Package: growR (via r-universe)

August 29, 2024

Type Package

Version 1.3.0.9001

Date 2024-05-23

Title Implementation of the Vegetation Model ModVege

Description Run grass growth simulations using a grass growth model based on ModVege (Jouven, M., P. Carrère, and R. Baumont ``Model Predicting Dynamics of Biomass, Structure and Digestibility of Herbage in Managed Permanent Pastures. 1. Model Description." (2006) <doi:10.1111/j.1365-2494.2006.00515.x>). The implementation in this package contains a few additions to the above cited version of ModVege, such as simulations of management decisions, and influences of snow cover. As such, the model is fit to simulate grass growth in mountainous regions, such as the Swiss Alps. The package also contains routines for calibrating the model and helpful tools for analysing model outputs and performance.

URL https://github.com/kuadrat/growR, https://kuadrat.github.io/growR/

BugReports https://github.com/kuadrat/growR/issues

License MIT + file LICENSE

RoxygenNote 7.2.3

Encoding UTF-8

Imports rlang, R6, utils, Rdpack

Suggests ggplot2, knitr, patchwork, rmarkdown, testthat (>= 3.0.0)

Depends R (>= 2.10)

RdMacros Rdpack

LazyData true

Config/testthat/edition 3

Config/testthat/start-first setup, pscan, run

VignetteBuilder knitr

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

RemoteUrl https://github.com/kuadrat/growr

RemoteRef HEAD

RemoteSha 017ae9af011ea63ddbffc7d4961da8a708189ba0

Contents

`aCO2_inverse` *Concentration representative year*

Description

Inverse of 'atmospheric_CO2': retrieve the year by which a given CO2 concentration is reached.

Usage

```
aCO2_inverse(aCO2)
```

Arguments

`aCO2`	Target CO2 concentration in ppm.

Details

Does not give a reasonable result for values below 317ppm, corresponding to the year 1949, as this is where the minimum of the parabola is located in the second order fit to the data that was used in aCO2.fct.

Value

year Approximate year (as floating point number) by which target concentration is reached.

Examples

```
aCO2_inverse(420)
aCO2_inverse(700)
# Insensible
aCO2_inverse(100)
```

`add_lines` *Add data to a ggplot*

Description

Add a lineplot of the *x_key* and *y_key* columns in *data* to the supplied ggplot object *ax*. If none is supplied, a new one is created.

Usage

```
add_lines(
  data,
  ax = NULL,
  y_key = "dBM_smooth",
  x_key = "DOY",
  style = "line",
  label = NULL,
  ...
)
```

Arguments

`data`	data.frame or similar object interpretable by ggplot.
`ax`	list as returned by ggplot() and related functions.
`x_key, y_key`	Column names in *data* to be plotted.
`style`	XXX in ggplot geom_XXX to use.
`label`	Codename for this line to be used in legend creation. If NULL, use *y_key*.
`...`	All further arguments are passed to the selected ggplot geom.

Value

ax A ggplot list (like the input *ax*).

Examples

```
library(ggplot2)
# Add first set of data
ax = add_lines(mtcars, x_key = "wt", y_key = "mpg", label = "First Line")

# Add one more line to the plot
ax = add_lines(mtcars, ax = ax, x_key = "wt", y_key = "qsec",
label = "Second Line")

print(ax)
```

`analyze_parameter_scan` *Analyze results of a parameter scan*

Description

Analyze results of a parameter scan

Usage

```
analyze_parameter_scan(
  parameter_scan_results,
  datafile = "",
  smooth_interval = 28
)
```

Arguments

`parameter_scan_results`	String or List. If a string, it is interpreted as the name of a `rds` file that contains the results of a parameter scan which is then loaded using `readRDS()`. Otherwise, it should be the output of `run_parameter_scan()` directly.
`datafile`	Name or path to a file containing measured data. The model results in *parameter_scan_results* are compared to the data therein. If empty, the site is inferred from the ModvegeSite objects in *parameter_scan_results* and a corresponding data file is searched for in 'getOption("growR.data_dir", default = "data").
`smooth_interval`	Int. Number of days over which the variable `dBM` is smoothened. Should be set to make experimental data and simulated data to be as comparable as possible.

Value

analyzed A list with five keys: `dBM`, `cBM`, `cBM_end`, `metrics` and `params`.

dBM A data.frame with `1 + n_params + n_metrics` columns where each row represents a different parameter combination. The first column (`n`) gives the row number and is used to identify a parameter combination. The subsequent `n_params` columns give the values of the parameters used in this combination. The final `n_metrics` columns give the resulting performance score of the model run with these parameters for each metric applied to model variable `dBM`.

cBM A data.frame of same format as for the key *dBM*. The first `n_params + 1` columns are identical to the data.frame in *dBM*. The difference is that the final `n_metrics` columns give performance scores with respect to the model variable `cBM`.

cBM_end A data.frame analogous to *dBM* and *cBM*, only this time the last `n_metrics` columns give performance scores with respect to the variable `cBM_end`, which is the final value of `cBM`, i.e. the cumulative grown biomass at the end of the year.

params A vector containing the names of the scanned parameters. These are also the column names of columns `2:(n_params+1)` in *results*.

metrics A vector containing the names of the employed performance metrics. These are also the column names of the last `n_metrics` columns in *results*.

See Also

`run_parameter_scan()`, `readRDS()`

Examples

```
# There needs to be data available with which the model is to be compared.
# For this example, use data provided by the package.
path = system.file("extdata", package = "growR")
datafile = file.path(path, "posieux1.csv")

# We also use example parameter scan data provided by the package.
# In the real world, you would generally create your own data using
# `run_parameter_scan()`.
analyze_parameter_scan(parameter_scan_example, datafile = datafile)
```

---

append_to_table *Write *data* to supplied file in append mode without generating a warning message.*

Description

This function essentially wraps ‘write.table‘ with a calling handler that suppresses appending warnings that would appear with the argument ‘col.names = TRUE‘.

Usage

```
append_to_table(data, filename, ...)
```

Arguments

data	Any object which can be handled by ‘write.table‘.
filename	Name of file to append to.
...	All additional arguments are passed to ‘write.table‘.

---

atmospheric_CO2 *Atmospheric CO2 concentration*

Description

Retrieve CO2 concentration (in ppm) for given calendar year.

Usage

```
atmospheric_CO2(year)
```

Arguments

year	Calender year for which to extract CO2 concentration.

Details

This function defines the CO2 concentration as a function of calendar year. it is based on a polynomial fit to the annual CO2 data published by NOAA <https://gml.noaa.gov/webdata/ccgg/trends/co2/co2_annmean_gl.txt>

Value

Approximate CO2 concentration in ppm for given year.

Note

This is only approximately valid for years in the range 1949 - 2020

Examples

```
atmospheric_CO2(1990)
atmospheric_CO2(2020)
# Insensible
atmospheric_CO2(1800)
```

Autocut *Autocut*

Description

An algorithm to determine grass cut dates if none are provided. This is an abstract class and not intended for direct use. Instead, use its subclasses that implement a `determine_cut()` method.

The expected number of cuts is estimated from management intensity and site altitude based on data for Swiss grasslands by Huguenin et al.

Public fields

`MVS` The ModvegeSite object for which to take the cut decision.

`n_cuts` Number of cuts expected for this altitude and management intensity.

Methods

Public methods:

- `Autocut$new()`
- `Autocut$get_expected_n_cuts()`
- `Autocut$determine_cut()`
- `Autocut$clone()`

Method `new()`**:** Constructor

Usage:

```
Autocut$new(MVS)
```

Arguments:

`MVS` The ModvegeSite object for which cuts shall be determined.
Get number of expected cuts
Return the number of expected cuts for a site at a given *elevation* and management *intensity*. This uses data.frame `management_parameters` as a lookup table and interpolates linearly in between the specified values.

Method `get_expected_n_cuts()`**:**

Usage:

```
Autocut$get_expected_n_cuts(elevation, intensity = "high")
```

Arguments:

`elevation` The elevation of the considered site in meters above sea level.

`intensity` One of ("high", "middle", "low", "extensive"). Management intensity for considered site.

Returns: Number of expected cuts per season.

Method `determine_cut()`**:** Empty method stub intended for overriding by inheriting subclasses.

Usage:

```
Autocut$determine_cut(DOY)
```

Arguments:

`DOY` Integer day-of-the-year.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

References

Huguenin-Elie I, Mosimann E, Schlegel P, Lüscher A, Kessler W, Jeangros B (2017). "Grundlagen für die Düngung landwirtschaftlicher Kulturen in der Schweiz (GRUD), Kapitel 9: Düngung von Grasland." *Agrarforschung Schweiz*. https://www.agrarforschungschweiz.ch/2017/06/9-duengung-von-grasland-grud-2017/.

See Also

management_parameters

`box_smooth` *Endpoint smoother*

Description

Smooth data in vector *x* to its endpoint.

Usage

```
box_smooth(x, box_width = 28)
```

Arguments

`x`	1D data to be smoothed.
`box_width`	Width (in units of vector steps) of the box used for smoothing.

Details

Employ an endpoint box filter (aka "running mean" or endpoint smoother) to the 1-D data in *x*:

```
x_smoothed[i] = mean(x[i-box_width:i])
```

Where *x* is considered to be *zero-padded* vor values of *i-box_width* < 1.

Value

x_smooth Smoothened version of *x*.

Examples

```
# Create a sine wave with noise
x = seq(0, 4*pi, 0.1)
y = sin(x) + runif(length(x))
# Apply endpoint smoothing
y_smooth = box_smooth(y, box_width = 5)
```

`browse` *Debugging utilities*

Description

Debug specified function *func* by entering a `browser()` right at the beginning (`browse()`) or end (`browse_end()`) of the function.

Usage

```
browse(func, ...)

browse_end(func, ...)
```

Arguments

`func`	An R function to be browsed.
`...`	Arguments to the function *func* that is to be browsed.

Details

These are convenience shorthands for R's builtin debug tools, like `debugonce()` and the `trace()`/`untrace()` combination.

Value

Returns the result of `func(...)`. Enters a `browser()`.

Functions

- `browse_end()`: Enter `browser()` at the end of the function call to `func(...)`. This only works, if the function can execute without error until its end. Otherwise, the error will be thrown.

See Also

browser(), debugonce(), trace()

trace()

Examples

```
# Define a simple function for this example
my_func = function(a) { for (i in 1:5) { a = a + i }; return(a) }

# Enter a browser at the beginning of the function
browse(my_func, 0)

# Enter a browser at the end of the function. This allows us to inspect
# the function's local variables without having to go through the whole loop.
browse_end(my_func, 0)
```

---

build_functional_group

Build the effective functional group as a weighted linear combination.

Description

Uses the weights found in :param:P to construct the effective functional groups and updates functional group parameters in P.

Usage

```
build_functional_group(P)
```

Arguments

P	list; name-value pairs of parameters. Should contain at least one non-zero functional group weight w_FGX with X in (A, B, C, D). Any weights not present are assumed to be 0.

Value

A FunctionalGroup object composed of a linear combination of the four groups *FG_A*, *FG_B*, *FG_C* and *FG_D*.

