

Package: plant (via r-universe)

October 13, 2026

Title A Package for Modelling Forest Trait Ecology and Evolution

Version 2.0.0.9002

Maintainer Daniel Falster <daniel.falster@unsw.edu.au>

Description Solves trait, size and patch structured model from
(Falster et al. 2016) using either method of characteristics or
as stochastic, finite-sized population.

Depends R (>= 4.5.0)

Encoding UTF-8

License AGPL (>= 3)

LinkingTo Rcpp, BH, odelia (== 0.7.3), phylloptim (== 0.10.2)

Imports Rcpp, R6, logger, crayon, magrittr, tibble, rlang, dplyr,
tidyselect (>= 1.2.0), tidyr, purrr, odelia (>= 0.7.3), withr,
parallel, tools

Suggests phylloptim, testthat, rprojroot, pkgdown, here, numDeriv,
RcppR6 (>= 0.2.3), knitr, rmarkdown, markdown, bench, ggplot2,
ggridges, patchwork, kableExtra

Remotes richfitz/RcppR6, traitecoevo/odelia@88cbff7,
traitecoevo/phylloptim@edaffe8

URL <https://github.com/traitecoevo/plant>

BugReports <https://github.com/traitecoevo/plant/issues>

Config/roxygen2/version 8.0.0

Config/testthat/edition 3

Config/testthat/parallel true

RoxygenNote 7.3.3

Config/pak/sysreqs libicu-dev

Repository <https://traitecoevo.r-universe.dev>

Date/Publication 2026-10-13 20:01:13 UTC

RemoteUrl <https://github.com/traitecoevo/plant>

RemoteRef HEAD

RemoteSha bd09645dec0f6d1850ffa96298dda1af6b295e07

Contents

add_strategies	3
check_driver_interpolation	5
control	6
Control	7
Disturbance_Regime	7
environment_type	7
events	8
export_patch_state	10
FF16_Environment	11
FF16_expand_state	11
FF16_generate_stand_report	12
FF16_hyperpar	13
FF16_Individual	13
FF16_Parameters	14
FF16_Pars	14
FF16_Strategy	15
fitness_window	15
fixed_point_fitness	16
generate_strategy	17
gradient_control	18
grow_individual_to_size	19
grow_individual_to_time	20
Individual	20
integrate_over_size_distribution	21
Internals	21
interpolate_to_heights	22
interpolate_to_times	22
K93_hyperpar	23
K93_Individual	23
K93_Pars	24
K93_Strategy	24
Leaf	25
make_FF16_hyperpar	26
make_hyperpar	27
make_initial_state	28
make_K93_hyperpar	29
make_TF24_hyperpar	29
model_version	31
No_Disturbance	32
node_schedule_times_default	32
node_schedule_times_periodic	33
NodeSchedule	34
OdeControl	34
optimise_individual_rate_at_size_by_trait	35
plant_log_console	36
plot_size_distribution	37
pulse_series_poisson	37

QK	38
rainfall_pulse	39
resource_compensation_point	40
run_plant_benchmarks	40
run_scm	41
run_stochastic_collect	42
run_until_converged	43
scenario_eval	45
scm_base_parameters	48
seq_log	49
set_initial_state	49
stand_census	50
stand_census_state_adjoint	51
stand_gradient	52
stand_gradient_compare	54
stand_gradient_refused	54
TF24_Environment	55
TF24_generate_stand_report	55
TF24_H_ANCHOR	56
TF24_hyperpar	56
TF24_Individual	57
TF24_K_s_from_whole_stem	57
TF24_Parameters	58
TF24_Pars	59
TF24_plot_diagnostics	59
TF24_Strategy	60
TF24f_generate_stand_report	60
TF24f_hyperpar	61
TF24f_Individual	61
TF24f_Parameters	62
TF24f_Strategy	63
tidy_individual	63
tidy_snapshots	64
tidy_species	65
trait_matrix	65
util_colour_set_opacity	66
Weibull_Disturbance_Regime	66
Index	67

Description

Add strategies to a `Parameters` object. `add_strategies` appends the new strategies to any existing residents (controlled by `keep_existing`); `add_mutant` introduces strategies as mutants (i.e. replacing the resident set). Both translate trait values into strategies via `generate_strategy` and are pipe-friendly (the `Parameters` object is the first argument).

Usage

```
add_strategies(
  p,
  traits,
  hyperpar = param_hyperpar(p),
  birth_rate = 1,
  keep_existing = TRUE
)

add_mutant(p, traits, hyperpar = param_hyperpar(p), birth_rate = 1)

expand_parameters(
  trait_matrix,
  p,
  hyperpar = param_hyperpar(p),
  birth_rate_list = 1,
  keep_existing_strategies = TRUE
)

mutant_parameters(
  trait_matrix,
  p,
  hyperpar = param_hyperpar(p),
  birth_rate_list = 1,
  keep_existing_strategies = FALSE
)
```

Arguments

<code>p</code>	A <code>Parameters</code> object.
<code>traits</code>	A matrix of traits corresponding to the new strategies to introduce (see <code>trait_matrix</code>).
<code>hyperpar</code>	Hyperparameter function to use. By default links to the standard function for this strategy type.
<code>birth_rate</code>	Birth rate(s), one per row of <code>traits</code> . See <code>generate_strategy</code> .
<code>keep_existing</code>	Should existing resident strategies be retained?
<code>trait_matrix</code>	Deprecated (<code>expand_parameters</code> / <code>mutant_parameters</code>); use <code>traits</code> .
<code>birth_rate_list</code>	Deprecated; use <code>birth_rate</code> .

`keep_existing_strategies`
 Deprecated; use `keep_existing`.

`check_driver_interpolation`
Check how a driver's control points survive interpolation

Description

Extrinsic drivers are interpolated with a cubic spline, which is a poor fit for intermittent forcing such as daily rainfall. Between mostly-zero control points a cubic overshoots in both directions: a realistic daily rainfall series with a ~10% wet-day fraction evaluates **negative** at roughly 45% of points, and overshoots pulse peaks several-fold. Negative rainfall is floored at zero inside the TF24 soil balance, but the distortion of pulse magnitudes remains, so it is worth inspecting a series before committing to a long run.

Usage

```
check_driver_interpolation(
  environment,
  driver_name,
  x,
  y,
  n_eval = 20001,
  warn = TRUE,
  tol_integral = 0.01
)
```

Arguments

<code>environment</code>	An <code>Environment</code> object with the driver already set (via <code>environment\$extrinsic_drivers_set_v</code> <code>x, y</code>).
<code>driver_name</code>	Name of the driver to check, e.g. "rainfall".
<code>x, y</code>	The control points that were supplied for this driver. Used to compute the intended integral and value range to compare against.
<code>n_eval</code>	Number of points on the fine evaluation grid.
<code>warn</code>	If <code>TRUE</code> (default), raise a warning when the interpolated series goes negative or its integral departs from the supplied series by more than <code>tol_integral</code> .
<code>tol_integral</code>	Relative tolerance on the integral mismatch before warning.

Details

This runs that inspection: it evaluates the driver on a fine grid and compares the result against the control points you supplied.

Value

Invisibly, a list with the evaluated `min`, `max`, `frac_negative`, the `integral_supplied` and `integral_evaluated` (trapezoid), their `integral_rel_error`, and `negative_area` — the amount spuriously removed by undershoot. Printed as a short summary as a side effect.

Examples

```
e <- TF24_Environment()
t <- seq(0, 2, length.out = 730)
# intermittent daily rainfall: wet season only, most days dry
set.seed(1)
y <- ifelse((t %% 1) > 0.5 & runif(length(t)) < 0.3, 5, 0)
e$extrinsic_drivers_set_variable("rainfall", t, y)
check_driver_interpolation(e, "rainfall", t, y)
```

`control`

Control presets

Description

Construct a **Control** object. `control()` is a lowercase alias for the `Control()` constructor, whose defaults are the pragmatic, fast-ish settings used for essentially all of plant's runs (see `control.cpp`). `control_accurate()` tightens the ODE and schedule tolerances for high-accuracy runs at the cost of speed.

Usage

```
control(...)

control_accurate(base = Control())
```

Arguments

<code>...</code>	Named control fields, passed to <code>Control()</code> .
<code>base</code>	An optional Control object to tighten; defaults are used if omitted.

Value

A **Control** object.

Control	<i>Control parameters</i>
---------	---------------------------

Description

Control parameters that tune various aspects of the numerical solvers.

Usage

```
Control(..., values = list(...))
```

Arguments

<code>..., values</code>	Values to initialise the struct with (either as variadic arguments, or as a list, but not both).
--------------------------	--

Disturbance_Regime	<i>Disturbance regime</i>
--------------------	---------------------------

Description

Base class of representing patch disturbance

Usage

```
Disturbance_Regime()
```

environment_type	<i>Creates an environment object of specified type</i>
------------------	--

Description

Creates an environment object of specified type

Usage

```
environment_type(type)
```

```
Environment(type = NULL)
```

Arguments

<code>type</code>	Any environment name as a string, e.g.: "FF16_Env".
-------------------	---

events
Discrete events applied during an SCM run.

Description

An event is a (**time**, **action**) pair applied between solver legs: the integrator is stopped at the event time, the action changes the patch, the system is recomputed, and integration resumes from the new state. Node introduction has always worked this way; these functions make the same mechanism available for resource pulses, harvest and climate extremes.

Usage

```
events(...)

events_default(p)

node_introductions(p)

resource_pulse(time, resource, amount)

harvest(time, fraction, size_min = 0, size_max = Inf, species = NULL)

climate_extreme(
  time,
  intensity,
  duration = 14/365,
  threshold = 40,
  sensitivity = 1,
  species = NULL
)
```

Arguments

...	For events , objects returned by the individual event constructors, or whole Events objects; these are concatenated. Each constructor is vectorised over its arguments, so a whole series of pulses is one call.
p	Parameters object, whose node_schedule_times supply the introduction times.
time	Event time(s), in years of patch age.
resource	Index of the environment's resource pool to add to. What a resource is depends on the environment: TF24's are its soil layers, so resource = 1 is the surface. See rainfall_pulse for that case under a name that reads correctly.
amount	How much to add, in whatever the environment measures that resource in. What the pool cannot hold is shed rather than forced in; scm\$event_log reports how much of each.

fraction	Fraction of individuals removed, in $[0, 1)$.
size_min, size_max	Only individuals whose size falls in this band are removed. Size is the individual's size coordinate — height, for plant's models. The defaults take the whole population.
species	Index of the species to act on; <code>NULL</code> (the default) acts on every species in the patch.
intensity	Peak level reached during the episode, in whatever the model measures the stressor in (deg C for a heatwave, say).
duration	Nominal duration of the episode (years; a fortnight is $14 / 365$). The action sub-integrates over it at half-hourly steps; patch time does not advance.
threshold	Level above which dose accrues; below it the episode does nothing.
sensitivity	Dose accrued per unit-year above threshold .

Details

The vocabulary here is deliberately model-agnostic, because the machinery is shared by every strategy and environment and nothing about it is specific to plants: a resource pulse is water in TF24 and could be anything countable in a size-structured animal model, and a climate extreme is heat in one model and could be cold or salinity in another. Names that are only true of one model live with that model — `rainfall_pulse` is a resource pulse of water into TF24's surface soil layer.

Each event carries when it happens, what kind of thing it is, what it acts on, and the values it needs. The target is `"environment"` (its resource pools and drivers), `"patch"` (every species) or `"species"` (one, named by `species`). There is deliberately no per-cohort target: a cohort has no stable address across a run, because nodes are appended and never removed and schedule refinement changes how many exist. Selecting particular cohorts is expressed as a size range instead, which is both well defined and what size-selective removal needs.

Events are instantaneous *to the solver*: patch time does not advance across one. An action is still free to reach its answer by integrating its own fast sub-model over a nominal duration — `climate_extreme` does — with demography frozen. The solver sees one jump either way.

Events sharing a time are applied in a fixed order: environment events first, then demographic ones, then node introductions — so a node introduced at that instant starts life in the post-event environment.

