Package 'BioCro'

March 7, 2025

Version 3.2.0

Date 2025-02-25

Title Modular Crop Growth Simulations

Description A cross-platform representation of models as sets of equations that facilitates modularity in model building and allows users to harness modern techniques for numerical integration and data visualization. Documentation is provided by several vignettes included in this package; also see Lochocki et al. (2022) <doi:10.1093/insilicoplants/diac003>.

Depends R (>= 3.6.0)

Imports stats

Suggests testthat (>= 3.2.0), knitr, rmarkdown, bookdown, lattice, deSolve

VignetteBuilder knitr

SystemRequirements C++11, GNU make

License MIT + file LICENSE

LazyData true

ByteCompile TRUE

URL https://github.com/biocro/biocro, https://biocro.github.io

Config/testthat/edition 3

NeedsCompilation yes

Author Justin M. McGrath [cre, aut] (<https://orcid.org/0000-0002-7025-3906>), Edward B. Lochocki [aut] (<https://orcid.org/0000-0002-4912-9783>), Yufeng He [aut] (<https://orcid.org/0000-0001-9895-1880>), Scott W. Oswald [aut] (<https://orcid.org/0000-0002-1906-0340>), Scott Rohde [aut] (<https://orcid.org/0000-0001-9030-0936>), Deepak Jaiswal [aut] (<https://orcid.org/0000-0002-4077-3919>), Megan L. Matthews [aut] (<https://orcid.org/0000-0002-5513-9320>), Fernando E. Miguez [aut] (<https://orcid.org/0000-0002-4627-8329>), Stephen P. Long [aut] (<https://orcid.org/0000-0002-8501-7164>), Dan Wang [ctb], David LeBauer [ctb] (<https://orcid.org/0000-0001-7228-053X>),

BioCro authors [cph],
Boost Organization [cph] (Copyright holder of included Boost library)

Maintainer Justin M. McGrath <jmcgrath@illinois.edu>

Repository CRAN

Date/Publication 2025-03-07 11:20:17 UTC

Contents

`add_time_to_weather_data` *Add a time component to input*

Description

Ensure, if possible, that input data that varies over time has a "time" component. See the documentation for `time` for more information about this quantity.

It is rare for users to call this function directly because it is called internally by `run_biocro`.

Usage

```
add_time_to_weather_data(drivers)
```

Arguments

`drivers`	A list or dataframe representing known system parameters that vary over time, such as weather data.

Value

If `drivers` has `doy` and `hour` columns, then it is assumed to represent weather data, and will be modified as follows:

- A new `time` column will be computed from `doy` and `hour`.
- The original `doy` and `hour` columns will be removed.

In this case, it is expected that the `BioCro:format_time` direct module will be used to re-compute `doy` and `hour` from `time`.

If `drivers` does not have `doy` and `hour` columns, then `drivers` will be returned as-is.

Note

Preconditions:

- If `drivers` is a list, the values should be vectors of equal length.
- If `drivers` already contains a `time` component, then it shouldn't contain either a `doy` or an `hour` component unless it contains both of them and the values are mutually consistent.

Why is the 'BioCro:format_time' module necessary?

If values of `doy` and `hour` are supplied to `run_biocro` in the drivers, undesired results may happen during interpolation. For example, if two sequential rows have `(time = 3599, doy = 150, hour = 23)` and `(time = 3600, doy = 151, hour = 0)`, and the results are to be returned at half-hour time intervals, then linear interpolation between these rows would produce `(time = 3599.5, doy = 150.5, hour = 11.5)`. Typically it is expected that `doy` takes only integer values, so this may cause issues. Using the `BioCro:format_time` module to calculate `doy` and `hour` from `time` will ensure that the result includes `(time = 3599.5, doy = 150, hour = 23.5)` instead.

Examples

```
# Add a time column to the buit-in 2002 weather data
new_weather <- add_time_to_weather_data(weather[['2002']])

# Compare column names
colnames(weather[['2002']])
colnames(new_weather)
```

`annualDB` *Miscanthus dry biomass data.*

Description

The first column is the thermal time. The second, third, fourth, and fifth columns are miscanthus stem, leaf, root, and rhizome dry biomass in Mg ha^{-1} (root is missing). The sixth column is the leaf area index. The `annualDB.c` version is altered so that root biomass is not missing and LAI is smaller. The purpose of this last modification is for testing some functions.

Format

Data frame of dimensions 5 by 6.

Source

Clive Beale and Stephen Long. 1997. Seasonal dynamics of nutrient accumulation and partitioning in the perennial C4 grasses Miscanthus x giganteus and Spartina cynosuroides. *Biomass and Bioenergy* 12 (6): 419–428.

`catm_data` *Global annual mean atmopspheric CO2 levels*

Description

Multiple years of globally averaged annual mean atmospheric CO2 levels and their uncertainties.

This data is included in the BioCro package so users can reproduce calculations in Lochocki *et al.* (2022) [doi:10.1093/insilicoplants/diac003] and for exploratory purposes; it is likely that most BioCro studies will require different data sets, and no attempt is made here to be exhaustive.

Usage

```
catm_data
```

Format

Data frame with 3 columns and 44 rows:

- `year`: the year
- `Catm`: CO2 concentration (micromol / mol)
- `unc`: the uncertainty associated with the CO2 concentration (micromol / mol)

Source

Data were obtained from the National Oceanic and Atmospheric Administration's Global Monitoring Laboratory (https://gml.noaa.gov/ccgg/trends/data.html) on 2024-02-07.

The exact link used was https://gml.noaa.gov/webdata/ccgg/trends/co2/co2_annmean_gl.txt.

Alternatively, the data can be accessed from https://gml.noaa.gov/ccgg/trends/gl_data.html by clicking the link to `Globally averaged marine surface annual mean data (CSV)`.

Note: the globally averaged value for 2023 was not yet available, so the 2023 Mauna Loa value was used instead as a temporary fix. This value is likely to be slightly higher than the global value (by around 1 ppm).

These data are provided here as a convenience to BioCro users; please visit the NOAA GML webpage for guidelines regarding the use of this data if you are intending to include it in a publication.

`cmi_soybean_weather_data`

Champaign, IL weather data for Soybean-BioCro

Description

Champaign, IL weather data specified at hourly intervals in the CST time zone for the years 2002, 2004, 2005, and 2006. The data includes typical inputs required for BioCro simulations, with the addition of `day_length`, which is specifically required for soybean simulations. Although this quantity can be calculated by modules during the course of a simulation, it is included in this weather data to speed up the simulations. The time range is restricted to the SoyFACE growing season that was used for each year.

This weather data is included in the BioCro package so users can reproduce the calculations of Matthews *et al.* (2022) [doi:10.1093/insilicoplants/diab032] and for exploratory purposes; it is likely that most BioCro studies will require different data sets, and no attempt is made here to be exhaustive.

Usage

```
soybean_weather
```

Format

A list of 4 named elements, where each element is a data frame corresponding to one year of weather data and the name of each element is a year, for example `'2004'`. Each data frame has 2952 - 3384 observations (representing hourly time points) of 14 variables:

- `year`: the year
- `doy`: the day of year
- `hour`: the hour
- `time_zone_offset`: the time zone offset relative to UTC (hr)
- `precip`: preciptation rate (mm / hr)
- `rh`: the ambient relative humidity (dimensionless)
- `dw_solar`: downwelling global solar radiation (J / m^2 / s)
- `up_solar`: upwelling global solar radiation (J / m^2 / s)
- `netsolar`: net global solar radiation (downwelling - upwelling) (J / m^2 /s)
- `solar`: the incoming photosynthetically active photon flux density (PPFD) measured on a ground area basis including direct and diffuse sunlight light just outside the crop canopy (micromol / m^2 / s)
- `temp`: the ambient air temperature (degrees Celsius)
- `windspeed`: the wind speed in the ambient air just outside the canopy (m / s)
- `zen`: the solar zenith angle (degrees)
- `day_length`: the length of the daily photoperiod (hours)

Source

Weather data were obtained from the public SURFRAD and WARM databases and processed according to the method described in Matthews *et al.* (2022) [doi:10.1093/insilicoplants/diab032]. See that paper for a full description of the data processing.

In brief, the columns in the data frames were determined from SURFRAD and WARM variables as follows:

- `precip`: from the `precip` variable in the WARM data set
- `rh`: from the `rh` variable in the SURFRAD data set
- `dw_solar`: from the `dw_solar` variable in the SURFRAD data set
- `up_solar`: from the `uw_solar` variable in the SURFRAD data set
- `netsolar`: from the `netsolar` variable in the SURFRAD data set
- `solar`: from the `par` variable in the SURFRAD data set; when these values are not available, the `netsolar` and `up_solar` variables are used to make an estimate; when these values are also not available, the `dw_solar` variable is used to make an estimate
- `temp`: from the `temp` variable in the SURFRAD data set
- `windspeed`: from the `windspd` variable in the SURFRAD data set
- `zen`: from the `zen` variable in the SURFRAD data set
- `day_length`: calculated from `solar` using an oscillator-based circadian clock

The WARM data set includes daily values. Hourly values for precipitation are derived from daily totals by assuming a constant rate of precipitation throughout the day.

The SURFRAD data set includes values at 1 or 3 minute intervals. Hourly values are determined by averaging over hourly intervals, where the value at hour `h` is the average over that hour. Some values are missing; any missing entries are filled by interpolating between neighboring hours.

To create this data frame, hourly values for all columns except `day_length` are extracted from the WARM and SURFRAD data. Then, BioCro is used to run the circadian clock model that determines photoperiod length. (See this page for additional information about the clock model: `soybean_clock`.) The result from this calculation is then appended to the weather data frame as a new column.

The `time_zone_offset` is set to a constant value of -6 since this data is specified in the CST time zone (i.e., UTC-6). Since the value of this quantity does not change, it could in principle be considered a parameter rather than a driver; however, it is included with the weather data for convenience.

To reduce size the in the BioCro repository, the raw data values are rounded. This was done using the commands in a script that is included with the BioCro package. This script can be located by typing `system.file('BioCro', 'extdata', 'get_soybean_weather_data.R')`.

`cmi_weather_data` *Champaign, IL weather data*

Description

Champaign, IL weather data specified at hourly intervals in the CST time zone for the years 1995–2023. The data includes typical inputs required for BioCro imulations. Note: some values are missing near the start of 1995 since those time points are not available from SURFRAD.

This weather data is included in the BioCro package so users can reproduce the calculations of Lochocki *et al.* (2022) [doi:10.1093/insilicoplants/diac003] and for exploratory purposes; it is likely that most BioCro studies will require different data sets, and no attempt is made here to be exhaustive.