See Also

FunctionalGroup

Examples

```
parameters = list(w_FGA = 0.5, w_FGB = 0.5)
build_functional_group(parameters)

# The w_FGX weights in the input are interpreted as relative to each other.
# Thus, they do not need to satisfy the sum rule. The following is
# equivalent to the previous example:
parameters = list(w_FGA = 1, w_FGB = 1)
build_functional_group(parameters)
```

---

check_for_package *Check if *package* is available*

Description

Some functions not pertaining to the package core require additional libraries. These libraries are listed as *suggested* in the 'DESCRIPTION' When such a function is called by a user who does not have the respective libraries installed, we should notice that and notify the user. This is the purpose of this *function* 'check_for_package'.

Usage

```
check_for_package(package, stop = TRUE)
```

Arguments

package	Name of the package to check for.
stop	Toggle whether an error should be thrown ('TRUE') or a warning generated ('FALSE').

Details

The function checks if *package* is installed and loaded. If not, it either produces a warning or throws an error, depending on the value of *stop*.

Value

'TRUE' if the package was found. 'FALSE' if it wasn't found and *stop* is 'FALSE'. Otherwise, an error will be thrown.

Combinator *Combinator*

Description

Helps to find all possible combinations for a given set of values.

Public fields

`combinations` list Once run, holds all valid parameter combinations as named lists.

`eps` float Numerical precision to require when checking the functional group weight sum criterion.

Methods

Public methods:

- `Combinator$create_combinations()`
- `Combinator$clone()`

Method `create_combinations()`**:** Find possible combinations

Usage:

```
Combinator$create_combinations(param_values)
```

Arguments:

`param_values` A list giving all options for the parameter values which are to be combined. As an example:

```
list(w_FGA = c(0, 0.5, 1), w_FGB = c(0, 0.5, 1), NI = c(0.5, 0.9))
```

This would generate the combinations

w_FGA	w_FGB	NI
0	1	0.5
0	1	0.9
0.5	0.5	0.5
0.5	0.5	0.9
1	0	0.5
1	0	0.9

`eps` Precision to be used when checking if the sum citerion of the functional groups (w_FGA + w_FGB + w_FGC + w_FGD = 1) is fulfilled.

Returns: combinations A list containing vectors of parameter value combinations.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

create_combinations()

---

compare.R *Compare simulation results*

Description

The script compare.R which ships with the growR package and is automatically put into the working directory with setup_directory() can be used to compare results of growR simulation runs.

It is a simple script, so it can and should be adjusted to your personal needs.

Details

The script makes use of the packages ggplot2, patchwork and some growR functions which facilitate data loading and plotting, like load_measured_data() and add_lines().

---

create_combinations *Create Valid Combinations*

Description

Generate a list which contains all possible combinations of the provided parameter values. This excludes combinations that are invalid because the sum criterion for functional groups w_FGA + w_FGB + w_FGC + w_FGD = 1 is not fulfilled.

Usage

```
create_combinations(param_values, eps = 0.02)
```

Arguments

param_values	A list giving all options for the parameter values which are to be combined. The format is list[[param_name]] = param_values where param_values is a vector with the values for the respective parameter. The parameter names for functional group weights (w_FGX with X in (A, B, C, D)) receive special treatment and therefore need to be spelled correctly.
eps	Float specifying the precision to which the sum criterion for functional group has to be satisfied. The criterion is considered satisfied, if "' abs(w_FGA + w_FGB + w_FGC + w_FGD) - 1) <= eps

Details

Assume for example the following list as argument *param_values*:

```
list(w_FGA = c(0, 0.5, 1), w_FGB = c(0, 0.5, 1), NI = c(0.5, 0.9))
```

This would generate the combinations

w_FGA	w_FGB	NI
0	1	0.5
0	1	0.9
0.5	0.5	0.5
0.5	0.5	0.9
1	0	0.5
1	0	0.9

One can see that the input *param_values* has to be set up carefully: one has to ensure that the given `w_FGX` values *can* actually add up to 1. The following would be a bad counterexample, where only one single valid combination is found, even though many values for `w_FGA` and `w_FGB` are provided:

```
list(w_FGA = seq(0.5, 1, 0.01), w_FGB = c(0.5, 1, 0.01))
```

Similarly, if the steps in the `w_FGX` don't match, we might not end up with many valid combinations, even though the ranges are reasonabl:

```
list(w_FGA = seq(0.5, 1, 0.1), w_FGB = c(0, 0.5, 0.25))
```

Here, no combination can be made with `w_FGA` in `c(0.6, 0.7, 0.8, 0.9)` or `w_FGB = 0.25`.

Value

combinations An unnamed list where every entry is a list containing the parameter values (named as in the input *param_values*) for a valid combination.

Examples

```
# Define the parameter steps you want to explore. This is a minimal example.
# A more realistic one follows below.
param_values = list(w_FGA = c(0, 0.5, 1),
                    w_FGB = c(0, 0.5, 1),
                    NI = c(0.5, 0.9)
)
# Create all valid combinations of the defined steps
create_combinations(param_values)

# More realistic example for an initial exploration of parameter space,
# where we suspect that functional groups A and B should be more prevalent
# than C and D. This produces 54 parameter combinations, which is a number
# of model evaluations that can run within a reasonable timeframe
```

```
# (depending on your system).
param_values = list(w_FGA = seq(0, 1, 0.33),
                    w_FGB = seq(0, 1, 0.33),
                    w_FGC = seq(0, 0.7, 0.33),
                    w_FGD = seq(0, 0.7, 0.33),
                    NI = seq(0.5, 1.0, 0.25)
)
length(create_combinations(param_values))

# The default value for *eps* made sure that combinations of 0.33 + 0.66 =
# 0.99 etc. are considered "valid". If we make *eps* too small, no valid
# combinations can be found:
length(create_combinations(param_values, eps = 1e-3))
```

`create_example_environment` *Provide an example ModvegeEnvironment*

Description

This is intended for testing and for the examples in the documentation.

Usage

```
create_example_environment(site = "posieux")
```

Arguments

`site`	Choose for which example site an environment is to be created. Options: `"posieux"`, `"sorens"`.

Value

E A `[ModvegeEnvironment]` instance based on the example data for *site* which is shipped with this package.

Examples

```
extdata = system.file("extdata", package = "growR")
print(extdata)
list.files(extdata, recursive = TRUE)
create_example_environment()
```

ensure_table_columns *Check if supplied table contains all *required* variables.*

Description

Logs an error if any variable is missing and lists the missing variables in the error message along with *data_name*.

Usage

```
ensure_table_columns(required, data, data_name = "the data table")
```

Arguments

required	List of names of required variables.
data	data.frame or similar object to be checked.
data_name	Name to be displayed in the error message if any variable is missing.

ensure_unique_filename *Replace given filename by a version that contains an incremental number in order to prevent overwriting existing files.*

Description

Replace given filename by a version that contains an incremental number in order to prevent overwriting existing files.

Usage

```
ensure_unique_filename(path, add_num = TRUE)
```

Arguments

path	string; Filename including path for which to check uniqueness.
add_num	boolean; if TRUE, add the incremental number anyways, even if no filename conflict exists.

Value

A unique filename.

`fCO2_growth_mod` *CO2 growth modifier*

Description

Function describing the effects of elevated CO2 on growth.

Usage

```
fCO2_growth_mod(c_CO2, b = 0.5, c_ref = 360)
```

Arguments

`c_CO2`	numeric Atmospheric CO2 concentration in ppm
`b`	numeric Strength of CO2 effect on growth. Kellner et al. report values bewtween 0 and 2 with the interval of highest likelihood (0.1, 0.3). However, Soltani and Sinclair discuss that b = 0.4 in C4 plants and b = 0.8 in C3 plants. The difference on the output of this function of choosing a small (0.1) and large (0.8) value for b has an effect on the result for an atmospheric concentration of 700 ppm of roughly 40 percent!.
`c_ref`	numeric Reference CO2 concentration in ppm.

Details

The function for the effects on growth is as proposed by Soltani et al (2012) and later adapted by equation (5) in Kellner et al. (2017)

References

Soltani A, Sinclair TR (2012). *Modeling Physiology of Crop Development, Growth and Yield*. CABI. ISBN 978-1-84593-971-7, xnHT6YOlk00C.

Kellner J, Multsch S, Houska T, Kraft P, Müller C, Breuer L (2017). "A Coupled Hydrological-Plant Growth Model for Simulating the Effect of Elevated CO2 on a Temperate Grassland." *Agricultural and Forest Meteorology*, **246**, 42–50. ISSN 0168-1923, doi:10.1016/j.agrformet.2017.05.017, https://www.sciencedirect.com/science/article/pii/S0168192317301831.

Examples

```
fCO2_growth_mod(420)
# The modifier is always relative to *c_ref*. This returns 1.
fCO2_growth_mod(420, c_ref = 420)
```

`fCO2_transpiration_mod` *CO2 transpiration modifier*

Description

Function describing the effects of elevated CO2 on transpiration.

Usage

```
fCO2_transpiration_mod(c_CO2, c_ref = 360)
```

Arguments

`c_CO2`	numeric Atmospheric CO2 concentration in ppm
`c_ref`	numeric Reference CO2 concentration in ppm.

Details

The function for the effect on transpiration is from equations (2-6) in Kruijt et al.

It appears in this paper that there is a small formal mistake in said equations. With the stated values, it is not possible to reproduce the tabulated values of c close to 1, as in their table 3. Instead, we conclude that equation (4) should read:

```
c = 1 + s_gs * s_T * F_T * deltaCO2
```

with the multiplicative terms giving small negative numbers. The factors s_gs, s_T and F_T for grasslands are taken from pages 260 and 261 in Kruijt et al. where we averaged over the stated ranges to get:

```
c ~= 1 + 0.0001 * deltaCO2
```

References

Kruijt B, Witte JM, Jacobs CMJ, Kroon T (2008). "Effects of Rising Atmospheric CO2 on Evapotranspiration and Soil Moisture: A Practical Approach for the Netherlands." *Journal of Hydrology*, **349**(3), 257–267. ISSN 0022-1694, doi:10.1016/j.jhydrol.2007.10.052, https://www.sciencedirect.com/science/article/pii/S0022169407006373.

Examples

```
fCO2_transpiration_mod(420)
# The modifier is always relative to *c_ref*. This returns 1.
fCO2_transpiration_mod(420, c_ref = 420)
```

FG_A *Functional group A*

Description

Functional group A

Usage

```
FG_A
```

Format

An object of class `FunctionalGroup` (inherits from `R6`) of length 23.

See Also

[FunctionalGroup]

FG_B *Functional group B*

Description

Functional group B

Usage

```
FG_B
```

Format

An object of class `FunctionalGroup` (inherits from `R6`) of length 23.

See Also

[FunctionalGroup]

FG_C *Functional group C*

Description

Functional group C

Usage

```
FG_C
```

Format

An object of class `FunctionalGroup` (inherits from `R6`) of length 23.

See Also

[FunctionalGroup]

FG_D *Functional group D*

Description

Functional group D

Usage

```
FG_D
```

Format

An object of class `FunctionalGroup` (inherits from `R6`) of length 23.

See Also

[FunctionalGroup]

fPAR *Radiation limitation*

Description

Threshold function representing growth limitation due to lack of photosynthetically active radiation (PAR).

Usage

```
fPAR(PAR)
```

Arguments

PAR	float Photosynthetically active radiation in MJ/m^2

Value

A value in the range [0, 1], acting as a multiplicative factor to plant growth.

