Package: phylloptim (via r-universe)

November 13, 2026

Title Leaf Gas Exchange with Explicit Plant Hydraulics

Version 0.36.0

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

Description An implementation of leaf-level gas exchange in which stomatal behaviour emerges from hydraulics rather than an empirical conductance model. Farquhar-von Caemmerer-Berry photosynthesis is coupled to an explicit soil-root-stem-leaf water transport path, with Weibull vulnerability curves for both xylem and roots, multi-layer soil and gravitational head, and the operating point is chosen by Sperry-style gain-risk profit maximisation over the root-collar water potential. Forward-mode automatic differentiation supplies exact derivatives of profit with respect to the collar potential. Extracted from the TF24 strategy in the 'plant' package so that it can be tested, profiled, extended and embedded independently. The model itself is a set of self-contained C++ headers, under inst/include, that use no R and no 'Rcpp' and depend only on Boost and the header-only parts of 'odelia': they compile and run with no R present at all, so the model can be linked directly into a C++ program or bound to another language such as Python without going through this package. R users get the same model through an 'RcppR6' layer that sits on top of those headers and is never included by them, and R packages get it by naming 'phylloptim' in their LinkingTo field.

Encoding UTF-8

License AGPL (>= 3)

Depends R (>= 4.2.0)

Imports graphics, R6, Rcpp, stats, utils

LinkingTo Rcpp, BH, odelia (>= 0.10.0)

Suggests codetools, dplyr, ggplot2, knitr, rmarkdown, testthat (>= 3.0.0), tibble, tidyr, xml2, yaml

VignetteBuilder knitr

Config/testthat/edition 3

URL https://github.com/traitecoevo/phylloptim, https://traitecoevo.github.io/phylloptim/

BugReports https://github.com/traitecoevo/phylloptim/issues

Roxygen list(markdown = TRUE)

Config/roxygen2/version 8.0.0

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

Date/Publication 2026-11-10 07:34:07 UTC

RemoteUrl https://github.com/traitecoevo/phylloptim

RemoteRef HEAD

RemoteSha af31832bb456a19667882ea8ce15b7c98c8ffd55

Contents

phylloptim-package . . . 3
align_curves . . . 4
cost_curve_names . . . 6
cost_curve_parameters . . . 7
gradient_output_names . . . 7
gradient_par_names . . . 8
Leaf . . . 9
leaf_batch . . . 11
leaf_control . . . 13
leaf_damage_gradient . . . 15
leaf_damage_trajectory . . . 16
leaf_gradient . . . 17
leaf_gradient_batch . . . 24
leaf_infeasible_codes . . . 26
leaf_model . . . 28
leaf_solve . . . 29
leaf_supply_singlelayer . . . 32
leaf_traits . . . 34
model_registry . . . 37
operating_point . . . 38
plot_model_overview . . . 39
plot_solver_convergence . . . 40
psi_at_plc . . . 41
registry_locations . . . 42
registry_resolve . . . 43
registry_values . . . 44
root_network_from_carbon . . . 44
RootNetwork . . . 46
series_resistance . . . 46
set_drivers . . . 47

set_traits 49
weibull_p50_c 50
weibull_p50_from_b 52
weibull_s50 53
with_phylloptim_conditions 53

Index **55**

`phylloptim-package` *phylloptim: leaf gas exchange with explicit plant hydraulics*

Description

A leaf gas-exchange model in which stomatal behaviour emerges from hydraulics. Farquhar-von Caemmerer-Berry photosynthesis is coupled to an explicit soil-root-stem-leaf water path, and the operating point is the maximum of an optimality objective over the root-collar water potential. Nine optimality models share one solver; `cost_curve_names()` lists them. The model is a set of R-free C++ headers under `inst/include`, and this package is an R layer on top of them.

Where to start

- Solve a leaf in one call: `leaf_solve()`, with `leaf_traits()` and `leaf_control()`.
- Keep a leaf and drive it yourself: `leaf_model()`, `set_drivers()`, `set_traits()` and `operating_point()`.
- Trait gradients: `leaf_gradient()` for one operating point, and `leaf_batch()` with `leaf_gradient_batch()` for a fit over many observations.
- Thermal damage: `leaf_damage_trajectory()` and `leaf_damage_gradient()`.
- Vulnerability curves from published quantities: `weibull_p50_c()` and `psi_at_plc()`.

Vignettes

- `vignette("phylloptim")` – a tour of the R interface.
- `vignette("the-models")` – the nine optimality models and the maths behind them.
- `vignette("fitting")` – fitting traits to data.
- `vignette("cpp-interface")` – using the model from C++ with no R.
- `vignette("derivative-surface")` – differentiating the operating point inside a larger model.

Author(s)

Maintainer: Daniel Falster `<daniel.falster@unsw.edu.au>` (ORCID)

Authors:

- Daniel Falster `<daniel.falster@unsw.edu.au>` (ORCID)
- Isaac Towers (Author of the leaf gas exchange and hydraulics model)

Other contributors:

- Rich FitzJohn (Root-finding utilities (uniroot.hpp)) [contributor]

See Also

Useful links:

- https://github.com/traitecoevo/phylloptim
- https://traitecoevo.github.io/phylloptim/
- Report bugs at https://github.com/traitecoevo/phylloptim/issues

---

`align_curves` *Align the cost curves' level parameters to a common optimum*

Description

Solves each curve's level parameter so that its leaf-route optimum has the same stomatal conductance as a target, under one reference environment. Aligned curves agree at the reference and differ only in how their optimum moves away from it, so a comparison across curves measures how each RESPONDS to the environment rather than where its parameters happen to put it.

Usage

```
align_curves(
  models = NULL,
  psi_soil = 1,
  PPFD = 900,
  atm_vpd = 2,
  ca = 40,
  leaf_temp = 25,
  atm_o2_kpa = 21,
  atm_kpa = 101.3,
  leaf_specific_conductance_max = 3.14e-05,
  traits = leaf_traits(),
  control = leaf_control(),
  supply = leaf_supply_singlelayer(),
  target = NULL,
  tol = 1e-12
)
```

Arguments

`models`	which curves to align; any subset of `"CF77"`, `"JS22"`, `"CMax"` and `"LeastCost"`, the curves whose definitions have an aligned parameter (`cost_curve_parameters()`). `NULL`, the default, aligns all of them.

`psi_soil`, `PPFD`, `atm_vpd`, `ca`, `leaf_temp`, `atm_o2_kpa`, `atm_kpa`
: the reference environment, as in `leaf_solve()`. The defaults are `leaf_solve()`'s own, with a moderately wet soil.