Because an event is also a stop time for the integrator, adding one changes the adaptive step sequence. A run with events legitimately differs from one without, at solver tolerance, even away from the events themselves.

What each event actually did — as against what was asked of it — is recorded, and readable afterwards as `scm$event_log`. The two differ routinely: a pulse is capped at what the pool can hold, and harvesting a size class removes whatever was in it.

`events_default(p)` is the schedule a run gets when no events are supplied: the node introductions from `p$node_schedule_times` and nothing else. Start from it when adding events to an otherwise ordinary run — `events(events_default(p), harvest(...))` — or pass it on its own, which reproduces the default run exactly.

harvest covers removal generally, not only the forestry sense: leave the size band at its defaults and it is an across-the-board knock-down; set **size_min** and it takes everything above a size; set both and it thins one size class. They are one action, differing only in which individuals are selected.

Value

An **Events** object: a list with **time**, **type**, **target**, **target_index** and **params**, in schedule order.

Examples

```
p <- scm_base_parameters("FF16")
p <- add_strategies(p, trait_matrix(1, "lma"))
ev <- events(
  events_default(p),
  harvest(time = 20, fraction = 0.3)
)
```

<code>export_patch_state</code>	<i>Export patch state from an SCM</i>
---------------------------------	---------------------------------------

Description

Export the full state of a patch from a (run) **SCM** so it can be re-imported to seed a new run (see [set_initial_state](#)). The exported object captures everything needed to reproduce the patch's forward trajectory: every node's ODE state plus the per-node birth bookkeeping (introduction time, patch-age density and survival probability at birth) that is not part of the ODE state but feeds the rates and lifetime-fitness integrals, the patch age, and the not-yet-introduced ("residual") portion of the node-introduction schedule. A resume re-derives the two birth weights from its own regime (see [set_initial_state](#)); they are exported as a record of the run that made them.

Usage

```
export_patch_state(scm, step = NULL)
```

Arguments

scm	An SCM object that has been run.
step	Optional 1-based index into <code>scm\$history</code> (requires the run to have been performed with <code>collect = TRUE</code>); when <code>NULL</code> (default) the SCM 's current/final patch is used.

Value

A list describing the patch state, suitable for `set_initial_state`: `time`, `n` (nodes per species), `ode_state` (flat), the per-species lists `node_times`, `patch_density` and `pr_patch_survival`, the residual `node_schedule_times`, and `events`, the run's events still to come – its residual introductions and every action the run had not yet applied when the state was taken (an action at the export time itself is pending in a collected snapshot and in a run that `SCM$run_to()` stopped there, and applied in a finished run's final patch). Events are a `run_scm` argument rather than a field of `Parameters`, so pass `state$events` to the resumed run's `events` argument; a resumed run given none re-runs the residual introductions alone and silently loses any pulse or harvest still ahead.

With the absorbing top layer on (`Control$absorb_epsilon`), each species' block of `ode_state` ends with the layer: its pinned individual's states, then its density, its two fitness weights and its two seed totals. A state exported without the layer has no such block, so resuming it with the layer on needs that many zeros inserted after each species' nodes.

See Also

`set_initial_state`, `run_scm`

FF16_Environment	Create an FF16_Environment object.
------------------	------------------------------------

Description

Create an FF16_Environment object.

Usage

```
FF16_Environment()
```

```
K93_Environment()
```

FF16_expand_state	Add additional state variables to the species component in output of a model.
-------------------	---

Description

Add additional state variables to the species component in output of a model.

Usage

```

FF16_expand_state(results)

K93_expand_state(results)

expand_state(results)

TF24_expand_state(results)

TF24f_expand_state(results)

```

Arguments

results from ‘tidy_patch’

Value

similar format to input, but with additional columns for additional state variables

FF16_generate_stand_report

Generates a report on stand grown with FF16 strategy

Description

Builds a detailed report on stand grown with FF16 strategy, based on the template Rmd file provided. The reports are rendered as html files and saved in the specified output folder.

Usage

```

FF16_generate_stand_report(
  results,
  output_file = "FF16_report.html",
  overwrite = FALSE,
  target_ages = NA,
  input_file = system.file("reports", "FF16_report.Rmd", package = "plant"),
  quiet = TRUE
)

```

Arguments

results	results of running <code>run_scm(..., collect = TRUE)</code>
output_file	name of output file
overwrite	logical value to determine whether to overwrite existing report
target_ages	Patches ages at which to make plots
input_file	report script (.Rmd) file to build study report
quiet	An option to suppress printing during rendering from knitr, pandoc command line and others.

Value

html file of the rendered report located in the specified output folder.

FF16_hyperpar	<i>Hyperparameter function for FF16 physiological model</i>
---------------	---

Description

Hyperparameter function for FF16 physiological model

Usage

```
FF16_hyperpar(m, s, filter = TRUE)
```

Arguments

m	A matrix of trait values, as returned by <code>trait_matrix</code>
s	A strategy object
filter	A flag indicating whether to filter columns. If TRUE, any numbers that are within eps of the default strategy are not replaced.

FF16_Individual	<i>Create a FF16 Individual</i>
-----------------	---------------------------------

Description

Create a FF16 Individual

Usage

```
FF16_Individual(s = FF16_Strategy())
```

Arguments

s	A FF16_Strategy object
----------	--

Examples

```
p1 <- FF16_Individual()
p1$height
```

FF16_Parameters

Setup an a model system with default or specified parameters

Description

Setup an a model system with default or specified parameters. This function enables you initialize a model system. Use the model name to start different models.

Usage

```
FF16_Parameters(...)
```

```
K93_Parameters()
```

Arguments

... Arguments to be passed to the model constructor. These include

- *‘patch_area’: Area of idnividfual patch. Only relevant for stochastic model. Default is 1.0m2.
- *‘max_patch_lifetime’: The maximum time in years we want to simulate
- *‘strategies’: A list of stratgies to simulate. The default is an empty list.
- *‘strategy_default’: Values for the default startegy. The default values are those specified in the C++ code for the model.
- *‘node_schedule_times_default’: Default vector of times at which to introduce nodes. The default is chosen to have close spacing at the start of the simulation.
- *‘node_schedule_times’: A list with each element containing the vector of times we want to introduce nodes for each strategy. The default is an empty list.
- *‘ode_times’: A vector of patch ages we want the ode solver to stop at

Examples

```
p1 <- FF16_Parameters()
p2 <- FF16_Parameters(max_patch_lifetime = 10.0, patch_area = 1.0, strategies = list(FF16_Strategy()), st
```

FF16_Pars

FF16 strategy parameters

Description

Biological trait/parameters for the FF16 strategy. Held as the nested **pars** element of an [FF16_Strategy](#) (access as `s$pars$lma`).

Usage

```
FF16_Pars(..., values = list(...))
```

Arguments

`..., values` Values to initialise the struct with (either as variadic arguments, or as a list, but not both).

<code>FF16_Strategy</code>	<i>Strategy parameters</i>
----------------------------	----------------------------

Description

Strategy parameters that tune various aspects of the biological model.

Usage

```
FF16_Strategy(..., values = list(...))
```

Arguments

`..., values` Values to initialise the struct with (either as variadic arguments, or as a list, but not both).

<code>fitness_window</code>	<i>Weight arrivals by a window</i>
-----------------------------	------------------------------------

Description

Weight arrivals by a window of the run: the continuous patch's regime (plant issue #519). Where the default "meta-population" patch type weights a cohort by the density of patches at its age under a Weibull disturbance regime, this weights it by when it arrived in one long run: a cohort born inside `[start, end]` carries the tabulated weight at its birth date and a cohort born outside carries zero. The net reproduction ratio is then the weighted mean, over the cohorts the run introduced, of a cohort's lifetime output: the trapezium over birth dates of output times weight, divided by the trapezium of the weight on the same grid, so a uniform window is the exact mean however its edges sit against the introduction grid. Patch survival is one.

Usage

```
fitness_window(p, start, end, weight_x = numeric(0), weight_y = numeric(0))
```

Arguments

`p` A `Parameters` object.

`start, end` The window, in years of the run.

`weight_x, weight_y` Optional knots of the weight over the window; `weight_x` increasing, `weight_y` non-negative. Omit both for a uniform window.

Details

With no knots the window is uniform, each birth date weighted $1 / (\text{end} - \text{start})$. Knots give a piecewise-linear weight, held at the end knots' values beyond them: a germination opportunity built from the same rainfall series that drives the run, say.

The run must extend past **end** by the longest lifespan that matters, or the last arrivals' output is truncated; and the window should open after the stand's burn-in, which `export_patch_state` lets a run pay once.

The window is closed at both ends, because the trapezium over `[start, end]` needs a cohort at each. Invaders introduced only inside the window need one at **end** too: `node_schedule_times_periodic(start, end, by, include_end = TRUE)`.

Value

The modified `Parameters` object, with `patch_type = "fitness_window"`.

See Also

`node_schedule_times_periodic`

<code>fixed_point_fitness</code>	<i>Invader fitness at a fixed point</i>
----------------------------------	---

Description

Fitness of invaders at a resident's fixed point

Usage

```
fixed_point_fitness(
  scm,
  p,
  traits,
  hyperpar = param_hyperpar(p),
  lifetime = 3000,
  survival_tol = 1e-08
)
```

Arguments

<code>scm</code>	The resident's SCM, run to its fixed point.
<code>p</code>	The resident's <code>Parameters</code> ; each invader's strategy is built from them as <code>add_mutant</code> builds one.
<code>traits</code>	A trait matrix, one row per invader.
<code>hyperpar</code>	Hyperparameter function, as for <code>add_strategies</code> .

<code>lifetime</code>	Years each invader runs for. It has to outlive the cohort: the call fails unless, at the end, none of the cohort survives, or its survival since establishment is below <code>survival_tol</code> and it has stopped growing. A cohort still growing may be on its way to the canopy, where most of its output lies, however few of it are left. An invader that never establishes scores zero.
<code>survival_tol</code>	The survival since establishment below which a cohort that has stopped growing counts as dead.

Details

A resident at a fixed point makes one field, so an invader of it needs no recording of the resident and no replay: it stands in that one field, held, for its whole life. This runs each strategy in `traits` as one cohort born at time zero into the field `scm` stands in now, until the cohort is dead, and returns each one's lifetime offspring per seed. Every arrival at a fixed point sees the same field, so that one cohort's output is the invader's fitness. The resident's own strategy scores a resident cohort's lifetime output per seed, which is one at the resident's demographic equilibrium (it is not `scm$net_reproduction_ratios`, which a continuous patch's ever-growing node list does not make a per-cohort quantity).

The field is taken from the state `scm` has reached at the first call, and held for every later one; run the resident to its fixed point first, for instance with `run_until_converged` under constant drivers and the absorbing top layer (`Control$absorb_epsilon`). The invaders run together, as one community, since nothing they do moves a held field. `scm` is consumed: afterwards it holds the invaders, and the field is released, so a later run of it builds its own.

The introduction grid has to make the stand a fixed point. Between two introductions the field swings, and a grid that is too coarse makes the stand a periodic orbit whose field, held at an introduction, is the top of the swing: FF16 at its equilibrium birth rate swings by 1.8 % in held fitness at a cohort a year and 0.02 % at four. `SCM$run_to()` stops only at introduction instants, so to hold a field mid-interval, and check the grid, end the resident's run there with `max_patch_lifetime`.

Value

Lifetime offspring per seed, one per row of `traits`.

See Also

[run_until_converged](#)

<code>generate_strategy</code>	<i>Generate strategies from traits</i>
--------------------------------	--

Description

Convert a matrix of trait values into a list of **Strategy** objects, one per row, by applying the hyperparameter function and inserting the resulting parameters into a copy of the default strategy.

