is done automatically when a simulation is created. There is no need to call this as user.

Usage:

```
metaRangeEnvironment$new(sourceSDS = NULL)
```

Arguments:

`sourceSDS` <SpatRasterDataset> created by `terra::sds()` that holds all the environmental values influencing the simulation. Note that the individual data sets should be sensibly named as their names will used throughout the simulation to refer to them.

Returns: A <metaRangeEnvironment> object

Examples:

```
# Note: Only for illustration purposes.
# The environment is automatically created when creating a simulation.
metaRangeEnvironment$new(
     sourceSDS = terra::sds(
         terra::rast(vals = 1, nrow = 2, ncol = 2)
     )
)
```

Method `set_current()`**:** Set current (active) time step / environment layer. No reason to call this as user. The current time step is set automatically by the simulation.

Usage:

```
metaRangeEnvironment$set_current(layer)
```

Arguments:

`layer` <integer> layer number.

Returns: <invisible self>

Examples:

```
# Note: Only for illustration purposes.
# The time step is automatically set by the simulation.
sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2, nlyr = 2))
names(sim_env) <- "env_01"
env <- metaRangeEnvironment$new(sourceSDS = sim_env)
env$set_current(layer = 1)
```

Method `print()`**:** Prints information about the environment to the console

Usage:

```
metaRangeEnvironment$print()
```

Returns: <invisible self>

Examples:

```
env <- metaRangeEnvironment$new(
    sourceSDS = terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2, nlyr = 2))
)
env$print()
```

Examples

```
## ------------------------------------------------
## Method `metaRangeEnvironment$new`
## ------------------------------------------------

# Note: Only for illustration purposes.
# The environment is automatically created when creating a simulation.
metaRangeEnvironment$new(
     sourceSDS = terra::sds(
         terra::rast(vals = 1, nrow = 2, ncol = 2)
     )
)

## ------------------------------------------------
## Method `metaRangeEnvironment$set_current`
## ------------------------------------------------

# Note: Only for illustration purposes.
# The time step is automatically set by the simulation.
sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2, nlyr = 2))
names(sim_env) <- "env_01"
env <- metaRangeEnvironment$new(sourceSDS = sim_env)
env$set_current(layer = 1)

## ------------------------------------------------
## Method `metaRangeEnvironment$print`
## ------------------------------------------------

env <- metaRangeEnvironment$new(
    sourceSDS = terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2, nlyr = 2))
)
env$print()
```

`metaRangePriorityQueue` *Process priority queue*

Description

Creates a priority queue in form of an R6 class, that manages the correct process execution order.

Value

<metaRangePriorityQueue> A metaRangePriorityQueue object.

Methods

Public methods:

- metaRangePriorityQueue$new()
- metaRangePriorityQueue$execute_next_process()
- metaRangePriorityQueue$enqueue()
- metaRangePriorityQueue$dequeue()
- metaRangePriorityQueue$sort_future_queue()
- metaRangePriorityQueue$update()
- metaRangePriorityQueue$is_empty()
- metaRangePriorityQueue$get_queue()
- metaRangePriorityQueue$get_future_queue()
- metaRangePriorityQueue$get_current_index()
- metaRangePriorityQueue$print()

Method `new()`**:** Creates a new metaRangePriorityQueue object. Note: No reason to call this as user. The priority queue is created automatically when a simulation is created.

Usage:

```
metaRangePriorityQueue$new()
```

Returns: <metaRangePriorityQueue> A metaRangePriorityQueue object.

Examples:

```
# Only for illustration purposes.
pr_queue <- metaRangePriorityQueue$new()
pr_queue
```

Method `execute_next_process()`**:** Executes the next process in the queue. No reason to call this as user. The next process is executed automatically, when the previous process is finished.

Usage:

```
metaRangePriorityQueue$execute_next_process(verbose)
```

Arguments:

verbose <logical> Should timing and process information be printed when the process is executed?

Returns: <logical> TRUE if the next process has been executed, FALSE if not, in which case the queue is empty.

Examples:

```
# Only for illustration purposes.
pr_queue <- metaRangePriorityQueue$new()
pr <- metaRangeProcess$new("A", "1", \() {message("test")}, 1, new.env())
pr_queue$enqueue(pr)
pr_queue$update()
pr_queue$execute_next_process(verbose = TRUE)
```

Method `enqueue()`**:** Add a process to the (future) queue. Users should only use this method if they added a process to the simulation via the add_process method of the simulation object with the argument queue = FALSE. Otherwise the process is added to the queue automatically.

Usage:

```
metaRangePriorityQueue$enqueue(process)
```

Arguments:

`process` <metaRangeProcess> A metaRangeProcess that should be added to the queue.

Returns: <boolean> `TRUE` on success `FALSE` on failure.

Examples:

```
pr_queue <- metaRangePriorityQueue$new()
pr <- metaRangeProcess$new("A", "1", \() {message("test")}, 1, new.env())
pr_queue$enqueue(pr)
pr_queue$get_future_queue()
```

Method `dequeue()`**:** Remove a process from the (future) queue. Useful to remove a process from the queue if it is no longer needed. E.g. if a species went extinct.

Usage:

```
metaRangePriorityQueue$dequeue(PID = NULL)
```

Arguments:

`PID` <string> the ID of the process, that should be dequeued.

Returns: <boolean> `TRUE` on success `FALSE` on failure.

Examples:

```
pr_queue <- metaRangePriorityQueue$new()
pr <- metaRangeProcess$new("A", "1", \() {message("test")}, 1, new.env())
pr_queue$enqueue(pr)
pr_queue$dequeue(pr$get_PID())
pr_queue$get_future_queue()
```

Method `sort_future_queue()`**:** Sort the (future) queue based on the execution priority. This method is called automatically when a process is added to the queue. Note: No reason to call this as user.

Usage:

```
metaRangePriorityQueue$sort_future_queue()
```

Returns: <invisible self>.