Examples

```
fPAR(4)
```

fT *Temperature limitation*

Description

Threshold function representing growth limitation by temperature.

Usage

```
fT(t, T0 = 4, T1 = 10, T2 = 20)
```

Arguments

t	float Temperature in degree Celsius.
T0	float Photosynthesis activation temperature in degree Celsius.
T1	float Photosynthesis plateau temperature in degree Celsius.
T2	float Photosynthesis max temperature in degree Celsius.

Details

Photosynthesis is suppressed below *T0*, increases until it reaches its maximum at temperatures in the interval (T1, T2). For temperatures exceeding *T2*, photosynthetic activity decreases again until it reaches 0 at a final temperature of 40 degree Celsius.

Value

A value in the range (0, 1), acting as a multiplicative factor to plant growth.

Examples

```
fT(4)
fT(10)
fT(15)
```

FunctionalGroup *Representation of a grassland plant population*

Description

A functional group is a representation of a grassland plant population with certain functional attributes.

It contains many plant parameters that are collected under the hood of functional groups. The class implements S3 style operator overloading such that one can do things like

```
mixed_functional_group = 0.8 * FG_A + 0.2 * FG_B
```

Public fields

- `SLA` Specific Leaf Area in m^2^ per g.
- `pcLAM` Percentage of laminae (number between 0 and 1).
- `ST1` Temperature sum in degree Celsiues days after which the seasonality function `SEA` starts to decrease from its maximum plateau. See also `SEA()`.
- `ST2` Temperature sum in degree Celsiues days after which the seasonality function `SEA` has decreased back to its minimum value. See also `SEA()`.
- `maxSEA` Maximum value of the seasonality function `SEA()`
- `minSEA` Minimum value of the seasonality function `SEA()`. Usually, `minSEA = 1 - (maxSEA - 1)`.
- `maxOMDGV` Maximum organic matter digestability for green vegetative matter in arbitrary units.
- `minOMDGV` Minimum organic matter digestability for green vegetative matter in arbitrary units.
- `maxOMDGR` Maximum organic matter digestability for green reproductive matter in arbitrary units.

- `minOMDGR` Minimum organic matter digestability for green reproductive matter in arbitrary units.
- `BDGV` Bulk density of green vegetative dry matter in g per m^3^.
- `BDDV` Bulk density of dead vegetative dry matter in g per m^3^.
- `BDGR` Bulk density of green reproductive dry matter in g per m^3^.
- `BDDR` Bulk density of dead reproductive dry matter in g per m^3^.
- `fg_parameter_names` Vector of strings of the variable names of all vegetation parameters governed by functional group composition.

Default values for parameters are taken from functional group A in Jouven et al.

Public fields

`fg_parameter_names` Names of the vegetation parameters governed by functional group composition.

Methods

Public methods:

- `FunctionalGroup$new()`
- `FunctionalGroup$get_parameters()`
- `FunctionalGroup$get_parameters_ordered()`
- `FunctionalGroup$set_parameters()`
- `FunctionalGroup$set_parameters_ordered()`
- `FunctionalGroup$clone()`

Method `new()`**:** Constructor

Usage:

```
FunctionalGroup$new(...)
```

Arguments:

`...` Key-value pairs of parameters to be set.

Method `get_parameters()`**:** Convenient getter

Returns all parameters with their names in a list.

Usage:

```
FunctionalGroup$get_parameters()
```

Method `get_parameters_ordered()`**:** Ordered getter

Returns all parameters in reproducible order in a vector.

Usage:

```
FunctionalGroup$get_parameters_ordered()
```

Method `set_parameters()`**:** Convenient setter

Set all specified parameters.

Usage:

```
FunctionalGroup$set_parameters(...)
```

Arguments:

`...` Key-value pairs of parameters to be set.

Method `set_parameters_ordered()`**:** Efficient setter, assumes parameters come in known order.

Usage:

```
FunctionalGroup$set_parameters_ordered(ordered_parameter_values)
```

Arguments:

`ordered_parameter_values` Parameter values to be set. Need to be in the same order as pFunctionalGroup]`$fg_parameter_names`.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

References

Jouven M, Carrère P, Baumont R (2006). "Model Predicting Dynamics of Biomass, Structure and Digestibility of Herbage in Managed Permanent Pastures. 1. Model Description." *Grass and Forage Science*, **61**(2), 112–124. ISSN 1365-2494, doi:10.1111/j.13652494.2006.00515.x, https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1365-2494.2006.00515.x.

`fW` *Water stress*

Description

Threshold function representing growth limitation due to water stress.

Usage

```
fW(W, PET)
```

Arguments

`W`	Water stress given as the ratio of water reserves to water holding capacity.
`PET`	Potential evapotranspiration in mm per day.

Details

After equation (6) in McCall et al. (2003).#'

Value

A value in the range (0, 1), acting as a multiplicative factor to plant growth.

References

McCall DG, Bishop-Hurley GJ (2003). "A Pasture Growth Model for Use in a Whole-Farm Dairy Production Model." *Agricultural Systems*, **76**(3), 1183–1205. ISSN 0308-521X, doi:10.1016/S0308521X(02)00104X, https://www.sciencedirect.com/science/article/pii/S0308521X0200104X.

Examples

```
fW(0.5, 7)
fW(0.5, 5)
fW(0.5, 3)
```

`get_bias` *Metric Functions*

Description

Functions to calculate different performance metrics.

In the case of *get_bias*: Calculate the bias b, i.e. the average difference between predicted y and observed z values:

```
bias = mean(y - z)
```

Usage

```
get_bias(predicted, observed, ...)

root_mean_squared(predicted, observed, ...)

mean_absolute_error(predicted, observed, ...)
```

Arguments

`predicted`	Vector containing the predictions y.
`observed`	Vector containing the observations z.
`...`	**relative** Boolean. If true give the result as a ratio to the average observation `mean(ovserved)`.

Value

m A number representing the relative or absolute value for the metric.

Functions

- `root_mean_squared()`: Calculate the square root of the average squared difference between prediction and observation:

  `RMSE = sqrt(sum(predicted - observed)^2) / length(predicted)`

- `mean_absolute_error()`: Calculate the average of the absolute differences between prediction and observation:

  `MAE = mean(abs(predicted - observed))`

Note

NA values are completely ignored.

See Also

`willmott()`

Examples

```
predicted = c(21.5, 22.2, 19.1)
observed = c(20, 20, 20)
get_bias(predicted, observed)
get_bias(predicted, observed, relative = FALSE)

root_mean_squared(predicted, observed)
root_mean_squared(predicted, observed, relative = FALSE)

mean_absolute_error(predicted, observed)
mean_absolute_error(predicted, observed, relative = FALSE)
```

`get_site_name` *Extract the name of a site from a filename*

Description

This function assumes the filenames to begin with the site name, potentially followed by an underscore and further characters.

Usage

```
get_site_name(filename)
```

Arguments

`filename`	String of a ‘ModvegeSite‘ output filename.

growR_package_options *Default options introduced by package growR*

Description

These are the default options, set when the package is loaded by 'library(growR)'. To get or change the current value of an option, use the 'options()' function.

growR.verbosity Integer that controls how much console output is generated by growR functions. Higher numbers mean more output. See [logger()].

growR.input_dir Name of the directory in which to look for input files.

growR.output_dir Name of the directory into which output files are written.

growR.data_dir Name of the directory in which to look for measured data files.

Usage

```
growR_package_options
```

Format

An object of class `list` of length 4.

See Also

[options()]

growR_run_loop *Run growR simulations*

Description

Start the loop over runs specified in the config file.

Usage

```
growR_run_loop(modvege_environments, output_dir = "", independent = TRUE)
```

Arguments

`modvege_environments`	A list of ModvegeEnvironment instances.
`output_dir`	string; name of directory to which output files are to be written. If `output_dir == ""` (default), no files are written.
`independent`	boolean; If `TRUE` (default) the simulation for each year starts with the same initial conditions, as specified in the parameters of the modvege_environments. If `FALSE`, initial conditions are taken as the final state values of the simulation of the previous year.

Details

By default, returns an empty list but writes output to the output files as specified in the *site_name* and *run_name* fields of the supplied ModvegeEnvironment instances. Change this behaviour through the `write_files` and `store_results` arguments.

Value

A list of the format `[[run]][[year]]` containing clones of the ModvegeSite instances that were run. Also write to files, if *output_dir* is nonempty.

Examples

```
env1 = create_example_environment(site = "posieux")
env2 = create_example_environment(site = "sorens")

growR_run_loop(c(env1, env2), output_dir = "")
```

---

load_measured_data *Load experimental data*

Description

Load all datasets stored in the supplied files.

Upon loading, the cumulative biomass growth *cBM* is automatically calculated from the given daily biomass growth *dBM* values.

Usage

```
load_measured_data(filenames, sep = ",")

load_data_for_sites(sites)

load_matching_data(filenames)
```

Arguments

`filenames`	Vector of strings representing simulation output filenames for which matching data files are searched and loaded.
`sep`	String Field separator used in the datafiles.
`sites`	Vector of site names for which data to load.

Details

`load_matching_data()` internally uses `get_site_name()` and makes the same assumptions about the output filename formats. It further assumes measured data to be located in "data/" and adhere to the filename format `x.csv` with `x` being the site name.

Value

measured_data list of data.frame each corresponding to one of the sites detected in *filenames*. Each data.frame contains the keys

- `dBM`: average daily biomass growth since last observation in kg/ha.
- `cBM`: cumulative biomass growth up to this DOY in kg/ha.
- `year`: year of observation.
- `DOY`: day of year of observation.

Functions

- `load_data_for_sites()`: Data filenames are generated on the convention 'SITE.csv' and are searched for in the subdirectory 'getOption("growR.data_dir")', which defaults to 'data/'.
- `load_matching_data()`: Accepts a vector of output filenames as generated by Modvege-Site`$write_output()` out of which the site names are inferred.

Data file format

The input data files are expected to consist of four columns containing the following fileds, in order:

date Date of measurement in yyyy-mm-dd format.

year Year of measurement. Identical to yyyy in *date* field.

DOY Day of year of the measurement. Jan 1st corresponds to 1, Dec 31st corresponds to 365 (except in gap years).

dBM Observed **average daily biomass growth** since last cut in kg/ha.

The first row is expected to be a header row containing the exact field names as in the description above. Columns may be separated by an arbitrary character, specified by the *sep* argument. The example data uses a comma (",").

`logger` *Primitive logger for debugging.*

Description

Primitive logger for debugging.

Usage

```
logger(msg = "", level = DEBUG, stop_on_error = TRUE)
```

Arguments

`msg`	The message to print.
`level`	The message will only be printed if its *level* is lower or equal to `getOption("growR.verbosity")`.
`stop_on_error`	Can be set to FALSE in order to continue execution despite emitting a message of *level* ERROR.

Value

None Prints console output.

See Also

set_growR_verbosity()

Examples

```
logger("A standard message", level = 3)
logger("A debug message", level = 4)
logger("A deep debug message", level = 5)
```

---

make_yearDOY *Create unique DOY + year identifier*

Description

Return numbers of the form YYYYDDD where YYYY is the year and DDD the DOY.

Usage

```
make_yearDOY(years, DOYs)
```

Arguments

years	int vector.
DOYs	int vector of same length as *years*.