Usage

```
generate_strategy(p, traits, hyperpar = param_hyperpar(p), birth_rate = 1)

strategy_list(
  x,
  parameters,
  hyperpar = param_hyperpar(parameters),
  birth_rate_list = 1
)
```

Arguments

p	A Parameters object containing a default strategy to modify. Any hyperparameterisation included will be applied.
traits	Trait values as a <i>matrix</i> , with column names corresponding to traits (see trait_matrix); one row per strategy.
hyperpar	Hyperparameter function to use. By default links to the standard function for this strategy type. It translates ecological traits (e.g. lma, wood density) into the low-level strategy parameters, encoding the model's trade-offs.
birth_rate	Birth rate(s) for each row of traits : a scalar or vector (constant birth rate, set as strategy\$birth_rate_y) or a list with x , y control points (a varying birth rate, which also sets strategy\$birth_rate_x and is_variable_birth_rate = TRUE).
x	Deprecated (strategy_list); use traits .
parameters	Deprecated (strategy_list); use p .
birth_rate_list	Deprecated; use birth_rate .

gradient_control	<i>The Control entries a census gradient depends on.</i>
-------------------------	--

Description

The first four move the trajectory the gradient is taken along, so two gradients taken at different values of any of them are gradients of different functions. **gradient_curvature_floor** is here for a second reason: it moves no forward number at all and still decides which rows exist, by refusing a collar response the profit curvature is too small to support. **stand_gradient** records them and **stand_gradient_compare** refuses a pair that disagrees.

Usage

```
gradient_control(scm)
```

Arguments

scm A run SCM object for the TF24 or FF16 model, on the birth-date coordinate (`control(node_density_in_birth_date = TRUE)`).

Value

A named numeric vector of length five.

`grow_individual_to_size`

Grow individual to given size

Description

Grow an individual up to particular sizes.

Usage

```
grow_individual_to_size(
  individual,
  sizes,
  size_name,
  env,
  time_max = Inf,
  warn = TRUE,
  filter = FALSE
)
```

```
grow_individual_to_height(individual, heights, env, ...)
```

Arguments

individual An **Individual** object.

sizes A vector of sizes to grow the plant to (increasing in size).

size_name The name of the size variable within `individual$rates` (e.g., height).

env An **Environment** object.

time_max Time to run the ODE out for – only exists to prevent an infinite loop (say, on an unreachable size).

warn Warn if requesting a plant that is too large?

filter Filter individuals that are too large?

heights Heights (when using `grow_individual_to_height`)

... Additional parameters passed to `grow_individual_to_size`.

Value

A list with elements **time** (the time that a given size was reached), **state** (the *ode state* at these times, as a matrix) and **plant** a list of individuals grown to the appropriate size. Note that if only a single size is given, a list of length 1 is returned.

```
grow_individual_to_time
```

Grow a plant

Description

Grow a plant up for particular time lengths

Usage

```
grow_individual_to_time(individual, times, env)
```

Arguments

individual	An Individual object
times	A vector of times
env	An Environment object

Individual	<i>Individual object</i>
-------------------	--------------------------

Description

Individual object

Usage

```
Individual(T, E)
```

Arguments

T	String containing class of plant strategy to create;
E	String containing class of environmmnet to create;

integrate_over_size_distribution

Integrate over the size distribution for each species at each time point, to give totals of each variable Integrations are performed using trapezium integration

Description

Integrate over the size distribution for each species at each time point, to give totals of each variable Integrations are performed using trapezium integration

Usage

```
integrate_over_size_distribution(tidy_species_data)
```

Arguments

`tidy_species_data`

output of either ‘tidy_patch’ or ‘tidy_species’

Value

a tibble whose columns provide metrics on integrated totals for each variable for each species at each time

Internals

Extract Internals from plant object

Description

Internals class holding vectors of states, thier associated rates and auxiliary (aux) state which is calculated from the state through running `compute_rates`

Usage

```
Internals(s_size, a_size)
```

Arguments

`s_size` ???

`a_size` ???

`interpolate_to_heights`

Interpolate data on size distributions for each species to specific heights at every time point

Description

Interpolate data on size distributions for each species to specific heights at every time point

Usage

```
interpolate_to_heights(
  tidy_species_data,
  heights,
  method = "natural",
  min_log_density = -100
)
```

Arguments

`tidy_species_data` output of either ‘tidy_patch’ or ‘tidy_species’

`heights` heights to interpolate to

`method` Method for interpolation. For more info see help on stats::spline

`min_log_density` Set minimum possible value of log_density

Value

Returns a tibble of similar format to input, but with all outputs interpolated to specified heights.

`interpolate_to_times` *Interpolate data on size distributions for each species to specific timer points, using specified interpolation method*

Description

Interpolate data on size distributions for each species to specific timer points, using specified interpolation method

Usage

```
interpolate_to_times(tidy_species_data, times, method = "natural")
```

Arguments

<code>tidy_species_data</code>	output of either ‘tidy_patch’ or ‘tidy_species’
<code>times</code>	times to interpolate to
<code>method</code>	Method for interpolation. For more info see help on stats::spline

Value

Returns a tibble of similar format to input, but with all outputs interpolated to specified hieghts.

<code>K93_hyperpar</code>	<i>Hyperparameter function for K93 physiological model</i>
---------------------------	--

Description

Hyperparameter function for K93 physiological model

Usage

```
K93_hyperpar(m, s, filter = TRUE)
```

Arguments

<code>m</code>	A matrix of trait values, as returned by <code>trait_matrix</code>
<code>s</code>	A strategy object
<code>filter</code>	A flag indicating whether to filter columns. If TRUE, any numbers that are within eps of the default strategy are not replaced.

<code>K93_Individual</code>	<i>Create a K93 Individual or Node</i>
-----------------------------	--

Description

Create a K93 Individual or Node

Usage

```
K93_Individual(s = K93_Strategy())
```

Arguments

<code>s</code>	A K93_Strategy object
----------------	---------------------------------------

Examples

```
p1 <- K93_Individual()
p1$height
```

K93_Pars	<i>K93 strategy parameters</i>
----------	--------------------------------

Description

Biological trait/parameters for the K93 strategy. Held as the nested `pars` element of a [K93_Strategy](#) (access as `s$pars$b_0`).

Usage

```
K93_Pars(..., values = list(...))
```

Arguments

`..., values` Values to initialise the struct with (either as variadic arguments, or as a list, but not both).

K93_Strategy	<i>Strategy parameters</i>
--------------	----------------------------

Description

Strategy parameters that tune various aspects of the biological model.

Usage

```
K93_Strategy(..., values = list(...))
```

Arguments

`..., values` Values to initialise the struct with (either as variadic arguments, or as a list, but not both).

Leaf

Leaf model

Description

Argument names match phylloptim's own 'Leaf()' binding exactly (#634): the arguments are positional in C++, so a name here that disagrees with the slot it lands in is a silent error rather than a compile one.

Usage

```
Leaf(
  vcmx_25,
  stem_c,
  stem_P50,
  root_c,
  root_P50,
  TF24_beta2,
  jmax_25,
  a,
  curv_fact_elec_trans,
  curv_fact_colim,
  GSS_tol_abs,
  vulnerability_curve_ncontrol,
  ci_abs_tol,
  ci_niter,
  TF24_cost_scale
)
```

Arguments

<code>vcmx_25</code>	maximum carboxylation rate at 25 C ($\text{umol m}^{-2} \text{s}^{-1}$)
<code>stem_c</code>	shape parameter of the STEM hydraulic vulnerability curve (unitless)
<code>stem_P50</code>	STEM potential at 50 positive magnitude). phylloptim derives 'stem_b' and 'psi_crit' from it – do not pass the scale.
<code>root_c</code>	shape parameter of the ROOT hydraulic vulnerability curve (unitless)
<code>root_P50</code>	ROOT potential at 50 positive magnitude)
<code>TF24_beta2</code>	exponent for the effect of hydraulic risk (unitless)
<code>jmax_25</code>	maximum rate of electron transport at 25 C ($\text{umol m}^{-2} \text{s}^{-1}$)
<code>a</code>	quantum yield of photosynthetic electron transport (mol mol^{-1})
<code>curv_fact_elec_trans</code>	curvature factor for the light response curve (unitless)
<code>curv_fact_colim</code>	curvature factor for the colimited photosynthesis equation

GSS_tol_abs tolerance parameter for golden section search
vulnerability_curve_ncontrol
 number of points for pre-calculating vulnerability curve
ci_abs_tol absolute tolerance value for root-solving ci
ci_niter maximum number of iterations for root-solving ci
TF24_cost_scale
 cost parameter for the TF24 profit model ($\text{umol m}^{-2} \text{s}^{-1}$)

make_FF16_hyperpar *Hyperparameters for FF16 physiological model*

Description

Hyperparameters for FF16 physiological model

Usage

```
make_FF16_hyperpar(
  lma_0 = 0.1978791,
  B_kl1 = 0.4565855,
  B_kl2 = 1.71,
  rho_0 = 608,
  B_dI1 = 0.01,
  B_dI2 = 0,
  B_ks1 = 0.2,
  B_ks2 = 0,
  B_rs1 = 4012,
  B_rb1 = 2 * 4012,
  B_f1 = 3,
  narea = 0.00187,
  narea_0 = 0.00187,
  B_lf1 = 5120.738 * 0.00187 * 24 * 3600/1e+06,
  B_lf2 = 0.5,
  B_lf3 = 0.04,
  B_lf4 = 21000,
  B_lf5 = 1,
  latitude = 0
)
```

Arguments

lma_0	Central (mean) value for leaf mass per area [kg /m ²]
B_kl1	Rate of leaf turnover at lma_0 [/yr]
B_kl2	Scaling slope for phi in leaf turnover [dimensionless]
rho_0	Central (mean) value for wood density [kg /m ³]

B_dI1	Rate of instantaneous mortality at rho_0 [/yr]
B_dI2	Scaling slope for wood density in intrinsic mortality [dimensionless]
B_ks1	Rate of sapwood turnover at rho_0 [/yr]
B_ks2	Scaling slope for rho in sapwood turnover [dimensionless]
B_rs1	CO_2 respiration per unit sapwood volume [mol / yr / m3]
B_rb1	CO_2 respiration per unit sapwood volume [mol / yr / m3]
B_f1	Cost of seed accessories per unit seed mass [dimensionless]
narea	nitrogen per leaf area [kg / m2]
narea_0	central (mean) value for nitrogen per leaf area [kg / m2]
B_lf1	Potential CO_2 photosynthesis at average leaf nitrogen [mol / d / m2]
B_lf2	Curvature of leaf photosynthetic light response curve [dimensionless]
B_lf3	Quantum yield of leaf photosynthetic light response curve [dimensionless]
B_lf4	CO_2 respiration per unit leaf nitrogen [mol / yr / kg]
B_lf5	Scaling exponent for leaf nitrogen in maximum leaf photosynthesis [dimensionless]
latitude	degrees from equator (0-90), used in solar model [deg]

make_hyperpar

Hyperparameters for physiological model

Description

Set a suitable hyperparameter function for chosen physiological model

Usage

```
make_hyperpar(type)
```

```
param_hyperpar(parameters)
```

```
hyperpar(type)
```

Arguments

type Any strategy name as a string, e.g.: "FF16".

parameters A parameters object

make_initial_state	<i>Build an initial size distribution</i>
--------------------	---

Description

Build an initial patch state from a specified size distribution, for seeding a patch at age 0 with pre-existing plants instead of growing it from empty (the ecological motivation of [plant](#) issue #304). The returned object is consumed by [set_initial_state](#).

Usage

```
make_initial_state(
  p,
  heights,
  densities = NULL,
  log_densities = NULL,
  env = NULL,
  ctrl = control()
)
```

Arguments

p	A Parameters object.
heights	Per-species node heights: a numeric vector (single species) or a list of numeric vectors (one per species).
densities	Per-species node densities (same shape as heights). Supply this or log_densities .
log_densities	Per-species node log-densities (alternative to densities).
env	Environment object (defaults to the strategy's environment).
ctrl	Control object.

Details

Each species' initial nodes are described by their heights and densities; the remaining ODE state (mortality, fecundity, accumulated reproduction, heartwood, ...) starts at zero, all nodes are introduced at patch age 0, and the recruitment schedule continues for $t > 0$ (the seeded distribution replaces the $t = 0$ recruit). Pathologically large/dense initial conditions can produce non-finite densities; [run_scm](#) guards against this and errors with a suggestion to use more plausible inputs.