Examples:

```
pr_queue <- metaRangePriorityQueue$new()
pr <- metaRangeProcess$new("A", "1", \() {message("test")}, 1, new.env())
pr_queue$enqueue(pr)
pr_queue$sort_future_queue()
# at least no error
```

Method `update()`**:** Update and reset the queue. This method is called automatically at the end of each time step. Note: No reason to call this as user.

Usage:

```
metaRangePriorityQueue$update()
```

Returns: <invisible self>.

Examples:

```
pr_queue <- metaRangePriorityQueue$new()
pr <- metaRangeProcess$new("A", "1", \() {message("test")}, 1, new.env())
pr_queue$enqueue(pr)
pr_queue$update()
pr_queue$get_queue()
```

Method `is_empty()`**:** Check if the queue is empty.

Usage:

```
metaRangePriorityQueue$is_empty()
```

Returns: <boolean> `TRUE` if queue is empty `FALSE` otherwise.

Examples:

```
pr_queue <- metaRangePriorityQueue$new()
stopifnot(pr_queue$is_empty())
```

Method `get_queue()`**:** Return the current queue.

Usage:

```
metaRangePriorityQueue$get_queue()
```

Returns: <named int vector> The current queue.

Examples:

```
pr_queue <- metaRangePriorityQueue$new()
pr <- metaRangeProcess$new("A", "1", \() {message("test")}, 1, new.env())
pr_queue$enqueue(pr)
pr_queue$update()
pr_queue$get_queue()
```

Method `get_future_queue()`**:** Return the future queue.

Usage:

```
metaRangePriorityQueue$get_future_queue()
```

Returns: <named int vector> The future queue.

Examples:

```
pr_queue <- metaRangePriorityQueue$new()
pr <- metaRangeProcess$new("A", "1", \() {message("test")}, 1, new.env())
pr_queue$enqueue(pr)
pr_queue$get_future_queue()
```

Method `get_current_index()`**:** Get the index of the process that will be executed next.

Usage:

```
metaRangePriorityQueue$get_current_index()
```

Returns: <integer> The index.

Examples:

```
pr_queue <- metaRangePriorityQueue$new()
pr <- metaRangeProcess$new("A", "1", \() {message("test")}, 1, new.env())
pr_queue$enqueue(pr)
pr_queue$update()
pr_queue$get_current_index()
```

Method `print()`**:** Prints information about the queue to the console.

Usage:

```
metaRangePriorityQueue$print()
```

Returns: `<invisible self>`.

Examples:

```
pr_queue <- metaRangePriorityQueue$new()
pr_queue$print()
```

Examples

```
## ------------------------------------------------
## Method `metaRangePriorityQueue$new`
## ------------------------------------------------

# Only for illustration purposes.
pr_queue <- metaRangePriorityQueue$new()
pr_queue

## ------------------------------------------------
## Method `metaRangePriorityQueue$execute_next_process`
## ------------------------------------------------

# Only for illustration purposes.
pr_queue <- metaRangePriorityQueue$new()
pr <- metaRangeProcess$new("A", "1", \() {message("test")}, 1, new.env())
pr_queue$enqueue(pr)
pr_queue$update()
pr_queue$execute_next_process(verbose = TRUE)

## ------------------------------------------------
## Method `metaRangePriorityQueue$enqueue`
## ------------------------------------------------

pr_queue <- metaRangePriorityQueue$new()
pr <- metaRangeProcess$new("A", "1", \() {message("test")}, 1, new.env())
pr_queue$enqueue(pr)
pr_queue$get_future_queue()

## ------------------------------------------------
## Method `metaRangePriorityQueue$dequeue`
## ------------------------------------------------

pr_queue <- metaRangePriorityQueue$new()
pr <- metaRangeProcess$new("A", "1", \() {message("test")}, 1, new.env())
```

```
pr_queue$enqueue(pr)
pr_queue$dequeue(pr$get_PID())
pr_queue$get_future_queue()

## ------------------------------------------------
## Method `metaRangePriorityQueue$sort_future_queue`
## ------------------------------------------------

pr_queue <- metaRangePriorityQueue$new()
pr <- metaRangeProcess$new("A", "1", \() {message("test")}, 1, new.env())
pr_queue$enqueue(pr)
pr_queue$sort_future_queue()
# at least no error

## ------------------------------------------------
## Method `metaRangePriorityQueue$update`
## ------------------------------------------------

pr_queue <- metaRangePriorityQueue$new()
pr <- metaRangeProcess$new("A", "1", \() {message("test")}, 1, new.env())
pr_queue$enqueue(pr)
pr_queue$update()
pr_queue$get_queue()

## ------------------------------------------------
## Method `metaRangePriorityQueue$is_empty`
## ------------------------------------------------

pr_queue <- metaRangePriorityQueue$new()
stopifnot(pr_queue$is_empty())

## ------------------------------------------------
## Method `metaRangePriorityQueue$get_queue`
## ------------------------------------------------

pr_queue <- metaRangePriorityQueue$new()
pr <- metaRangeProcess$new("A", "1", \() {message("test")}, 1, new.env())
pr_queue$enqueue(pr)
pr_queue$update()
pr_queue$get_queue()

## ------------------------------------------------
## Method `metaRangePriorityQueue$get_future_queue`
## ------------------------------------------------

pr_queue <- metaRangePriorityQueue$new()
pr <- metaRangeProcess$new("A", "1", \() {message("test")}, 1, new.env())
pr_queue$enqueue(pr)
pr_queue$get_future_queue()

## ------------------------------------------------
## Method `metaRangePriorityQueue$get_current_index`
## ------------------------------------------------
```

```
pr_queue <- metaRangePriorityQueue$new()
pr <- metaRangeProcess$new("A", "1", \() {message("test")}, 1, new.env())
pr_queue$enqueue(pr)
pr_queue$update()
pr_queue$get_current_index()

## ------------------------------------------------
## Method `metaRangePriorityQueue$print`
## ------------------------------------------------

pr_queue <- metaRangePriorityQueue$new()
pr_queue$print()
```

`metaRangeProcess` *metaRangeProcess object*

Description

Creates an metaRangeProcess object in form of an R6 class that stores and handles all the parts that define a process.

Value

`<metaRangeProcess>` A metaRangeProcess object.