Value

int vector of same length as *years* containing numbers of the form YYYYDDD, where the first four digits represent a year and the last four represent a DOY.

`ManagementData` *Management Data Class*

Description

Management Data Class

Management Data Class

Details

Data structure that contains management data which can serve as input to a ModvegeSite simulation run.

Public fields

`management_file` string The file that was read.

`is_empty` boolean Used to determine if management data is present or not. In the latter case, ModvegeSite will simulate management decisions automatically.

`years` List of unique years for which data is available.

`cut_years` numeric Vector of length *N* where *N* is the total number of cuts read from the input file. Gives the year in which corresponding cut was made.

`cut_DOY` numeric Vector of length *N* giving the day of year (as an integer) on which a cut was made.

`intensity` string Management intensity for "autocut". One of c("high", "middle", "low").

Methods

Public methods:

- `ManagementData$new()`
- `ManagementData$read_management()`
- `ManagementData$ensure_file_integrity()`
- `ManagementData$get_management_for_year()`
- `ManagementData$clone()`

Method `new()`**:** Create a new ManagementData object.

Usage:

```
ManagementData$new(management_file = NULL, years = NULL)
```

Arguments:

`management_file` string Path to file containing the management data to be read.

`years` numeric Vector of years for which the management is to be extracted.

Method `read_management()`**:** Read management data from supplied *management_file*.

Usage:

```
ManagementData$read_management(management_file, years = NULL)
```

Arguments:

`management_file` Path to or name of file containing management data.

`years` Years for which the management is to be extracted. Default (NULL) is to read in all found years.

Returns: None The object's field are filled.

Method `ensure_file_integrity()`**:** Check that all required columns are present and that cut DOYs are only increasing in a given year.

Usage:

```
ManagementData$ensure_file_integrity(cut_data)
```

Arguments:

`cut_data` data.frame containing the cut data.

Method `get_management_for_year()`**:** Extract management data for given year

This simply filters out all data not matching *year* and returns a list with the relevant keys.

Usage:

```
ManagementData$get_management_for_year(year)
```

Arguments:

`year` integer Year for which to extract management data.

Returns: M A list containing the keys:

is_empty boolean Used to determine if management data is present or not. In the latter case, ModvegeSite will simulate management decisions automatically.

cut_years numeric Vector of length *N* where *N* is the total number of cuts for this *year*, as read from the input file. Gives the year in which corresponding cut was made.

cut_DOY numeric Vector of length *N* giving the day of year (as an integer) on which a cut was made.

intensity string Management intensity for "autocut". One of c("high", "middle", "low").

n_cuts integer Number of cuts occurring in given year.

The two vectors in `cut_DOY` and `cut_years` differ from this object's respective fields in that only data for selected year is present.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

ManagementData`$read_management()`

management_parameters *Management practices for Swiss grasslands*

Description

Expected yields, uncertainties and average number of cuts as function of altitude and management intensity.

Usage

```
management_parameters
```

Format

A data.frame with 15 rows and 5 variables:

intensity Management intensity

altitude Altitude in m.a.s.l.

n_cuts Average number of cuts

yield Expected gross dry matter yield in t / ha

sigma_yield Uncertainty on yield in t / ha

Details

Data after table 1b in

Olivier Huguenin et al.

References

Huguenin-Elie I, Mosimann E, Schlegel P, Lüscher A, Kessler W, Jeangros B (2017). "Grundlagen für die Düngung landwirtschaftlicher Kulturen in der Schweiz (GRUD), Kapitel 9: Düngung von Grasland." *Agrarforschung Schweiz*. https://www.agrarforschungschweiz.ch/2017/06/9-duengung-von-grasland-grud-2017/.

metric_map *List of Performance Metrics*

Description

This list provides some common metrics of model performance along with their "best value".

Usage

```
metric_map
```

Format

A list where each item is a sublist containing the keys *func* and *target*.

func The function used to calculate given metric.

target The value that would be reached in the case of optimal performance.

limits Reasonable limits to be used when plotting.

`ModvegeEnvironment` *growR environment data*

Description

Data structure that contains inputs (parameters pertaining to a site, to the vegetation, to the weather and to the management) to growR simulations.

Details

This class contains site parameters, weather and management data for one simulation run of growR on a given site over several years. Methods are provided to allow access to relevant data for a given year.

All inputs are read in from data files through the respective data classes WeatherData, ManagementData and ModvegeParameters. These parameters can be simultaneously specified through a config file using `read_config()`.

Public fields

`site_name` Name of site to be simulated.

`run_name` Name of simulation run. Allows distinguishing between different simulations at the same site. Defaults to "-" for *no name*.

`run_name_in_filename` How the run name will be represented in an output file. If `run_name` is the default "-", indicating no name, this will be an empty string. Otherwise, it will be the `run_name` prepended by and underscore _.

`years` Years for which environment data (weather & management) is present.

`param_file` Name of supplied parameter file.

`weather_file` Name of supplied weather file.

`management_file` Name of supplied management file.

`parameters` A ModvegeParameters object.

`weather` A WeatherData object.

`management` A ManagementData object.

`input_dir` Directory in which parameter, weather and management files are searched for. Defaults to ‘getOption("growR.input_dir").

Methods

Public methods:

- ModvegeEnvironment$new()
- ModvegeEnvironment$set_run_name()
- ModvegeEnvironment$load_inputs()
- ModvegeEnvironment$make_filename_for_run()
- ModvegeEnvironment$get_environment_for_year()
- ModvegeEnvironment$clone()

Method `new()`**:** Instantiate a new ModvegeEnvironment

Usage:

```
ModvegeEnvironment$new(
  site_name,
  run_name = "-",
  years = NULL,
  param_file = "-",
  weather_file = "-",
  management_file = "-",
  input_dir = NULL
)
```

Arguments:

`site_name` string Name of the simulated site.

`run_name` string Name of the simulation run. Used to differentiate between different simulation conditions at the same site. Defaults to "-", which indicates no specific run name.

`years` numeric Vector of integer years to be simulated.

`param_file` string Name of file that contains site and vegetation parameters. If default value `"-"` is provided, it is assumed to be "SITENAME_parameters.csv".

`weather_file` string Analogous to *param_file*.

`management_file` string Analogous to *param_file*.

`input_dir` string Path to directory containing input files. Defaults to `getOption("growR.input_dir")`.

Method `set_run_name()`**:** Set run name and update *run_name_in_filename*.

Usage:

```
ModvegeEnvironment$set_run_name(run_name)
```

Arguments:

`run_name` Str. New value of `self$run_name`.

Method `load_inputs()`**:** Load simulation inputs.

Stores parameters, management and weather data from files specified in self$parameter_file, self$weather_file and self$management_file, respectively.

Usage:

```
ModvegeEnvironment$load_inputs()
```

Method `make_filename_for_run()`**:** Ensure a readable filename for given *run_name*.

Usage:

```
ModvegeEnvironment$make_filename_for_run(run_name)
```

Arguments:

`run_name` Name of run to be converted into a filename.

Returns: A version of *run_name* that can be used in a filename.

Method `get_environment_for_year()`**:** Get weather and environment for given year

Convenience function to retrieve environmental and management inputs for given *year* from multi-year data containers `self$weather` and `self$management`.

Usage:

```
ModvegeEnvironment$get_environment_for_year(year)
```

Arguments:

`year` int; year for which to extract data.

Returns: `list(W, M)` where `W` is the WeatherData and `M` the ManagementData object for given year.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

`read_config()`

WeatherData`$get_weather_for_year()`, ManagementData`$get_management_for_year()`

`ModvegeParameters` *Parameter Data Object*

Description

Data structure that contains site and vegetation parameters necessary for the configuration of an growR simulation run.

Parameter description

The following is a list and description of model parameters, including the vegetation parameters, which are defined through the functional group composition.

Site and model parameters:

- `LON` geographic longitude of site in degree.
- `LAT` geographic latitude of site in degree.

- ELV geographic elevation of site in m.a.s.l.
- WHC water-holding capacity of site in mm.
- NI site nutritional index (dimensionless).
- RUEmax maximum radiuation use efficiency in g DM per MJ.
- w_FGA relative weight of functional group A.
- w_FGB relative weight of functional group B.
- w_FGC relative weight of functional group C.
- w_FGD relative weight of functional group D.
- sigmaGV rate of GV respirative biomass loss (dimensionless).
- sigmaGR rate of GR respirative biomass loss (dimensionless).
- T0 photosynthesis activation temperature (degree C).
- T1 photosynthesis plateau temperature (degree C).
- T2 photosynthesis max temperature (degree C).
- KGV basic senescence rate GV (dimensionless).
- KGR basic senescence rate GR (dimensionless).
- KlGV basic abscission rate GV (dimensionless).
- KlGR basic abscission rate GR (dimensionless).
- maxOMDDV organic matter digestibility in gram per gram DV.
- minOMDDR organic matter digestibility in gram per gram DR.
- CO2_growth_factor strength of effect of CO2 concentration on growth. See parameter *b* in fCO2_growth_mod().
- crop_coefficient multiplicative factor K~c~ by which reference evapotranspiration ET~0~ has to be multiplied to get the crop evapotranspiration ET~c~: ET~c~ = K~c~ ET~0~
- senescence_cap fraction c~s~ of *GRO* to which *SEN* is limited: SEN~i~^max^ = c~s~ GRO~i~ for *i* in *GV*, *GR*. Makes it less likely for grass population to die out. Can be set to large values in order to effectively disable senescence capping.
- stubble_height float. Minimum height the grass can assume. The biomass will not fall below that height. This effectively presents a simple model of *plant reserves*.
- SGS_method string. Choice of method to determine the start of the growing season. Can be either "MTD" for the multicriterial thermal definition (see start_of_growing_season_mtd()) or "simple" for a commonly used approach as described in start_of_growing_season()).

Initial conditions:

- AgeGV Age of green vegetative matter in degree Celsius days.
- AgeGR Age of green reproductive matter in degree Celsius days.
- AgeDV Age of dead vegetative matter in degree Celsius days.
- AgeDR Age of dead reproductive matter in degree Celsius days.
- BMGV biomass of GV (kg DM per ha).
- BMGR biomass of GR (kg DM per ha).
- BMDV biomass of DV (kg DM per ha).
- BMDR biomass of DR (kg DM per ha).
- BMDR biomass of DR (kg DM per ha).
- SENG senescence of GV (kg DM per ha).

- SENG senescence of GR (kg DM per ha).
- ABSG abscission of DV (kg DM per ha).
- ABSG abscission of DR (kg DM per ha).
- ST thermal time (degree days).
- cBM cumulative total biomass (kg per ha).

Vegetation parameters:

- SLA Specific Leaf Area in m^2^ per g.
- pcLAM Percentage of laminae (number between 0 and 1).
- ST1 Temperature sum in degree Celsiues days after which the seasonality function SEA starts to decrease from its maximum plateau. See also SEA().
- ST2 Temperature sum in degree Celsiues days after which the seasonality function SEA has decreased back to its minimum value. See also SEA().
- maxSEA Maximum value of the seasonality function SEA()
- minSEA Minimum value of the seasonality function SEA(). Usually, minSEA = 1 - (maxSEA - 1).
- maxOMDGV Maximum organic matter digestability for green vegetative matter in arbitrary units.
- minOMDGV Minimum organic matter digestability for green vegetative matter in arbitrary units.
- maxOMDGR Maximum organic matter digestability for green reproductive matter in arbitrary units.
- minOMDGR Minimum organic matter digestability for green reproductive matter in arbitrary units.
- BDGV Bulk density of green vegetative dry matter in g per m^3^.
- BDDV Bulk density of dead vegetative dry matter in g per m^3^.
- BDGR Bulk density of green reproductive dry matter in g per m^3^.
- BDDR Bulk density of dead reproductive dry matter in g per m^3^.
- fg_parameter_names Vector of strings of the variable names of all vegetation parameters governed by functional group composition.