`leaf_specific_conductance_max`
: maximum leaf-specific hydraulic conductance, kg m^-2 s^-1 MPa^-1

`traits`
: a `leaf_traits()` object

`control`
: a `leaf_control()` object

`supply`
: the supply path. The leaf routes need `leaf_supply_singlelayer()`, which is the default.

`target`
: the stomatal conductance to match, mol CO2 m^-2 s^-1. `NULL` (the default) takes TF24's at the reference, with `traits`.

`tol`
: the tolerance on the log of each level parameter.

Details

The package's own defaults are this function's answer at its defaults: `JS22_gamma` and `CMax_a` in `leaf_traits()`, and the `CF77_lambda` and `LeastCost_beta` that `leaf_solve()` and `leaf_batch()` use when those are left `NA`. The defaults' anchor is TF24, which does not move.

Value

A named numeric vector, one level parameter per curve, named as the argument or trait that carries it: `CF77_lambda`, `JS22_gamma`, `CMax_a`, `LeastCost_beta`. Attribute `"target"` holds the conductance matched.

Which curves can be aligned

Four curves have a level parameter: `CF77` (`CF77_lambda`, its price of water), `JS22` (`JS22_gamma`), `CMax` (`CMax_a`, with `CMax_b` held at its trait value) and `LeastCost` (`LeastCost_beta`). Each is solved on a log scale by `stats::uniroot()`.

The others are not, and this function refuses them:

- `SOX`, `JW26` and `ProfitMax` have no level parameter. The shared supply sets their optimum, and matching them through `kmax` would mean they no longer share the supply every other curve uses.
- `TF24_floor` reads TF24's own cost and adds a price `lambda_o` on top, so the only `lambda_o` that reaches TF24's conductance is zero, where the curve is TF24. That zero is its default.

`CF77` aligned to TF24 is an identity rather than a fit: at TF24's emergent `lambda` the two first-order conditions coincide, so the price returned is `operating_point()`'s `lambda` for TF24 at the reference.

See Also

`cost_curve_names()`, `leaf_traits()`, `leaf_solve()`

Examples

```
a <- align_curves()
a
# The defaults are this answer, so these two agree:
leaf_solve(psi_soil = 1, model = "JS22", route = "leaf",
           supply = leaf_supply_singlelayer())$gc
attr(a, "target")

# Align to a wetter reference instead, and use the result:
w <- align_curves(psi_soil = 0.5)
leaf_solve(psi_soil = 0.5, model = "CMax", route = "leaf",
           supply = leaf_supply_singlelayer(),
           traits = leaf_traits(CMax_a = w[["CMax_a"]]))$gc
```

---

cost_curve_names *The cost curves, in the order C++ indexes them*

Description

Every optimality model this package implements, as names, in the order of the `CostCurve` enumeration. R selects a curve by its POSITION in this vector, so the vector is read back out of C++ rather than restated in R: appending a curve is safe and reordering the enumeration would silently solve a different model.

Usage

```
cost_curve_names()
```

Value

A character vector of curve names.

See Also

`leaf_gradient()`, `leaf_gradient_batch()`

Examples

```
cost_curve_names()
```

`cost_curve_parameters` *Each cost curve's parameters, from its definition*

Description

One row per parameter a cost curve owns, curves in `cost_curve_names()` order: the curve, the parameter's name (a `gradient_par_names()` entry, and the `leaf_traits()` argument or `Leaf` field that carries it), its default, whether it is a trait, and whether its default is aligned by `align_curves()`. `NA` as a default means the curve refuses to solve until the parameter is set. Read out of the C++ definitions, so the R tables that route these parameters cannot drift from the curves.

Usage

```
cost_curve_parameters()
```

Details

A parameter that is not a trait is owned by its curve: `leaf_solve()` and `leaf_batch()` take it as an argument named without the trailing underscore, and every other curve refuses it.

Value

A data frame with columns `curve`, `parameter`, `default`, `trait` and `aligned`.

See Also

`cost_curve_names()`, `align_curves()`

Examples

```
cost_curve_parameters()
```

`gradient_output_names` *The differentiated outputs, in the order C++ indexes them*

Description

`A`, `gc`, `psi_leaf`, `collar` and `profit`. Unlike `gradient_par_names()`, which R holds its own copy of and a test compares, R **reads** this one — so the list exists once and adding an output is one edit rather than two that can disagree.

Usage

```
gradient_output_names()
```

Details

The first four are what a gas-exchange calibration observes; `profit` is what a demographic consumer bills, and it is the one output the envelope theorem reaches. See `leaf_gradient()`.

Value

A character vector of five names.

See Also

`leaf_gradient()`, `leaf_gradient_batch()`

Examples

```
gradient_output_names()
```

---

gradient_par_names *The differentiable parameters, in the order C++ indexes them*

Description

The sixteen `leaf_traits()` in `set_traits()`'s argument order, then `leaf_specific_conductance_max`, `resistance`, the parameters the cost curves own (`cost_curve_parameters()`) and the thermal-damage inputs. Exported so that R's own copy of this order can be compared against it in a test: R passes integer positions into this enumeration, so appending to it is safe and reordering it would silently differentiate the wrong parameter.

Usage

```
gradient_par_names()
```

Value

A character vector, one entry per differentiable parameter.

See Also

`leaf_gradient_batch()`

Examples

```
gradient_par_names()
```

`Leaf` *Leaf-level gas exchange with explicit hydraulics.*

Description

Stomatal behaviour emerges from a gain-risk profit maximisation over the root-collar water potential rather than from an empirical conductance model, coupled to Farquhar-von Caemmerer-Berry photosynthesis and an explicit soil-root-stem-leaf water transport path. See the package README for the model and for how it differs from `plantecophys`, `tealeaves` and `bigleaf`.

Usage

```
Leaf(
  vcmax_25,
  stem_c,
  stem_P50,
  root_c,
  root_P50,
  TF24_beta2,
  jmax_25,
  phi_J,
  curv_fact_elec_trans,
  curv_fact_colim,
  GSS_tol_abs,
  vulnerability_curve_ncontrol,
  ci_abs_tol,
  ci_niter,
  TF24_cost_scale
)
```

Arguments

`vcmax_25`	maximum carboxylation rate at 25 C (umol m^-2 s^-1)
`stem_c`	shape parameter of the STEM hydraulic vulnerability curve (unitless)
`stem_P50`	STEM potential at 50% loss of hydraulic conductivity (MPa, positive magnitude)
`root_c`	shape parameter of the ROOT hydraulic vulnerability curve (unitless)
`root_P50`	ROOT potential at 50% loss of hydraulic conductivity (MPa, positive magnitude)
`TF24_beta2`	exponent for the effect of hydraulic risk (unitless)
`jmax_25`	maximum rate of electron transport at 25 C (umol m^-2 s^-1)
`phi_J`	quantum yield of electron transport per ABSORBED photon (mol mol^-1); J reads phi_J * alpha_PAR_ * PPFD. Was `a` (incident basis) before 0.13.0: phi_J = a / alpha_PAR_.