Value

A state list suitable for [set_initial_state](#).

See Also

[set_initial_state](#), [export_patch_state](#)

make_K93_hyperpar	<i>Hyperparameters for K93 physiological model</i>
-------------------	--

Description

Construct hyperparameter object for K93 physiological model

Usage

```
make_K93_hyperpar(
  b_0 = 0.059,
  b_1 = 0.012,
  b_2 = 0.00041,
  c_0 = 0.008,
  c_1 = 0.00044,
  d_0 = 0.00073,
  d_1 = 0.044,
  eta = 12,
  k_I = 0.01
)
```

Arguments

b_0	Growth intercept year-1
b_1	Growth asymptote year-1.(ln cm)-1
b_2	Growth suppression rate m2.cm-2.year-1
c_0	Mortality intercept year-1
c_1	Mortality suppression rate m2.cm-2.year-1
d_0	Recruitment rate (cm2.year-1)
d_1	Recruitment suppression rate (m2.cm-2)
eta	Crown shape parameter
k_I	Extinction coefficient used when estimating competitive effect

make_TF24_hyperpar	<i>Hyperparameters for TF24 physiological model</i>
--------------------	---

Description

Hyperparameters for TF24 physiological model

Usage

```

make_TF24_hyperpar(
  lma_0 = 0.1978791,
  B_kl1 = 0.4565855,
  B_kl2 = 1.71,
  rho_0 = 608,
  B_dI1 = 0.01,
  B_dI2 = 0,
  B_hks1 = 7.5,
  B_hks2 = 0,
  B_ks1 = 0.2,
  B_ks2 = 0,
  B_rs1 = 4012,
  B_rb1 = 2 * 4012,
  B_f1 = 3,
  a_lf1 = 0.535,
  B_lf1 = 0.009,
  B_lf2 = 0.004,
  B_lf3 = 8e-04,
  B_lf4 = 21000,
  B_lf5 = 40000,
  latitude = 0,
  B_Hv1 = 0.365915653419241,
  B_Hv2 = -0.2,
  B_c1 = 2.04,
  B_c2 = 0
)

make_TF24f_hyperpar(...)

```

Arguments

<code>lma_0</code>	Central (mean) value for leaf mass per area [kg /m ²]
<code>B_kl1</code>	Rate of leaf turnover at <code>lma_0</code> [/yr]
<code>B_kl2</code>	Scaling slope for <code>phi</code> in leaf turnover [dimensionless]
<code>rho_0</code>	Central (mean) value for wood density [kg /m ³]
<code>B_dI1</code>	Rate of instantaneous mortality at <code>rho_0</code> [/yr]
<code>B_dI2</code>	Scaling slope for wood density in intrinsic mortality [dimensionless]
<code>B_hks1</code>	Intercept for the <code>TF24_cost_scale ~ rho</code> relationship at <code>rho_0</code> [dimensionless]
<code>B_hks2</code>	Scaling slope for <code>rho</code> in the <code>TF24_cost_scale</code> relationship [dimensionless]
<code>B_ks1</code>	Rate of sapwood turnover at <code>rho_0</code> [/yr]
<code>B_ks2</code>	Scaling slope for <code>rho</code> in sapwood turnover [dimensionless]
<code>B_rs1</code>	CO ₂ respiration per unit sapwood volume [mol / yr / m ³]
<code>B_rb1</code>	CO ₂ respiration per unit sapwood volume [mol / yr / m ³]

B_f1	Cost of seed accessories per unit seed mass [dimensionless]
a_lf1	intercept for empirical relationship between narea and vcmax, lma (Dong et al. 2022)
B_lf1	Beta coefficient for empirical relationship between narea_ls ~ lma [g/m2] (Dong et al. 2022)
B_lf2	Beta coefficient for empirical relationship between narea_lp ~ vcmax [g/m2] (Dong et al. 2022)
B_lf3	Beta coefficient for empirical relationship between narea_lp ~ jmax [umol / m2 / s] (Dong et al. 2022)
B_lf4	CO_2 respiration per unit structural leaf nitrogen [mol / yr / kg]
B_lf5	CO_2 respiration per unit photosynthetic leaf nitrogen [mol / yr / kg]
latitude	degrees from equator (0-90), used in solar model [deg]
B_Hv1	p50 at terminal-segment K_s = 1 [-MPa]. The relation is keyed on a tip conductivity rather than a whole-stem one, so the intercept describes a different physical quantity than it did before the stem path integral existed; it was re-anchored to hold stem_P50 fixed at the default parameters. Whether the Ks-p50 relation should be keyed on tip conductivity at all is open – twig segments are what people usually measure – and is logged as a staged inconsistency in notes/plan-tf24-height-hydraulics.md.
B_Hv2	Scaling slope for K_s in p50 [dimensionless]
B_c1	Shape parameter stem_c of the vulnerability curve at stem_P50 = 0 [dimensionless]
B_c2	Scaling slope for stem_P50 in the vulnerability-curve shape parameter stem_c [dimensionless]
...	Arguments passed to make_TF24_hyperpar

model_version	<i>Scientific version of a physiological model</i>
---------------	--

Description

The scientific version increments only when a model's equations or default parameters change the simulation output for identical inputs. It is independent of the package **Version** (which also moves for refactors, performance and interface changes). Downstream tools such as **logpile** use it to decide when archived simulations must be re-run: reruns happen when the scientific version changes, not on every software release.

Usage

```
model_version(type)
```

```
model_id(type)
```

Arguments

type Any strategy name as a string, e.g.: "FF16".

Details

The version is returned as a string. It is usually a single integer ("1"), but may be compound for a model defined relative to another: TF24f is a fast approximation of TF24, so its version is "<TF24 version>.<approximation revision>" (e.g. "2.1"); the major component auto-tracks TF24, so a TF24 scientific change also invalidates TF24f.

The number is authored in C++ (the `scientific_version` / `approximation_revision` constants on each strategy class, see `inst/include/plant/models/*_strategy.h`) and read here through the compiled `strategy_scientific_version()`; there is no duplicated copy in R.

Value

For `model_version`, a version string (e.g. "1" or "2.1"). For `model_id`, a string of the form "FF16@v1" (model name and scientific version).

No_Disturbance	<i>No disturbance regime No_Disturbance_Regime control object</i>
-----------------------	---

Description

A disturbance-free regime for running individual patches

Usage

`No_Disturbance()`

node_schedule_times_default	<i>Generate Default Node Introduction Times</i>
------------------------------------	---

Description

Generate a suitable set of default node introduction times, biased so that introductions are more closely packed at the beginning of time, become increasingly spread out.

Usage

`node_schedule_times_default(max_time)`

Arguments

max_time Time to generate introduction times up to (the last introduction time will be at least `max_time`).

Details

The reason for the stepped distribution is to keep step sizes as series of doublings. Doing this limits the range of possible introduction times from an infinite set of possible values to a very limited subset of values (based on combinations of 1, 0.5, 0.25, 0.125 etc). The reason for doing this is to minimise the number of unique introduction times across all species. The ODE stepper needs to stop at each point where a node is introduced. If each species was selecting a bunch of points that was essentially unique (compared to those selected for all other species), the number of unique node introductions times could get very large, requiring more ODE steps.

Value

Vector of introduction times.

Author(s)

Rich FitzJohn, adapted from original C++ code by Daniel S. Falster.

node_schedule_times_periodic

Periodic node introduction times

Description

Node introduction times on a uniform grid, for a run that introduces a cohort periodically rather than on the front-loaded default schedule of `node_schedule_times_default`: the continuous patch (plant issue #519). Times run from `start` in steps of `by`. By default they stop short of `end`, so consecutive spans tile without repeating a time: `periodic(0, 100, 1)` and `periodic(100, 200, 1)` meet at 100 once. A cohort introduced at a run's `max_patch_lifetime` would never be integrated either.

Usage

```
node_schedule_times_periodic(start, end, by, include_end = FALSE)
```

Arguments

<code>start</code>	First introduction time.
<code>end</code>	The span's end.
<code>by</code>	Spacing between introductions, in years.
<code>include_end</code>	Whether a time landing on <code>end</code> is kept.

Details

A schedule for a `fitness_window` over `[start, end]` is the other case: the window weighs a cohort at each end, so invaders introduced only inside it need `include_end = TRUE` to have the cohort at `end` the resident has. Without it their window average misses the last arrival (2 % low for a copy of the resident over ten).

Value

A numeric vector of introduction times, for one species of `p$node_schedule_times`.

See Also

`fitness_window`

NodeSchedule	<i>Schedule of node introduction times</i>
--------------	--

Description

Schedule of node introduction times

Usage

`NodeSchedule(n_species)`

Arguments

`n_species` number of species

OdeControl	<i>ODE Control parameters</i>
------------	-------------------------------

Description

Control parameters for the ode system

Usage

`OdeControl(..., values = list(...))`

Arguments

`..., values` Values to initialise the struct with (either as variadic arguments, or as a list, but not both).

```
optimise_individual_rate_at_size_by_trait
```

The function ‘optimise_individual_rate_at_height_by_trait’ and ‘optimise_individual_rate_at_size_by_trait’ solve for the maximum of some rate (e.g. growth rate) at a specified height within the interval of the bounds of a given trait

Description

The function ‘optimise_individual_rate_at_height_by_trait’ and ‘optimise_individual_rate_at_size_by_trait’ solve for the maximum of some rate (e.g. growth rate) at a specified height within the interval of the bounds of a given trait

Usage

```
optimise_individual_rate_at_size_by_trait(
  type = "FF16",
  bounds,
  log_scale = TRUE,
  tol = 0.001,
  size = 1,
  size_name = "height",
  rate = size_name,
  params = scm_base_parameters(type),
  env = Environment(type),
  hyperpars = hyperpar(type),
  set_state_directly = FALSE
)

optimise_individual_rate_at_height_by_trait(..., height = 1)
```

Arguments

type	The type of model to use (e.g. "FF16"). Defaults to "FF16"
bounds	A vector giving the lower and upper bounds of the trait
log_scale	Should the trait be optimised on a log scale? Defaults to TRUE
tol	The tolerance for the optimisation
size	The size of the individual to optimise the rate at
size_name	The name of the size variable specified by size
rate	The name of the rate to optimise. Defaults to size_name
params	The parameters of the model
env	The environment of the model
hyperpars	The hyperparameter function of the model

<code>set_state_directly</code>	If TRUE, set the state directly to the size, otherwise grows the plant to that size. Defaults to FALSE
<code>...</code>	Additional parameters passed to <code>optimise_individual_rate_at_size_by_trait</code>
<code>height</code>	Height at which grow is optimised. Defaults to 1

Author(s)

Isaac Towers, Daniel Falster and Andrew O'Reilly-Nugent

<code>plant_log_console</code>	<i>Activate logging</i>
--------------------------------	-------------------------

Description

Activate logging

Usage

```
plant_log_console(file_name = "console", colour = TRUE, threshold = "INFO")
```

Arguments

<code>file_name</code>	File to save output (default = "console")
<code>colour</code>	Use colour in console output?
<code>threshold</code>	Minimum log level to emit: "DEBUG", "INFO", etc.

Details

By default plant prints little information about its progress. This can be modified by enabling logging. Log entries include a timestamp and, where applicable, a routine label indicating which part of the model is running.

"Schedule" events (splitting) are sent to the DEBUG stream, everything else is sent to INFO.

```
plot_size_distribution
```

Title

Description

Title

Usage

```
plot_size_distribution(data_species)
```

Arguments

```
data_species  ??
```

```
pulse_series_poisson
```

Series of resource pulses

Description

Series of resource pulses

Usage

```
pulse_series_poisson(from, to, rate, mean_amount)
```

```
pulse_series_regular(from, to, interval, amount)
```

```
pulse_series_record(
  time,
  amount,
  from = 0,
  to = from + period,
  period = ceiling(max(time))
)
```

Arguments

<code>from, to</code>	The span the series covers: pulses fall in <code>(from, to]</code> .
<code>rate</code>	Mean number of pulses per year.
<code>mean_amount</code>	Mean amount per pulse, in the units the pulse constructor takes (TF24: metres of water).
<code>interval</code>	Years between pulses.
<code>amount</code>	Amount per pulse (<code>pulse_series_regular</code>), or per recorded pulse (<code>pulse_series_record</code>).