Public fields

required_parameter_names Names of parameters that do not have a default value and are therefore strictly required.

parameter_names Names of all required and optional parameters and state variables.

n_parameters Number of total parameters.

functional_group The FunctionalGroup instance holding the vegetation parameters.

fg_parameter_names Names of vegetation parameters defined by the functional group composition.

initial_condition_names Names of initial conditions.

param_file Name of the parameter file from which initial parameter values were read.

Methods

Public methods:

- ModvegeParameters$new()
- ModvegeParameters$read_parameters()
- ModvegeParameters$set_parameters()
- ModvegeParameters$update_functional_group()
- ModvegeParameters$check_parameters()
- ModvegeParameters$clone()

Method `new()`**:** Constructor

Usage:

```
ModvegeParameters$new(param_file = NULL)
```

Arguments:

`param_file` Name of file containing the site and vegetation parameters.

Method `read_parameters()`**:** Read parameters from parameter file

Reads in parameters from the supplied *param_file* and stores them in internal fields.

This function carries out some basic sanity checks of the supplied *param_file* and reports on unidentified and missing parameter names.

Usage:

```
ModvegeParameters$read_parameters(param_file)
```

Arguments:

`param_file` Path or name of file to read parameters from.

Returns: P List with field names as in the class variable `parameter_names`.

Method `set_parameters()`**:** Savely update the given parameters

This is the preferred method for changing the internal parameter values, because special care is taken to account for potential changes to functional group weights.

Usage:

```
ModvegeParameters$set_parameters(params)
```

Arguments:

`params` List of name-value pairs of the parameters to update.

Method `update_functional_group()`**:** Update functional group parameters

Should be run whenever the functional group composition is changed in order to reflect the changes in the parameter list `self$P`.

Usage:

```
ModvegeParameters$update_functional_group()
```

Method `check_parameters()`**:** Parameter Sanity Check Ensure that the supplied *params* are valid ModVege parameters and, if requested, check that all required parameters are present. Issues a warning for any invalid parameters and throws an error if completeness is not satisfied (only when `check_for_completeness = TRUE`).

Usage:

```
ModvegeParameters$check_parameters(param_names, check_for_completeness = TRUE)
```

Arguments:

`param_names` A list of parameter names to be checked.

`check_for_completeness` Boolean Toggle whether only the validity of supplied *param_names* is checked or whether we want to check that all required parameters to be present (default). In the latter case, if any required parameter is missing, an error is thrown.

Returns: not_known The list of unrecognized parameter names.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Note

Programmatically speaking, all parameters described under *Parameter description* are also fields of this R6Class.

`ModvegeSite` *ModvegeSite*

Description

Implements the ModVege grass growth model based off of Jouven et al. (2006).

This class contains model and site parameters and state variables as attributes and has methods for running ModVege with weather and management input.

Use the `run()` method to carry out a simulation for a given year. The results are stored in the state variables in this instance and can be written to file using `write_output()`.

Model variables

See Jouven et al. (2006) for a thorough description of all model variables.

State Variables:

- `BM` Standing biomass in kg DM per ha.
- `BMG` Standing green biomass (kg DM / ha).
- `cBM` Cumulativeley grown biomass (kg DM / ha).
- `dBM` Daily grown biomass (kg DM / ha).
- `hvBM` Cumulative harvested biomass (kg DM / ha).
- `OMD` Organic matter digestibility (kg / kg).
- `OMDG` OMD of green matter (kg / kg).

- ST Temperature sum in degree Celsius days.
- REP Reproductive function. Gives the fraction of growth that is assigned to reproductive growth. The remainder goes into vegetative growth. Dimensionless.
- PGRO Potential growth in kg DM / ha.
- GRO Effective growth in kg DM / ha.
- LAI Leaf area index, accounting for the proportion of light intercepted by the sward. Dimensionless.
- LAIGV LAI of green vegetative biomass. Dimensionless.
- AET Actual evapotranspiration in mm.
- WR Water reserves in mm.
- ENV Function representing environmental effects on growth. Acts as a multiplicative factor. Dimensionless.
- ENVfPAR Part of ENV due to strength of incident radiation. Dimensionless.
- ENVfT Part of ENV due to temperature. Dimensionless.
- ENVfW Part of ENV due to water limitation. Dimensionless.

Initial conditions:

- AgeGV Age of green vegetative matter in degree Celsius days.
- AgeGR Age of green reproductive matter in degree Celsius days.
- AgeDV Age of dead vegetative matter in degree Celsius days.
- AgeDR Age of dead reproductive matter in degree Celsius days.
- BMGV biomass of GV (kg DM per ha).
- BMGR biomass of GR (kg DM per ha).
- BMDV biomass of DV (kg DM per ha).
- BMDR biomass of DR (kg DM per ha).
- BMDR biomass of DR (kg DM per ha).
- SENG senescence of GV (kg DM per ha).
- SENG senescence of GR (kg DM per ha).
- ABSG abscission of DV (kg DM per ha).
- ABSG abscission of DR (kg DM per ha).
- ST thermal time (degree days).
- cBM cumulative total biomass (kg per ha).

Public fields

`time_step` Used time step in the model in days (untested).

`state_variable_names` Vector containing the names of the model's state variables.

`n_state_variables` Number of state variables.

`version` Version number of the growR package. Is written into output files.

`site_name` Name of the site to be simulated.

`run_name` Name of the simulation run. Used to distinguish between different runs at the same site.

`year` Year to be simulated.

`days_per_year` Number of days in this year.

`j_start_of_growing_season` Index (DOY) of the day the growing season was determined to begin.

`cut_height` Height of remaining grass after cut in m.

`parameters` A ModvegeParameters object.

`determine_cut` Function used to decide whether a cut occurs on a given DOY. Is overloaded depending on whether management data is provided or not.

`cut_DOYs` List of DOYs on which a cut occurred.

`cut_during_growth_preriod` Boolean to indicate whether a cut occurred during the growth period, in which case reproductive growth is stopped.

`BM_after_cut` Amount of biomass that remains after a cut #' (determined through cut_height and biomass densities BDGV, BDDV, BDGR, BDDR).

`weather` A list created by a WeatherData object's `get_weather_for_year()` method.

`management` A list containing management data as returned by ModvegeEnvironment's `get_environment_for_year()` method. If its `is_empty` field is `TRUE`, the Autocut routine will be employed.

`Autocut` A subclass of Autocut. The algorithm used to determine cut events.

Methods

Public methods:

- `ModvegeSite$new()`
- `ModvegeSite$get_weather()`
- `ModvegeSite$get_management()`
- `ModvegeSite$set_SGS_method()`
- `ModvegeSite$determine_cut_from_input()`
- `ModvegeSite$run()`
- `ModvegeSite$write_output()`
- `ModvegeSite$set_parameters()`
- `ModvegeSite$plot()`
- `ModvegeSite$plot_bm()`
- `ModvegeSite$plot_limitations()`
- `ModvegeSite$plot_water()`
- `ModvegeSite$plot_growth()`
- `ModvegeSite$plot_var()`
- `ModvegeSite$clone()`

Method `new()`**:** Constructor

Usage:

```
ModvegeSite$new(parameters, site_name = "-", run_name = "-")
```

Arguments:

`parameters` A ModvegeParameters object.

`site_name` string Name of the simulated site.

`run_name` string Name of the simulation run. Used to differentiate between different simulation conditions at the same site. Defaults to "-", which indicates no specific run name.

Method `get_weather()`**:** Return weather data if it exists

Usage:

```
ModvegeSite$get_weather()
```

Returns: The WeatherData object, if it exists.

Method `get_management()`**:** Return management data if it exists

Usage:

```
ModvegeSite$get_management()
```

Returns: The ManagementData object, if it exists.

Method `set_SGS_method()`**:** Choose which method to be used for determination of SGS
Options for the determination of the start of growing season (SGS) are:

MTD Multicriterial thermal definition, `start_of_growing_season_mtd()`
simple Commonly used, simple SGS definition based on temperature sum, `start_of_growing_season()`

Usage:

```
ModvegeSite$set_SGS_method(method)
```

Arguments:

`method` str Name of the method to use. Options: "MTD", "simple".

Returns: none

Method `determine_cut_from_input()`**:** Read from the input whether a cut occurs on day *DOY*.

Usage:

```
ModvegeSite$determine_cut_from_input(DOY)
```

Arguments:

`DOY` Integer day of the year for which to check.

Returns: Boolean `TRUE` if a cut happens on day *DOY*.

Method `run()`**:** Carry out a ModVege simulation for one year.

Usage:

```
ModvegeSite$run(year, weather, management)
```

Arguments:

`year` Integer specifying the year to consider.
`weather` Weather list for given year as returned by WeatherData`$get_weather_for_year`.
`management` Management list for given year as provided by ModvegeEnvironment`$get_environment_for_year()`.

Returns: None Fills the state variables of this instance with the simulated values. Access them programmatically or write them to file using `write_output()`.

Method `write_output()`**:** Write values of ModVege results into given file.
A header with metadata is prepended to the actual data.

Usage:

```
ModvegeSite$write_output(filename, force = FALSE)
```

Arguments:

`filename` Path or name of filename to be created or overwritten.

`force` Boolean If `TRUE`, do not prompt user before writing.

Returns: None Writes simulation results to file *filename*.

Method `set_parameters()`**:** Savely update the values in `self$parameters`.

This is just a shorthand to the underlying `ModvegeParameters` object's `set_parameters()` function. Special care is taken to account for potential changes to functional group weights.

Usage:

```
ModvegeSite$set_parameters(params)
```

Arguments:

`params` List of name-value pairs of the parameters to update.

Returns: None Updates this object's parameter values.

Method `plot()`**:** Create an overview plot for 16 state variables.

Creates a simple base R plot showing the temporal evolution of 16 modeled state variables.

Can only be sensibly run *after* a simulation has been carried out, i.e. after this instance's `run()` method has been called.

Usage:

```
ModvegeSite$plot(...)
```

Arguments:

`...` Further arguments are discarded.

Returns: NULL Creates a plot of the result in the active device.

Method `plot_bm()`**:** Create an overview plot for biomass.

Creates a simple base R plot showing the BM with cutting events and, if applicable, target biomass, dBM, cBM and hvBM. Can only be sensibly run *after* a simulation has been carried out, i.e. after this instance's `run()` method has been called.

Usage:

```
ModvegeSite$plot_bm(smooth_interval = 28, ...)
```

Arguments:

`smooth_interval` Int. Number of days over which the variable `dBM` is smoothened.

`...` Further arguments are discarded.

Returns: NULL Creates a plot of the result in the active device.

Method `plot_limitations()`**:** Create an overview plot of limiting factors.

Creates a simple base R plot showing the different environmental limitation functions over time. Can only be sensibly run *after* a simulation has been carried out, i.e. after this instance's `run()` method has been called.