`curv_fact_elec_trans`
: curvature factor for the light response curve (unitless)

`curv_fact_colim`
: curvature factor for the colimited photosynthesis equation

`GSS_tol_abs`
: absolute tolerance for the collar route's golden-section FALLBACK, and for the width below which it treats the feasible collar interval as a single point. It does NOT set how well the operating point is determined – the collar solve root-finds its own first-order condition – and it does NOT reach the leaf route at all.

`vulnerability_curve_ncontrol`
: number of control points for the pre-computed vulnerability curves

`ci_abs_tol`
: absolute tolerance for root-solving ci

`ci_niter`
: maximum number of iterations for root-solving ci

`TF24_cost_scale`
: cost parameter for the TF24 profit model (umol m^-2 s^-1)

Details

Set the drivers with `$set_physiology()`, solve with `$optimise()`, then read the operating point off the object: `$opt_psi_leaf_`, `$profit_`, `$assim_colimited_`, `$transpiration_`, `$lambda`, `$g1_eff`. `$find_root_collar_psi()` solves the seated curve on the collar route, whatever route and method are seated: on the default model, the same solve as `$optimise()`. It is kept because that is what `plant`'s own sources spell.

Which model is configuration, not a call argument. `$set_model(curve, route)` seats it and `$optimise()` takes nothing, because the curves' constants already live on the object – `$CF77_lambda_` and `$TF24_floor_lambda_o` are fields, and `JS22_gamma`, `CMax_a` and `CMax_b` are traits. `curve` is any name from `cost_curve_names()`; `route` is `"collar"` (the full soil-to-collar path, any supply) or `"leaf"` (psi_leaf with the upstream potential pinned at `psi_soil`, single soil layer only). A third argument, `method`, chooses HOW the operating point is reached: `"exact"` (the default, a root-find of the first-order condition) or `"closed"` (the closed form, for `TF24`, `CF77` and `TF24_floor` on the leaf route only). `$model_curve()`, `$model_route()` and `$model_method()` report what is seated. The defaults are `"TF24_floor"`, `"leaf"` and `"exact"`, on the single-potential supply, as in R's functions.

```
l$set_model("ProfitMax", "leaf")
l$optimise()
```

`ProfitMax` writes `$carbon_gain_`, `$hydraulic_cost_norm_` and `$thermal_cost_`, which are unitless and are NOT the same quantities as `$hydraulic_cost_`, which it sets to `NA` as SOX, JW26 and LeastCost do; `$profitmax_curve()` returns its whole cost, gain and profit curve in one call.

`$profit_` carries four kinds of number across the nine curves – carbon for the five difference objectives, a dimensionless normalised profit for `ProfitMax`, carbon times a dimensionless factor for the two products, and carbon per unit of weighted resource for `LeastCost`. Comparing it BETWEEN curves is not meaningful; comparing each curve's argmax is.

`$lambda`, the seated curve's marginal cost of water at its operating point, is the one output every curve reports on the same axis.
`$curve_cost()`, `$curve_profit()` and `$curve_lambda()` evaluate any curve by name at a leaf potential, whatever is seated; the per-curve methods (`$hydraulic_cost_CF77()`, `$lambda_SOX()` and the rest) are the same numbers, kept for existing code.

All water potentials are **positive magnitudes in MPa**. There is one representation throughout and it is asserted, not documented: a negative `stem_P50` or `root_P50` is an error here, and a negative `psi_soil` is an error in `set_physiology`.

`leaf_batch` *The observations a batch gradient is taken over*

Description

Resolves N observations' drivers once and holds them C++-side, together with one `Leaf`. Hand the result to `leaf_gradient_batch()`, repeatedly: the drivers do not change across a fit, only the parameters do.

Usage

```
leaf_batch(
  psi_soil,
  PPFD = 900,
  soil_depth = NULL,
  root_network = NULL,
  leaf_specific_conductance_max = 3.14e-05,
  atm_vpd = 2,
  ca = 40,
  leaf_temp = 25,
  atm_o2_kpa = 21,
  atm_kpa = 101.3,
  traits = leaf_traits(),
  control = leaf_control(),
  supply = leaf_supply_singlelayer(),
  CF77_lambda = NA_real_,
  TF24_floor_lambda_o = NA_real_,
  LeastCost_beta = NA_real_
)
```

Arguments

`psi_soil`	soil water potential, MPa, **positive magnitude**. Length sets the number of soil layers.
`PPFD`	INTERCEPTED photosynthetic photon flux density per unit leaf area, umol m^-2 s^-1: the light incident on the leaf, before its absorptance (`alpha_PAR_`). On the energy-balance path the leaf also needs `$NIR_int_`, the intercepted near-infrared in W m^-2, set before this call.

`soil_depth` cumulative depth to the bottom of each layer, m. Defaults to 1 m layers.

`root_network` the soil-to-collar hydraulic resistances, a `RootNetwork()`. **The same argument on both supply paths**: per rooted layer from `root_network_from_carbon()` on the multi-layer path, or one series resistance from `series_resistance()` on the single-potential path. Defaults to a nominal value on either — 20 kg C m^-2 leaf split evenly, or 1e3 — written out in the body where you can see and replace it. Stand-ins, not recommendations.

`leaf_specific_conductance_max` maximum leaf-specific hydraulic conductance, kg m^-2 s^-1 MPa^-1

`atm_vpd` atmospheric vapour pressure deficit, kPa. Must lie in `(0, e_s(leaf_temp))`, a relative humidity between 0 and 1; a deficit at or past saturation is refused.

`ca` atmospheric CO2 partial pressure, Pa

`leaf_temp` leaf temperature, deg C

`atm_o2_kpa` atmospheric O2 partial pressure, kPa

`atm_kpa` atmospheric pressure, kPa. Note this is not decorative: the ppm-to-Pa conversion is derived from it, and getting it wrong moves assimilation by percent, not by rounding.

`traits` a `leaf_traits()` object

`control` a `leaf_control()` object

`supply` how water reaches the root collar: `leaf_supply_singlelayer()` (the default) or `leaf_supply_multilayer()`. On the single-potential path `soil_depth` must be omitted, `root_network` is a `series_resistance()` (defaulted if omitted), and each `psi_soil` is one value rather than a profile.