Public fields

`fun <function>` The processes function. This will be called when the process is executed.

Methods

Public methods:

- `metaRangeProcess$new()`
- `metaRangeProcess$get_PID()`
- `metaRangeProcess$get_name()`
- `metaRangeProcess$get_priority()`
- `metaRangeProcess$get_env_label()`
- `metaRangeProcess$print()`

Method `new()`**:** Creates a new metaRangeProcess object

Usage:

```
metaRangeProcess$new(
  process_name,
  id = "",
  process_fun,
  execution_priority,
```

```
  env,
  env_label = NULL
)
```

Arguments:

`process_name` <string> name of the process.

`id` <string> optional ID of the process.

`process_fun` <function> The function to be called when the process is executed. It will be executed in the specified environment (see argument: env) and has access to all the variables in that environment. This function may not have any arguments, i.e. `is.null(formals(process_fun))` must be `TRUE`.

`execution_priority` <integer> the priority of the process. The lower the number the earlier the process is executed (1 == highest priority). Note that the priority is only used to sort the processes in the priority queue. The actual execution order is determined by the order of the processes in the queue.

`env` <environment> the environment where the process should be executed.

`env_label` <string> optional name of the execution environment. Just used as a human readable label for debug purposes.

Returns: <metaRangeProcess> A metaRangeProcess object.

Examples:

```
# Note: Only for illustration purposes. Use the add_process method of the
# simulation object to add processes to a simulation.
pr <- metaRangeProcess$new(
   process_name = "ecological_process",
   process_fun = function() {
      cat("Execute ecological process!")
   },
   execution_priority = 1L,
   env = new.env(),
   env_label = "a_species_name"
)
pr
```

Method `get_PID()`**:** get the process ID

Usage:

```
metaRangeProcess$get_PID()
```

Returns: <string> The process ID

Examples:

```
pr <- metaRangeProcess$new("A", "1", \() {}, 1, new.env())
pr$get_PID()
```

Method `get_name()`**:** get the process name

Usage:

```
metaRangeProcess$get_name()
```

Returns: <string> The process name

Examples:

```
pr <- metaRangeProcess$new("A", "1", \() {}, 1, new.env())
pr$get_name()
```

Method `get_priority()`**:** get the process execution priority

Usage:

```
metaRangeProcess$get_priority()
```

Returns: `<integer>` The process execution priority

Examples:

```
pr <- metaRangeProcess$new("A", "1", \() {}, 1, new.env())
pr$get_priority()
```

Method `get_env_label()`**:** get the name of the process execution environment

Usage:

```
metaRangeProcess$get_env_label()
```

Returns: `<string>` The name of the process execution environment or NULL

Examples:

```
pr <- metaRangeProcess$new("A", "1", \() {}, 1, new.env(), "human_readable_label")
pr$get_env_label()
```

Method `print()`**:** Prints information about the process to the console

Usage:

```
metaRangeProcess$print()
```

Returns: `<invisible self>`

Examples:

```
pr <- metaRangeProcess$new("A", "1", \() {}, 1, new.env())
pr$print()
```

See Also

metaRangePriorityQueue

Examples

```
## ------------------------------------------------
## Method `metaRangeProcess$new`
## ------------------------------------------------

# Note: Only for illustration purposes. Use the add_process method of the
# simulation object to add processes to a simulation.
pr <- metaRangeProcess$new(
   process_name = "ecological_process",
   process_fun = function() {
      cat("Execute ecological process!")
   },
   execution_priority = 1L,
```

```
   env = new.env(),
   env_label = "a_species_name"
)
pr

## ------------------------------------------------
## Method `metaRangeProcess$get_PID`
## ------------------------------------------------

pr <- metaRangeProcess$new("A", "1", \() {}, 1, new.env())
pr$get_PID()

## ------------------------------------------------
## Method `metaRangeProcess$get_name`
## ------------------------------------------------

pr <- metaRangeProcess$new("A", "1", \() {}, 1, new.env())
pr$get_name()

## ------------------------------------------------
## Method `metaRangeProcess$get_priority`
## ------------------------------------------------

pr <- metaRangeProcess$new("A", "1", \() {}, 1, new.env())
pr$get_priority()

## ------------------------------------------------
## Method `metaRangeProcess$get_env_label`
## ------------------------------------------------

pr <- metaRangeProcess$new("A", "1", \() {}, 1, new.env(), "human_readable_label")
pr$get_env_label()

## ------------------------------------------------
## Method `metaRangeProcess$print`
## ------------------------------------------------

pr <- metaRangeProcess$new("A", "1", \() {}, 1, new.env())
pr$print()
```

`metaRangeSimulation` *metaRangeSimulation object*

Description

Creates an simulation object in form of an R6 class that stores and handles all the parts that are necessary to run a simulation.

Value

A <metaRangeSimulation> object.

Public fields

`ID <string>` simulation identification.

`globals <environment>` a place to store global variables. I.e. variables and data that are no specific to one species.

`environment <metaRangeEnvironment>` A metaRangeEnvironment that holds all the environmental values influencing the simulation.

`number_time_steps <integer>` read-only. Will likely be removed in the future in favor of `get_number_of_time_steps()`. Number of time steps in the simulation.

`time_step_layer <integer>` read-only. Will likely be removed in the future in favor of `get_time_layer_mapping()`. Vector of layer IDs that describe which environmental layer to use at each time step. Use `set_stime_layer_mapping` to set these values.

`queue <metaRangePriorityQueue>` manages the order in which the processes should be executed.

`processes <list>` of global (simulation level) `<metaRangeProcess>` (es).

`seed <integer>` seed for the random number generator.

Methods

Public methods:

- `metaRangeSimulation$new()`
- `metaRangeSimulation$add_globals()`
- `metaRangeSimulation$set_time_layer_mapping()`
- `metaRangeSimulation$get_time_layer_mapping()`
- `metaRangeSimulation$get_current_time_step()`
- `metaRangeSimulation$get_number_of_time_steps()`
- `metaRangeSimulation$add_species()`
- `metaRangeSimulation$species_names()`
- `metaRangeSimulation$add_process()`
- `metaRangeSimulation$add_traits()`
- `metaRangeSimulation$exit()`
- `metaRangeSimulation$begin()`
- `metaRangeSimulation$print()`
- `metaRangeSimulation$summary()`

Method `new()`**:** Creates a new metaRangeSimulation object.