<code>time</code>	Times of the recorded pulses, in years from the start of the record, in <code>(0, period]</code> and increasing: day of record / 365.25, say.
<code>period</code>	The record's length in years, the spacing at which it repeats. Defaults to its last time rounded up to a whole year.

Details

Three ways to build a series of pulses for a run, each returning the pulses' times and amounts as a data frame, sorted by time, to hand to a model's pulse constructor: for TF24, `rainfall_pulse(s$time, s$amount)`.

`pulse_series_poisson()` draws a stationary marked Poisson process: pulses arrive at `rate` per year, and each one's amount is exponential with mean `mean_amount`, so the long-run supply is `rate * mean_amount` per year. It draws from R's random number stream, so `set.seed()` fixes the series, and one realisation shared by every run that is compared is what makes a comparison of fitness a comparison of traits (common random numbers). It draws the count, then the times, then the amounts, so a series over a longer span does not contain a shorter one.

`pulse_series_regular()` puts a pulse of `amount` every `interval` years.

`pulse_series_record()` takes an observed record – each wet day a pulse, say, with times in years – and, given to beyond the record's end, repeats it end to end to cover a longer run: a 100-year record behind a burn-in and a window of several centuries. Repeating a record makes the series periodic, which a window shorter than the record does not see.

Value

A data frame with columns `time` and `amount`.

See Also

[rainfall_pulse](#), [events](#)

Examples

```
set.seed(1)
s <- pulse_series_poisson(0, 100, rate = 20, mean_amount = 0.014)
nrow(s)                # about 2000 pulses
sum(s$amount) / 100    # about 0.28 m of water a year
p <- add_strategies(scm_base_parameters("TF24"), trait_matrix(0.0825, "lma"))
ev <- events(events_default(p), rainfall_pulse(s$time, s$amount))
```

QK

Gauss-Kronrod Quadrature

Description

Gauss-Kronrod Quadrature

Usage

```
QK(rule)
```

Arguments

rule Degree of the rule; valid sizes are 15, 21, 31, 41, 51, 61.

rainfall_pulse	<i>A rainfall pulse for TF24.</i>
-----------------------	-----------------------------------

Description

TF24's resources are its soil layers, so a rainfall pulse is a [resource_pulse](#) of water into one of them. This is the motivating case for the event mechanism (#522): TF24 otherwise takes rainfall as a smooth cubic spline, which smears out exactly the variability that matters — large pulses.

Usage

```
rainfall_pulse(time, depth, layer = 1, env = NULL)
```

Arguments

time Event time(s), in years of patch age.

depth Depth of water delivered, in m — so 13 mm is 0.013.

layer Soil layer to deliver it to. The default, 1, is the surface, i.e. rain. A deeper layer is irrigation, or a water table rising into the profile.

env Optionally, the **Environment** the run will use. When given, `rainfall_pulse()` warns if that environment still carries a non-zero continuous `rainfall` driver — pulses *add to* the continuous forcing rather than replacing it, so supplying both silently double-counts the water. Set the driver to zero to model rainfall as pulses alone.

Details

The surface layer accepts what its free capacity allows and sheds the rest, which is not a corner case: a realistic dryland event (~13 mm) already exceeds a moderately wet layer's capacity. `scm$event_log` reports how much of each pulse landed and how much was shed.

Value

Event rows, to be combined with [events](#).

See Also

[events](#), [resource_pulse](#)

Examples

```
p <- scm_base_parameters("TF24")
p <- add_strategies(p, trait_matrix(1, "lma"))
ev <- events(
  events_default(p),
  rainfall_pulse(time = c(1.5, 3.2), depth = c(0.013, 0.050))
)
```

```
resource_compensation_point
```

Whole plant light compensation point

Description

Compute the whole plant light compensation point for a single plant: the light level at which whole-plant carbon gain is zero.

Usage

```
resource_compensation_point(p, ...)
```

Arguments

<code>p</code>	An <code>Individual</code> , with strategy, height, etc set.
<code>...</code>	Additional arguments that are ignored

Author(s)

Rich FitzJohn

```
run_plant_benchmarks
```

Check performance on current system using package bench

Description

Check performance on current system using package bench

Usage

```
run_plant_benchmarks(
  strategy_types = list(FF16 = FF16_Strategy),
  iterations = 1
)
```

Arguments

- strategy_types** A list of name strategy types to be tests
- iterations** The number of iterations to be run

Details

For branch-to-branch performance comparisons, run ‘make’ before ‘devtools::load_all(quiet = TRUE)’ so compiled code is rebuilt with the package’s intended optimization flags.

Value

A dataframe of results

<code>run_scm</code>	<i>Run SCM</i>
----------------------	----------------

Description

Run the SCM.

Usage

```
run_scm(
  p,
  env = NULL,
  ctrl = control(),
  refine_schedule = FALSE,
  collect = FALSE,
  record_trajectory = FALSE,
  events = NULL,
  run = TRUE
)
```

Arguments

- p** Parameters object
- env** Environment object (defaults to the strategy’s environment)
- ctrl** Control object
- refine_schedule** Should the node-introduction schedule be adaptively refined before/while running (using `schedule_eps` and `schedule_nsteps` from `ctrl`)? Refinement records the ODE schedule its final run took into `p$ode_times` and `p$ode_step_sizes`, so a later run of those parameters replays it exactly rather than choosing its own steps again.
- collect** Should tidied results be collected at every step and returned (instead of the SCM object)?

record_trajectory	Should the run keep the state at every accepted step? A gradient sweeps those states and cannot recover them from a finished run, so a run that did not keep them is repeated – one whole forward integration. Asked for here because the flag has to be set before the run that fills it, and this function is where that run happens. With <code>refine_schedule = TRUE</code> the refinement’s own runs do not keep them; only the final run does.
events	An <code>events</code> object giving the discrete events to apply during the run — rainfall pulses, harvest, and node introductions themselves. When <code>NULL</code> (the default) the schedule is taken from <code>p\$node_schedule_times</code> , i.e. introductions only.
run	Run the SCM? <code>FALSE</code> returns it built and unrun, for a caller that drives it leg by leg: <code>run_until_converged</code> and <code>SCM\$run_to()</code> . It cannot be combined with <code>collect</code> or <code>refine_schedule</code> , which are about the run.

Details

The node-introduction schedule can be adaptively refined in C++ by setting `refine_schedule = TRUE` (this replaces the former `build_schedule` function). Setting `collect = TRUE` returns tidied output collected at every ODE step (replacing the former `run_scm_collect`); otherwise the `SCM` object itself is returned for interrogation.

Value

When `collect = FALSE`, an `SCM` object. When `collect = TRUE`, a list of tidied patch output with `offspring_production`, `net_reproduction_ratios` and the (possibly refined) parameters `p`.

Author(s)

Rich FitzJohn

`run_stochastic_collect`

Run a stochastic patch, Collecting Output

Description

Run a stochastic simulation of a patch, given a `Parameters`

Usage

```
run_stochastic_collect(p, env = NULL, ctrl = Control(), random_schedule = TRUE)
```

Arguments

<code>p</code>	A FF16_Parameters object
<code>env</code>	Environment object
<code>ctrl</code>	Control object
<code>random_schedule</code>	setting to TRUE causes algorithm to generate a random schedule based on offspring arrival and area.

Details

This one might need to be made differently so that different schedules can be added easily. Not sure if this is how we will generally want to do this. Consider this function liable to change.

Author(s)

Rich FitzJohn

`run_until_converged` *Run until the stand's block means settle*

Description

Run a patch until its stand has settled, judged on block means of its snapshots: the burn-in of a continuous patch (plant issue #519), paid once and then exported with [export_patch_state](#).

Usage

```
run_until_converged(
  scm,
  block = 20,
  tol = 0.02,
  t_max = scm$parameters$max_patch_lifetime,
  statistics = c("density", "height_max", "environment"),
  snapshot_interval = 1,
  environment_entries = NULL,
  window = 3,
  tol_abs = 0
)
```

Arguments

<code>scm</code>	An SCM built and not yet run to the end, as <code>run_scm(p, ..., run = FALSE)</code> returns one, whose schedule reaches <code>t_max</code> . Its <code>snapshot_interval</code> is set to <code>snapshot_interval</code> if it is zero.
------------------	---

block	Years per block.
tol	Relative range of the last window block means below which a statistic counts as settled.
t_max	Time to give up at; defaults to the run's max_patch_lifetime .
statistics	Which of " density " (the species' carried density summed), " height_max " (the tallest cohort with density over the species), " n_live " (cohorts with density, summed) and " environment " (entries of the environment's state) to judge. The default leaves out " height_max " when the absorbing top layer is on, since the layer pins it at the absorbing height whatever the stand does.
snapshot_interval	Years between snapshots when the SCM has none set.
environment_entries	Which entries of the environment's state the " environment " statistic reads. NULL (the default) reads every entry but the cumulative flux accumulators (TF24's last five), which grow without bound, so that their relative change shrinks with time whether or not anything has settled.
window	Number of consecutive block means that must agree.
tol_abs	Absolute range below which a statistic counts as settled whatever its size: for one that settles at or near zero. Zero, the default, judges every statistic relatively.

Details

The run advances one block at a time with `scm$run_to()`. After each block the mean over that block's snapshots is taken of the chosen statistics, and the run is converged once the last **window** block means of every statistic agree: their range is below **tol** times the largest of them in magnitude, or below **tol_abs**. Judging the range over a window rather than the change between neighbouring blocks is what lets a steady drift accumulate into the test instead of passing block by block. It stops regardless at **t_max**, or when the schedule is consumed, and says which it was; a last block shorter than **block** runs the patch on to **t_max** without being tested. Under a stochastic driver the block has to span several of the driver's correlation times, or the test measures the driver and not the stand. A statistic that is NA in a block – the tallest cohort of a stand with none alive – cannot settle, and the function warns which.

A settled stand statistic is not a settled fitness. A drift slower than **tol** per **window * block** years passes, and an invader's fitness can be far more sensitive to the stand than the statistic is: a TF24 stand under constant rain at its equilibrium birth rate was still gaining 0.1 % density a century at year 5000, and the resident's fitness read off it at years 300 and 500 differed sixfold. Check the quantity you will use at two times.

Value

A list: **converged**, **time** (where the run stands), and **trace**, a data frame with one row per block of the block means and the relative range of each over the window ending there (**rel_***, NA until **window** blocks have run).

See Also

[tidy_snapshots](#), [export_patch_state](#)

<code>scenario_eval</code>	<i>Scenario evaluation framework for the TF24 / TF24f hydraulic models.</i>
----------------------------	---

Description

These functions turn a table of qualitatively-described model scenarios (e.g. `inst/scenarios/model_scenarios_h`, added in PR #555) into concrete TF24 parameterisations, run each through the SCM, classify the run as a success or a failure, and score the observed outcome against the expected outcome recorded in the table. The result is a provenance-stamped scorecard that can be re-run across branches as a gateway check: many scenarios are expected to fail on the current model and become targets to fix as new features (e.g. NSC storage, #554) land.