`CF77_lambda` Cowan-Farquhar's prescribed marginal value of water, in umol C (kg H2O)^-1, one per observation. Only `model = "CF77"` reads it, and `leaf_gradient_batch()` refuses it under any other model, as `leaf_solve()` does; `NA_real_` (the default) uses its aligned default (see `align_curves()`).

`TF24_floor_lambda_o` the `TF24_floor` curve's price of water as transpiration goes to zero, same units, one per observation. Only `model = "TF24_floor"` reads it, and it is refused under any other model; `NA_real_` (the default) uses its default, 0, where that curve is `TF24`.

`LeastCost_beta` least-cost theory's unit-cost ratio (Prentice et al. 2014's beta): the cost of maintaining carboxylation capacity relative to the cost of transpiration, dimensionless, one per observation. Only `model = "LeastCost"` reads it, and it is refused under any other model; `NA_real_` (the default) uses its aligned default (see `align_curves()`). Zero is refused, since at zero the capacity term vanishes and what is left is water-use efficiency, which rises without bound as transpiration falls. The published value is 146.

Value

A `leaf_batch` object.

Why this is a separate call

Because the conversion is expensive and the gradient is not. A `RootNetwork()` costs 60-100 µs to cross the R boundary — some 60 times a trivial `.Call`, because it is an `RcppR6` `list:` class and every crossing rebuilds a five-element named list. Converting 1,327 of them per likelihood evaluation would cost ~80 ms and swamp the ~13 ms of gradient it was carrying. The same argument applies to the `Leaf:` constructing one is the largest single cost in the R layer (the README's "Performance from R" has the measured ratio), so a fit builds it once here rather than once per gradient.

The result holds a C++ pointer, so it does not survive `saveRDS()` or a new session. Rebuild it rather than restoring it; `leaf_gradient_batch()` says so rather than crashing if you try.

Recycling

The same rules as `leaf_solve()`. `PPFD`, `atm_vpd`, `ca`, `leaf_temp`, `atm_o2_kpa`, `atm_kpa` and `leaf_specific_conductance_max` are recycled to a common length. `psi_soil` is recycled when it is a plain numeric vector, each element then being a single-layer observation; pass a list of numeric vectors for multi-layer ones, and `soil_depth` follows the same rule. `root_network` is either one `RootNetwork()` used for every observation, or a list of them.

See Also

`leaf_gradient_batch()`, `leaf_gradient()` for one observation.

Examples

```
b <- leaf_batch(psi_soil = c(1.0, 1.5, 2.0), PPFD = 900)
b
leaf_gradient_batch(b, pars = c("vcmax_25", "stem_P50"))$gradient[, , "A"]
```

`leaf_control` *Numerical settings for a leaf solve*

Description

Tolerances and iteration counts. These are not traits: they control how precisely the solve is carried out, not what is being solved, and separating them from `leaf_traits()` is the point of having two functions.

Usage

```
leaf_control(
  GSS_tol_abs = 0.001,
  vulnerability_curve_ncontrol = .leaf_ncontrol_default(),
  ci_abs_tol = 0.001,
  ci_niter = 1000,
  integration_rule = 21,
  integration_tol = 1e-08
)
```

Arguments

`GSS_tol_abs` a width on the collar potential, for the collar solve's golden-section fallback. **This does not set how well the operating point is determined.** Every solver in the package root-finds its own first-order condition instead of searching the objective, so changing this leaves the answer bit-identical. What it still does, and it is less than the name suggests: it is read in exactly two places, both on the COLLAR route – the width below which the feasible collar interval is treated as a single point, and the tolerance of the golden-section FALLBACK used when either bracket endpoint has no usable gradient (no driver sweep has reached that fallback).

It does not reach the leaf route at all. That refines by a root-find on `dJ/dpsi == 0` at a hardcoded tolerance; the `(cell width) * 1e-4` figure is its own golden-section fallback, not its normal path. Tightening this to sharpen a leaf-route argmax does nothing.

`vulnerability_curve_ncontrol`
number of knot intervals in each Weibull vulnerability curve's table (a C2 quintic of its cumulative integral). Higher is more accurate and slower to construct; it does not affect solve speed. The default, 800, is read from the C++ `Leaf::ncontrol_default`, so the R and C++ defaults cannot differ.

`ci_abs_tol` absolute tolerance for root-solving intercellular CO2

`ci_niter` maximum iterations for root-solving intercellular CO2

`integration_rule`
accepted and ignored. It selected a Gauss-Kronrod rule order in plant's compiled quadrature; the header-only adaptive Simpson that replaced it has no rule order. Retained because plant's R bindings expose it.

`integration_tol`
absolute tolerance for the direct quadrature in `transpiration_full_integration()`. Not on the production path, which reads the vulnerability table instead.

Value

A `leaf_control` object; a named list.

What is deliberately not here

A range for the leaf temperature. Nothing clamps it: with the energy balance on it is affine in the transpiration, and `set_drivers()` refuses drivers under which a leaf the solve can reach would be past what the model can evaluate (R_d's Q10 reaching zero, about 119 C at the defaults), rather than holding the temperature at a bound.

See Also

`leaf_traits()`, `leaf_model()`

Examples

```
leaf_control()
leaf_control(GSS_tol_abs = 1e-5)
```

`leaf_damage_gradient` *Gradient of the leaf's thermal-damage trajectory*

Description

The exact sensitivities of `leaf_damage_trajectory()`'s states to the layer's dynamics parameters and initial states: what a fit of the layer to a leaf experiment needs. Computed by forward-mode differentiation of the same RK4 the trajectory runs, in one pass that carries every input's direction, so it is the derivative of the numbers `leaf_damage_trajectory()` returns, not of an idealised solution. Its cost does not depend on how many inputs are asked for.

Usage

```
leaf_damage_gradient(x, times, leaf_temp, A0 = 0, I0 = 0, D0 = 0, pars = NULL)
```

Arguments

`x`	a `Leaf`, from `leaf_model()`
`times`	days, strictly increasing
`leaf_temp`	leaf temperature at each time, deg C
`A0`, `I0`, `D0`	the acclimation state and the impaired and damaged fractions at `times[1]`
`pars`	which inputs to differentiate with respect to; all of them by default.

Details

The inputs, named as the leaf's fields: the impairment rate (`k_i_`, `k_i_Ea_`, `k_i_Tiso_`), repair (`k_rec_`, `k_rec_Ea_`), maturation (`k_mat_`, `k_mat_Ea_`), acclimation (`acclim_alpha_`, `acclim_beta_`, `acclim_T_`, `acclim_s_`), the acclimated shift of the isokinetic temperature (`dTimpair_max_`, `K_A_`) and the constitutive offset (`topt_offset_`), then the initial states `A0`, `I0` and `D0`.