Usage

```
weather
```

Format

A list of 29 named elements, where each element is a data frame corresponding to one year of weather data and the name of each element is a year, for example `'2004'`. Each data frame has 8760 or 8784 observations (representing hourly time points) of 9 variables:

- `year`: the year
- `doy`: the day of year
- `hour`: the hour
- `time_zone_offset`: the time zone offset relative to UTC (hr)

- precip: preciptation rate (mm / hr)
- rh: the ambient relative humidity (dimensionless)
- solar: the incoming photosynthetically active photon flux density (PPFD) measured on a ground area basis including direct and diffuse sunlight light just outside the crop canopy (micromol / m^2 / s)
- temp: the ambient air temperature (degrees Celsius)
- windspeed: the wind speed in the ambient air just outside the canopy (m / s)

Source

Weather data were obtained from the public SURFRAD and WARM databases and processed according to the method described in Lochocki *et al.* (2022) [doi:10.1093/insilicoplants/diac003]. See version 1.2.0 of the `eloch216/oscillator-based-circadian-clock-analysis` GitHub repository for a full description of the data processing.

In brief, the columns in the data frames were determined from SURFRAD and WARM variables as follows:

- precip: from the precip variable in the WARM data set
- rh: from the rh variable in the SURFRAD data set
- solar: from the par variable in the SURFRAD data set; when these values are not available, the direct_n, diffuse, and zen variables are used to make an estimate
- temp: from the temp variable in the SURFRAD data set
- windspeed: from the windspd variable in the SURFRAD data set

The WARM data set includes daily values. Hourly values for precipitation are derived from daily totals by assuming a constant rate of precipitation throughout the day.

The SURFRAD data set includes values at 1 or 3 minute intervals. Hourly values are determined by averaging over hourly intervals, where the value at hour h is the average over the hour-long interval centered at h. Some values are missing; any missing entries are filled via an interpolation procedure based on the assumption that values at the same hour of sequential days should be similar.

The time_zone_offset is set to a constant value of -6 since this data is specified in the CST time zone (i.e., UTC-6). Since the value of this quantity does not change, it could in principle be considered a parameter rather than a driver; however, it is included with the weather data for convenience.

To reduce size the in the BioCro repository, the raw data values are rounded. This was done using the commands in a script that is included with the BioCro package. This script can be located by typing `system.file('BioCro', 'extdata', 'rounding_weather_values.R')`.

compare_model_output *Compare new and stored results for a BioCro model test case*

Description

BioCro models can be tested using test cases, which are sets of known outputs that correspond to particular inputs. The compare_model_output function facilitates manual comparisons between new and stored results.

Note that *model tests* are distinct from the *module tests* described in module_testing.

Usage

```
compare_model_output(mtc, columns_to_keep = NULL)
```

Arguments

mtc	A single module test case, which should be created using model_test_case.
columns_to_keep	A vector of column names that should be included in the return value. If columns_to_keep is NULL, all columns that are in both the new and stored result will be included.

Details

The compare_model_output function is a key part of the BioCro model testing system. See model_testing for more information.

This function will run the model to get a new result, and load the stored result associated with the test case. The two data frames will be combined using rbind, where a new column named version indicates whether each row is from the new or stored result.

It is intended that quantities from the resulting data frame will be plotted to visually look for changes in the model output.

Value

A data frame as described above.

See Also

- model_testing
- model_test_case
- run_model_test_cases
- compare_model_output

Examples

```
# Define a test case for the miscanthus model and save the model output to a
# temporary directory
miscanthus_test_case <- model_test_case(
    'miscanthus_x_giganteus',
    miscanthus_x_giganteus,
    get_growing_season_climate(weather$'2005'),
    TRUE,
    tempdir()
)

update_stored_model_results(miscanthus_test_case)

# Now we can use `compare_model_output` to compare the saved result to a new one
comparison_df <- compare_model_output(miscanthus_test_case)

# This will be a boring example because the new and stored results will be
# exactly the same
lattice::xyplot(
  Leaf + Stem + Root ~ time,
  group = version,
  data = comparison_df,
  type = 'l',
  auto = TRUE,
  grid = TRUE
)
```

`crop_model_definitions` *Crop model definitions*

Description

In BioCro, a crop model is defined by sets of direct modules, differential modules, initial values, and parameters, along with an ordinary differential equation (ODE) solver. To run a model, these values, along with a set of weather data, are passed to the `run_biocro` function. For convenience, several crop model definitions are included in the BioCro R package. A full list can be obtained by typing `??crop_models` into the R terminal.

Details

Each crop model definition is stored as a list with the following named elements:

- `direct_modules`: A list of direct module names; can be passed to `run_biocro` as its `direct_module_names` argument.
- `differential_modules`: A list of differential module names; can be passed to `run_biocro` as its `differential_module_names` argument.
- `ode_solver`: A list specifying details of a numerical ODE solver; can be passed to `run_biocro` as its `ode_solver` argument.

- `initial_values`: A list of named quantity values; can be passed to `run_biocro` as its `initial_values` argument.
- `parameters`: A list of named quantity values; can be passed to `run_biocro` as its `parameters` argument, and also can be passed to `evaluate_module` and `module_response_curve` when investigating the behavior of one of the crop's modules.

These model definitions are not sufficient for running a simulation because `run_biocro` also requires drivers; for these crop growth models, the drivers should be sets of weather data. The `soybean` model is intended to be used along with the specialized soybean weather data (see `cmi_soybean_weather_data`). The other crops should be used with the other weather data (see `cmi_weather_data`).

Some quantities in the crop model definitions, such as the values of photosynthetic parameters, would remain the same in any location; others, such as the latitude or longitude, would need to change when simulating crop growth in different locations. Care must be taken to understand each input quantity before attempting to run simulations in other places or for other cultivars.

Typically, the modules in a crop model definition are defined as lists with some named elements; the names facilitate on-the-fly module swapping via the `within` function. For example, to change the soybean canopy photosynthesis module to the `BioCro:ten_layer_rue_canopy` module, one could pass `within(soybean$direct_modules, {canopy_photosynthesis = "BioCro:ten_layer_rue_canopy"})` as the `direct_module_names` argument when calling `run_biocro` instead of `soybean$direct_modules`.

Because each crop model definition is stored as a list with named elements, it is possible to use the `with` function to save some typing when calling `run_biocro` or related functions such as `partial_run_biocro` or `validate_dynamical_system_inputs`. For an example, compare `Example 1` and `Example 2` below. Besides shortening the code, using `with` also makes it easy to modify a command to simulate the growth of a different crop; if the two models can use the same drivers, this switch can be accomplished with one small change (`Example 3`).

See Also

- `run_biocro`
- `modules`

Examples

```
# Example 1: Simulating Miscanthus growth using its model definition list
result1 <- run_biocro(
  miscanthus_x_giganteus$initial_values,
  miscanthus_x_giganteus$parameters,
  get_growing_season_climate(weather$'2002'),
  miscanthus_x_giganteus$direct_modules,
  miscanthus_x_giganteus$differential_modules,
  miscanthus_x_giganteus$ode_solver
)

# Example 2: Performing the same simulation as in Example 1, but making use of
# the `with` command to reduce repeated references to the model definition list
result2 <- with(miscanthus_x_giganteus, {run_biocro(
  initial_values,
  parameters,
  get_growing_season_climate(weather$'2002'),
```

```
  direct_modules,
  differential_modules,
  ode_solver
)})

# Example 3: Simulating willow growth using the same weather data as Examples 1
# and 2, which just requires one change relative to Example 2
result3 <- with(willow, {run_biocro(
  initial_values,
  parameters,
  get_growing_season_climate(weather$'2002'),
  direct_modules,
  differential_modules,
  ode_solver
)})
```

---

default_ode_solvers *Default ODE solver settings*

Description

A collection of reasonable settings to use with each ODE solver type. Users may need or wish to modify them for particular applications.

Usage

```
default_ode_solvers
```

Format

A list of 6 named elements, where each name is one of the possible ODE solver types. Each element is itself a list of 5 named elements that can be passed to `run_biocro` as its `ode_solver` input argument.

Details

A full list of solver types can be obtained with the `get_all_ode_solvers` function.

---

dynamical_system *Validating dynamical system inputs*

Description

Utility function for checking inputs to `run_biocro` without running it

Usage

```
validate_dynamical_system_inputs(
    initial_values = list(),
    parameters = list(),
    drivers,
    direct_module_names = list(),
    differential_module_names = list(),
    verbose = TRUE
)
```

Arguments

initial_values	Identical to the corresponding argument from run_biocro.
parameters	Identical to the corresponding argument from run_biocro.
drivers	Identical to the corresponding argument from run_biocro.
direct_module_names	Identical to the corresponding argument from run_biocro.
differential_module_names	Identical to the corresponding argument from run_biocro.
verbose	Identical to the corresponding argument from run_biocro.

Details

validate_dynamical_system_inputs accepts the same input arguments as run_biocro with the exception of ode_solver (which is not required to check the validity of a dynamical system).

validate_dynamical_system_inputs checks a set of parameters, drivers, modules, and initial values to see if they can properly define a dynamical system and can therefore be used as inputs to run_biocro. Although the run_biocro function performs the same validity checks, the validate_dynamical_system_inputs includes additional information, such as a list of parameters whose values are not used as inputs by any modules, since in principle these parameters could be removed for clarity.

When using one of the pre-defined crop growth models, it may be helpful to use the with command to pass arguments to validate_dynamical_system_inputs; see the documentation for crop_model_definitions for more information.

Value

A boolean indicating whether or not the inputs are valid.

See Also

run_biocro

Examples

```
# Example 1: missing a parameter and an initial value
validate_dynamical_system_inputs(
  within(soybean$initial_values, rm(Leaf)),         # remove the initial `Leaf` value
  within(soybean$parameters, rm(leaf_reflectance)), # remove `leaf_reflectance`
  soybean_weather$'2002',
  soybean$direct_modules,
  soybean$differential_modules
)

# Example 2: a valid set of input arguments
validate_dynamical_system_inputs(
  soybean$initial_values,
  soybean$parameters,
  soybean_weather$'2002',
  soybean$direct_modules,
  soybean$differential_modules
)
```

---

get_all *Get lists of modules, quantities, and solvers*

Description

`get_all_modules` returns the fully-qualified names (of the form `library_name:local_module_name`) for all modules available in a BioCro module library package.

`get_all_quantities` returns information about all quantities used as inputs or outputs by modules available in a BioCro module library package.