Usage:

```
metaRangeSimulation$new(source_environment, ID = NULL, seed = NULL)
```

Arguments:

`source_environment <SpatRasterDataset>` created by `terra::sds()` that represents the environment. The individual data sets represent different environmental variables (e.g. temperature or habitat availability) and the different layer of the data sets represent the different time steps of the simulation. The function metaRangeSimulation`$set_time_layer_mapping()` can be used to extend/ shorten the simulation time steps and set the mapping between each time step and a corresponding environmental layer. This can be used e.g. to repeat the first (few) layer as a burn-in period. The number of layers must be the same for all data sets.

ID <string> optional simulation identification string. Will be set automatically if none is specified.

seed <integer> optional seed for the random number generator. Will be set automatically if none is specified.

Returns: A <metaRangeSimulation> object.

Examples:

```
sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2))
sim <- metaRangeSimulation$new(source_environment = sim_env)
sim
```

Method `add_globals()`**:** Add global variables to the simulation

Usage:

```
metaRangeSimulation$add_globals(...)
```

Arguments:

... <any> the variables to add. Variables to add to the simulation. They will be saved and accessible through the 'globals' field.

Returns: <invisible self>

Examples:

```
sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2))
sim <- metaRangeSimulation$new(source_environment = sim_env)
sim$add_globals(a = 1, b = 2)
sim$globals$a
#> [1] 1
```

Method `set_time_layer_mapping()`**:** Set the time layer mapping of the simulation.

Usage:

```
metaRangeSimulation$set_time_layer_mapping(x)
```

Arguments:

x <integer> vector of layer indices that describe which environmental layer to use for each time step. The length of this vector is equal to the number of time steps.

Returns: <invisible self>

Examples:

```
sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2, nlyr = 4))
sim <- metaRangeSimulation$new(source_environment = sim_env)
sim$set_time_layer_mapping(1:2)
stopifnot(identical(sim$get_time_layer_mapping(), 1:2))
```

Method `get_time_layer_mapping()`**:** Return the time layer mapping of the simulation.

Usage:

```
metaRangeSimulation$get_time_layer_mapping()
```

Returns: <integer> vector of time layer for each time step of the simulation.

Examples:

```
sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2, nlyr = 4))
sim <- metaRangeSimulation$new(source_environment = sim_env)
sim$set_time_layer_mapping(1:2)
sim$get_time_layer_mapping()
```

Method `get_current_time_step()`**:** Get the index of the current time step

Usage:

```
metaRangeSimulation$get_current_time_step()
```

Returns: <integer> the current time step index.

Examples:

```
sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2))
sim <- metaRangeSimulation$new(source_environment = sim_env)
sim$get_current_time_step()
#> [1] 1
```

Method `get_number_of_time_steps()`**:** Get the number of time steps

Usage:

```
metaRangeSimulation$get_number_of_time_steps()
```

Returns: <integer> the number of time steps.

Examples:

```
sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2))
sim <- metaRangeSimulation$new(source_environment = sim_env)
sim$get_number_of_time_steps()
#> [1] 1
```

Method `add_species()`**:** Adds new species to the simulation

Usage:

```
metaRangeSimulation$add_species(names)
```

Arguments:

`names` <character> names of the species to add.

Returns: <invisible boolean> TRUE on success FALSE on failure.

Examples:

```
sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2))
sim <- metaRangeSimulation$new(source_environment = sim_env)
sim$add_species(c("species_1", "species_2"))
sim$species_1
```

Method `species_names()`**:** Returns the names of all species in the simulation.

Usage:

```
metaRangeSimulation$species_names()
```

Returns: <character> vector of species names

Examples:

```
sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2))
sim <- metaRangeSimulation$new(source_environment = sim_env)
sim$add_species("species_1")
sim$add_species("species_2")
sim$species_names()
#> [1] "species_1" "species_2"
```

Method `add_process()`**:** Adds a process to the simulation.

Usage:

```
metaRangeSimulation$add_process(
  species = NULL,
  process_name,
  process_fun,
  execution_priority,
  queue = TRUE
)
```

Arguments:

`species` <character> Names of the species that the process should be added to. If `NULL` the process will be "global", i.e added to the simulation object itself.

`process_name` <string> Name of the process to add.

`process_fun` <named function> The function to call when the process gets executed.

`execution_priority` <positive integer> This number decides when the process should be executed within each time step. The samller the number the earlier it will executed (1 == highest priority i.e. this function will be the executed first). In case multiple processes in the simulation have the same priority, it is assumed that they are independent from each other and there execution order does not matter (i.e. they could be executed in parallel).

`queue` <boolean> If `TRUE` the process will be added to the process execution queue directly. If `FALSE` the process will be added to the simulation but not to the queue, which means that in order to execute the process, it has to be added manually via the metaRangePriorityQueue`$enqueue()` method.

Returns: <invisible self>.

Examples:

```
sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2))
sim <- metaRangeSimulation$new(source_environment = sim_env)
sim$add_species("species_1")
sim$add_process("species_1", "species_process_1", function() {message("process_1")}, 1)
sim$species_1$processes$species_process_1
sim$add_process(species = NULL, "global_process_2", function() {message("process_2")}, 2)
sim$processes$global_process_2
```

Method `add_traits()`**:** Adds traits to a species.

Usage:

```
metaRangeSimulation$add_traits(species, population_level = TRUE, ...)
```

Arguments:

`species` <character> Names of the species that the traits should be added to.

population_level <boolean> If TRUE the traits will be added at the population level (i.e. as a matrix with same dimensions (nrow/ncol) as the environment with one value for each population). This means that the traits either need to be single values that will be extended to such a matrix via base::matrix() or they already need to be a matrix with these dimension. If FALSE the traits will be added without any conversion and may have any type and dimension.