The sub-step count is chosen from the parameter VALUES, so it is piecewise constant in them: the gradient is exact for the discrete map at that count, and a finite difference straddling a change of count differs from it by the integrator's own error (~1e-8 relative).

Value

A list: `value`, the `leaf_damage_trajectory()` data.frame, and `gradient`, an array `[time, state, input]` with states `A`, `I`, `D`.

Examples

```
l <- leaf_model()
t <- seq(0, 2, by = 1 / 24)
g <- leaf_damage_gradient(l, t, ifelse(t < 0.25, 46, 25),
                          pars = c("k_i_Tiso_", "k_rec_"))
g$gradient[length(t), , ]
```

`leaf_damage_trajectory` *Integrate the leaf's thermal-damage states over a temperature series*

Description

The leaf's lasting damage model, run on its own: three fractions of leaf area, functional impaired → damaged, plus an acclimation state, driven by a leaf-temperature time series. This is the route for fitting the layer's parameters to a leaf experiment without a whole-plant model; the comparison with measurements (Fv/Fm, say) belongs to the caller.

Usage

```
leaf_damage_trajectory(x, times, leaf_temp, A0 = 0, I0 = 0, D0 = 0)
```

Arguments

`x`	a `Leaf`, from `leaf_model()`
`times`	days, strictly increasing
`leaf_temp`	leaf temperature at each time, deg C
`A0`, `I0`, `D0`	the acclimation state and the impaired and damaged fractions at `times[1]`

Details

The dynamics, per day, with T the leaf temperature and $F = 1 - I - D$:

```
dA/dt    = acclim_alpha * softplus(T - acclim_T; acclim_s) - acclim_beta * A
dI/dt    = k_imp(T) * F - (k_rec(T) + k_mat(T)) * I
dD/dt    = k_mat(T) * I
k_imp(T) = k_i   * exp[(k_i_Ea/R)(1/T_iso - 1/T)]
k_rec(T) = k_rec * exp[(k_rec_Ea/R)(1/T_25 - 1/T)]
k_mat(T) = k_mat * exp[(k_mat_Ea/R)(1/T_25 - 1/T)]
```

with T_{iso} = `k_i_Tiso_` shifted by `topt_offset_` and the acclimation, and $T_{25} = 25$ C. The impairment rate is the first-order PSII heat inactivation of Bison & Michaletz (2026, Nature Ecology & Evolution, doi:10.1038/s41559026031960x), and its defaults their 177-species fit: the rate common to every species at the isokinetic temperature (276 per day, at 50.5 C) and the mean activation energy (316 kJ/mol), so it rises tenfold about every 6 K. Nothing switches the rates on or off.

Every parameter is a field of `x` (`$k_i_`, `$k_i_Tiso_`, `$acclim_beta_`, ...); set them on the leaf before calling. Neither the gate `use_thermal_damage_` nor the drivers are read. The rates themselves are `x$damage_rates(leaf_temp, A, I, D)`.

Temperature is linear between the given times. The integrator is fixed-step RK4 with each interval cut so the fastest rate times the step stays below 0.1 and the temperature moves at most 0.25 K per step, so a coarse output grid does not coarsen the solution.

Value

A data.frame with `time`, `leaf_temp`, `A`, `I`, `D` and the functional fraction `F = 1 - I - D`.

Examples

```
l <- leaf_model()
# a 6-hour 46 C heatwave, then two days at 25 C
t <- seq(0, 2.25, by = 1 / 24)
temp <- ifelse(t < 0.25, 46, 25)
d <- leaf_damage_trajectory(l, t, temp)
tail(d, 3)
```

`leaf_gradient` *Trait gradients of a solved operating point*

Description

The derivatives of the solved outputs with respect to the traits: `dA/dtheta`, `dgc/dtheta`, `dpsi_leaf/dtheta`, `dcollar/dtheta` and `dprofit/dtheta`, at one operating point.

Usage

```
leaf_gradient(
  psi_soil,
  PPFD = 900,
  soil_depth = NULL,
  root_network = NULL,
  leaf_specific_conductance_max = 3.14e-05,
  atm_vpd = 2,
  ca = 40,
  leaf_temp = 25,
  atm_o2_kpa = 21,
  atm_kpa = 101.3,
  x = NULL,
  traits = leaf_traits(),
  control = leaf_control(),
  supply = leaf_supply_singlelayer(),
  pars = NULL,
  psi = NULL,
  dpsi_dtheta = NULL,
```

```
  model = "TF24_floor",
  route = NULL
)
```

Arguments

psi_soil	soil water potential, MPa, **positive magnitude**. Length sets the number of soil layers.
PPFD	INTERCEPTED photosynthetic photon flux density per unit leaf area, umol m^-2 s^-1: the light incident on the leaf, before its absorptance (alpha_PAR_). On the energy-balance path the leaf also needs $NIR_int_, the intercepted near-infrared in W m^-2, set before this call.
soil_depth	cumulative depth to the bottom of each layer, m. Defaults to 1 m layers.
root_network	the soil-to-collar hydraulic resistances, a RootNetwork(). **The same argument on both supply paths**: per rooted layer from root_network_from_carbon() on the multi-layer path, or one series resistance from series_resistance() on the single-potential path. Defaults to a nominal value on either — 20 kg C m^-2 leaf split evenly, or 1e3 — written out in the body where you can see and replace it. Stand-ins, not recommendations.
leaf_specific_conductance_max	maximum leaf-specific hydraulic conductance, kg m^-2 s^-1 MPa^-1
atm_vpd	atmospheric vapour pressure deficit, kPa. Must lie in (0, e_s(leaf_temp)), a relative humidity between 0 and 1; a deficit at or past saturation is refused.
ca	atmospheric CO2 partial pressure, Pa
leaf_temp	leaf temperature, deg C
atm_o2_kpa	atmospheric O2 partial pressure, kPa
atm_kpa	atmospheric pressure, kPa. Note this is not decorative: the ppm-to-Pa conversion is derived from it, and getting it wrong moves assimilation by percent, not by rounding.
x	an existing Leaf from leaf_model() to reuse instead of building one, for a loop that takes many gradients. traits is required with it, and control/supply must be omitted. See the section above.
traits	a leaf_traits() object: the point in trait space to differentiate at
control	a leaf_control() object
supply	how water reaches the root collar: leaf_supply_singlelayer() (the default) or leaf_supply_multilayer()
pars	what to differentiate with respect to. Any of the sixteen leaf_traits() names, plus "leaf_specific_conductance_max", on the single-potential path's collar route only "resistance" (the leaf route never reads it), on model = "CF77" only "CF77_lambda_", on model = "TF24_floor" only "TF24_floor_lambda_o", and on model = "LeastCost" only "LeastCost_beta". On a reused leaf x with use_thermal_damage_ on, also the layer's inputs that reach the solve: its states "A_", "I_", "D_", the optimum shift

"topt_offset_", "dTopt_max_", "K_A_", and every quantity a maintenance cost multiplies ("k_i_", "k_i_baseline_", "k_rec_", "k_rec_Ea_", "c_acclim_maint_", "c_repair_maint_", "c_protect_maint_", "c_repair_flux_"). The layer's dynamics parameters move no solved output and are refused; their gradient is leaf_damage_gradient()'s. Defaults to all the route allows.