Usage:

```
ModvegeSite$plot_limitations(...)
```

Arguments:

`...` Further arguments are discarded.

Returns: NULL Creates a plot of the result in the active device.

Method `plot_water()`**:** Create an overview plot of the water balance.

Creates a simple base R plot showing different variables pertaining to the water balance, namely water reserves *WR*, actual evapotranspiration *AET*, leaf area index *LAI* and LAI of the green vegetative compartment *LAIGV*.

Can only be sensibly run *after* a simulation has been carried out, i.e. after this instance's `run()` method has been called.

Usage:

```
ModvegeSite$plot_water(...)
```

Arguments:

`...` Further arguments are discarded.

Returns: NULL Creates a plot of the result in the active device.

Method `plot_growth()`**:** Create an overview plot of growth dynamics.

Creates a simple base R plot showing different variables pertaining to the growth dynamics, namely potential growth *PGRO*, effective growth *GRO*, the reproductive function *REP* and the temperature sum *ST*.

Can only be sensibly run *after* a simulation has been carried out, i.e. after this instance's `run()` method has been called.

Usage:

```
ModvegeSite$plot_growth(...)
```

Arguments:

`...` Further arguments are discarded.

Returns: NULL Creates a plot of the result in the active device.

Method `plot_var()`**:** Plot the temporal evolution of a modeled state variable.

Usage:

```
ModvegeSite$plot_var(var, ...)
```

Arguments:

`var` String. Name of the state variable to plot.

`...` Further arguments are passed to the base `plot()` function.

Returns: None, but plots to the current device.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

References

Jouven M, Carrère P, Baumont R (2006). "Model Predicting Dynamics of Biomass, Structure and Digestibility of Herbage in Managed Permanent Pastures. 1. Model Description." *Grass and Forage Science*, **61**(2), 112–124. ISSN 1365-2494, doi:10.1111/j.13652494.2006.00515.x, https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1365-2494.2006.00515.x.

See Also

Autocut

`start_of_growing_season_mtd()`, `start_of_growing_season()`

`ModvegeParameters$set_parameters()`

`parameter_scan_example`

Example results of a parameter scan

Description

The function `run_parameter_scan()` can take a significant time to execute, as it typically requires a few dozen model evaluations or more. In order to still showcase what its output can look like, and to facilitate testing and giving examples in the documentation of tools that make use of the output of `run_parameter_scan()` (such as e.g. `analyze_parameter_scan()`), this example dataset is provided.

Usage

```
parameter_scan_example
```

Format

A list containing an entry for each supplied parameter set in *param_values*. Each entry is itself a list containing the following keys:

params The parameter set that was used to run growR for this entry.

data A list containing for each simulated year a ModvegeSite object which was run for the respective year and therefore carries the respective results.

Details

The input for the parameter scan that produced this output was:

-
```
w_FGB = seq(0.25, 1, 0.25),
w_FGC = seq(0, 0.25, 0.25),
w_FGD = c(0),
NI = seq(0.75, 1.0, 0.25)
```

```
- `eps = 2e-2`
)
```

See Also

run_parameter_scan()

---

parse_year_strings *Parse and generate lists of years.*

Description

Parse and generate lists of years.

Usage

```
parse_year_strings(year_strings)
```

Arguments

year_strings	A vector of strings that each either represents a single year or a sequence of year in the format 'start:stop'.

Value

run_years List of integer vectors, representing the years to simulate for each run.

PetersenAutocut *Petersen autocut algorithm*

Description

Simulation routine to realistically predict grass cutting events. This follows an implementation described in Petersen et al. (2021).

Details

The decision to cut is made based on two criteria. First, it is checked whether a *target biomass* is reached on given DOY. The defined target depends on the DOY and is given through :func:get_target_biomass. If said biomass is present, return TRUE.

Otherwise, it is checked whether a given amount of time has passed since the last cut. Depending on whether this is the first cut of the season or not, the relevant parameters are :int:last_DOY_for_initial_cut and :int:max_cut_period. If that amount of time has passed, return TRUE, otherwise return FALSE.

The target biomass for a given day is determined following the principles described in Petersen et al.

The exact regression for the target biomass is based on Fig. S2 in the supplementary material of Petersen et al.

A refinement to expected yield as function of altitude has been implemented according to Table 1a in Huguenin et al. (2017).

Super class

`growR::Autocut` -> `PetersenAutocut`

Public fields

`last_DOY_for_initial_cut` Start cutting after this DOY, even if yield target is not reached.

`max_cut_period` Maximum period to wait between subsequent cuts.

`dry_precipitation_limit` Maximum amount of allowed precipitation (mm) to consider a day.

`dry_days_before_cut` Number of days that shold be dry before a cut is made.

`dry_days_after_cut` Number of days that shold be dry after a cut is made.

`max_cut_delay` Number of days a farmer is willing to wait for dry conditions before a cut is made anyways.

`cut_delays` Vector to keep track of cut delay times.

`dry_window` Logical that indicates if DOY at index is considered dry enough to cut.

`target_biomass` Biomass amount that should to be reached by given DOY for a cut to be made.

`end_of_cutting_season` Determined DOY after which no more cuts are made.

Methods

Public methods:

- `PetersenAutocut$new()`
- `PetersenAutocut$get_annual_gross_yield()`
- `PetersenAutocut$get_target_biomass()`
- `PetersenAutocut$get_relative_cut_contribution()`
- `PetersenAutocut$get_end_of_cutting_season()`
- `PetersenAutocut$determine_cut()`
- `PetersenAutocut$clone()`

Method `new()`**:** Constructor

Usage:

```
PetersenAutocut$new(MVS)
```

Arguments:

`MVS` The ModvegeSite object for which cuts shall be determined.

Method `get_annual_gross_yield()`**:** Lookup table returning expected annual gross yields as function of elevation and management intensity.

Based on data from Table 1a in Lookup Table of expected yield as functions of height and management intensity after Olivier Huguenin et al. (2017).

Usage:

```
PetersenAutocut$get_annual_gross_yield(elevation, intensity = "high")
```

Arguments:

`elevation` The elevation of the considered site in meters above sea level.

`intensity` One of ("high", "middle", "low", "extensive"). Management intensity for considered site.

Returns: Annual gross yield in t / ha (metric tons per hectare). Note that 1 t/ha = 0.1 kg/m^2.

Method `get_target_biomass()`**:** Get target value of biomass on given *DOY*, which determines whether a cut is to occur.

The regression for the target biomass is based on Fig. S2 in the supplementary material of Petersen, Krischan, David Kraus, Pierluigi Calanca, Mikhail A. Semenov, Klaus Butterbach-Bahl, and Ralf Kiese. "Dynamic Simulation of Management Events for Assessing Impacts of Climate Change on Pre-Alpine Grassland Productivity." European Journal of Agronomy 128 (August 1, 2021): 126306. https://doi.org/10.1016/j.eja.2021.126306.

A refinement to expected yield as function of altitude has been implemented according to Table 1a in Huguenen-Elie et al. "Düngung von Grasland", Agrarforschung Schweiz, 8, (6), 2017, https://www.agrarforschungschweiz.ch/2017/06/9-duengung-von-grasland-grud-2017/

Usage:

```
PetersenAutocut$get_target_biomass(DOY, intensity = "high")
```

Arguments:

`DOY` Integer day of the year to consider.

`intensity` One of ("high", "middle", "low") specifying management intensity.

Returns: target Biomass (kg / ha) that should be reached on day *DOY* for this management *intensity*.

Method `get_relative_cut_contribution()`**:** Relative cut contribution

Get the fraction of the total annual harvested biomass that a cut at given *DOY* is expected to contribute.

The regression for the target biomass is based on Fig. S2 in the supplementary material of Petersen et al. (2021).

Usage:

```
PetersenAutocut$get_relative_cut_contribution(DOY)
```

Arguments:

`DOY` Integer representing the day of the year on which a cut occurs.

Returns: The fraction (between 0 and 1) of biomass harvested at the cut at given *DOY* divided by the total annual biomass.

Method `get_end_of_cutting_season()`**:** Last day of cutting season

Estimate the last day on which it still makes sense to cut. This is done by checking at which point the expected target biomass (see `self$get_relative_cut_contribution()`) goes below the minimally harvestable standing biomass.

Usage:

```
PetersenAutocut$get_end_of_cutting_season(
  min_biomass,
  elevation,
  intensity = "high"
)
```

Arguments:

`min_biomass` float A standing biomass below this value cannot even be harvested,

`elevation` float Altitude in m.a.s.l.

`intensity` string Management intensity. One of "high", "middle", "low"

Returns: float Last (fractional) day of the year on which a cut still makes sense.

Method `determine_cut()`**:** Decide based on simple criteria whether day of year *DOY* would be a good day to cut.

This follows an implementation described in Petersen, Krischan, David Kraus, Pierluigi Calanca, Mikhail A. Semenov, Klaus Butterbach-Bahl, and Ralf Kiese. "Dynamic Simulation of Management Events for Assessing Impacts of Climate Change on Pre-Alpine Grassland Productivity." European Journal of Agronomy 128 (August 1, 2021): 126306. https://doi.org/10.1016/j.eja.2021.126306.

The decision to cut is made based on two criteria. First, it is checked whether a *target biomass* is reached on given DOY. The defined target depends on the DOY and is given through :func:`get_target_biomass`. If said biomass is present, return `TRUE`.

Otherwise, it is checked whether a given amount of time has passed since the last cut. Depending on whether this is the first cut of the season or not, the relevant parameters are :int:`last_DOY_for_initial_cut` and :int:`max_cut_period`. If that amount of time has passed, return `TRUE`, otherwise return `FALSE`.

Usage:

```
PetersenAutocut$determine_cut(DOY)
```

Arguments:

`DOY` Integer day of the year for which to make a cut decision.

Returns: Boolean `TRUE` if a cut happens on day *DOY*.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

References

Huguenin-Elie I, Mosimann E, Schlegel P, Lüscher A, Kessler W, Jeangros B (2017). "Grundlagen für die Düngung landwirtschaftlicher Kulturen in der Schweiz (GRUD), Kapitel 9: Düngung von Grasland." *Agrarforschung Schweiz*. https://www.agrarforschungschweiz.ch/2017/06/9-duengung-von-grasland-grud-2017/. Petersen K, Kraus D, Calanca P, Semenov MA, Butterbach-Bahl K, Kiese R (2021). "Dynamic Simulation of Management Events for Assessing Impacts of Climate Change on Pre-Alpine Grassland Productivity." *European Journal of Agronomy*, **128**, 126306. ISSN 1161-0301, doi:10.1016/j.eja.2021.126306, https://www.sciencedirect.com/science/article/pii/S1161030121000782.

Huguenin-Elie I, Mosimann E, Schlegel P, Lüscher A, Kessler W, Jeangros B (2017). "Grundlagen für die Düngung landwirtschaftlicher Kulturen in der Schweiz (GRUD), Kapitel 9: Düngung von Grasland." *Agrarforschung Schweiz*. https://www.agrarforschungschweiz.ch/2017/06/9-duengung-von-grasland-grud-2017/.

Petersen K, Kraus D, Calanca P, Semenov MA, Butterbach-Bahl K, Kiese R (2021). "Dynamic Simulation of Management Events for Assessing Impacts of Climate Change on Pre-Alpine Grassland Productivity." *European Journal of Agronomy*, **128**, 126306. ISSN 1161-0301, doi:10.1016/j.eja.2021.126306, https://www.sciencedirect.com/science/article/pii/S1161030121000782.