... <atomic> (see base::is.atomic()) The named traits to be added. Named means: Name = value e.g. a = 1.

Returns: <invisible self>.

Examples:

```
sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2))
sim <- metaRangeSimulation$new(source_environment = sim_env)
sim$add_species("species_1")
sim$add_traits("species_1", population_level = TRUE, a = 1)
sim$add_traits("species_1", population_level = FALSE, b = 2, c = "c")
sim$species_1$traits$a
#>      [,1] [,2]
#> [1,]    1    1
#> [2,]    1    1
sim$species_1$traits$b
#> [1] 2
sim$species_1$traits$c
#> [1] "c"
```

Method exit(): When called, will end the simulation (prematurely) once the current process is finished. Useful to e.g. end the simulation safely (i.e. without an error) when no species is alive anymore and there would be no benefit to continue the execution until the last time step.

Usage:

```
metaRangeSimulation$exit()
```

Returns: invisible NULL

Examples:

```
sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2, nlyr = 4))
names(sim_env) <- "env_var_name"
sim <- metaRangeSimulation$new(source_environment = sim_env)
sim$add_species("species_1")
sim$add_process("species_1", "species_process_1", function() {self$sim$exit()}, 1)
sim$begin()
```

Method begin(): Begins the simulation

Usage:

```
metaRangeSimulation$begin()
```

Returns: <invisible self> The finished simulation

Examples:

```
sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2, nlyr = 4))
names(sim_env) <- "env_var_name"
sim <- metaRangeSimulation$new(source_environment = sim_env)
sim$add_process(
     species = NULL,
     "timestep_counter",
     function() {
         message("timestep: ", self$get_current_time_step())
     },
     1
)
sim$begin()
```

Method `print()`**:** Prints information about the simulation to the console

Usage:

```
metaRangeSimulation$print()
```

Returns: <invisible self>

Examples:

```
sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2))
sim <- metaRangeSimulation$new(source_environment = sim_env)
sim$print()
```

Method `summary()`**:** Summarizes information about the simulation and outputs it to the console

Usage:

```
metaRangeSimulation$summary()
```

Returns: <invisible self>

Examples:

```
sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2))
sim <- metaRangeSimulation$new(source_environment = sim_env)
sim$summary()
```

Examples

```
## ------------------------------------------------
## Method `metaRangeSimulation$new`
## ------------------------------------------------

sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2))
sim <- metaRangeSimulation$new(source_environment = sim_env)
sim

## ------------------------------------------------
## Method `metaRangeSimulation$add_globals`
## ------------------------------------------------

sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2))
sim <- metaRangeSimulation$new(source_environment = sim_env)
```

```
sim$add_globals(a = 1, b = 2)
sim$globals$a
#> [1] 1

## ------------------------------------------------
## Method `metaRangeSimulation$set_time_layer_mapping`
## ------------------------------------------------

sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2, nlyr = 4))
sim <- metaRangeSimulation$new(source_environment = sim_env)
sim$set_time_layer_mapping(1:2)
stopifnot(identical(sim$get_time_layer_mapping(), 1:2))

## ------------------------------------------------
## Method `metaRangeSimulation$get_time_layer_mapping`
## ------------------------------------------------

sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2, nlyr = 4))
sim <- metaRangeSimulation$new(source_environment = sim_env)
sim$set_time_layer_mapping(1:2)
sim$get_time_layer_mapping()

## ------------------------------------------------
## Method `metaRangeSimulation$get_current_time_step`
## ------------------------------------------------

sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2))
sim <- metaRangeSimulation$new(source_environment = sim_env)
sim$get_current_time_step()
#> [1] 1

## ------------------------------------------------
## Method `metaRangeSimulation$get_number_of_time_steps`
## ------------------------------------------------

sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2))
sim <- metaRangeSimulation$new(source_environment = sim_env)
sim$get_number_of_time_steps()
#> [1] 1

## ------------------------------------------------
## Method `metaRangeSimulation$add_species`
## ------------------------------------------------

sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2))
sim <- metaRangeSimulation$new(source_environment = sim_env)
sim$add_species(c("species_1", "species_2"))
sim$species_1

## ------------------------------------------------
## Method `metaRangeSimulation$species_names`
## ------------------------------------------------
```

```
sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2))
sim <- metaRangeSimulation$new(source_environment = sim_env)
sim$add_species("species_1")
sim$add_species("species_2")
sim$species_names()
#> [1] "species_1" "species_2"

## ------------------------------------------------
## Method `metaRangeSimulation$add_process`
## ------------------------------------------------

sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2))
sim <- metaRangeSimulation$new(source_environment = sim_env)
sim$add_species("species_1")
sim$add_process("species_1", "species_process_1", function() {message("process_1")}, 1)
sim$species_1$processes$species_process_1
sim$add_process(species = NULL, "global_process_2", function() {message("process_2")}, 2)
sim$processes$global_process_2

## ------------------------------------------------
## Method `metaRangeSimulation$add_traits`
## ------------------------------------------------

sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2))
sim <- metaRangeSimulation$new(source_environment = sim_env)
sim$add_species("species_1")
sim$add_traits("species_1", population_level = TRUE, a = 1)
sim$add_traits("species_1", population_level = FALSE, b = 2, c = "c")
sim$species_1$traits$a
#>      [,1] [,2]
#> [1,]    1    1
#> [2,]    1    1
sim$species_1$traits$b
#> [1] 2
sim$species_1$traits$c
#> [1] "c"

## ------------------------------------------------
## Method `metaRangeSimulation$exit`
## ------------------------------------------------

sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2, nlyr = 4))
names(sim_env) <- "env_var_name"
sim <- metaRangeSimulation$new(source_environment = sim_env)
sim$add_species("species_1")
sim$add_process("species_1", "species_process_1", function() {self$sim$exit()}, 1)
sim$begin()

## ------------------------------------------------
## Method `metaRangeSimulation$begin`
## ------------------------------------------------

sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2, nlyr = 4))
```

```
names(sim_env) <- "env_var_name"
sim <- metaRangeSimulation$new(source_environment = sim_env)
sim$add_process(
     species = NULL,
     "timestep_counter",
     function() {
         message("timestep: ", self$get_current_time_step())
     },
     1
)
sim$begin()

## ------------------------------------------------
## Method `metaRangeSimulation$print`
## ------------------------------------------------

sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2))
sim <- metaRangeSimulation$new(source_environment = sim_env)
sim$print()

## ------------------------------------------------
## Method `metaRangeSimulation$summary`
## ------------------------------------------------

sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2))
sim <- metaRangeSimulation$new(source_environment = sim_env)
sim$summary()
```

`metaRangeSpecies` *metaRangeSpecies object*

Description

Creates an species object in form of an R6 class that stores and handles all the parts that define a species.