`get_all_ode_solvers` returns the names of all ordinary differential equation (ODE) solvers available in the BioCro framework.

Usage

```
get_all_modules(library_name)

get_all_quantities(library_name)

get_all_ode_solvers()
```

Arguments

`library_name`	The name of a BioCro module library

Details

These "get_all" functions return the modules, quantities, and ODE solvers available within the BioCro framework or a BioCro module library package.

Developer details: The `get_all_modules` and `get_all_quantities` expect a module library package to include unexported functions called `get_all_modules_internal` and `get_all_quantities_internal`, respectively. These functions should not have any input arguments, and their return values should follow the requirements described below for `get_all_modules` and `get_all_quantities`. Any module library package created by forking from the skeleton library will automatically include these functions without any modifications to the package's R code.

Value

`get_all_modules`
: A character vector of fully-qualified module names

`get_all_quantities`
: A data frame with three columns: `quantity_type` (input or output), `quantity_name`, and `module_name`. A quantity will appear multiple times if it is use as both an input and an output, or if it is used by multiple modules.

`get_all_ode_solvers`
: A character vector of ODE solver names

See Also

- `modules`
- `module_paste`
- `run_biocro`

Examples

```
# Example 1: Getting a sorted list of distinct quantities defined by modules in
# the `BioCro` module library. Doing this can be useful when writing a new
# module that is intended to work along with pre-existing modules.
all_quantities <- get_all_quantities('BioCro')
all_quantity_names <- all_quantities$quantity_name
distinct_quantities <- sort(unique(all_quantity_names))

# Example 2: Getting a list of all modules in the `BioCro` module library that
# have "ci" as an input or output, using `tolower()` to account for any possible
# variations in capitalization.
all_quantities <- get_all_quantities('BioCro')
ci_modules <- subset(all_quantities, tolower(quantity_name) == "ci")
```

`get_growing_season_climate` *Truncate weather data to one growing season*

Description

Attempt to restrict a year of weather data to a growing season; not intended to be a general-use function (see below for a detailed discussion of its shortcomings).

Usage

```
get_growing_season_climate(climate, threshold_temperature = 0)
```

Arguments

`climate`	A data frame representing one year of weather data, typically intended to be passed to `run_biocro` as its `drivers` argument. This data frame must have columns for the day of year (`doy`) and the air temperature in degrees C (`temp`).
`threshold_temperature`	The value of air temperature in degrees C to use when locating the beginning and end of the growing season.

Details

DISCLAIMER: This function is included here primarily to reproduce the output of older BioCro calculations, where it used to be hard-coded into every simulation. It has several severe limitations which are discussed below, and is not intended to be a general-use function for subsetting weather data.

To determine the growing season, this function locates its beginning and end based on the air temperature data. The start of the growing season is set by the last day in the first half of the year where the air temperature is below (or equal to) the threshold temperature, or day 90, whichever is later. The end of the growing season is set by the first day of the second half of the year where the air temperature is below (or equal to) the threshold temperature, or day 330, whichever is earlier.

This is not a sophisticated function and no attempt is made to ensure that the output is reasonable. For example, if the air temperature never exceeds the threshold value, a growing season beginning on day 183 (the last day of the first half of the year) and ending on day 184 (the first day of the second half of the year) will be returned. If the air temperature always exceeds the threshold value, the growing season will go from day 90 to day 330.

This function also assumes that the air temperature generally increases early in the year and generally decreases later in the year, and is only applicable for locations where this is the case. It is therefore unlikely to work properly in the Southern Hemisphere or the tropics.

In general, an appropriate threshold temperature would depend on the species that is being modeled. For a perennial grass, the growth season might be said to begin after the last freeze, requiring a threshold temperature of 0 degrees C. Of course, this is an oversimplification of a complicated biological process, and a plant has no way of knowing when it has experienced the last freezing day of the year.

On the other hand, annual crops like maize or soybean are not typically sown until conditions are warmer and might require a higher threshold. Again, this is an oversimplification of a complicated process. Farmers typically take trends in temperature, historical data, soil conditions, and weather predictions into account when deciding to sow, and they may also be constrained by external factors like the availability of machinery, seeds, or labor.

It should also be noted that as the threshold temperature increases, the likelihood of that air temperature occurring at night, even in the middle of summer, also increases. Consequently, if the threshold is set too high, an unrealistically short growing season may be predicted. For example, calling `get_growing_season_climate(weather$'2005', 15)` returns a two-day growing season (days 183–184) because the temperatures in the late night of day 183 and the early morning of day 184 both dip below 15 degrees C.

Thus, the logic encoded here is an oversimplification in several ways. It is likely not appropriate in many situations, and more tailored approaches would be required.

Value

A copy of the `climate` data frame truncated to the growing season.

Examples

```
# Truncate the 2002 Champaign, Illinois weather data to an estimated growing
# season
truncated_weather <- get_growing_season_climate(weather[['2002']])

# We can see which days were included
list(
  doy_start = min(truncated_weather$doy),
  doy_end = max(truncated_weather$doy)
)
```

`miscanthus_x_giganteus` *Miscanthus model definition*

Description

Initial values, parameters, direct modules, differential modules, and a differential equation solver that can be used to run *Miscanthus x giganteus* growth simulations in Champaign, Illinois and other locations.

To represent *Miscanthus* growth in Champaign, IL, these values must be paired with the Champaign weather data (`cmi_weather_data`). The parameters already include the `clay_loam` values from the `soil_parameters` dataset, which is the appropriate soil type for Champaign.

Some specifications, such as the values of photosynthetic parameters, would remain the same in any location; others, such as the latitude or longitude, would need to change when simulating crop growth in different locations. Care must be taken to understand each input quantity before attempting to run simulations in other places or for other cultivars.

Usage

```
miscanthus_x_giganteus
```

Format

A list of 5 named elements that are suitable for passing to `run_biocro`, as described in the help page for `crop_model_definitions`.

Source

This model was originally described in Miguez *et al.* (2009) [doi:10.1111/j.17571707.2009.01019.x] and Miguez *et al.* (2012) [doi:10.1111/j.17571707.2011.01150.x]. Since its original parameterization, the behavior of several of its core modules has changed as bugs have been identified and fixed, so this model likely needs to be reparameterized before it can be used for realistic simulations.

See Also

- `run_biocro`
- `modules`
- `crop_model_definitions`

---

model_testing *The BioCro model testing system*

Description

BioCro provides several functions for defining, modifying, and running model test cases. These functions together allow model developers to easily create regression tests that ensure the models continue to function correctly.

Note that *model tests* are distinct from the *module tests* described in `module_testing`.

Details

Together, `model_test_case`, `run_model_test_cases`, `update_stored_model_results`, and `compare_model_output` form a simple and convenient system for defining and running model test cases. Such tests form a critical component of BioCro's regression testing system, and test cases should be defined for all BioCro models in all BioCro-related repositories. These functions are not required in order to use the BioCro package, but they are critical to understand when creating or modifying models, or the modules they use.

A model test case consists of a model definition, a set of drivers, a short name, and a few additional settings that specify some of the testing behavior. To run a test, the model definition and drivers are passed to `run_biocro` to ensure the model is well-defined, and then the results are (optionally) compared against saved results to ensure the model behavior has not changed. Multiple test cases can be defined in a single list and passed to `run_model_test_cases`, which will run all of them.

In this system, stored data for a test case with name `'test_name'` must be stored in a CSV file called `'test_name_simulation.csv'`. The `update_stored_model_results` function can be used to generate a suitable file.

Typically, a BioCro-related repository will include a model testing file that defines test cases and runs them to check for issues. An example can be found in the `tests/testthat/test.CropModels.R` file. The associated stored test results can be found in the `tests/testthat/test_data` directory.

If any of the initial values, parameters, modules or weather data change, or if the behavior of any of these modules changes, the stored data for one or more model test cases will likely need to be updated. This can be done using the `update_stored_model_results` function.

Sometimes these changes are not expected to alter key outputs like the biomass values. In this case, it is helpful to visually compare the new and old biomass values. This can be done using the `compare_model_output` function before updating the results.

See Also

- `crop_model_definitions`
- `model_test_case`
- `update_stored_model_results`
- `compare_model_output`
- `run_model_test_cases`

`model_test_case` *Define BioCro model test cases*

Description

BioCro models can be tested using test cases, which are sets of known outputs that correspond to particular inputs. The `model_test_case` function defines such a test case.

Note that *model tests* are distinct from the *module tests* described in `module_testing`.

Usage

```
model_test_case(
  test_case_name,
  model_definition,
  drivers,
  check_outputs,
  directory = '.',
  quantities_to_ignore = character(),
  row_interval = 24,
  digits = 5,
  relative_tolerance = 1e-3
)
```

Arguments

test_case_name	A string describing the test case.
model_definition	A list defining a model, as described in the documentation for crop_model_definitions.
drivers	A set of drivers to be passed to run_biocro along with the model_definition.
check_outputs	A logical value indicating whether to compare the simulation output against a stored result.
directory	A relative or absolute path to a directory containing a stored simulation result. Only used when check_outputs is TRUE.
quantities_to_ignore	A character vector of any quantities that should not be compared against the stored results. Only used when check_outputs is TRUE.
row_interval	Determines which rows are saved and compared when using update_stored_model_results, compare_model_output, or run_model_test_cases. Only used when check_outputs is TRUE.
digits	Passed to signif to round values when storing saved results. Only used when check_outputs is TRUE.
relative_tolerance	A relative tolerance to be used when comparing new values against stored ones. This value will be passed to all.equal as its tolerance input argument. Only used when check_outputs is TRUE.

Details

The model_test_case function forms the basis for the BioCro model testing system. See model_testing for more information.

With the default settings:

- Every 24 rows of the simulation output will be stored and compared. When using drivers with an hourly time step, this corresponds to one row for each day.
- Values in the stored simulation results will be rounded to five significant digits. This reduces the size of the stored result file.
- The value of the relative tolerance was chosen to be the smallest value that enabled the tests to pass on all operating systems.

These default settings have proven useful for the BioCro miscanthus_x_giganteus, willow, and soybean models.

Value

A list that defines a model test case, which can be passed to update_stored_model_results, compare_model_output, or run_model_test_cases.

See Also

- model_testing
- crop_model_definitions
- update_stored_model_results
- compare_model_output
- run_model_test_cases

Examples

```
# Define a test case for the miscanthus model
miscanthus_test_case <- model_test_case(
    'miscanthus_x_giganteus',
    miscanthus_x_giganteus,
    get_growing_season_climate(weather$'2005'),
    TRUE,
    tempdir(),
    'soil_evaporation_rate'
)

# The result is a specially formatted list
str(miscanthus_test_case)
```

---

modules *BioCro module functions*

Description

BioCro modules are named sets of equations, and each module is available from a BioCro module library. Each module is identified by a fully-qualified name that includes the name of its library and its local name within that library. The functions here provide ways to access information about modules and to calculate their output values from sets of input values.