See Also

PhenologicalAutocut

`get_relative_cut_contribution()`

`get_target_biomass()`

`PhenologicalAutocut` *Autocut based on phenology*

Description

An algorithm to determine grass cut dates if none are provided. This uses empirical data for Switzerland to determine the first and last cut dates of the season from meteorological data. The number of cuts is inferred from Huguenen et al. and these cut events are distributed equally between first and last cut dates.

Super class

`growR::Autocut` -> `PhenologicalAutocut`

Public fields

`cut_DOYs` vector containing the integer day-of-year's on which cuts occur.

Methods

Public methods:

- `PhenologicalAutocut$new()`
- `PhenologicalAutocut$determine_cut()`
- `PhenologicalAutocut$clone()`

Method `new()`**:** Constructor

Valid cut dates are calculated upon initialization.

Usage:

```
PhenologicalAutocut$new(MVS)
```

Arguments:

`MVS` The ModvegeSite object for which cuts shall be determined.

Method `determine_cut()`**:** Does a cut occur on *DOY*?

Check if *DOY* is in `self$cut_DOYs`. If so, return `TRUE`. Return `FALSE` otherwise.

Usage:

```
PhenologicalAutocut$determine_cut(DOY)
```

Arguments:

`DOY` Integer day of the year (1-366).

Returns: Boolean

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

References

Huguenin-Elie I, Mosimann E, Schlegel P, Lüscher A, Kessler W, Jeangros B (2017). "Grundlagen für die Düngung landwirtschaftlicher Kulturen in der Schweiz (GRUD), Kapitel 9: Düngung von Grasland." *Agrarforschung Schweiz*. https://www.agrarforschungschweiz.ch/2017/06/9-duengung-von-grasland-grud-2017/.

See Also

management_parameters, PetersenAutocut

Autocut

`plot.ModvegeSite` *Plot ModVege simulation result overview*

Description

This wraps the `ModvegeSite` instance's `plot()` method.

Usage

```
## S3 method for class 'ModvegeSite'
plot(x, ...)
```

Arguments

`x`	A ModvegeSite instance.
`...`	Arguments are passed on to ModvegeSite`$plot()`.

Value

NULL, but plots to the active device.

See Also

The different `[modvegeSite]$plot_XXX()` methods.

plot_parameter_scan *Plot Parameter Scan Results*

Description

Visualize the results of a parameter scan and allow interactive inspection of model performance in parameter space.

Usage

```
plot_parameter_scan(analyzed, variable = "dBM", interactive = TRUE)
```

Arguments

analyzed	List; Output of analyze_parameter_scan().
variable	Str; Name of variable in *analyzed* to visualize. Can be changed later with PscanPlotter$set_variable(). Allowed values are the keys in *analyzed* except for params and metrics.
interactive	boolean; Toggle between just creating a static plot (interactive = FALSE) or entering a small, interactive analysis setting (interactive = TRUE, default).

Details

Under the hood this function just creates a PscanPlotter object and calls its analyze method.

Value

A PscanPlotter object.

See Also

analyze_parameter_scan(), PscanPlotter$analyze()

Examples

```
# There needs to be data available with which the modle is to be compared.
# For this example, use data provided by the package.
path = system.file("extdata", package = "growR")
datafile = file.path(path, "posieux1.csv")

# Analyze example output of `run_parameter_scan()`.
results = analyze_parameter_scan(parameter_scan_example, datafile = datafile)
# The following plots the results.
psp = plot_parameter_scan(results, interactive = FALSE)

# The interactive session can still be entered later from the returned
# PscanPlotter object
psp$analyze()
```

posieux_weather *Example Weather Data*

Description

Datasets containing the weather input parameters as used by growR. The same data is made available as plain text files by the package and automatically found in the `input` directory created by `setup_directory()` if the `include_examples` option is set to `TRUE` (default).

Usage

```
posieux_weather
```

Format

A data.frame with 3652 rows and 10 variables:

year Year as an integer

DOY Day of year as an integer

Ta Average temperature of the day in degree Celsius

Tmin Minimum temperature of the day in degree Celsius

Tmax Maximum temperature of the day in degree Celsius

precip Daily precipitation in mm

rSSD Relative sunshine duration in percent

SRad Sun irradiance in J / s / m^2. This can be converted into photosynthetically active radiation (PAR) in MJ / m^2 as: PAR = SRad * 0.47 * 24 * 60 * 60 / 1e6

ET0 Evapotranspiration in mm.

snow Precipitation in the form of snow in mm

Details

For use in growR, a WeatherData object has to be created from a plain text file. Therefore, this dataset is only provided for convenient inspection. In order to run growR, use the plain text files provided by the package. Use `system.file("extdata", package = "growR")` to locate them.

The *snow* column is not actually used by growR but rather calculated through precipitation and temperatures in WeatherData`$read_weather()`.

Likewise, the *rSSD* column is deprecated, currently unused and only kept for backwards compatibility.

See Also

`setup_directory()`, WeatherData

PscanPlotter *Plot Parameter Scan Results*

Description

This class facilitates interactive visual analysis of parameter scan results.

Public fields

`analyzed` List, as output by `analyze_parameter_scan()`.

`params` Vector of names of scanned parameters.

`metrics` Vector of names of model performance metrics to use.

`n_params` Number of scanned parameters.

`n_metrics` Number of performance metrics to apply.

`res` data.frame holding parameter scan results. It contains `n_params` + `n_metrics` + 1 columns: one column for each scanned parameter, one for each employed metric and an additional column (name `n`) to give each parameter combination (i.e. each row) an identifying number.

`n_combinations` Number of rowns in `res`.

`sorted` List containing copies of `res` which are each sorted by a different performance metric. List keys are the values in `self$metrics`.

`selection` Vector of integers corresponding to the ID number of combinations (column `n` in `self$res`) that is to be highlighted.

Methods

Public methods:

- `PscanPlotter$new()`
- `PscanPlotter$set_variable()`
- `PscanPlotter$analyze()`
- `PscanPlotter$plot()`
- `PscanPlotter$print_info()`
- `PscanPlotter$clone()`

Method `new()`**:** Construct and set up a PscanPlotter instance.

Usage:

```
PscanPlotter$new(analyzed, variable = "dBM")
```

Arguments:

`analyzed` List; Output of `analyze_parameter_scan()`.

`variable` Str; Name of variable in *analyzed* to visualize. Can be changed later with `set_variable()`. Allowed values are the keys in *analyzed* except for `params` and `metrics`.

Method `set_variable()`**:** Choose which variable to visualize.

Usage:

```
PscanPlotter$set_variable(variable)
```

Arguments:

`variable` Chosen variable name. One of "dBM", "cBM", "cBM_end"

Method `analyze()`**:** Enter analysis loop.

This plots the analysis results and enters a simple command-line interface through which more insights can be gathered. Particularly, it allows highlighting specific parameter combinations, either by their index number or by selecting the best performers according to a given metric.

Usage:

```
PscanPlotter$analyze()
```

Method `plot()`**:** Plot parameter scan results.

For every combination of scanned parameter and metric, a subplot is generated in which the parameter values are plotted against performance score in that metric for every parameter combination.

The result of this is static. Use `PscanPlotter$analyze()` for an interactive version.

Usage:

```
PscanPlotter$plot()
```

Method `print_info()`**:** Print information on selected parameter combinations.

The parameter values and performance scores of all combinations referred to by the integers in *selection* are printed to console.

Usage:

```
PscanPlotter$print_info(selection)
```

Arguments:

`selection` Vector of integers representing IDs of parameter combinations (i.e. column `n` in `self$res`).

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

`plot_parameter_scan()`

`analyze_parameter_scan()`

`PscanPlotter$plot()`

`read_config` *Read simulation run configurations from file*

Description

The format of the configuration file is expected to contain 6 space-separated columns representing, in order:

site_name Name of simulated site. This is used, for example, when an output file is created.

run_name Name of this simulation run. Used to differentiate between different runs at the same site. Can be `-` to indicate no particular name, in which case nothing will be appended in the resulting output file.

year(s) Specification of years to be simulated. Either a single number or a sequence in R's `:` notation, i.e. `2013:2022` to indicate all years from 2013 to (including) 2022.

param_file Filename (not full path) of parameter file to use. The file is assumed to be located in *input_dir* (confer documentation for that parameter).

weather_file Filename (not full path) of weather file. See also *param_file*.

management_file Filename (not full path) of management file. See also *param_file*. Can be set to `high`, `middle`, `low` or `-` if no management data is to be used and the *autocut* routine shall be employed to simulate cutting events.

Rows starting with a # are skipped.

Usage

```
read_config(config_file, input_dir = NULL)
```

Arguments

`config_file`	Path to the configuration file to be read.
`input_dir`	Path to directory where input files are located. Defaults to `getOptions("growR.input_dir", default = file.path("input"))`.

Value

A list of ModvegeEnvironment instances corresponding to the configurations in the order they appear in *config_file*.

Examples

```
# First, we set up the expected directory structure in a temporary place
tmp = file.path(tempdir(), "test-read_config")
dir.create(tmp)

# We need `force = TRUE` here in order to make the example work in
# non-interactive settings.
setup_directory(root = tmp, include_examples = TRUE, force = TRUE)
```

```
# Now we can test `read_config`.
read_config(file.path(tmp, "example_config.txt"),
            input_dir = file.path(tmp, "input"))
```

`run_parameter_scan` *Parameter Scan*

Description

Run ModVege for a different sets of parameters.

Usage

```
run_parameter_scan(environment, param_values, force = FALSE, outfilename = "")
```

Arguments

`environment`	Either a ModvegeEnvironment instance with all the site, management and weather inputs expected by ModvegeSite`$run()` **or** a string representing the name of a config file to read in order to generate the ModvegeEnvironment with `read_config()`. Note that, in the latter case, only the first found configuration is used if there are more than one valid uncommented lines in the config file.
`param_values`	A named list where each key stands for a ModVege parameter, i.e. a member of ModvegeParameters`$parameter_names`. Each list entry then has to be a vector containing the allowed values for the respective parameter. All possible allowed combinations of these parameter values are then generated and fed into a ModVege run.
`force`	Boolean. By default (`force = FALSE`), the function first counts the number of parameter combinations that need to be run and asks the user, if it should proceed. This can be suppressed by letting `force = TRUE`.
`outfilename`	String. If nonempty, the results are stored as an `rds` file with filename *outfilename* using the `saveRDS()` function.

Value

results A list containing an entry for each supplied parameter set in *param_values*. Each entry is itself a list containing the following keys:

params The parameter set that was used to run ModVege for this entry.

data A list containing for each simulated year a ModvegeSite object which was run for the respective year and therefore carries the respective results.

Note

Special care has to be taken in the creation of the *param_values* argument. It's possible to choose values that do not allow for any valid combination. Confer `create_combinations()`.

See Also

ModvegeParameters, `saveRDS()`, `create_combinations()`

Examples

```
env = create_example_environment()
# We're creating a trivial list of parameters to explore here in order to
# prevent the example from requiring a long time to execute. See
# [create_combinations()] for more realistic uses of param_values.
param_values = list(w_FGA = c(0, 1), w_FGB = c(0, 1))
run_parameter_scan(env, param_values, force = TRUE)
```

SEA *Seasonal effect on growth*

Description

Function representing the strategy of plants adjusting their roots:shoots ratios during the season.