At 25 C the peaked Arrhenius is normalised to its reference, so topt_offset_ and dTopt_max_ have an exactly zero effect there – a true zero, not a missing row. The protection cost is signed, k_i_baseline_ - k_i_, so its columns are the same linear slope on both sides of the default, where the two are equal.

dY/dvcmax_25 is a PARTIAL at fixed respiration: R_d_25 is its own trait, so a fit that wants respiration to follow Vcmax moves both and adds the two columns.

beta_R_H and beta_R_V are not parameters here: they parameterise the root-architecture model, which the leaf does not run, so this function has no way to reach them. Getting a gradient in either means solving at two networks yourself. The perturbation itself is cheap — root_network_from_carbon() is homogeneous of degree 1 in each constant, so scaling beta_R_H scales r_R_H_min by the same factor and needs no rebuild — but the two solves are still two solves.

psi — the decision variable to evaluate at, in MPa as a positive magnitude, instead of solving for the profit-maximising one: the collar potential on the collar route, the leaf potential on a leaf route. NULL (the default) solves. See the section above.

dpsi_dtheta — how the prescribed psi itself responds to each parameter, as one value per entry of pars (or one value recycled, or a vector named by pars). Defaults to zero, which is the partial derivative at fixed collar. Must be finite — an infinite value would return infinite rows, and NaN wherever dY/dpsi is zero. Only meaningful with psi.

model — which optimality model to differentiate: any name from cost_curve_names(), on route; "TF24_floor" by default. See *Choosing which model to differentiate*.

route — "collar" or "leaf", the potential the curve is maximised over; NULL (the default) is the leaf route. See *Choosing which model to differentiate*.

Value

A list with

gradient a matrix, one row per trait in pars, with columns A (umol C m^-2 s^-1 per trait unit), gc, psi_leaf, collar and profit (also umol C m^-2 s^-1 per trait unit)

value the solved outputs the gradient is taken at

status "interior", "pinned" or "no-gradient" on the solving path; "prescribed", "refused" or "no-gradient" with psi

H the curvature of profit in the decision variable, at an interior optimum or at the psi given; NA elsewhere

`M` the mixed partial `d2profit/dpsi dtheta`, one per entry of `pars`, at an interior optimum or a prescribed `psi`; `NA` elsewhere

`dY_dpsi` `dY/dpsi` at fixed traits, one per output. `profit`'s entry is the analytic zero at an interior optimum and the exact `dprofit/dpsi` elsewhere – see the envelope section

`psi` the decision variable the outputs were evaluated at (the collar, or `psi_leaf` on a leaf route): `psi*` on the solving path, and the `psi` argument on the prescribed path, refused or not

`model`, `route` the model differentiated and its route

An entry that is not determined, such as a refused point's gradient or a leaf route's `collar`, is `NaN`.

Choosing which model to differentiate

A model is a cost curve and a route, and every curve has a gradient on both routes. `model` names the curve and `route` the potential it is maximised over: `"collar"`, the root-collar potential, on either supply path; or `"leaf"`, `psi_leaf` with the upstream potential pinned at `psi_soil`, which needs the single-potential supply path. The defaults are one set, everywhere: `"TF24_floor"` (plant's model; at its default price, `TF24_floor_lambda_o = 0`, it is TF24) on the leaf route over the single-potential supply. The collar route and the multi-layer supply are asked for.

The product objectives, `SOX` and `JW26`, need no special case. They maximise `A * g(psi)`, which is the `Log` benefit link applied to the same difference form everything else uses: `d/dpsi[h(A) - C] = h'(A) * dA/dpsi - dC/dpsi` with `h' = 1/A`. One derivative serves all nine.

The two routes are different models, not two ways of asking one question, even for one curve. They optimise different variables over different supply topologies, for TF24 disagree on 20 of 30 driver rows, and the gap does not close as the root resistance goes to zero.

`model` **SEATS the leaf, including one you passed as** `x`**.** The curve and route are applied with `$set_model()` and left there when this returns, alongside the base-point traits and solve. A loop that alternates models on one `x` is fine; a loop that takes a gradient and then reads `$optimise()` off the same object expecting the default is not.

On a leaf route the decision variable is `psi_leaf`**, so** `collar` **comes back non-finite.** The collar is not solved for at all there, and an absent number is the truthful answer rather than a gap – reading one would be reading state from whichever solve ran last.

Every curve on every route is differentiated the same way. `ProfitMax` normalises by `|A|max`, which the gradient carries as a function of the parameters, so its gradient is the total derivative, not a partial at a held normaliser.

A zero column for another curve's parameter means two different things, and they are not distinguishable from the number. On a `collar` or `TF24` route, `JS22_gamma`, `CMax_a`, `CMax_b` and the rest come back exactly zero, and read one way that is correct: those parameters do not enter the TF24 cost, so their derivative is genuinely zero. Read the other way it is a trap: if you are *fitting* one of those curves, the zero is telling you this differentiated a different model, not that the parameter does nothing. Measured on the single-potential path at `psi_soil` 1.5, the TF24 collar optimum sits at 2.09 MPa and the JS22 optimum at 3.15.

An optimiser handed such a column concludes the parameter is inert and leaves it at its starting value, which looks like convergence. **Name the curve you are fitting in** `model`; `.gradient_available_pars()` reports which parameters a given route can move.

How the gradient is computed

The reported outputs are evaluated at the solved decision variable `psi*` (the collar potential, or `psi_leaf` on a leaf route), so a parameter moves them two ways: directly, and by moving `psi*`. The solve records what kind of point `psi*` is, and that decides how its response is taken.

At an interior optimum `dprofit/dpsi = 0`, and differentiating that condition gives `psi*`'s response without re-solving the model:

$$d\psi^*/d\theta = -M/H, \quad M = \partial^2 profit/\partial\psi\partial\theta, \quad H = \partial^2 profit/\partial\psi^2$$

$$dY/d\theta = \partial Y/\partial\theta|_\psi + (\partial Y/\partial\psi)(d\psi^*/d\theta)$$

At a pinned optimum `psi*` is the bound that holds it, and its response is the bound's (see *The active set*).