Value

A `<metaRangeSpecies>` object.

Public fields

`name` `<string>` name or ID of the species.

`processes` `<list>` of `<metaRangeProcesses>`. The processes that describe how the species interacts with the environment, itself and other species.

`traits` `<environment>` holds the traits of the species.

`sim` `<metaRangeSimulation>` A reference to the metaRangeSimulation simulation object that the species is part of. Useful to access environmental data or data of other species.

Methods

Public methods:

- metaRangeSpecies$new()
- metaRangeSpecies$print()

Method `new()`**:** Creates a new metaRangeSpecies object

Usage:

```
metaRangeSpecies$new(name, sim)
```

Arguments:

`name` <string> name or ID of the species.

`sim` <metaRangeSimulation> A reference to the metaRangeSimulation simulation object that the species is part of. Useful to access environmental data or data of other species.

Returns: A <metaRangeSpecies> object.

Examples:

```
# The following is only for illustration purposes!
# species should be added to a simulation via the `add_species` method
# of the simulation object.
sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2))
test_sim <- metaRangeSimulation$new(source_environment = sim_env)
sp <- metaRangeSpecies$new(name = "species_01", sim = test_sim)
sp
```

Method `print()`**:** Prints information about the species to the console

Usage:

```
metaRangeSpecies$print()
```

Returns: <invisible self>

Examples

```
## ------------------------------------------------
## Method `metaRangeSpecies$new`
## ------------------------------------------------

# The following is only for illustration purposes!
# species should be added to a simulation via the `add_species` method
# of the simulation object.
sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2))
test_sim <- metaRangeSimulation$new(source_environment = sim_env)
sp <- metaRangeSpecies$new(name = "species_01", sim = test_sim)
sp
```

`negative_exponential_function` *Negative Exponential kernel*

Description

Negative Exponential kernel

Usage

```
negative_exponential_function(x, mean_dispersal_dist)
```

Arguments

`x`	<numeric> distance at which the probability should be calculated.
`mean_dispersal_dist`	<numeric> mean dispersal distance. Needs to be (>0).

Details

The negative exponential kernel is defined as:

$$f(x) = \frac{1}{2\pi a^2} e^{-\frac{x}{a}}$$

where a is the mean dispersal distance divided by 2.

Value

<numeric> The probability at distance x.

References

Nathan, R., Klein, E., Robledo-Arnuncio, J.J. and Revilla, E. (2012) Dispersal kernels: review. in: *Dispersal Ecology and Evolution* pp. 187–210. (eds J. Clobert, M. Baguette, T.G. Benton and J.M. Bullock), Oxford, UK: Oxford Academic, 2013. doi:10.1093/acprof:oso/9780199608898.003.0015

Examples

```
negative_exponential_function(1, 1)
```

`plot.metaRangeEnvironment` *Plotting function*

Description

Plots the specified current environment of a metaRangeSimulation object.

Usage

```
## S3 method for class 'metaRangeEnvironment'
plot(x, env_name, col, as_timeseries = FALSE, main = NULL, ...)
```

Arguments

`x`	`<metaRangeEnvironment>` metaRangeEnvironment object.
`env_name`	`<string>` name of the (sub) environment to plot.
`col`	`<character>` colors to use. Defaults to `grDevices::hcl.colors()` with `n = 50` and a random palette.
`as_timeseries`	`<logical>` if `TRUE`, plot the mean of each layer of the (source) environment as a line graph over time, if `FALSE` plot the (current) environment as a raster.
`main`	`<string>` optional title of the plot. Will be labeled automatically when NULL.
`...`	additional arguments passed to terra::plot or base::plot.

Value

`<invisible NULL>`.

Examples

```
layer <- 100
sim_env <- terra::sds(
    terra::rast(
        vals = rnorm(4 * layer),
        nrow = 2,
        ncol = 2,
        nlyr = layer
    )
)
names(sim_env) <- "env_01"
test_sim <- metaRangeSimulation$new(source_environment = sim_env)
test_sim$environment$set_current(1)
plot(test_sim$environment, "env_01")
plot(test_sim$environment, "env_01", as_timeseries = TRUE)
```

plot.metaRangeSimulation *Plotting function*

Description

Plots the specified element of a metaRangeSimulation object.

Usage

```
## S3 method for class 'metaRangeSimulation'
plot(x, obj, name, col, ...)
```

Arguments

x	<metaRangeSimulation> metaRangeSimulation object.
obj	<string> either the string environment or the name of a species.
name	<string> either the name of an environment of the name of a species trait.
col	<character> colors to use. Defaults to grDevices::hcl.colors() with n = 50 and a random palette.
...	additional arguments passed to terra::plot or base::plot.

Value

<invisible NULL>.

Examples

```
sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2))
names(sim_env) <- "env_01"
test_sim <- metaRangeSimulation$new(source_environment = sim_env)
plot(test_sim, "environment", "env_01")

test_sim$add_species("species_01")
test_sim$add_traits("species_01", trait_01 = matrix(1, nrow = 2, ncol = 2))
plot(test_sim, "species_01", "trait_01")

test_sim$add_globals("global_01" = 1:10)
plot(test_sim, "globals", "global_01")
```

plot.metaRangeSpecies *Plotting function*

Description

Plots the specified trait of a metaRangeSpecies object.