`module_info` returns essential information about a BioCro module.

`quantity_list_from_names` initializes a list of named numeric elements from a set of names.

`evaluate_module` runs a BioCro module using a list of input quantity values.

`module_response_curve` runs a BioCro module repeatedly with different input quantity values to produce a response curve.

Usage

```
module_info(module_name, verbose = TRUE)

quantity_list_from_names(quantity_names)

evaluate_module(module_name, input_quantities)

module_response_curve(module_name, fixed_quantities, varying_quantities)
```

Arguments

module_name	A string specifying one BioCro module, formatted like library_name:local_module_name, where library_name is the name of a library that contains a module with local name local_module_name; such fully-qualified module names can be formed manually or with module_paste.
verbose	A boolean indicating whether or not to print information to the R console.
input_quantities	A list of named numeric elements representing the input quantities required by the module; any extraneous quantities will be ignored by the module.
quantity_names	A vector of strings.
fixed_quantities	A list of named numeric elements representing input quantities required by the module whose values should be considered to be constant; any extraneous quantities will be ignored by the module.
varying_quantities	A data frame where each column represents an input quantity required by the module whose value varies across the response curve.

Details

By providing avenues for retrieving information about a module and evaluating a module's equations, the module_info and evaluate_module functions form the main interface to individual BioCro modules from within R. The quantity_list_from_names function is a convenience function for preparing suitable quantity lists to pass to evaluate_module.

The module_response_curve function provides a convenient way to calculate a module response curve. To do this, a user must specify a module to use, the values of any fixed input quantities (input_quantities), and a sequence of values for other quantities that vary across the response curve (varying_quantities). The returned data frame includes all the information that would be required to reproduce the curve: the full-qualified module name, all inputs (including ones with constant values), and the outputs. Note: if one quantity q is both an input and output of the module, its input value will be stored in the q column of the returned data frame and its output value will be stored in the q.1 column; this renaming is performed automatically by the make.unique function.

Value

module_info — An invisible list of several named elements containing essential information about the module:

- module_name: The module's (not-fully-qualified) name
- inputs: A character vector of the module's inputs
- outputs: A character vector of the module's outputs
- type: The module's type represented as a string (either 'differential' or 'direct')
- euler_requirement: Indicates whether the module requires a fixed-step Euler ODE solver when used in a BioCro simulation
- creation_error_message: Describes any errors that occurred while creating an instance of the module

`quantity_list_from_names`
: A list of named numeric elements, where the names are set by `quantity_names` and each value is set to 1.

`evaluate_module`
: A list of named numeric elements representing the values of the module's outputs as calculated from the `input_quantities` according to the module's equations.

`module_response_curve`
: A data frame where the first column is the fully-qualified name of the module that produced the response curve and the remaining columns are the module's input and output quantities. Each row corresponds to a row in the `varying_quantities`.

See Also

- `get_all_modules`
- `module_paste`
- `module_testing`
- `partial_evaluate_module`

Examples

```
# Example 1: printing information about the 'BioCro' module library's
# 'c3_assimilation' module to the R console
module_info('BioCro:c3_assimilation')

# Example 2: getting the inputs to the 'BioCro' module library's
# 'thermal_time_linear' module, generating a default input list, and using it to
# run the module
info <- module_info('BioCro:thermal_time_linear', verbose = FALSE)
inputs <- quantity_list_from_names(info$inputs) # All inputs will be set to 1
outputs <- evaluate_module('BioCro:thermal_time_linear', inputs)

# Example 3: calculating the temperature response of light saturated net
# assimilation at several values of relative humidity in the absence of water
# stress using the 'BioCro' module library's 'c3_assimilation' module and
# the default soybean parameters. Here, the leaf temperature and humidity values
# are independent of each other, so we use the `expand.grid` function to form a
# data frame of all possible combinations of their values. Then we set the
# ambient temperature equal to the leaf temperature.
rc <- module_response_curve(
  'BioCro:c3_assimilation',
  within(soybean$parameters, {Qabs = 2000; StomataWS = 1; gbw = 1.2}),
  within(
    expand.grid(
      Tleaf = seq(from = 0, to = 40, length.out = 201),
      rh = c(0.2, 0.5, 0.8)
    ),
    {temp = Tleaf}
  )
)
```

```
caption <- paste(
  "Response curves calculated with several RH\nvalues and Q =",
  unique(rc$Qp),
  "micromol / m^2 / s\nusing the",
  unique(rc$module_name),
  "module"
)

lattice::xyplot(
  Assim ~ Tleaf,
  group = rh,
  data = rc,
  auto = TRUE,
  type = 'l',
  main = caption
)
```

`module_case_files` *Define and modify BioCro module test case files*

Description

Test cases for testing modules can be stored in files. The functions here provide ways to create and update those files.

`initialize_csv` helps define test cases for module testing by initializing the `csv` file for one module based on either a set of default input values or user-supplied ones.

`add_csv_row` helps define test cases for module testing by adding one test case to a module's `csv` file based on the user-supplied inputs and description.

`update_csv_cases` helps define cases for module testing by updating the expected output values for each case stored in a module's csv file.

Note that *module tests* are distinct from the *model tests* described in `model_testing`.

Usage

```
initialize_csv(
  module_name,
  directory,
  nonstandard_inputs = list(),
  description = "automatically-generated test case",
  overwrite = FALSE
)

add_csv_row(module_name, directory, inputs, description)

update_csv_cases(module_name, directory)
```

Arguments

module_name	A string specifying one BioCro module, formatted like library_name:local_module_name, where library_name is the name of a library that contains a module with local name local_module_name; such fully-qualified module names can be formed manually or with module_paste.
directory	The directory where module test case files are stored, e.g. file.path('tests', 'module_test_cases').
inputs	A list of module inputs, i.e., a list of named numeric elements corresponding to the module's input quantities.
description	A string describing the test case, e.g. "temp above tbase". The description should be succinct and not contain any newline characters.
nonstandard_inputs	An optional list of input quantities whose values will override the default value of 1.0; see the inputs entry above.
overwrite	A logical value indicating whether an existing file should be overwritten.

Details

Module test case files form a critical component of BioCro's regression testing system. For more details, see the help page for module_testing.

The initialize_csv function will evaluate the module for a set of input quantities and store the results as a test case csv file. Typically, both of its optional arguments can be omitted. However, some modules produce errors when all inputs are set to 1.0. In this case, it would be necessary to supply some nonstandard inputs and (possibly) an alternate case description.

The add_csv_row function will evaluate the module for a set of input quantities, define a test case from the resulting outputs and the description, and add it to the module's corresponding csv file. If no csv file exists, one will be initialized with the new case.

The update_csv_cases function will evaluate the module for all input values specified in its csv case file and update the stored values of the corresponding outputs. Any output columns not present in the file will be added automatically and filled in with the correct values. Although the output columns are optional, the description column must exist in the csv file.

If a module test fails and update_csv_cases is used to update the test, care should be taken to ensure that the new outputs are sensible. This function should not be used to blindly ensure that tests pass, since a test failure may indicate a real problem with a module.

Note that update_csv_cases can be used to batch-initialize test cases. To do this, manually create a test case csv file with the proper name that only includes columns for the inputs and the description; now, calling update_csv_cases will automatically fill in the outputs for each case. With this method, care must be taken when manually specifying the values of the description column; the descriptions must be double quoted, and if they contain internal double quotes, those quotes must be doubled. Generally it is safest to simply avoid double quotes in the descriptions. (See qmethod in the help file for write.csv for more details about quoting.)

Value

A message indicating whether a file was created, overwritten, or not written.

See Also

- modules
- module_paste
- module_testing
- test_module_library
- test_module

Examples

```
# First, we will initialize a test case file for the 'BioCro' library's
# 'thermal_time_linear' module, which will be saved in a temporary directory as
# 'BioCro_thermal_time_linear.csv'. Then, we will add a new case to the file.
# Finally, we will update the file. Note that the call to `update_csv_cases`
# will not actually modify the file unless it is manually edited beforehand to
# change an input or output value.

td <- tempdir()

initialize_csv(
  'BioCro:thermal_time_linear',
  td,
  nonstandard_inputs = list(temp = -1),
  overwrite = TRUE
)

writeLines(readLines(file.path(td, 'BioCro_thermal_time_linear.csv')))

add_csv_row(
  'BioCro:thermal_time_linear',
  td,
  list(fractional_doy = 101, sowing_fractional_doy = 100, tbase = 20, temp = 44),
  'temp above tbase'
)

writeLines(readLines(file.path(td, 'BioCro_thermal_time_linear.csv')))

update_csv_cases('BioCro:thermal_time_linear', td)
```

---

module_creators *Create instances of modules*

Description

Creates pointers to module wrapper objects

Usage

```
module_creators(module_names)
```

Arguments

module_names	A vector of module names

Details

This function is used internally by several other BioCro functions, where its purpose is to create instances of module wrapper pointers using BioCro's module library and return pointers to those wrappers. In turn, module wrappers can be used to obtain information about a module's inputs, outputs, and other properties, and can also be used to create a module instance. The See Also section contains a list of functions that directly rely on module_creators.

Although the description of externalptr objects is sparse, they are briefly mentioned in the R documentation: externalptr-class.

This function should not be used directly, and each module library package must have its own version. For these reasons, this function is not exported to the package namespace and can only be accessed using the package name via the ::: operator.

Value

A vector of R externalptr objects that each point to a module_creator C++ object

See Also

- run_biocro
- module_info
- evaluate_module

---

module_paste *Prepend library name to module names*

Description

Prepends a library name to a set of module names to create a suitably-formatted set of fully-qualified module names that can be passed to run_biocro or other BioCro functions.

Usage

```
module_paste(lib_name, local_module_names)
```

Arguments

lib_name	A string specifying a module library name.
local_module_names	A vector or list of module name strings.

Details

module_paste is a convenience function for specifying multiple modules from the same library; it prepends the library name to each module name, preserving the names and class of local_module_names.

Note that a simple call to paste0(lib_name, ':', local_module_names) will produce a similar output with two important differences: (1) paste0 will not preserve names if local_module_names has any named elements and (2) paste0 will always return a character vector, even if local_module_names is a list.

Value

A vector or list of fully-qualified module name strings formatted like lib_name:local_module_name.