Usage

```
SEA(ST, minSEA = 0.65, maxSEA = 1.35, ST1 = 800, ST2 = 1450)
```

Arguments

ST	float Temperature sum in degree(C)-days.
minSEA	float < 1. Minimum value of SEA.
maxSEA	float > 1. Maximum value of SEA.
ST1	float Temperature sum after which SEA declines from the maximum plateau.
ST2	float Temperature sum after which SEA reaches and remains at its minimum.

Examples

```
SEA(800)
```

setup_directory *Initialize growR directory structure*

Description

Creates directories in which growR by default looks for or deposits certain files. Also, optionally populates these directories with example files, which are useful to familiarize oneself with the growR simulation framework.

Usage

```
setup_directory(root, include_examples = TRUE, force = FALSE)
```

Arguments

root	Path to directory in which to initialize.
include_examples	If TRUE (default), include example data and input parameters in the appropriate directories.
force	boolean If TRUE, the user will not be asked for permission before we write to the filesystem.

Examples

```
# Prepare a temporary directory to write to
tmp = file.path(tempdir(), "test-setup_directory")
dir.create(tmp)

# We need `force = TRUE` here in order to make the example work in
# non-interactive settings.
setup_directory(root = tmp, include_examples = FALSE, force = TRUE)

# The `input`, `output` and `data` directories are now present.
list.files(tmp)

# Warnings are issued if directories are already present. Example files
# are still copied and potentially overwritten.
setup_directory(root = tmp, include_examples = TRUE, force = TRUE)

# Example files are now present
list.files(tmp, recursive = TRUE)

# End of the example. The following code is for cleaning up.
unlink(tmp, recursive = TRUE)
```

set_growR_verbosity *Set verbosity of growR output.*

Description

Set verbosity of growR output.

Usage

```
set_growR_verbosity(level = 3)
```

Arguments

level	Integer representing one of the following levels: 1: ERROR, 2: WARNING, 3: INFO, 4: DEBUG, 5: TRACE Messages with a level higher than the specified *level* are suppressed. In other words, higher values of *level* lead to more output and vice versa.

Value

None Sets the option “"growR.verbosity"‘.

Examples

```
# At level 3, only one of the three following messages are printed.
set_growR_verbosity(3)
logger("Message on level 5.", level = 5)
logger("Message on level 4.", level = 4)
logger("Message on level 3.", level = 3)
# At level 5, all three are printed.
set_growR_verbosity(5)
logger("Message on level 5.", level = 5)
logger("Message on level 4.", level = 4)
logger("Message on level 3.", level = 3)
# Reset to default.
set_growR_verbosity()
```

start_of_growing_season
Determine start of growing season

Description

This implements a conventional method for the determination of the start of the growing season (SGS) based on daily average temperatures.

Usage

```
start_of_growing_season(temperatures)
```

Arguments

temperatures	vector Daily average temperatures in degree Celsius.

Details

A temperature sum is constructed using [weighted_temperature_sum()], i.e. by summing the average daily temperature for each day, but applying a weight factor of 0.5 for January and 0.75 for February.

The SGS is defined as the first day where the so constructed temperature sum crosses 200 degree days.

See Also

[start_of_growing_season_mtd()], [weighted_temperature_sum()]

Examples

```
ts = rep(2, 365)
start_of_growing_season(ts)
```

---

start_of_growing_season_mtd

Multicriterial Thermal Definition

Description

Find the start of the growing season based on daily average temperatures.

Usage

```
start_of_growing_season_mtd(temperatures, first_possible_DOY = 1)
```

Arguments

temperatures	vector Daily average temperatures in degree Celsius.
first_possible_DOY	int Only consider days of the year from this value onward.

Details

This function implements the *multicriterial thermal definition* (MTD) as described in chapter 2.3.1.3 of the dissertation of Andreas Schaumberger: Räumliche Modelle zur Vegetations- und Ertragsdynamik im Wirtschaftsgrünland, 2011, ISBN-13: 978-3-902559-67-8

Value

int DOY of the growing season start according to the MTD.

See Also

[start_of_growing_season()]

Examples

```
# Create fake temperatures
ts = c(0, 0, 0, 0, 0, 0, 0, 6, 0, 0, 6, 6, 6, 6, 3, 3, 3, 3, 3, 6, 6, 6,
6, 5, 6, 7, 8, 9, 10, 11, 12)
start_of_growing_season_mtd(ts)
```

WeatherData *Weather Data Object*

Description

Data structure containing weather data for a given site for several years.

Details

All fields representing weather variables are vectors of length 365 times N, where *N* is the number of years for which weather data is stored. In other words, every variable has one value for each of the 365 of each of the *N* years.

Weather inputs

The weather input file should be organized as space separated columns with a `year` column and at least the following parameters as headers (further columns are ignored):

- `DOY` day of year in given year
- `Ta` average temperature of given day (Celsius).
- `precip` precipitation in millimeter per day.
- `PAR` photosynthetically active radiation in MJ/m^2^. Can be calculated from average sunlight irradiance `SRad` in J/s/m^2^ as: `PAR = SRad * 0.47 * 24 * 60 * 60 / 1e6`
- `ET0` evapotranspiration in mm.

These parameters are stored in this object in the respective `PARAM_vec` fields.

Snow model

The precipitation and temperature inputs are used in order to estimate the snow cover for each day by use of a snow model. The employed model is as formulated by Kokkonen et al. 2006 and makes use of parameters from Rango and Martinec, 1995.

Public fields

`weather_file` Name of provided weather data file.

`years` numeric Integer representation of the contained years.

`vec_size` Length of the PARAM_vec vectors, which is equal to *number of contained years* times 365.

`year_vec` Vector of length *vec_size*, holding the year for the respective index.

`W` A list generated by `get_weather_for_year()` which contains weather data only for a given year. The keys in the list are:

- aCO2 (atmospheric CO2 concentration in ppm)
- year
- DOY
- Ta
- Ta_sm (smoothed daily average temperature)
- PAR
- PP
- PET
- liquidP
- melt
- snow
- ndays (number of days in this year)

Methods

Public methods:

- `WeatherData$new()`
- `WeatherData$read_weather()`
- `WeatherData$ensure_file_integrity()`
- `WeatherData$calculate_day_length()`
- `WeatherData$get_weather_for_year()`
- `WeatherData$clone()`

Method `new()`**:** Create a new WeatherData object.

Usage:

```
WeatherData$new(weather_file = NULL, years = NULL)
```

Arguments:

`weather_file` string Path to file containing the weather data to be read.

`years` numeric Vector of years for which the weather is to be extracted.

Method `read_weather()`**:** Read weather data from supplied *weather_file*.

Usage:

```
WeatherData$read_weather(weather_file, years = NULL)
```

Arguments:

`weather_file` Path to or name of file containing weather data.

`years` Years for which the weather is to be extracted. Default (NULL) is to read in all found years.

Method `ensure_file_integrity()`**:** Check if supplied input file is formatted correctly.
Check if required column names are present and fix NA entries.

Usage:

```
WeatherData$ensure_file_integrity(weather)
```

Arguments:

`weather` data.table of the read input file with `header = TRUE`.

Method `calculate_day_length()`**:** Calculate the expected length of day based on a site's geographical latitude.

Usage:

```
WeatherData$calculate_day_length(latitude)
```

Arguments:

`latitude` numeric; geographical latitude in degrees.

Method `get_weather_for_year()`**:** Extract state variables to the weather data for given year and return them as a list.

Usage:

```
WeatherData$get_weather_for_year(year)
```

Arguments:

`year` integer Year for which to extract weather data.

Returns: W List containing the keys aCO2, year, DOY, Ta, Ta_sm, PAR, PP, PET, liquidP, melt, snow, ndays.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

References

Rango A, Martinec J (1995). "Revisiting the Degree-Day Method for Snowmelt Computations." *JAWRA Journal of the American Water Resources Association*, **31**(4), 657–669. ISSN 1752-1688, doi:10.1111/j.17521688.1995.tb03392.x.

Kokkonen T, Koivusalo H, Jakeman A, Norton J (2006). "Construction of a Degree-Day Snow Model in the Light of the Ten Iterative Steps in Model Development." In *iEMSs Third Biennial Meeting: "Summit on Environmental Modelling and Software". International Environmental Modelling and Software Society, Burlington, USA, July 2006*.

See Also

WeatherData`$read_weather()`

weighted_temperature_sum *Create a weighted temperature sum*

Description

A temperature sum is constructed by summing the average daily temperature for each day, but applying a weight factor of 0.5 for January and 0.75 for February.

Usage

```
weighted_temperature_sum(temperatures, negative = FALSE)
```

Arguments

temperatures	vector Daily average temperatures in degree Celsius.
negative	boolean Whether to include negative temperature values in the summation. By default, negative values are set to 0, meaning that the temperature sum is monotonically increasing.

Value

Weighted temperature sum.

Examples

```
# Use fake temperatures
ts = rep(2, 365)
weighted_temperature_sum(ts)
```

willmott *Willmott Index*

Description

Willmott's index of model performance as described in Willmott (2012).

Usage

```
willmott(predicted, observed, ...)
```

Arguments

predicted	Vector containing the predictions *y*.
observed	Vector containing the observations *z*.
...	Scaling factor **c** in the denominator in the Willmott index. The originally proposed value of 2 should be fine.

Details

This index takes on values from -1 to 1, where values closer to 1 are generally indicating better model performance. Values close to -1 can either mean that the model predictions differ strongly from the observation, or that the observations show small variance (or both).

Value

willmott Value between -1 and 1

References

Willmott CJ, Robeson SM, Matsuura K (2012). "A Refined Index of Model Performance." *International Journal of Climatology*, **32**(13), 2088–2094. ISSN 1097-0088, doi:10.1002/joc.2419, https://rmets.onlinelibrary.wiley.com/doi/abs/10.1002/joc.2419.

See Also

get_bias()

Examples

```
predicted = c(21.5, 22.2, 19.1)
observed = c(20, 20, 20)
# The Willmott index "fails" in this case, as the variance in the
# observation is 0.
willmott(predicted, observed)

# Try with more realistic observations
observed = c(20.5, 19.5, 20.0)
willmott(predicted, observed)
```

---

yield_parameters *Parameters for expected yields in Switzerland*

Description

The dataset contains the parameters *a* and *b* used to model the expected gross dry matter yield (in t / ha) as a function of altitude (in m.a.s.l.) as yield = a + b * altitude.

Usage

```
yield_parameters
```

Format

A data.frame with 4 rows and 3 variables:

intensity Management intensity

a Offset *a* in t / ha

b Slope *b* in t / ha / m

Details

Lookup Table of expected yield as functions of height and management intensity after table 1a in Olivier Huguenin et al.

References

Huguenin-Elie I, Mosimann E, Schlegel P, Lüscher A, Kessler W, Jeangros B (2017). "Grundlagen für die Düngung landwirtschaftlicher Kulturen in der Schweiz (GRUD), Kapitel 9: Düngung von Grasland." *Agrarforschung Schweiz*. https://www.agrarforschungschweiz.ch/2017/06/9-duengung-von-grasland-grud-2017/.

Index