Every term is exact. The outputs, the first-order condition and the bounds are closed forms in the parameters, and one forward pass of automatic differentiation carries their derivatives in every parameter at once. Only `H` is a difference, of the exact marginal `dprofit/dpsi` at the solved point. So a gradient costs the same whatever `pars` asks for, and its rows agree with a central difference of the whole solve to that difference's own precision: the package's tests hold every curve on every route to it.

The second term is not a correction. For `TF24_cost_scale`, `TF24_beta2` and `stem_c` it is 100% of the answer, and for `vcmax_25` 52%.

profit, which is the one output the envelope theorem reaches

The first four outputs are what a gas-exchange calibration observes. `profit` is here because it is what a **demographic** consumer bills: plant's carbon budget reads the leaf's carbon, not `assim_colimited_`, and without it no gradient from this package would reach a demographic model.

At an interior optimum `dprofit/dpsi = 0`, so the indirect term above vanishes identically and

$$dprofit/d\theta = \partial profit/\partial\theta|_{\psi^*}$$

– the direct partial alone. That is the envelope theorem. The gradient reads profit at the held `psi*` there, so the column is the direct term and `dY_dpsi["profit"]` is the analytic zero, not the solve's residual.

The envelope does not survive the active set. At a pinned optimum `psi*` is a trait-dependent bound and `dprofit/dpsi` is not zero there, so the indirect term is real and the gradient carries it through the bound.

A collar potential you supply, instead of the one this solves for

Pass `psi` and the outputs are evaluated at that collar rather than at the argmax; on a leaf route `psi` is the leaf potential, that route's decision variable, and everything below holds with it in the collar's place. This is for a caller whose model **tracks** the optimum instead of finding it — plant's TF24f carries the collar potential as an ODE state, `dpsi/dt = k * dprofit/dpsi`, so at finite gain its `dprofit/dpsi` is deliberately non-zero and *is* the acclimation rate.

There the derivation above simplifies rather than breaks. `psi` is exogenous, so the indirect term is whatever the caller says it is:

$$dY/d\theta = \partial Y/\partial\theta|_{\psi} + (\partial Y/\partial\psi)(d\psi/d\theta)$$

with `dpsi_dtheta` supplied — defaulting to zero, the partial at fixed collar. Nothing is derived from `-M/H`, so nothing needs stationarity, and `profit` is read at the collar you gave, with its exact `dprofit/dpsi`.

`M`, `H` and `dY_dpsi` come back in the result, because a caller with a *dynamic* `psi` cannot supply `dpsi_dtheta` as a constant: for the gradient-ascent law above it obeys `ds/dt = k(M + H s)`, and those are its coefficients. Its fixed point is `-M/H`, which is what the solving path returns — the two agree in the large-gain limit. So `psi = <the solved psi*>` with `dpsi_dtheta = -M/H` reproduces the solving path, to rounding, which the tests assert.

A `psi` outside the feasible interval is refused, and gets no gradient. The leaf does not project it onto the interval's end, which would make the outputs move with the *bound* rather than with `dpsi_dtheta`. `status` reports `"refused"`, `psi` in the result is the one you gave, and everything else is `NA`. This is reported rather than thrown because a fit reaches these points routinely: a trait perturbation moves the interval past a fixed `psi`.

A feasible `psi` whose leaf potential lies in the stem curve's tail has a value and no gradient, reported as `"no-gradient"`. That is the collar a shut leaf seats, and the last sliver before the dry end of the interval: there inverting the flux for the leaf potential divides by a conductance near `e^-40`, and every row would be rounding.

The active set

Stationarity is the premise of `-M/H`, and it fails when the optimum is pinned to an end of the feasible interval. There `psi*` is a bound, and `-M/H` is not the bound's derivative. It does not fail loudly: measured at a boundary-soil point the true gradient is about `1e-08` and `-M/H` gives `O(1)`, plausible-looking and wrong by seven orders of magnitude. About one operating point in six is pinned across this package's test grid, all at soil suctions of 3-6 MPa, the dry end a drought calibration will visit.

So the gradient does not test for stationarity; it reads what the solve found. The solve records whether it root-found the first-order condition inside the interval, or a bound held the optimum, and which: the wet end, where the leaf declines to open, or the dry end, where it transpires all its stem can carry. `status` reports it, `"interior"` or `"pinned"`, and at a pinned point the gradient differentiates the bound.

That matters beyond accuracy. The dry end – `psi_crit` for the curves it bounds, the end of the stem curve for the rest – does not appear in the profit function at all, so `-M/H` would return **exactly zero** for a parameter that moves only the end. At a dry pin that end *is* the

binding constraint, and a zero would tell an optimiser the parameter does nothing. Placing the point on the bound carries the term.

`H` and `M` are `NA` at a pinned point. An interior point whose curvature is not clearly negative is `"no-gradient"`: there every row would amplify the solve's own error.

Two of these are not traits

`pars` also accepts `leaf_specific_conductance_max` and, on the single-potential path's collar route, `resistance`. They are here because a calibration fits them: of a companion calibration study's four free parameters two are traits and two are these, so restricting `pars` to `leaf_traits()` left half of that fit without an exact gradient.

Nothing in the derivation cares that the parameter is a trait — the implicit function theorem is applied to `dprofit/dpsi = 0`, and any parameter the profit function depends on goes through it identically. What differs is only which setter applies the perturbation. `leaf_specific_conductance_max` is also how plant's height reaches the leaf, so the same route gives a gradient with respect to plant state and not only with respect to traits.

Both move the feasible collar interval, not just the profit function — `resistance` directly and `leaf_specific_conductance_max` through the supply curve — so at a pinned point their rows run through the bound, as a trait's do.

What it costs

With `N` observations, a gradient costs

$$T_{exact} = N\,(t_{construct} + t_{solve} + t_{pass})$$

where `t_pass` is one forward pass carrying every parameter, a few solves' worth, whatever `pars` asks for. Differencing the objective instead costs `2 P_fit` solves per observation, for the `P_fit` parameters an optimiser moves. So the exact gradient wins once a fit moves more than a couple of parameters, and by more the more it moves; `vignette("fitting")` and the README's "Performance from R" give the measured figures.

`pars` selects rows, not work: asking for one parameter costs the same as asking for all of them.

The intercept is a fresh `Leaf` per call. Pass `x` to reuse one across calls; without it, construction is most of a one-observation gradient.

Precision

Do not ask for more than about `1e-09` from any of this. That is the achievable precision of the solved outputs themselves, set by the tolerance of the intercellular-CO2 root-find.