See Also

- modules
- run_biocro

Examples

```
# Example: Specifying several modules from the `BioCro` module library.
modules <- module_paste(
  'BioCro',
  list('total_biomass', canopy_photosynthesis = 'c3_canopy')
)

# Compare to the output from `paste0`
modules2 <- paste0(
  'BioCro',
  ':',
  list('total_biomass', canopy_photosynthesis = 'c3_canopy')
)

str(modules)
str(modules2)
```

---

module_testing *The BioCro module testing system*

Description

BioCro provides several functions for defining, modifying, and running module test cases. These functions together allow module developers to easily create regression tests that ensure the modules continue to function correctly.

Note that *module tests* are distinct from the *model tests* described in model_testing.

Details

Together, test_module_library, test_module, case, cases_from_csv, initialize_csv, add_csv_row, and update_csv_cases form a simple and convenient system for defining and running module test cases. Such tests form a critical component of BioCro's regression testing system, and test cases should be defined for all BioCro modules in all BioCro module libraries. These functions are not required in order to use the BioCro package, but they are critical to understand when creating or modifying modules.

A module test case consists of a set of module inputs, a set of module outputs, and a short description of the case. To run the test, the inputs are passed to the module, and then the calculated outputs are compared to the expected ones. If the outputs match, the test is passed; otherwise, it fails. This operation is handled by the test_module function.

For simple on-the-fly testing, it is possible to define a test case using the case function and run it using test_module. However, a more robust method is available to facilitate regression testing, where module test cases are stored in suitably-formatted csv files, allowing multiple test cases to be defined for each module and easily checked afterwards. If test case files for each module in a module library are stored in a single directory, all the test cases can be checked with one call to test_module_library.

In this system, test cases for a module with fully-qualified name module_name must be stored in module_name.csv, where the colon in the module name has been replaced by an underscore; for example, the module named BioCro:total_biomass would be associated with BioCro_total_biomass.csv. The first row of a test case file must be the quantity types (input or output), the second row must be the quantity names, and the remaining rows must each specify input quantity values along with the expected output values they should produce. There must also be a description column (with description in the first row) containing short descriptions of the test cases. These formatting requirements will automatically be satisfied for any test case file produced by initialize_csv or modified by add_csv_row or update_csv_cases. Such files can be read from R using cases_from_csv, and the resulting case objects can be passed to test_module.

Although it is possible, directly editing the case files is not recommended since initialize_csv, add_csv_row, and update_csv_cases are easier to use. There are several exceptions to this suggestion: (1) when a case must be deleted, (2) when a module input must be added or removed, and (3) during the initialization of a test file, where a user may wish to batch-initialize using update_csv_cases (see its documentation for an explanation of batch-initialization).

Case files can easily be viewed using Excel or other spreadsheet viewers, and are also nicely formatted when viewed on the GitHub website for the repository.

Examples of module test case files can be found in the tests/module_test_cases directory, while code that uses the testthat package to automatically run all the defined test cases for the standard BioCro module library via test_module_library can be found in the tests/testthat/test.Modules.R file.

See Also

- modules
- module_case_files
- test_module_library
- test_module

`module_write` *Generate a BioCro module header file.*

Description

To facilitate the creation of new BioCro modules, `module_write` generates a BioCro module header file. Given a set of input and output variables, `module_write` will create a C++ header file ('.h' file) by filling in a template with the input and output variables, ensuring the correct C++ syntax for a BioCro module.

Usage

```
module_write(
  module_name,
  module_library,
  module_type,
  inputs,
  outputs,
  output_equations = NULL,
  input_units = NULL,
  output_units = NULL
)
```

Arguments

`module_name`	A string for the module's name.
`module_library`	A string for the module's library namespace. E.g., `'biocro'`.
`module_type`	A string setting the module type: `'direct'` or `'differential'`.
`inputs`	A character vector of the module's input variables.
`outputs`	A character vector of the module's output variables.
`output_equations`	A character vector. The module's output variables will be updated with these variables. If `NULL`, a zero is inserted instead.
`input_units`	A character vector of the inputs' units. If `NULL`, no units are embedded.
`output_units`	A character vector of the outputs' units. If `NULL`, no units are embedded.

Details

`type` should be either `'direct'` or `'differential'`; however, `module_write` does not enforce this in case new module types are created in the future.

Value

A string containing a new BioCro module header file.

Note

This function returns a string and has no file I/O. Use `writeLines` to print the output to console, or to save the output. See examples below. Note that it is customary to name the header file with the same name as the module.

`module_write` checks for duplicate input or output variables, and if detected, it will raise an error. In theory, this check should ensure that the generated module will compile correctly. However, it is still possible to define an module that is circular and will not pass the checks in `validate_dynamical_system_inputs`. See `Example 4`.

Examples

```
# Example 1
# Inputs as character vector
xs = c('x1','x2','x3')

# Units
xs_units <- c('Mg / ha', 'Mg / ha / hr', 'dimensionless')

# Outputs
ys = c('y1','y2')

out <- module_write('testmod', 'testlib', 'direct',
    inputs=xs, input_units= xs_units, outputs=ys)

# Use writeLines to print to console
writeLines(out)

## Not run:
  # Use writeLines to save as a `.h` file
  writeLines(out, "./testmod.h")

## End(Not run)

# Example 2: A differential module
xs <- c('var_1','var_2')
out <- module_write('testmod', 'testlib', 'differential', xs, xs)
writeLines(out)

# Example 3: A module with pairwise names
# Here we use an outer product to generate pairwise combinations of
# tissues and pool types
tissues <- c('leaf', 'stem', 'root')
pools <- c('carbon', 'nitrogen')
xs <- as.vector(outer(tissues, pools, paste, sep = '_'))
out <- module_write('testmod', 'testlib', 'differential', xs, xs)
writeLines(out)

# Example 4: Circular modules

## Not run:
  out <- module_write(inputs = c('x' ,'x'))
```

```
  # Will compile, but will cause a "circular quantities" error if it is used
  # in a BioCro simulation:
  out <- module_write('inconsistent', 'examplelib', type='direct',
          inputs = 'x', outputs = 'x')

## End(Not run)
```

---

obsBea *Miscanthus assimilation field data*

Description

Assimilation in Miscanthus as measured in Beale, Bint, and Long 1996. The first column is the observed net assimilation rate (micromoles m^-2 s^-1). The second column is the observed quantum flux (micromoles m^-2 s^-1). The third column is the temperature (degrees Celsius). Relative humidity was not reported and thus was assumed to be 0.7.

Format

Data frame of dimensions 27 by 4.

Source

C. V. Beale, D. A. Bint, S. P. Long. 1996. Leaf photosynthesis in the C4-grass Miscanthus x giganteus, growing in the cool temperate climate of southern England. *J. Exp. Bot.* 47 (2): 267–273.

---

obsBeaC *Complete Miscanthus assimilation field data*

Description

Assimilation and stomatal conductance in Miscanthus as measured in Beale, Bint, and Long 1996. (Missing data are also included.) The first column is the date, the second the hour. Columns 3 and 4 are assimilation and stomatal conductance respectively.

Format

Data frame of dimensions 35 by 6.

Details

The third column is the observed net assimilation rate (micromoles m^-2 s^-1).

The fifth column is the observed quantum flux (micromoles m^-2 s^-1).

The sixth column is the temperature (degrees Celsius).

Source

C. V. Beale, D. A. Bint, S. P. Long. 1996. Leaf photosynthesis in the C4-grass Miscanthus x giganteus, growing in the cool temperate climate of southern England. *J. Exp. Bot.* 47 (2): 267–273.

`obsNaid` *Miscanthus assimilation data*

Description

Assimilation in Miscanthus as measured in Naidu et al. (2003). The first column is the observed net assimilation rate (micromoles m^-2 s^-1). The second column is the observed quantum flux (micromoles m^-2 s^-1). The third column is the temperature (degrees Celsius). The fourth column is the observed relative humidity in proportion (e.g. 0.7).

Format

Data frame of dimensions 16 by 4.

Source

S. L. Naidu, S. P. Moose, A. K. AL-Shoaibi, C. A. Raines, S. P. Long. 2003. Cold Tolerance of C4 photosynthesis in Miscanthus x giganteus: Adaptation in Amounts and Sequence of C4 Photosynthetic Enzymes. *Plant Physiol.* 132 (3): 1688–1697.

`partial_application` *Convenience Functions for Partial Application*

Description

Convenience functions for using partial application with BioCro

Usage

```
partial_run_biocro(
    initial_values = list(),
    parameters = list(),
    drivers,
    direct_module_names = list(),
    differential_module_names = list(),
    ode_solver = BioCro::default_ode_solvers$homemade_euler,
    arg_names,
    verbose = FALSE
)

partial_evaluate_module(module_name, input_quantities, arg_names)
```

Arguments

arg_names	A vector of strings specifying input quantities whose values should not be fixed when using partial application.
initial_values	Identical to the corresponding argument from run_biocro.
parameters	Identical to the corresponding argument from run_biocro.
drivers	Identical to the corresponding argument from run_biocro.
direct_module_names	Identical to the corresponding argument from run_biocro.
differential_module_names	Identical to the corresponding argument from run_biocro.
ode_solver	Identical to the corresponding argument from run_biocro.
verbose	Identical to the corresponding argument from run_biocro.
module_name	Identical to the corresponding argument from evaluate_module.
input_quantities	A list of named numeric elements representing any input quantities required by the module that are not included in arg_names; any extraneous quantities will be ignored by the module.

Details

Partial application is the technique of fixing some of the input arguments to a function, producing a new function with fewer inputs. In the context of BioCro, partial application can often be useful while varying some parameters, initial values, or drivers while performing optimization or sensitivity analysis. Optimizers (such as optim) typically require a function with a single input argument, so the partial application tools provided here help to create such functions.

Both partial_run_biocro and partial_evaluate_module accept the same arguments as their "regular" counterparts (run_biocro and evaluate_module) with the addition of arg_names, which specifies the input quantities that should not be fixed.

For partial_run_biocro, each element of arg_names must be the name of a quantity that is one of the initial_values, parameters, or drivers. For partial_evaluate_module, each element of arg_names must be the name of one of the module's input quantities.

When using one of the pre-defined crop growth models, it may be helpful to use the with command to pass arguments to partial_run_biocro; see the documentation for crop_model_definitions for more information.