Usage

```
read_scenario_table(path = scenario_file("model_scenarios_hydraulic.csv"))

read_scenario_mapping(path = scenario_file("scenario_mapping.csv"))

scenario_to_config(row, mapping)

scenario_control()

build_scenario(
  config,
  max_patch_lifetime = 100,
  ctrl = scenario_control(),
  birth_rate = 1
)

classify_scm_run(p, env, ctrl = scenario_control())

evaluate_scenario(
  row,
  mapping,
  ctrl = scenario_control(),
  max_patch_lifetime = 100
)

run_scenarios(
  scenarios = read_scenario_table(),
  mapping = read_scenario_mapping(),
  ctrl = scenario_control(),
  max_patch_lifetime = 100,
```

```

    workers = 1L,
    cache = NULL
  )

scenario_model_fingerprint()

scenario_run_metadata()

scenario_summary(scorecard)

scenario_generate_report(
  scorecard,
  output_file = "scenario_scorecard.html",
  input_file = system.file("reports", "scenario_scorecard.Rmd", package = "plant"),
  overwrite = FALSE,
  quiet = TRUE
)

```

Arguments

<code>path</code>	Path to the CSV file.
<code>row</code>	A one-row data frame / tibble from <code>read_scenario_table</code> .
<code>mapping</code>	A mapping tibble from <code>read_scenario_mapping</code> .
<code>config</code>	A config list from <code>scenario_to_config</code> .
<code>max_patch_lifetime</code>	Patch lifetime (years) for the SCM run.
<code>ctrl</code>	A <code>Control</code> object. Defaults to <code>scenario_control()</code> , which integrates in birth date rather than height – see there for why that is the right coordinate for TF24.
<code>birth_rate</code>	Birth rate passed to <code>add_strategies</code> .
<code>p</code>	A built <code>Parameters</code> object (e.g. <code>build_scenario()\$p</code>).
<code>env</code>	An <code>Environment</code> object (e.g. <code>build_scenario()\$env</code>).
<code>scenarios</code>	A scenario tibble from <code>read_scenario_table</code> .
<code>workers</code>	Number of parallel workers. <code>> 1</code> uses fork-based parallelism (<code>parallel::mclapply</code>). Forking is deliberate: it inherits the currently-loaded namespace and compiled library, so it works when <code>plant</code> is loaded for development via <code>pkgload::load_all</code> / <code>devtools::load_all</code> . A <code>PSOCK</code> / <code>future::multisession</code> cluster would spawn fresh R sessions that see only the <i>installed</i> package, not the dev build, and would silently run stale (or missing) code — so it is not used here. Forking is unavailable on Windows, where the run falls back to sequential regardless of <code>workers</code> .
<code>cache</code>	Optional path to an <code>.rds</code> cache. When supplied, each scenario's result is keyed by a content hash of its resolved config, <code>max_patch_lifetime</code> and the model fingerprint (<code>scenario_model_fingerprint</code>); a scenario is rerun only when that key changes. The key covers every input that affects a result: the resolved config, <code>max_patch_lifetime</code> , all <code>ctrl</code> settings,

and the model fingerprint. So a model recompile / R-source edit reruns everything (the fingerprint moves), a `ctrl` change reruns everything, and editing one mapping cell reruns only the scenarios it touches. The cache deliberately errs toward rerunning.

<code>scorecard</code>	A scorecard tibble from <code>run_scenarios</code> .
<code>output_file</code>	Output HTML path for the rendered report.
<code>input_file</code>	The report template (<code>.Rmd</code>).
<code>overwrite</code>	Overwrite an existing <code>output_file</code> ?
<code>quiet</code>	Passed to <code>rmarkdown::render</code> .

Details

The qualitative-to-quantitative translation is data-driven: every `High/Low/descriptor` cell is looked up in an editable mapping table (`inst/scenarios/scenario_mapping.csv`) so the biology can be recalibrated without touching code.

Value

`read_scenario_table` returns a tibble of scenarios with an added `scenario_id` and `is_duplicate` flag; the raw descriptor columns are kept verbatim.

`scenario_to_config` returns a list with `traits` (named numeric), `env` (named list of Environment fields), `driver` (named list of extrinsic-driver settings) and `expected` ("failure" or "success").

`scenario_control` returns the `Control` the gateway runs under: `control()` with `node_density_in_birth_date = TRUE`.

Every scenario here is TF24, and TF24 is the model the two density coordinates genuinely disagree on (#590). The transport equation's compression term is the total derivative of growth along a cohort's own trajectory, which equals dg/dh only when growth is a function of size; TF24's reserve gate (#517) breaks that, so in height coordinates the solver carries an accurate derivative of a quantity the plant never experiences. The tell is that refining the node schedule does not close the height-vs-birth-date gap the way it does for FF16 and K93, where growth *is* size-only and the two coordinates converge at ~2nd order.

Measured on this gateway at `max_patch_lifetime = 100`, the coordinate change raises R_0 on every scenario, by 2.4x (S02) to 47x (S07), and moves S01 across $R_0 = 1$. So the choice is not cosmetic, and running the TF24 gateway in the package default coordinate would score the model on a compression term that is wrong for it.

`build_scenario` returns a list with the `Parameters` (`p`), configured `Environment` (`env`) and `Control` (`ctrl`) ready for `run_scm`.

`classify_scm_run` returns a list describing the run: `status` ("success"/"failure"), `offspring_production`, `persists`, `finite`, `error_message`, `warnings` and `run_seconds`. A run is a success when it completes with finite, positive total offspring production; any thrown error or non-finite output is a failure. The C++ layer already fails fast on non-finite state, so classification does not depend on error wording. `persists` is the separate, stricter ecological question — whether the strategy replaces itself, $R_0 \geq 1$ — which a run can fail while still counting as a numerical success.

`evaluate_scenario` returns a one-row scorecard tibble.

`run_scenarios` returns a scorecard tibble (one row per scenario) with a "metadata" attribute recording provenance. A crash in one scenario never aborts the batch. Scenario runs are deterministic (no RNG), so parallel and sequential runs produce identical scorecards.

`scenario_model_fingerprint` returns a hash string that changes whenever the model that produces scenario results could change: the package version, the compiled shared library, and all package R source plus the scenario data files. It is intentionally broad — the cache should rerun rather than risk a stale result.

`scenario_run_metadata` returns a list of provenance fields (git commit / branch / dirty flag, package version, R version, platform, timestamp) so a scorecard records the exact build it came from.

`scenario_summary` returns a one-row tibble reporting **two separate axes** (#572), because the gateway conflated them and so returned no signal:

Numerical viability — did the model run? `n`, `n_ran`, `n_crashed`, `viability_rate`.

This is what the scenario CSV's "Model failure" means (#549, #550) and what the hydraulic/NSC work targets.

Ecological persistence — does the strategy replace itself? `n_persists`, `persistence_rate`, judged at $R0 \geq 1$ (see `persists_at`). Now that the crashes are fixed this is the axis the gateway is actually useful for.

`n_match` / `match_rate` and the expected-failure vs expected-success breakdown are still reported, but as **agreement with the CSV's crash predictions**, not as a headline quality score: with the crashes fixed, the match rate mostly measures how well those predictions have aged. Read `n_persists` against `n` — a scorecard where every run "succeeds" but almost none persists is reporting that the model no longer crashes, not that the strategies live.

`scenario_generate_report` renders the scorecard report and returns the output path (invisibly).

`scm_base_parameters` *Basic default parameters for a given strategy*

Description

Basic default settings for a given strategy, environment only really used for templating initially and will be overloaded later by passing an environment to the SCM API (suggesting perhaps the template could be removed).

Usage

```
scm_base_parameters(type = NA, env = environment_type(type))
```

Arguments

<code>type</code>	Any strategy name as a string, e.g.: "FF16".
<code>env</code>	And environment object

Author(s)

Rich FitzJohn

seq_log	<i>Sequence in log space</i>
---------	------------------------------

Description

Sequence in log space

Usage`seq_log(from, to, length.out)``seq_log_range(r, length.out)``seq_range(r, length.out)`**Arguments**

<code>from</code>	Starting point
<code>to</code>	Ending point
<code>length.out</code>	Number of points to generate
<code>r</code>	range (i.e., <code>c(from, to)</code>)

Details

Unlike the billions of options for `seq`, only `length.out` is supported here, and both `from` and `to` must be provided. For completeness, `seq_range` generates a range in non-log space.

Author(s)

Rich FitzJohn

set_initial_state	<i>Seed Parameters with an initial patch state</i>
-------------------	--

Description

Write an exported patch state (from `export_patch_state`) into a `Parameters` object so that the next `run_scm` starts from that state instead of an empty patch. The state is carried on the `Parameters` object (rather than passed separately) so the run stays self-describing and reproducible, and so the seeding survives the reset at the start of every run / schedule refinement.

Usage

```
set_initial_state(p, state)
```

Arguments

p A **Parameters** object (its strategies must match the exported state).
state An exported state list from `export_patch_state`.

Details

Each seeded cohort's patch density and survival at birth are read from the resuming parameters' own regime at its birth date, not taken from the export: a cohort seeded from a burn-in into a `fitness_window` run weighs what the window gives it. Under the regime that exported it they are the same numbers. The exported `patch_density` and `pr_patch_survival` are checked only for their lengths.

Value

The modified **Parameters** object. The state's `events` are not carried on it: pass them to `run_scm` as `events = state$events`.

See Also

`export_patch_state`, `run_scm`

<code>stand_census</code>	<i>Metrics of a solved stand.</i>
---------------------------	-----------------------------------

Description

`stand_census` reports the census metrics: the trapezium integral of $n_k\psi(\text{state}_k)$ over the size distribution, for each per-plant quantity ψ , summed over the species. The grid is the coordinate the density is carried in – the birth date, or the height – and the inflow boundary node closes it, so the interval between it and the smallest cohort is in the sum.

Usage

```
stand_census(scm)

stand_fitness(scm)
```

Arguments

scm A run SCM object for the TF24 or FF16 model, on the birth-date coordinate (`control(node_density_in_birth_date = TRUE)`).

Details

`stand_fitness` reports, for a model that declares its fitness differentiable (FF16), each species' net reproduction ratio and offspring production, named "<species>.net_reproduction_ratio" and "<species>.offspring_production". They are the same numbers as `scm$net_reproduction_ratios` and `scm$offspring_production`, reported under the names `stand_gradient` differentiates them by.

Value

A named numeric vector, one entry per metric.

`stand_census_state_adjoint`

Sensitivity of the stand metrics to the ODE state at the current time.

Description

One row per metric – the census metrics, then for a model that declares its fitness differentiable each species' net reproduction ratio and offspring production, named as `stand_gradient` names them – and one column per ODE state entry, in the order `scm$patch$ode_state` writes. This is what the reverse pass is seeded with. The cohort-height columns carry the trapezium weights as well as the integrand only where the height is the coordinate integrated over; on the birth-date coordinate the weights are constants.

Usage

```
stand_census_state_adjoint(scm)
```

Arguments

<code>scm</code>	A run SCM object for the TF24 or FF16 model, on the birth-date coordinate (<code>control(node_density_in_birth_date = TRUE)</code>).
------------------	--

Value

A numeric matrix, metrics by ODE state entries.

<code>stand_gradient</code>	<i>Gradient of a stand's metrics with respect to traits.</i>
-----------------------------	--

Description

Doubles in and doubles out: the active scalar is created and destroyed inside one call and never crosses into R.

Usage

```
stand_gradient(scm, metrics = NULL, traits = NULL, hyperpar = NULL)
```

Arguments

<code>scm</code>	A run SCM object for the TF24 or FF16 model, on the birth-date coordinate (<code>control(node_density_in_birth_date = TRUE)</code>).
<code>metrics</code>	Metric names to differentiate; defaults to every metric the model declares: the census metrics (<code>stand_census</code>) and, for a model that declares its fitness differentiable, each species' " <code><species>.net_reproduction_ratio</code> " and " <code><species>.offspring_production</code> " (<code>stand_fitness</code>). Each metric costs one sweep of the recorded trajectory, so name the ones wanted.
<code>traits</code>	Column names to differentiate with respect to; defaults to every differentiable parameter every species declares. A column is named for its species and its parameter, as " <code>1.lma</code> ". With <code>hyperpar</code> , these name the traits to differentiate through it, and must be given.
<code>hyperpar</code>	NULL, or the hyperparameter function the stand was built with, as <code>TF24_hyperpar</code> or <code>FF16_hyperpar</code> . See the section on traits and parameters.

Details

Two terms, because a census reads the traits as well as the state. The reverse pass is seeded on the states the census reads at the end of the run and runs back over the steps the adaptive pass resolved; added to it is the census's own reading of the traits at that state, which is not a sensitivity of the state and so no sweep produces it.