Usage

```
## S3 method for class 'metaRangeSpecies'
plot(x, trait_name, col, main = NULL, ...)
```

Arguments

x	<metaRangeSpecies> metaRangeSpecies object.
trait_name	<string> name of the trait to plot.
col	<character> colors to use. Defaults to grDevices::hcl.colors() with n =50 and a random palette.
main	<string> optional title of the plot. Will be labeled automatically when NULL.
...	additional arguments passed to terra::plot or base::plot.

Value

<invisible NULL>.

Examples

```
sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2))
names(sim_env) <- "env_01"
test_sim <- metaRangeSimulation$new(source_environment = sim_env)
test_sim$add_species("species_01")
test_sim$add_traits("species_01", trait_01 = matrix(1:4, nrow = 2, ncol = 2))
plot(test_sim$species_01, "trait_01")
```

print.metaRangeVariableStorage *Print traits or globals*

Description

Print method for species traits and simulation globals.

Usage

```
## S3 method for class 'metaRangeVariableStorage'
print(x, ...)
```

Arguments

x	<metaRangeVariableStorage> The object to print.
...	<any> ignored.

Value

<invisible x>

Examples

```
sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2))
names(sim_env) <- "env_01"
test_sim <- metaRangeSimulation$new(source_environment = sim_env)
test_sim$add_species("species_01")
test_sim$add_traits(species = "species_01", a = 1)
print(test_sim$species_01$traits)
test_sim$add_globals(b = 2)
print(test_sim$globals)
```

---

ricker_allee_reproduction_model

Ricker reproduction model with Allee effects

Description

An implementation of the Ricker reproduction model with Allee effects based on (Cabral and Schurr, 2010) with variable overcompensation and an extension to handle negative reproduction rates.

Usage

```
ricker_allee_reproduction_model(
  abundance,
  reproduction_rate,
  carrying_capacity,
  allee_threshold,
  overcomp_factor = as.numeric(c(1))
)
```

Arguments

abundance	<numeric> vector (or matrix) of abundances.
reproduction_rate	<numeric> vector (or matrix) of reproduction rates.
carrying_capacity	<numeric> vector (or matrix) of carrying capacities.
allee_threshold	<numeric> vector (or matrix) of Allee thresholds.

`overcomp_factor`
: <numeric> overcompensation factor (default: 1.0). Higher values lead to stronger overcompensation. Can also be a vector or matrix.

Details

Equations::

If $reproduction\_rate >= 0$ (based on: Cabral and Schurr, 2010):

$$N_{t+1} = N_t e^{br \frac{(K-N_t)(N_t-C)}{(K-C)^2})}$$

If $reproduction\_rate < 0$:

$$N_{t+1} = N_t \cdot e^r$$

With:

- N_t = abundance at time t
- N_{t+1} = abundance at time t+1
- r = reproduction rate
- K = carrying capacity
- C = (critical) Allee threshold
- b = overcompensation factor

Note that:

- `abundance` should generally be greater than 0.
- `reproduction_rate`, `carrying_capacity` and `allee_threshold` should either all have the same size as the input abundance or all be of length 1.
- `carrying_capacity` should be greater than 0. If it is 0 or less, the abundance will be set to 0.
- `allee_threshold` should be less than `carrying_capacity`. If it is greater than or equal, the abundance will be set to 0.

Important Note: To optimize performance, the functions modifies the abundance in-place. This mean the input abundance will be modified (See Examples). Since the result of this function is usually assigned to the same variable as the input abundance, this is unnoticable in most use cases. Should you wish to keep the input abundance unchanged, you can `rlang::duplicate()` it before passing it to this function.

Value

<numeric> vector (or matrix) of abundances.

References

Cabral, J.S. and Schurr, F.M. (2010) Estimating demographic models for the range dynamics of plant species. *Global Ecology and Biogeography*, **19**, 85–97. doi:10.1111/j.14668238.2009.00492.x

Examples

```
ricker_allee_reproduction_model(
    abundance = 50,
    reproduction_rate = 2,
    carrying_capacity = 100,
    allee_threshold = -100
)
ricker_allee_reproduction_model(
    abundance = 50,
    reproduction_rate = 2,
    carrying_capacity = 100,
    allee_threshold = -100,
    overcomp_factor = 4
)
ricker_allee_reproduction_model(
    abundance = matrix(10, 5, 5),
    reproduction_rate =  0.25,
    carrying_capacity =  100,
    allee_threshold =  20
)
ricker_allee_reproduction_model(
    abundance = matrix(10, 5, 5),
    reproduction_rate =  matrix(seq(-0.5, 0.5, length.out = 25), 5, 5),
    carrying_capacity =  matrix(100, 5, 5),
    allee_threshold =  matrix(20, 5, 5)
)
ricker_allee_reproduction_model(
    abundance = matrix(10, 5, 5),
    reproduction_rate =  matrix(1, 5, 5),
    carrying_capacity =  matrix(100, 5, 5),
    allee_threshold =  matrix(seq(0, 100, length.out = 25), 5, 5)
)
ricker_allee_reproduction_model(
    abundance = matrix(10, 5, 5),
    reproduction_rate =  matrix(seq(0, -2, length.out = 25), 5, 5),
    carrying_capacity =  matrix(100, 5, 5),
    allee_threshold =  matrix(20, 5, 5)
)
# Note that the input abundance is modified in-place
abu <- 10
res <- ricker_allee_reproduction_model(
    abundance = abu,
    reproduction_rate = 0.25,
    carrying_capacity = 100,
    allee_threshold = -100
)
stopifnot(identical(abu, res))
```

`ricker_reproduction_model` *Ricker reproduction model*

Description

An implementation of the Ricker reproduction model (Ricker, 1954) with an extension to handle negative reproduction rates.

Usage

```
ricker_reproduction_model(abundance, reproduction_rate, carrying_capacity)
```

Arguments

`abundance`	<numeric> vector (or matrix) of abundances.
`reproduction_rate`	<numeric> vector (or matrix) of reproduction rates.
`carrying_capacity`	<numeric> vector (or matrix) of carrying capacities.