Value

partial_run_biocro	A function that calls run_biocro with all of the inputs (except those specified in arg_names) set to the values specified by the original call to partial_run_biocro. The new function has one input (x), which can be a vector or list specifying the values of the quantities in arg_names. If x has no names, its elements must be supplied in the same order as in the original arg_names. If x has names, they must be identical to the elements of arg_names but can be in any order. Elements of x corresponding to drivers must be vectors having the same length as

the other drivers; they can be specified as a named element of a list or as sequential elements of a vector without names. The return value of the new function is a data frame as would be produced by run_biocro.

partial_evaluate_module
: A function that calls evaluate_module with the input quantities (except those specified in arg_names) set to the values specified by the original call to partial_evaluate_module. The new function has one input (x), which can be a vector or list specifying the values of the quantities in arg_names. If x has no names, its elements must be supplied in the same order as in the original arg_names. If x has names, they must be identical to the elements of arg_names but can be in any order. The return value of the new function is a list with two elements (inputs and outputs), each of which is a list of named numeric elements representing the module's input and output values. (Note that this differs from the output of evaluate_module, which only returns the outputs.)

See Also

- run_biocro
- evaluate_module

Examples

```
# Specify weather data to use in these examples
ex_weather <- get_growing_season_climate(weather$'2005')

# Example 1: varying the thermal time values at which senescence starts for
# different organs in a simulation; here we set them to the following values
# instead of the defaults:
#  - seneLeaf: 2000 degrees C * day
#  - seneStem: 2100 degrees C * day
#  - seneRoot: 2200 degrees C * day
#  - seneRhizome: 2300 degrees C * day
senescence_simulation <- partial_run_biocro(
  miscanthus_x_giganteus$initial_values,
  miscanthus_x_giganteus$parameters,
  ex_weather,
  miscanthus_x_giganteus$direct_modules,
  miscanthus_x_giganteus$differential_modules,
  miscanthus_x_giganteus$ode_solver,
  c('seneLeaf', 'seneStem', 'seneRoot', 'seneRhizome')
)
senescence_result <- senescence_simulation(c(2000, 2100, 2200, 2300))

# Example 2: a crude method for simulating the effects of climate change; here
# we increase the atmospheric CO2 concentration to 500 ppm and the temperature
# by 2 degrees C relative to 2005 temperatures. The commands below that call
# `temperature_simulation` all produce the same result.
temperature_simulation <- partial_run_biocro(
  miscanthus_x_giganteus$initial_values,
  miscanthus_x_giganteus$parameters,
  ex_weather,
```

```
  miscanthus_x_giganteus$direct_modules,
  miscanthus_x_giganteus$differential_modules,
  miscanthus_x_giganteus$ode_solver,
  c("Catm", "temp")
)
hot_result_1 <- temperature_simulation(c(500, ex_weather$temp + 2.0))
hot_result_2 <- temperature_simulation(list(Catm = 500, temp = ex_weather$temp + 2.0))
hot_result_3 <- temperature_simulation(list(temp = ex_weather$temp + 2.0, Catm = 500))

# Note that these commands will both produce errors:
# hot_result_4 <- temperature_simulation(c(Catm = 500, temp = ex_weather$temp + 2.0))
# hot_result_5 <- temperature_simulation(stats::setNames(
#   c(500, ex_weather$temp + 2.0),
#   c("Catm", rep("temp", length(ex_weather$temp)))
# ))

# Note that this command will produce a strange result where the first
# temperature value will be incorrectly interpreted as a `Catm` value, and the
# `Catm` value will be interpreted as the final temperature value.
# hot_result_6 <- temperature_simulation(c(ex_weather$temp + 2.0, 500))

# Example 3: varying the base and air temperature inputs to the
# 'thermal_time_linear' module from the 'BioCro' module library. The commands
# below that call `thermal_time_rate` all produce the same result.
thermal_time_rate <- partial_evaluate_module(
  'BioCro:thermal_time_linear',
  within(miscanthus_x_giganteus$parameters, {fractional_doy = 1}),
  c("temp", "tbase")
)
rate_result_1 <- thermal_time_rate(c(25, 10))
rate_result_2 <- thermal_time_rate(c(temp = 25, tbase = 10))
rate_result_3 <- thermal_time_rate(c(tbase = 10, temp = 25))
rate_result_4 <- thermal_time_rate(list(temp = 25, tbase = 10))
rate_result_5 <- thermal_time_rate(list(tbase = 10, temp = 25))
```

`run_biocro` *Simulate Crop Growth with BioCro*

Description

Runs a full crop growth simulation using the BioCro framework

Usage

```
run_biocro(
    initial_values = list(),
    parameters = list(),
    drivers,
    direct_module_names = list(),
    differential_module_names = list(),
```

```
    ode_solver = BioCro::default_ode_solvers$homemade_euler,
    verbose = FALSE
)
```

Arguments

initial_values	A list of named quantities representing the initial values of the differential quantities, i.e., the quantities whose derivatives are calculated by differential modules
parameters	A list of named quantities that don't change with time; must include a 'timestep' parameter (see 'drivers' for more info)
drivers	A data frame of quantities with rows at equally spaced time intervals specified in the 'parameters' as 'timestep'. The drivers must include either (1) 'time' or (2) 'doy' and 'hour' columns. In the latter case, 'time' will be automatically computed from 'doy' and 'hour' using add_time_to_weather_data, and the BioCro:format_time module will be added to the direct modules if it is not already present.
direct_module_names	A character vector or list of the fully-qualified names of the direct modules to use in the system; lists of available modules can be obtained via the get_all_modules function.
differential_module_names	A character vector or list of the fully-qualified names of the differential modules to use in the system; lists of available modules can be obtained via the get_all_modules function.
ode_solver	A list specifying details about the numerical ODE solver. The required elements are: • type: A string specifying the name of the algorithm to use; a list of available options can be obtained using the get_all_ode_solvers function. • output_step_size: The output time step size in units of 'timestep'. For example, if output_step_size is 0.25 and 'timestep' is 2, the output will have time points spaced by 0.25 * 2 = 0.5. • adaptive_rel_error_tol: used to set the relative error tolerance for adaptive step size methods • adaptive_abs_error_tol: used to set the absolute error tolerance for adaptive step size methods • adaptive_max_steps: determines how many times an adaptive step size method will attempt to find a new step size before indicating failure
verbose	A logical variable indicating whether or not to print dynamical system validation information. (More detailed startup information can be obtained with the validate_dynamical_system_inputs function.)

Details

run_biocro is the most important function in the BioCro package. The input arguments to this function are used to define a dynamical system and solve for its time evolution during a desired time period. For more details about how this function operates, see Lochocki *et al.* (2022) [doi:10.1093/insilicoplants/diac003].

When using one of the pre-defined crop growth models, it may be helpful to use the `with` command to pass arguments to `run_biocro`; see the documentation for `crop_model_definitions` for more information.

Value

A data frame where each column represents one of the quantities included in the simulation (with the exception of the parameters, since their values are guaranteed to not change with time) and each row represents a time point

See Also

- `get_all_modules`
- `get_all_ode_solvers`
- `validate_dynamical_system_inputs`
- `partial_run_biocro`

Examples

```
# Example: running a miscanthus simulation using weather data from 2005
result <- run_biocro(
  miscanthus_x_giganteus$initial_values,
  miscanthus_x_giganteus$parameters,
  get_growing_season_climate(weather$'2005'),
  miscanthus_x_giganteus$direct_modules,
  miscanthus_x_giganteus$differential_modules,
  miscanthus_x_giganteus$ode_solver
)

lattice::xyplot(
  Leaf + Stem + Root + Grain ~ TTc,
  data=result,
  type='l',
  auto=TRUE
)
```

`run_model_test_cases` *Run BioCro model test cases*

Description

BioCro models can be tested using test cases, which are sets of known outputs that correspond to particular inputs. The `run_model_test_cases` function runs one or more of these tests.

Note that *model tests* are distinct from the *module tests* described in `module_testing`.

Usage

```
run_model_test_cases(model_test_cases)
```

Arguments

model_test_cases
: A list of module test cases, each of which should be created using model_test_case.

Details

The run_model_test_cases function is a key part of the BioCro model testing system. See model_testing for more information.

For each test case, the following checks will be performed:

- The model definition must be valid according to validate_dynamical_system_inputs.
- The model will be run, which should not cause any errors or warnings.

For each test case where check_outputs was set to TRUE, the following additional checks comparing the new result to a saved result will be performed:

- The new result should have the same number of rows as the old result.
- With the exception of any columns in quantities_to_ignore, all columns in the stored result should be included in the new result.
- With the exception of any columns in quantities_to_ignore, all columns in the stored result should have the same values in the new result (to within the specified tolerance). This check will be made using all.equal with tolerance set to relative_tolerance.

When comparing the values of each column, values will only be checked for every Nth row of the new result, where N is the value of row_interval specified when defining the test case.

For each test case where check_outputs is TRUE, the stored result should be created using the update_stored_model_results function.

If any of the above checks fail for any of the supplied test cases, an error will be thrown with a descriptive message.

Besides the checks above, a warning message will also be sent to the user if there are columns in the new result that are not included in the saved result.

Value

If no issues are found, the function will return TRUE.

See Also

- model_testing
- model_test_case
- update_stored_model_results
- compare_model_output

Examples

```
# Define and run a test case for the miscanthus model
miscanthus_test_case <- model_test_case(
    'miscanthus_x_giganteus',
    miscanthus_x_giganteus,
    get_growing_season_climate(weather$'2005'),
    FALSE
)

run_model_test_cases(
  list(
    miscanthus_test_case
  )
)
```

---

soil_parameters *Soil properties*

Description

A collection of soil property data.

Usage

```
soil_parameters
```

Format

A list of named elements, where each element represents the hydraulic properties of one type of soil. The soil types are defined following the USDA soil texture classification scheme, and 11 of the 12 possible types are included ("silt" is not available). The following names are used to indicate the various soil types:

- `sand`
- `loamy_sand`
- `sandy_loam`
- `loam`
- `silt_loam`
- `sandy_clay_loam`
- `clay_loam`
- `silty_clay_loam`
- `sandy_clay`
- `silty_clay`
- `clay`

For each soil type, the following parameter values are provided:

- `soil_silt_content` (dimensionless)
- `soil_clay_content` (dimensionless)
- `soil_sand_content` (dimensionless)
- `soil_air_entry` (J / kg)
- `soil_b_coefficient` (dimensionless)
- `soil_saturated_conductivity` (J * s / m^3)
- `soil_saturation_capacity` (dimensionless)
- `soil_field_capacity` (dimensionless)
- `soil_wilting_point` (dimensionless)
- `soil_bulk_density` (Mg / m^3)

Source

These soil property values are based on Table 9.1 from Campbell and Norman's textbook *An Introduction to Environmental Biophysics* (1998). Bulk density values are taken from function `getsoilprop.c` from Melanie (Colorado). The bulk density of sand in `getsoilprop.c` is 0, which isn't sensible, and here a value of `1.60 Mg / m^3` is used instead.