Value

A list with `value`, the metrics at the end of the run; `gradient`, a metrics-by-traits matrix; `refusal`, one entry per metric holding the reason and where it was found, or NULL where the metric answered; and `control`, the entries the gradient was taken at. With `hyperpar`, also `jacobian`: for each trait, the derivative of every parameter the hyperparameter function derives from it.

Traits and parameters

****Without hyperpar, each column is a PARTIAL derivative with every other parameter held fixed**, including those a hyperparameter function derives from the trait. TF24_hyperpar sets `k_l`, `r_l` and `nmass_l` from `lma`, so `gradient[, "1.lma"]` is not the derivative a trait fit wants: on a 3-year stand it was 2.4 to 4 times a finite difference of `lma` through `add_strategies()`.**

Pass `hyperpar` for the total derivative with respect to the traits as a trait matrix gives them. Each trait's column is its own partial plus every derived parameter's partial times that parameter's derivative with respect to the trait. The hyperparameter function is closed-form, so that derivative is taken by central difference of the function itself, costing no model run. Pass the function the stand was built with: a run does not record it.

A refused metric's whole row is `NaN`: a sum has no defined value with an undefined term, so refusal is metric-level and carries no localisation within a metric. Every other number is one the sweep computed, an exact zero included — a parameter no gradient exists for cannot be asked for, so it never reaches a column. `psi_crit` and `root_psi_crit` are two such: each is derived from its curve's (P50, c) pair, which `phylloptim` derives for itself, so setting one reaches no equation and asking for its gradient errors rather than returning a zero.

An invader against a recorded resident

Set `scm$record_trajectory <- TRUE` before `scm$run_mutant(p)` and the invasion replay keeps its states; a gradient taken then is of the invaders' metrics with the canopy held at the recorded resident — the field is exogenous to the invader and carries no derivative — which is the rare-mutant derivative. Several invaders in `p` share one replay and each has its own columns, but each metric is still one sweep, so a set of invaders costs about what running them one at a time does. An invader of a resumed resident starts at the resume time, as `run_mutant` requires. A sweep of an invasion that kept no states is refused rather than re-run.

The birth rate

For a species run at a constant birth rate the column "`<species>.birth_rate`" is the derivative with respect to that rate, which reaches the density every cohort is born at and the offspring integral. A species with a variable birth rate has no such column.

A single entry can dominate a second moment

A gradient taken where the run crosses an unresolved event — a cohort reaching `hmat` between two steps, say — is a derivative of the trajectory's step placement rather than of the model, and it can be two orders larger than its neighbours. One measured cell was 211 times the median of its grid.

****Do not average such a grid, and do not form a covariance from one.**** A single outlying cell dominates any second moment, which is the first thing an active-subspace or sensitivity workflow computes, so the reduction inherits the step placement instead of the ecology. Treat a gradient far outside its neighbourhood as a diagnostic: nudge the trait a fraction of a percent and take the answer that is stable, or resolve the event by forcing a step at the crossing.

stand_gradient_compare
Compare two stand gradients.

Description

Two gradients are comparable only if they were taken at the same **Control**: four of the five entries change the trajectory and so change the function being differentiated, and the fifth, **gradient_curvature_floor**, decides which rows exist. See [gradient_control](#).

Usage

```
stand_gradient_compare(a, b)
```

Arguments

a, b Results of **stand_gradient**.

Value

The element-wise difference **a\$gradient - b\$gradient**, over the metrics and traits both carry.

stand_gradient_refused
Which of a gradient's metrics were refused.

Description

Refusal is metric-level: a sum has no defined value with an undefined term, so a refused metric's whole row is NaN and one reason serves it.

Usage

```
stand_gradient_refused(g)
```

Arguments

g A result of **stand_gradient**.

Value

A named logical, one entry per metric.

TF24_Environment	Create an TF24_Environment object.
------------------	------------------------------------

Description

This environment is used for the TF24 strategy, which includes soil moisture and hydraulic conductivity parameters.

Usage

```
TF24_Environment()
```

TF24_generate_stand_report	Generates a report on stand grown with TF24 strategy
----------------------------	--

Description

Builds a detailed report on stand grown with TF24 strategy, based on the template Rmd file provided. The reports are rendered as html files and saved in the specified output folder.

Usage

```
TF24_generate_stand_report(
  results,
  output_file = "TF24_report.html",
  overwrite = FALSE,
  target_ages = NA,
  input_file = system.file("reports", "TF24_report.Rmd", package = "plant"),
  quiet = TRUE
)
```

Arguments

results	results of running run_scm(..., collect = TRUE)
output_file	name of output file
overwrite	logical value to determine whether to overwrite existing report
target_ages	Patches ages at which to make plots
input_file	report script (.Rmd) file to build study report
quiet	An option to suppress printing during rendering from knitr, pandoc command line and others.

Value

html file of the rendered report located in the specified output folder.

TF24_H_ANCHOR	<i>Height at which the stem hydraulic path is anchored [m].</i>
---------------	---

Description

TF24's resistance depends on 'theta' and 'K_s' only through their ratio, so there is exactly one free scalar and the height-dependent model can be made to agree with the older height-linear one at exactly **one** height – never two. 'TF24_H_ANCHOR' is that height. It is the pivot of a rotation: below it every plant is more resistant than under the linear model, above it every plant is less.

Usage

TF24_H_ANCHOR

Details

Anchored at 1 m, so resistance is unchanged at 1 m stature and falls progressively above it – reaching 0.21x by 60 m. Only sub-metre plants pay more than they did. It is deliberately a fixed constant rather than being read from 'pars\$hmat': sweeping 'hmat' as a trait must not silently move the hydraulics with it.

This is provenance for a default, not a model parameter, which is why it does not live in 'TF24_Pars'. Exposing it would add a fourth knob to the grouping 'theta * L_tip^beta / K_s', which is already identifiable only as a whole – and inviting independent calibration of its members is exactly what invariance criterion I5 forbids.

TF24_hyperpar	<i>Hyperparameter function for TF24 physiological model</i>
---------------	---

Description

Hyperparameter function for TF24 physiological model

Usage

TF24_hyperpar(m, s, filter = TRUE)

Arguments

m	A matrix of trait values, as returned by <code>trait_matrix</code>
s	A strategy object
filter	A flag indicating whether to filter columns. If TRUE, any numbers that are within eps of the default strategy are not replaced.

TF24_Individual	Create a TF24 Plant or Node
-----------------	-----------------------------

Description

Create a TF24 Plant or Node

Usage

```
TF24_Individual(s = TF24_Strategy())
```

Arguments

s A [TF24_Strategy](#) object

Examples

```
pl <- TF24_Individual()
pl$height
```

TF24_K_s_from_whole_stem	<i>Convert a whole-stem sapwood-specific conductivity to a terminal-segment one.</i>
--------------------------	--

Description

‘K_s’ is a property of the terminal segment, but conductivity is usually **reported** for a stem section at some height – and it was defined that way in TF24 before the stem path integral existed. This converts such a value into the terminal-segment ‘K_s’ that reproduces it at ‘H_anchor’.

Usage

```
TF24_K_s_from_whole_stem(
  K_s,
  D_c = 0.2,
  theta_c = 0,
  L_tip = 0.02,
  H_anchor = TF24_H_ANCHOR,
  eta = 12
)
```

Arguments

<code>K_s</code>	Whole-stem sapwood-specific conductivity.
<code>D_c</code>	Conduit widening exponent.
<code>theta_c</code>	Huber-profile exponent.
<code>L_tip</code>	Terminal segment length [m].
<code>H_anchor</code>	Height at which the whole-stem value was defined [m].
<code>eta</code>	Canopy shape parameter, used for the crown-centre coordinate.

Details

The ratio is obtained by calling the same ‘effective_path_length’ the model itself uses, at ‘beta’ and at zero, rather than being written as a literal: that way the ‘eta_c’ convention (the path runs to the leaf-area-weighted mean leaf height, not the apex) and the thin-crown approximation cancel out of the ratio instead of being smuggled into a hand-computed constant.

At the shipped defaults the ratio is 2.9782, so a whole-stem 1 becomes 0.3357738.

Value

The terminal-segment ‘K_s’.

TF24_Parameters	<i>Setup an a TF24 system with default or specified parameters</i>
-----------------	--

Description

Setup an a model system with default or specified parameters.

Usage

```
TF24_Parameters(...)
```

Arguments

... Arguments to be passed to the model constructor. These include

- *‘patch_area’: Area of idnividfual patch. Only relevant for stochastic model. Default is 1.0m2.
- *‘max_patch_lifetime’: The maximum time in years we want to simulate
- *‘strategies’: A list of stratgies to simulate. The default is an empty list.
- *‘strategy_default’: Values for the default stategy. The default values are those specified in the C++ code for the model.
- *‘node_schedule_times_default’: Default vector of times at which to introduce nodes. The default is chosen to have close spacing at the start of the simulation.
- *‘node_schedule_times’: A list with each element containing the vector of times we want to introduce nodes for each strategy. The default is an empty list.
- *‘ode_times’: A vector of patch ages we want the ode solver to stop at

Examples

```
p1 <- TF24_Parameters()
p2 <- TF24_Parameters(max_patch_lifetime = 10.0)
```

TF24_Pars	<i>TF24 strategy parameters</i>
-----------	---------------------------------

Description

Biological trait/parameters for the TF24 strategy. Held as the nested `pars` element of a [TF24_Strategy](#) (access as `s$pars$lma`).

Usage

```
TF24_Pars(..., values = list(...))
```

Arguments

<code>..., values</code>	Values to initialise the struct with (either as variadic arguments, or as a list, but not both).
--------------------------	--

TF24_plot_diagnostics	<i>Diagnostic panel plot for a TF24 stand</i>
-----------------------	---

Description

Assembles a multi-panel diagnostic figure from a collected TF24 SCM run: patch leaf area, the size distribution, soil water potential, stem and root water potentials, profit, stomatal conductance, transpiration (total and per leaf area), the rainfall driver and leaf temperature. Panels are combined with ‘patchwork’.

Usage

```
TF24_plot_diagnostics(results, x, y)
```

Arguments

<code>results</code>	A collected TF24 SCM run, i.e. the output of ‘ <code>run_scm(..., collect = TRUE)</code> ’ for a TF24 strategy/environment. Carries the tidied patch state (‘ <code>results\$species</code> ’) and the environment series (‘ <code>results\$env\$soil_moist</code> ’, ‘ <code>soil_depth</code> ’, ‘ <code>soil_moist_cumulative_flux</code> ’).
<code>x, y</code>	Numeric vectors giving the rainfall time series — time (years) and rainfall rate (m yr^{-1}) — as supplied to the environment’s extrinsic “rainfall” driver.

Details

The leaf-temperature panel is flat at the prescribed ‘leaf_temp’ driver unless the strategy ran with ‘pars\$use_energy_balance’ non-zero; see the ‘Tleaf’ auxiliary variable.

‘ggplot2’ and ‘patchwork’ are Suggested packages; the function errors if either is unavailable. The returned plot is neither drawn nor saved — print it or pass it to [ggplot2::ggsave()] as needed.

Value

The assembled ‘patchwork’ plot.

TF24_Strategy	<i>Strategy parameters</i>
---------------	----------------------------

Description

Strategy parameters that tune various aspects of the biological model.

Usage

```
TF24_Strategy(..., values = list(...))
```

Arguments

..., values	Values to initialise the struct with (either as variadic arguments, or as a list, but not both).
-------------	--

TF24f_generate_stand_report	<i>Generates a report on stand grown with TF24f strategy</i>
-----------------------------	--

Description

Builds a detailed report on stand grown with TF24f strategy, based on the template Rmd file provided. The reports are rendered as html files and saved in the specified output folder.