Details

Equations::

If $reproduction\_rate >= 0$ (Ricker, 1954):

$$N_{t+1} = N_t e^{r(1-\frac{N_t}{K})}$$

If $reproduction\_rate < 0$:

$$N_{t+1} = N_t \cdot e^r$$

With:

- N_t = abundance at time t
- N_{t+1} = abundance at time t+1
- r = reproduction rate
- K = carrying capacity

Note that:

- `abundance` should generally be greater than 0.
- `reproduction_rate` and `carrying_capacity` should either both have the same size as the input abundance or both be of length 1.
- `carrying_capacity` should generally be greater than 0. If it is 0 or less, the abundance will be set to 0.

Important Note: To optimize performance, the functions modifies the abundance in-place. This mean the input abundance will be modified (See Examples). Since the result of this function is usually assigned to the same variable as the input abundance, this is unnoticable in most use cases. Should you wish to keep the input abundance unchanged, you can `rlang::duplicate()` it before passing it to this function.

Value

<numeric> vector (or matrix) of abundances.

References

Ricker, W.E. (1954) Stock and recruitment. *Journal of the Fisheries Research Board of Canada*, **11**, 559–623. doi:10.1139/f54039

Examples

```
ricker_reproduction_model(
    abundance = 10,
    reproduction_rate = 0.25,
    carrying_capacity = 100
)
ricker_reproduction_model(
    abundance = matrix(10, 5, 5),
    reproduction_rate =  0.25,
    carrying_capacity =  100
)
ricker_reproduction_model(
    abundance = matrix(10, 5, 5),
    reproduction_rate =  matrix(seq(-0.5, 0.5, length.out = 25), 5, 5),
    carrying_capacity =  matrix(100, 5, 5)
)
ricker_reproduction_model(
    abundance = matrix(10, 5, 5),
    reproduction_rate =  matrix(seq(0, -2, length.out = 25), 5, 5),
    carrying_capacity =  matrix(100, 5, 5)
)
# Note that the input abundance is modified in-place
abu <- 10
res <- ricker_reproduction_model(
    abundance = abu,
    reproduction_rate = 0.25,
    carrying_capacity = 100
)
stopifnot(identical(abu, res))
```

---

save_species *Save function*

Description

Saves the specified traits of a metaRangeSpecies object.

Usage

```
save_species(x, traits = NULL, prefix = NULL, path, overwrite = FALSE, ...)
```

Arguments

x	<metaRangeSpecies> metaRangeSpecies object.
traits	<character> NULL or a character vector specifying the trait to save. If NULL, all traits are saved.
prefix	<string> prefix for the file names or NULL.
path	<string>path to the directory where the files are saved.
overwrite	<boolean> overwrite existing files.
...	additional arguments passed to terra::writeRaster.

Details

The generated file names are of the form `file.path(path, paste0(prefix, species_name, "_", trait_name, ".file_extension"))`. If the trait is in a matrix or raster form, the file extension is `.tif`. Otherwise it is `.csv`. The prefix is optional and mainly useful to add a time step to the file name, in case the trait is saved multiple times during a simulation.

Value

<invisible character> the paths to the saved files.

Examples

```
sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2))
names(sim_env) <- "env_01"
test_sim <- metaRangeSimulation$new(source_environment = sim_env)
test_sim$add_species("species_01")
test_sim$add_traits(
    "species_01",
    trait_01 = matrix(1, nrow = 2, ncol = 2),
    trait_02 = matrix(2, nrow = 2, ncol = 2)
)

file_prefix <- "This_could_be_a_time_step"
directory_name <- tempdir()

res_path <- save_species(
    test_sim$species_01,
    traits = "trait_01",
    prefix = file_prefix,
    path = directory_name
)
# the following should be TRUE
# but might fail due to floating point errors (that's why we round the values)
identical(
    round(terra::as.matrix(terra::rast(res_path), wide = TRUE)),
    round(test_sim$species_01$traits[["trait_01"]])
)

# test overwrite
res_path2 <- save_species(
    test_sim$species_01,
```

```
    traits = "trait_01",
    prefix = file_prefix,
    path = directory_name,
    overwrite = TRUE
)
stopifnot(identical(res_path, res_path2))

# Saving all traits
res_path3 <- save_species(
    test_sim$species_01,
    prefix = basename(tempfile()),
    path = directory_name
)
res_path3
# cleanup
unlink(c(res_path, res_path3))
stopifnot(all(!file.exists(res_path, res_path3)))
```

`set_verbosity` *Set verbosity of metaRange simulation*

Description

Just a wrapper for `options(metaRange.verbose = [0 | 1 | 2])` but documented. If `0`, metaRange functions will print no messages to the console. If `1`, metaRange functions will print some messages to the console. If `2`, metaRange functions will print many messages to the console.

Usage

```
set_verbosity(verbose)
```

Arguments

`verbose`	`<integer>` message verbosity (see description).

Value

`<invisible list>` a list with the previous verbosity setting.

Examples

```
set_verbosity(0)
getOption("metaRange.verbose")
```

summary.metaRangeSimulation *Summary for metaRange simulation*

Description

Print a summary of the simulation to the console.

Usage

```
## S3 method for class 'metaRangeSimulation'
summary(object, ...)
```

Arguments

object	<metaRangeSimulation> The metaRangeSimulation object to summarize.
...	<any> ignored.

Value

<invisible NULL>

Examples

```
sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2))
names(sim_env) <- "env_01"
test_sim <- metaRangeSimulation$new(source_environment = sim_env)
test_sim$add_species("species_01")
summary(test_sim)
```

summary.metaRangeSpecies *Summary for metaRange species*

Description

Summary for metaRange species

Usage

```
## S3 method for class 'metaRangeSpecies'
summary(object, ...)
```

Arguments

object	<metaRangeSpecies> The metaRangeSpecies object to summarize.
...	<any> ignored.

Value

<invisible NULL>

Examples

```
sim_env <- terra::sds(terra::rast(vals = 1, nrow = 2, ncol = 2))
names(sim_env) <- "env_01"
test_sim <- metaRangeSimulation$new(source_environment = sim_env)
test_sim$add_species("species_01")
summary(test_sim$species_01)
```

Index