The wilting point value of 0.21 (corrected from 0.32) for silty clay loam is based on the list of book corrections available from Brian Hornbuckle's teaching website using the Wayback Machine, since it does not seem to be available on his current site.

`soybean` *Soybean-BioCro model definition*

Description

Initial values, parameters, direct modules, differential modules, and a differential equation solver that can be used to run soybean growth simulations in Champaign, Illinois and other locations. Along with the soybean circadian clock specifications (`soybean_clock`), these values define the soybean growth model of Matthews *et al.* (2022) [doi:10.1093/insilicoplants/diab032], which is commonly referred to as *Soybean-BioCro*.

To represent soybean growth in Champaign, IL, these values must be paired with the Champaign weather data (`cmi_soybean_weather_data`). This weather data includes the output from the soybean circadian clock model (`soybean_clock`), so the clock components do not need to be included when running a soybean growth simulation using this weather data. The parameters already include the `clay_loam` values from the `soil_parameters` dataset, which is the appropriate soil type for Champaign.

Some specifications, such as the values of photosynthetic parameters, would remain the same in any location; others, such as the latitude or longitude, would need to change when simulating crop growth in different locations. Care must be taken to understand each input quantity before attempting to run simulations in other places or for other cultivars.

Usage

```
soybean
```

Format

A list of 5 named elements that are suitable for passing to `run_biocro`, as described in the help page for `crop_model_definitions`.

Details

As improvements are made to the BioCro modules, their behavior changes, and the soybean model parameters must be updated. Following significant module updates, reparameterization is performed using the same method and data as used in Matthews *et al.* (2022). The following is a summary of reparameterizations that have occurred since the original publication of the Soybean-BioCro model:

- *2023-06-18*: Several modules have been updated, and the value of the atmospheric transmittance has been changed from 0.85 to 0.6 based on Campbell and Norman, An Introduction to Environmental Biophysics, 2nd Edition, Pg 173. Due to these changes, reparameterization of the following was required: `alphaLeaf`, `alphaRoot`, `alphaStem`, `alphaShell`, `betaLeaf`, `betaRoot`, `betaStem`, `betaShell`, `rateSeneLeaf`, `rateSeneStem`, `alphaSeneLeaf`, `betaSeneLeaf`, `alphaSeneStem`, and `betaSeneStem`.
- *2023-03-15*: Several modules have been updated. The most significant changes are that (1) the `BioCro:no_leaf_resp_neg_assim_partitioning_growth_calculator` now reduces the leaf growth rate in response to water stress and (2) the partitioning modules now include a new tissue type (`shell`). The new component allows us to distinguish between components of the soybean pod, where `shell` represents the pericarp and `grain` represents the seed. This distinction has been found to be important for accurately predicting seed biomass, which is more important in agricultural settings than the entire pod mass, since the pericarp is not included in typical yield measurements. Due to these changes, reparameterization of the following was required: `alphaLeaf`, `alphaRoot`, `alphaStem`, `alphaShell`, `betaLeaf`, `betaRoot`, `betaStem`, `betaShell`, `rateSeneLeaf`, `rateSeneStem`, `alphaSeneLeaf`, `betaSeneLeaf`, `alphaSeneStem`, and `betaSeneStem`. It was also necessary to add a new direct module to the model definition: `BioCro:leaf_water_stress_exponential`. This module calculates the fractional reduction in leaf growth rate due to water stress.
- *2024-09-12*: Several changes have been made: (1) The `mrc1` and `mrc2` were renamed to `grc_stem` and `grc_root`, respectively. These two parameters are used to scale the assimilate rate, which is commonly called growth respiration coefficient (grc). (2) A new module, `BioCro:maintenance_respiration`, has been added to account for maintenance respiration during the biomass partitioning. This module removes a fraction from each organ by a constant parameter called `mrc_*` (e.g., `mrc_leaf`) and also by a temperature-dependent Q10 scaling factor. Among these `mrc_*` parameters, `mrc_leaf` and `mrc_stem` are set equal to represent maintenance respiration for the shoot, while `mrc_grain` is assigned a negligible value to prevent grain biomass reduction at the season end. No decreasing trends have been seen in the observed data. (3) Parameter optimizations against the 2002-2006 biomass datasets were performed to accommodate these changes.

Whenever a reparameterization is made, this list should be updated, and any vignettes using the soybean model should be checked to see if any axis limits, etc., need to change.

Source

This model is described in detail in Matthews *et al.* (2022) [doi:10.1093/insilicoplants/diab032]. Here we make a few notes about some of its components:

- For this model, the ODE solver type should not be `boost_rosenbrock` or `auto` (which defaults to `boost_rosenbrock` when a fixed step size Euler ODE solver is not required, as in this case) since the integration will fail unless the tolerances are stringent (e.g., `output_step_size = 0.01`, `adaptive_rel_error_tol = 1e-9`, `adaptive_abs_error_tol = 1e-9`).
- For the initial total seed mass per land area, we use the following equation: `Number of seeds per meter * weight per seed / row spacing`. The number of seeds per meter is 20 and the row spacing is 0.38 m, as reported in Morgan *et al.* (2004) [doi:10.1104/pp.104.043968]. The weight per seed is based on the average of .12 to .18 grams, as reported by Feedipedia. Thus, we have an initial biomass of `(20 seeds / m) * (0.15 g / seed) / (0.38 m) = 7.89 g / m^2`, equivalent to `0.0789 Mg / ha` in the typical BioCro units. Since this model does not have a seed component, this value is used to determine the initial `Leaf`, `Stem`, and `Root` biomass, assuming 80% leaf, 10% stem, and 10% root.
- For historical reasons, the seed tissue in this model is called `Grain`. The entire pod biomass can be calculated by adding the `Grain` and `Shell` biomass.
- For historical reasons, this model includes a `Rhizome` tissue. Soybean does not have a rhizome, so the rhizome in the model does not grow or senesce. To achieve this, the `kRhizome_emr` and `rateSeneRhizome` parameters must be set to 0. It is also necessary to specify values for several other quantities such as `alphaSeneRhizome`, `betaSeneRhizome`, and the initial rhizome mass, although the actual values of these quantities will have no effect on the simulation output.
- For historical reasons, some of the modules that define Soybean-BioCro require input quantities that are not actually used for any calculations; these "extraneous" parameters are identified in `data/soybean.R`.
- The `sowing_fractional_doy` input to the `soybean_development_rate_calculator` module is set to 0 because Soybean-BioCro uses the weather data to set the sowing time. In other words, the weather data is truncated so it begins at the beggining of the simulation.
- Leaf reflectance and transmittance in the PAR band are estimated from [doi:10.2134/agronmonogr31.c7], [doi:10.2134/agronj1971.00021962006300010038x], and [doi:10.2134/agronj1991.00021962008300030026x]. Reflectance and transmittance in the NIR band are from [doi:10.2134/agronmonogr31.c7].

See Also

- `run_biocro`
- `modules`
- `crop_model_definitions`
- `soybean_clock`

soybean_clock *Soybean-BioCro circadian clock model definition*

Description

Initial values, parameters, direct modules, differential modules, and a differential equation solver that can be used to run soybean circadian clock simulations in Champaign, Illinois and other locations. Along with the soybean growth specifications (soybean), these values define the soybean growth model of Matthews *et al.* (2022) [doi:10.1093/insilicoplants/diab032], which is commonly referred to as *Soybean-BioCro*.

To represent a soybean circadian clock in Champaign, Illinois, these values must be paired with the weather data from cmi_weather_data.

Usage

```
soybean_clock
```

Format

A list of 5 named elements that are suitable for passing to run_biocro, as described in the help page for crop_model_definitions.

Source

This model is described in detail in Matthews *et al.* (2022) [doi:10.1093/insilicoplants/diab032] and Lochocki & McGrath (2021) [doi:10.1093/insilicoplants/diab016].

Here, we use initial phases for the dawn and dusk oscillators of `200.0` and `80.0` radians, respectively. These values are optimized for simulations beginning at midnight on January 1, and should require minimal time for transient signals to die down. These values were determined by running a simulation for one year starting on January 1, and recording the oscillator states at the end of December 31.

See Also

- run_biocro
- modules
- crop_model_definitions
- soybean

system_derivatives *Calculate Derivatives for Differential Quantities*

Description

Solving a BioCro model using one of R's available differential equation solvers

Usage

```
system_derivatives(
  parameters = list(),
  drivers,
  direct_module_names = list(),
  differential_module_names = list()
)
```

Arguments

parameters	Identical to the corresponding argument from run_biocro.
drivers	Identical to the corresponding argument from run_biocro.
direct_module_names	Identical to the corresponding argument from run_biocro.
differential_module_names	Identical to the corresponding argument from run_biocro.

Details

system_derivatives accepts the same input arguments as run_biocro with the exceptions of ode_solver and initial_values; this function is intended to be passed to an ODE solver in R, which will solve for the system's time dependence as its diffferential quantities evolve from their initial values, so ode_solver and initial_values are not required here.

When using one of the pre-defined crop growth models, it may be helpful to use the with command to pass arguments to system_derivatives; see the documentation for crop_model_definitions for more information.

Value

The return value of system_derivatives is a function with three inputs (t, differential_quantities, and parms) that returns derivatives for each of the differential quantities in the dynamical system determined by the original inputs (parameters, drivers, direct_module_names, and differential_module_names).

This function signature and the requirements for its inputs are set by the LSODES function from the deSolve package. The t input should be a single time value and the differential_quantities input should be a vector with the names of the differential quantities defined by the modules. parms is required by LSODES, but we don't use it for anything.

This function can be passed to LSODES as an alternative integration method, rather than using one of BioCro's built-in solvers.