Usage

```
TF24f_generate_stand_report(
  results,
  output_file = "TF24f_report.html",
  overwrite = FALSE,
  target_ages = NA,
  input_file = system.file("reports", "TF24f_report.Rmd", package = "plant"),
  quiet = TRUE
)
```

Arguments

<code>results</code>	results of running <code>run_scm(..., collect = TRUE)</code>
<code>output_file</code>	name of output file
<code>overwrite</code>	logical value to determine whether to overwrite existing report
<code>target_ages</code>	Patches ages at which to make plots
<code>input_file</code>	report script (.Rmd) file to build study report
<code>quiet</code>	An option to suppress printing during rendering from knitr, pandoc command line and others.

Value

html file of the rendered report located in the specified output folder.

TF24f_hyperpar	<i>Hyperparameter function for TF24f physiological model</i>
----------------	--

Description

Hyperparameter function for TF24f physiological model

Usage

```
TF24f_hyperpar(m, s, filter = TRUE)
```

Arguments

<code>m</code>	A matrix of trait values, as returned by <code>trait_matrix</code>
<code>s</code>	A strategy object
<code>filter</code>	A flag indicating whether to filter columns. If TRUE, any numbers that are within eps of the default strategy are not replaced.

TF24f_Individual	<i>Create a TF24f Plant or Node</i>
------------------	-------------------------------------

Description

Create a TF24f Plant or Node

Usage

```
TF24f_Individual(s = TF24f_Strategy())
```

Arguments

`s` A `TF24f_Strategy` object

Examples

```
p1 <- TF24f_Individual()
p1$height
```

<code>TF24f_Parameters</code>	<i>Set up a TF24f system with default or specified parameters</i>
-------------------------------	---

Description

Set up a model system with default or specified parameters.

Usage

```
TF24f_Parameters(...)
```

Arguments

`...` Arguments to be passed to the model constructor. These include

- *`'patch_area'`: Area of an individual patch. Only relevant for the stochastic model. Default is 1.0m2.
- *`'max_patch_lifetime'`: The maximum time in years we want to simulate
- *`'strategies'`: A list of strategies to simulate. The default is an empty list.
- *`'strategy_default'`: Values for the default strategy. The default values are those specified in the C++ code for the model.
- *`'node_schedule_times_default'`: Default vector of times at which to introduce nodes. The default is chosen to have close spacing at the start of the simulation.
- *`'node_schedule_times'`: A list with each element containing the vector of times we want to introduce nodes for each strategy. The default is an empty list.
- *`'ode_times'`: A vector of patch ages we want the ode solver to stop at

Examples

```
p1 <- TF24f_Parameters()
p2 <- TF24f_Parameters(max_patch_lifetime = 10.0)
```

TF24f_Strategy	<i>Strategy parameters</i>
----------------	----------------------------

Description

Strategy parameters that tune various aspects of the biological model.

Usage

```
TF24f_Strategy(..., values = list(...))
```

Arguments

<code>..., values</code>	Values to initialise the struct with (either as variadic arguments, or as a list, but not both).
--------------------------	--

<code>tidy_individual</code>	<i>Turn ‘results’ of plant solver, when solving individuals into a tidy data object</i>
------------------------------	---

Description

Turn ‘results’ of plant solver, when solving individuals into a tidy data object

Usage

```
tidy_individual(results)
```

Arguments

<code>results</code>	plant solver output.
----------------------	----------------------

Value

a tibble whose columns provide metrics on each individual over time

tidy_snapshots	<i>Tidy a run's snapshots</i>
----------------	-------------------------------

Description

Flatten the thinned record of a run. With `scm$snapshot_interval` set before the run, the SCM keeps one summary per interval instead of a copy of the whole patch per leg: per species the node counts, the tallest cohort carrying density, the trapezium of the carried density over the birth-date grid and every node's lifetime offspring output, plus the environment's own state.

Usage

```
tidy_snapshots(scm)
```

Arguments

<code>scm</code>	An SCM that was run with a positive <code>snapshot_interval</code> .
------------------	--

Details

Read on the birth-date coordinate, which a continuous patch runs on: `density` is the trapezium of the carried density over the nodes' birth dates, a number of individuals per unit area, without the closing interval to the boundary node that a census adds; on the height coordinate it is the same trapezium over birth dates of a density in height, which is not a count. `n_retired` counts the nodes carrying no density, frozen or not, so it is `n_nodes - n_live`. The per-node `fecundity` vectors grow with the node list, so their memory grows with the square of the run's length (about 6 MB at a snapshot a year over 1000 years of one species); a coarser interval cuts it.

Value

A list of three: `species`, a data frame with one row per snapshot and species (`time`, `species`, `n_nodes`, `n_live`, `n_retired`, `height_max`, `density`, which includes what the absorbing top layer carries, and `top_density`); `environment`, a data frame with one row per snapshot of `time` and the environment's state entries (`env_1`, ...); and `fecundity`, a list over snapshots of lists over species of each node's lifetime offspring output.

See Also

[run_until_converged](#)

<code>tidy_species</code>	<i>Turn ‘species’ component of plant solver output into a tidy data object</i>
---------------------------	--

Description

Turn ‘species’ component of plant solver output into a tidy data object

Turn ‘env’ component of solver output into a tidy data object

Turns output of plant solver into a tidy data object

Usage

```
tidy_species(results)
```

```
tidy_env(results)
```

```
tidy_patch(results)
```

Arguments

`results` output of `run_scm(..., collect = TRUE)`

Value

a list, containing outputs of plant solver in tidy format

<code>trait_matrix</code>	<i>Create trait matrix</i>
---------------------------	----------------------------

Description

Helper function to create trait matrices suitable for [generate_strategy](#) and [add_strategies](#).

Usage

```
trait_matrix(x, trait_name)
```

Arguments

`x` Values

`trait_name` Name of a single trait

Author(s)

Rich FitzJohn

```
util_colour_set_opacity
```

Make colours transparent

Description

Make colours transparent

Usage

```
util_colour_set_opacity(col, opacity = 0.5)
```

Arguments

<code>col</code>	Vector of colours
<code>opacity</code>	Vector of opacities

Examples

```
util_colour_set_opacity("red", seq(0, 1, length.out=6))
util_colour_set_opacity(c("red", "blue"), .5)
```

```
Weibull_Disturbance_Regime
```

Weibull disturbance regime The frequency of disturbance in a meta-population of patches follows a Weibull distribution

Description

Weibull_Disturbance_Regime control object.

Usage

```
Weibull_Disturbance_Regime(max_patch_lifetime)
```

Arguments

<code>max_patch_lifetime</code>	Maximum lifetime of a patch, determines length of a simulation
---------------------------------	--

Index

- * `scenario_eval`
 - `scenario_eval`, 45
- `add_mutant`, 16
- `add_mutant` (*add_strategies*), 3
- `add_strategies`, 3, 65
- `build_scenario` (*scenario_eval*), 45
- `check_driver_interpolation`, 5
- `classify_scm_run` (*scenario_eval*), 45
- `climate_extreme`, 9
- `climate_extreme` (*events*), 8
- `Control`, 7
- `control`, 6
- `control_accurate` (*control*), 6
- `Disturbance_Regime`, 7
- `Environment` (*environment_type*), 7
- `environment_type`, 7
- `evaluate_scenario` (*scenario_eval*), 45
- `events`, 8, 38, 39, 42
- `events_default` (*events*), 8
- `expand_parameters` (*add_strategies*), 3
- `expand_state` (*FF16_expand_state*), 11
- `export_patch_state`, 10, 16, 28, 43, 45, 49, 50
- `FF16_Environment`, 11
- `FF16_expand_state`, 11
- `FF16_generate_stand_report`, 12
- `FF16_hyperpar`, 13
- `FF16_Individual`, 13
- `FF16_Parameters`, 14, 43
- `FF16_Pars`, 14
- `FF16_Strategy`, 13, 14, 15
- `fitness_window`, 15, 33, 34, 50
- `fixed_point_fitness`, 16
- `generate_strategy`, 4, 17, 65
- `gradient_control`, 18, 54
- `grow_individual_to_height`
 - (*grow_individual_to_size*), 19
- `grow_individual_to_size`, 19
- `grow_individual_to_time`, 20
- `harvest` (*events*), 8
- `hyperpar` (*make_hyperpar*), 27
- `Individual`, 20
- `integrate_over_size_distribution`, 21
- `Internals`, 21
- `interpolate_to_heights`, 22
- `interpolate_to_times`, 22
- `K93_Environment` (*FF16_Environment*), 11
- `K93_expand_state` (*FF16_expand_state*), 11
- `K93_hyperpar`, 23
- `K93_Individual`, 23
- `K93_Parameters` (*FF16_Parameters*), 14
- `K93_Pars`, 24
- `K93_Strategy`, 23, 24, 24
- `Leaf`, 25
- `make_FF16_hyperpar`, 26
- `make_hyperpar`, 27
- `make_initial_state`, 28
- `make_K93_hyperpar`, 29
- `make_TF24_hyperpar`, 29, 31
- `make_TF24f_hyperpar`
 - (*make_TF24_hyperpar*), 29
- `model_id` (*model_version*), 31
- `model_version`, 31
- `mutant_parameters` (*add_strategies*), 3
- `No_Disturbance`, 32
- `node_introductions` (*events*), 8
- `node_schedule_times_default`, 32, 33

node_schedule_times_periodic, 16, 33
 NodeSchedule, 34

 OdeControl, 34
 optimise_individual_rate_at_height_by_trait
 (*optimise_individual_rate_at_size_by_trait*), 35
 optimise_individual_rate_at_size_by_trait, 35

 param_hyperpar (*make_hyperpar*), 27
 plant_log_console, 36
 plot_size_distribution, 37
 pulse_series_poisson, 37
 pulse_series_record
 (*pulse_series_poisson*), 37
 pulse_series_regular
 (*pulse_series_poisson*), 37

 QK, 38

 rainfall_pulse, 8, 9, 38, 39
 read_scenario_mapping
 (*scenario_eval*), 45
 read_scenario_table (*scenario_eval*), 45
 resource_compensation_point, 40
 resource_pulse, 39
 resource_pulse (*events*), 8
 run_plant_benchmarks, 40
 run_scenarios (*scenario_eval*), 45
 run_scm, 11, 41, 49, 50
 run_stochastic_collect, 42
 run_until_converged, 17, 42, 43, 64

 scenario_control (*scenario_eval*), 45
 scenario_eval, 45
 scenario_generate_report
 (*scenario_eval*), 45
 scenario_model_fingerprint, 46
 scenario_model_fingerprint
 (*scenario_eval*), 45
 scenario_run_metadata
 (*scenario_eval*), 45
 scenario_summary (*scenario_eval*), 45
 scenario_to_config (*scenario_eval*), 45
 scm_base_parameters, 48
 seq_log, 49
 seq_log_range (*seq_log*), 49
 seq_range (*seq_log*), 49
 set_initial_state, 10, 11, 28, 49
 stand_census, 50, 52
 stand_census_state_adjoint, 51
 stand_fitness, 52
 stand_fitness (*stand_census*), 50
 stand_gradient, 51, 52
 stand_gradient_compare, 54
 stand_gradient_refused, 54
 strategy_list (*generate_strategy*), 17

 TF24_Environment, 55
 TF24_expand_state
 (*FF16_expand_state*), 11
 TF24_generate_stand_report, 55
 TF24_H_ANCHOR, 56
 TF24_hyperpar, 56
 TF24_Individual, 57
 TF24_K_s_from_whole_stem, 57
 TF24_Parameters, 58
 TF24_Pars, 59
 TF24_plot_diagnostics, 59
 TF24_Strategy, 57, 59, 60
 TF24f_expand_state
 (*FF16_expand_state*), 11
 TF24f_generate_stand_report, 60
 TF24f_hyperpar, 61
 TF24f_Individual, 61
 TF24f_Parameters, 62
 TF24f_Strategy, 62, 63
 tidy_env (*tidy_species*), 65
 tidy_individual, 63
 tidy_patch (*tidy_species*), 65
 tidy_snapshots, 45, 64
 tidy_species, 65
 trait_matrix, 4, 18, 65

 util_colour_set_opacity, 66

 Weibull_Disturbance_Regime, 66