See Also

run_biocro

Examples

```
# Note: Example 3 below may take several minutes to run. Patience is required!

# Example 1: calculating a single derivative using a soybean model

soybean_system <- system_derivatives(
  soybean$parameters,
  soybean_weather$'2002',
  soybean$direct_modules,
  soybean$differential_modules
)

derivs <- soybean_system(0, unlist(soybean$initial_values), NULL)

# Example 2: a simple oscillator with only one module

times = seq(0, 5, by = 1) # times spaced by `timestep`

oscillator_system_derivatives <- system_derivatives(
  list(
    timestep = 1,
    mass = 1,
    spring_constant = 1
  ),
  data.frame(time = times),
  c(),
  'BioCro:harmonic_oscillator'
)

result <- as.data.frame(deSolve::lsodes(
  c(position=0, velocity=1),
  times,
  oscillator_system_derivatives
))

lattice::xyplot(
  position + velocity ~ time,
  type='l',
  auto=TRUE,
  data=result
)

# Example 3: solving 500 hours of a soybean simulation. This will run slowly
# compared to a regular call to `run_biocro`.



soybean_system <- system_derivatives(
```

```
  soybean$parameters,
  soybean_weather$'2002',
  soybean$direct_modules,
  soybean$differential_modules
)

times = seq(from=0, to=500, by=1)

result <- as.data.frame(deSolve::lsodes(unlist(soybean$initial_values), times, soybean_system))

lattice::xyplot(Leaf + Stem ~ time, type='l', auto=TRUE, data=result)
```

`test_module` *Run BioCro module test cases*

Description

Modules can be tested using test cases, which are sets of known outputs that correspond to particular inputs. The functions here provide ways to create test cases and test modules.

`test_module` runs one test case for a module, returning an error message if its output does not match the expected value.

`case` helps define test cases for module testing by combining the required elements into a list with the correct names as required by `test_module`.

`cases_from_csv` helps define test cases for module testing by creating a list of test cases from a `csv` file.

Note that *module tests* are distinct from the *model tests* described in `model_testing`.

Usage

```
test_module(module_name, case_to_test)

case(inputs, expected_outputs, description)

cases_from_csv(module_name, directory)
```

Arguments

`module_name`	A string specifying one BioCro module, formatted like `library_name:local_module_name`, where `library_name` is the name of a library that contains a module with local name `local_module_name`; such fully-qualified module names can be formed manually or with `module_paste`.
`case_to_test`	A list with three named elements that describe a module test case: • `inputs`: A list of module inputs, i.e., a list of named numeric elements corresponding to the module's input quantities.

- `expected_outputs`: A list of expected module outputs, i.e., a list of named numeric elements corresponding to the expected values of the module's output quantities.
- `description`: A string describing the test case, e.g. `"temp below tbase"`. The description should be succinct and not contain any newline characters.

`inputs` See the corresponding entry in `test_case` above.

`expected_outputs`
See the corresponding entry in `test_case` above.

`description` See the corresponding entry in `test_case` above.

`directory` The directory where module test case files are stored, e.g. `file.path('tests', 'module_test_cases')`

Details

The `test_module` function forms the basis for the BioCro module testing system. (See `module_testing` for more information.) The functions `case` and `cases_from_csv` are complementary to `test_module` because they help to pass suitably-formatted test cases to `test_module`.

Value

`test_module` If the test passes, an empty string; otherwise, an informative message about what went wrong.

`case` A list with three named elements (`inputs`, `expected_outputs`, and `description`) formed from the input arguments.

`cases_from_csv` A list of test cases created by the `case` function that are each suitable for passing to the `test_module` function.

See Also

- `modules`
- `module_case_files`
- `module_paste`
- `module_testing`
- `test_module_library`

Examples

```
# Example 1: Defining an individual test case for the 'BioCro' module library's
# 'thermal_time_linear' module and running the test. This test will pass, so the
# return value will be an empty string: `character(0)`
test_module(
  'BioCro:thermal_time_linear',
  case(
    list(fractional_doy = 101, sowing_fractional_doy = 100, tbase = 20, temp = 44),
    list(TTc = 1.0),
    'temp above tbase'
  )
```

```
)

# Example 2: Defining an individual test case for the 'BioCro' module library's
# 'thermal_time_linear' module and running the test. This test will fail, so the
# return value will be a message indicating the failure.
test_module(
  'BioCro:thermal_time_linear',
  case(
    list(fractional_doy = 101, sowing_fractional_doy = 100, tbase = 20, temp = 44),
    list(TTc = 2.0),
    'temp above tbase'
  )
)

# Example 3: Loading a set of test cases from a file and running one of them.
# Note: here we use the `initialize_csv` function first to ensure that there is
# a properly defined test file in a temporary directory.

td <- tempdir()

module_name <- 'BioCro:thermal_time_linear'
initialize_csv(module_name, td)
cases <- cases_from_csv(module_name, td)
test_module(module_name, cases[[1]])
```

`test_module_library` *Run module test cases for an entire BioCro module library*

Description

Modules can be tested using test cases, which are sets of known outputs that correspond to particular inputs. The `test_module_library` function provides a way to run all test cases for all modules in a BioCro module library.

Note that *module tests* are distinct from the *model tests* described in `model_testing`.

Usage

```
test_module_library(library_name, directory, modules_to_skip = c())
```

Arguments

`library_name`	The name of a BioCro module library.
`directory`	The directory where module test case files are stored, e.g. `file.path('tests', 'module_test_cases')`
`modules_to_skip`	A vector of local module name strings indicating any modules from the library that should not be tested. This feature should be used sparingly, since there are very few legitimate reasons to skip a module test.

Details

For each CSV file in the specified directory, `test_module_library` determines the corresponding module name and checks to make sure it is part of the specified library. If there are test cases for modules not in the library, `test_module_library` throws an error with a message containing the "unexpected" module test cases.

For each module in the specified library, `test_module_library` loads stored test cases from the specified directory and runs each test case, storing information about any test failures or other issues that may occur. If any problems are detected, `test_module_library` throws an error with a message describing the issues.

For an example of how this function can be used along with the testthat package, see `tests/testthat/test.Modules.R`.

Value

None

See Also

- modules
- module_case_files
- module_testing
- test_module

Examples

```
# Here we will initialize a module test case file in a temporary directory, and
# then use `test_module_library` to test it. We will need to skip most of the
# modules in the library, since we only have a test case for one of them.

td <- file.path(tempdir(), 'module_test_cases')
dir.create(td, showWarnings = FALSE)

initialize_csv(
  'BioCro:thermal_time_linear',
  td,
  nonstandard_inputs = list(temp = -1),
  overwrite = TRUE
)

# Get a list of local module names, excluding the module that has a test case
all_modules <- get_all_modules('BioCro')
skip <- all_modules[all_modules != 'BioCro:thermal_time_linear']
skip <- gsub('BioCro:', '', skip)

test_module_library('BioCro', td, skip)

# If we attempt to test the entire library, we will get errors since only one
# module actually has an associated case file
tryCatch(
  {
```

```
    test_module_library('BioCro', td)
  },
  error = function(e) {print(e)}
)
```

`Time Variable` *The 'time' variable in BioCro*

Description

Even when using an adaptive ODE solver, `run_biocro` returns values at evenly spaced time intervals given by the 'timestep' parameter. It is assumed that the drivers are provided at intervals spaced by this 'timestep'. To prevent user error, `run_biocro` will check that the 'drivers" are spaced by 'timestep' but to do so, the 'drivers' must contain a 'time'.

Details

When differential modules are passed, BioCro will check that the drivers have a 'time' variable that satisfies: 'time[n+1] - time[n] = timestep' for all 'n' (no checks are required for direct_modules). BioCro assumes that differential modules return rates-of-change in the same units as 'timestep'.

Unless using modules which require time to correpond to a calendar time (a date and time), the starting 'time[1]' has no special meaning. If 'time[1] = 0' then 'time' is the amount of time that has passed since the simulation's start (in the same units as 'timestep').

Certain modules expect 'time' to specify a date and time with respect to the calendar. For most BioCro simulations, the rates of change are measured per hour, therefore 'timestep' has units hour.

Therefore, the time variable is defined as the number of hours that have passed since midnight January 1st of the same year. For example for year 2023: + 'time = 0' is '2023-01-01 00:00:00' + 'time = 1' is '2023-01-01 01:00:00' + 'time = 50'is '2023-01-03 02:00:00' Note in version 3.1.3 and earlier of BioCro 'time' counted the number of days rather than hours.

`update_stored_model_results`
Updated stored result for a BioCro model test case

Description

BioCro models can be tested using test cases, which are sets of known outputs that correspond to particular inputs. The `update_stored_model_results` function stores the model outputs so they can be used for testing.

Note that *model tests* are distinct from the *module tests* described in `module_testing`.

Usage

```
update_stored_model_results(mtc)
```

Arguments

mtc	A single module test case, which should be created using model_test_case.

Details

The update_stored_model_results function is a key part of the BioCro model testing system. See model_testing for more information.

This function will run the model with the supplied drivers and store the results in an appropriately-named CSV file in the specified directory.

To save space, the values in the result will be rounded using signif, where the number of digits is specified in the module test case. Also, only every Nth row will be retained, where N is the value of row_interval specified in the module test case.

The saved result created by this function will be retrieved by run_model_test_cases when checking the test case.

Value

This function has no return value.

See Also

- model_testing
- model_test_case
- run_model_test_cases
- compare_model_output

Examples

```
# Define a test case for the miscanthus model and save the model output to a
# temporary directory
miscanthus_test_case <- model_test_case(
    'miscanthus_x_giganteus',
    miscanthus_x_giganteus,
    get_growing_season_climate(weather$'2005'),
    TRUE,
    tempdir(),
    'soil_evaporation_rate'
)

update_stored_model_results(miscanthus_test_case)

# The output file's name will be based on the test case description
fpath <- file.path(tempdir(), 'miscanthus_x_giganteus_simulation.csv')

# Check that the output file exists and then load it
if (file.exists(fpath)) {
  saved_result <- read.csv(fpath)
  str(saved_result)
}
```

`willow` *Willow model definition*

Description

Initial values, parameters, direct modules, differential modules, and a differential equation solver that can be used to run willow growth simulations in Champaign, Illinois and other locations.

To represent willow growth in Champaign, IL, these values must be paired with the Champaign weather data (`cmi_weather_data`). The parameters already include the `clay_loam` values from the `soil_parameters` dataset, which is the appropriate soil type for Champaign.

Some specifications, such as the values of photosynthetic parameters, would remain the same in any location; others, such as the latitude or longitude, would need to change when simulating crop growth in different locations. Care must be taken to understand each input quantity before attempting to run simulations in other places or for other cultivars.

Usage

```
willow
```

Format

A list of 5 named elements that are suitable for passing to `run_biocro`, as described in the help page for `crop_model_definitions`.

Source

This model was originally described in Wang *et al.* (2015) [doi:10.1111/pce.12556]. Since its original parameterization, the behavior of several of its core modules has changed as bugs have been identified and fixed, so this model likely needs to be reparameterized before it can be used for realistic simulations.

See Also

- `run_biocro`
- `modules`
- `crop_model_definitions`

Index