Reusing a leaf across gradients

Constructing a `Leaf` is the largest single cost on this surface, and this function does it once per call: it is **most of a gradient** on a fresh leaf, paid once per observation by a fit that differentiates per observation. The measured ratios are in the README's "Performance from R", the one place the package quotes them. Pass `x` to reuse one:

```
l <- leaf_model(traits, control, supply)
for (i in seq_len(n)) leaf_gradient(psi_soil = ps[i], x = l, traits = traits, pars = FIT)
```

`x` is a **vessel, not the point**: `traits` still says where the gradient is taken and is applied to `x`, rather than read from it (a `Leaf` does not expose its traits). That is why `traits` is required with `x` — defaulting it would differentiate at the package defaults on somebody else's leaf and return a plausible answer for a point nobody asked about.

`control` and `supply` are fixed when `x` is built, so passing them alongside it is an error rather than silently resolved.

`x` **is left solved at the base point**, on the model it seated, including after a prescribed `psi` or an error. The restore costs one solve, a small fraction of the construction it saves.

See Also

`leaf_solve()` for the operating point itself, `set_traits()`.

Examples

```
g <- leaf_gradient(psi_soil = 2.0, PPFD = 900)
g$status         # "interior"
g$gradient[c("vcmax_25", "stem_P50"), ]

# On the collar route, in dry soil at high VPD, the optimum pins against the
# edge of the feasible range, and the gradient is the bound's.
dry <- leaf_gradient(psi_soil = 4.5, PPFD = 900, atm_vpd = 3.0,
                     supply = leaf_supply_multilayer(), route = "collar")
dry$status       # "pinned"
dry$H            # NA: no curvature is taken off an interior point
```

---

leaf_gradient_batch *Trait gradients over a batch of observations*

Description

`leaf_gradient()`, composed in C++ and vectorised over the observations of a `leaf_batch()`. One crossing of the R boundary per call, in place of 112 per observation.

Usage

```
leaf_gradient_batch(
  batch,
  traits = NULL,
  pars = NULL,
  theta = NULL,
  psi = NULL,
  dpsi_dtheta = NULL,
  model = "TF24_floor",
  route = NULL
)
```

Arguments

`batch`	a `leaf_batch()`
`traits`	a `leaf_traits()` object: the point in trait space to differentiate at. Defaults to the one `batch` was built with. Ignored when `theta` is given.
`pars`	what to differentiate with respect to, as in `leaf_gradient()`. It selects rows, not work: the default, all of them, costs the same as one.
`theta`	the model parameters as a matrix, for the case where they vary by observation. See the section above.
`psi`	one decision variable per observation to evaluate at (positive magnitudes, MPa) instead of solving for the profit-maximising one: the collar potential on the collar route, the leaf potential on a leaf route. See `leaf_gradient()`'s section on it. Recycling a single value across the batch is deliberately not offered — rows sharing an imposed collar is a coincidence rather than a design, and broadcasting one would make a length mismatch look intentional.
`dpsi_dtheta`	how each prescribed `psi` responds to each parameter: an `n` × `length(pars)` matrix, or one vector of `length(pars)` shared by every observation. Defaults to zero. Must be finite, for the reason `leaf_gradient()` records. Only meaningful with `psi`.
`model`, `route`	which optimality model to differentiate, and on which route, as in `leaf_gradient()`: any `cost_curve_names()` entry, `"TF24_floor"` by default, on its leaf route unless `route = "collar"`.

Value

A list with

`gradient` an `n` × `length(pars)` × 5 array, the last dimension being `A`, `gc`, `psi_leaf`, `collar` and `profit`

`value` an `n` × 5 matrix of the solved outputs

`status` `"interior"`, `"pinned"`, `"no-gradient"`, `"error"`, or with `psi` one of `"prescribed"`, `"refused"`, `"no-gradient"`

`H` the curvature at each interior optimum or prescribed `psi`; `NA` elsewhere

`M` an `n` × `length(pars)` matrix of mixed partials `d2profit/dpsi dtheta`, `NA` where `leaf_gradient()` has none

`dY_dpsi` an `n` × 5 matrix of `dY/dpsi` at fixed traits

`psi` the decision variable each row was evaluated at

`message` `""`, or why that row failed

What it computes, and where the maths is written down

The same five derivatives at the same solved operating point, by the same pass. **Read `leaf_gradient()`'s documentation for all of it** — the implicit function theorem, why the second term is not a correction, how a pinned point is differentiated, and what `status` means. Nothing here differs: `leaf_gradient()` is this function with one observation.

Speed, and which side of the boundary the figure belongs to

Through `leaf_gradient()`, almost all of a gradient's cost per observation is dispatch and the R interpreter; the model work — a solve and one pass — is a small part of it. Composed here in C++, the same gradient costs over an order of magnitude less per observation. The README's "Performance from R" has the measured ratios.

That is a per-observation figure and it needs a batch to be realised. Calling this with one observation pays the whole R-side call overhead for one row's worth of work and will not be 20× anything. The number of observations is the lever.

Per-row status, not an error

A proposal during a fit **will** reach operating points the solve cannot handle. That has to cost those rows rather than the whole dataset, so every row reports its own `status` and a failure is `"error"` with a `message` rather than a condition. A failed row's gradient is all `NA`, never partly filled: returning entries next to `status == "error"` would invite a caller who checks the status per row to read a gradient with a hole in it. `value` and `H` are kept where they were determined, because they describe the point rather than the derivative.

Supplying `theta` directly

By default the parameters are `traits` — the batch's, or the ones passed here — with `leaf_specific_conductance_max` and `resistance` taken from the batch's own drivers. Pass `theta` instead when they vary by observation, which is what a hierarchical fit needs: a matrix with one row per observation (or one row shared by all of them) and one column per `gradient_par_names()` entry, in that order. `resistance` is ignored on the multi-layer path and may be `NA` there. An `NA` in the column of the level parameter `model` owns (`CF77_lambda_` or `LeastCost_beta`) takes its aligned default, as it does when `theta` is built for you.

See Also

`leaf_batch()`, `leaf_gradient()` for one observation and for the derivation.

Examples

```
b <- leaf_batch(psi_soil = seq(0.5, 4, length.out = 6), PPFD = 900)
g <- leaf_gradient_batch(b, pars = c("vcmax_25", "stem_P50"))
g$status
g$gradient[, "vcmax_25", "A"]
```

`leaf_infeasible_codes` *Which failures are reported as infeasible*

Description

The `code` values a `phylloptim_infeasible` condition can carry, and what each one means. Use these rather than matching on the message text, which changes whenever an error message is improved.

Usage

```
leaf_infeasible_codes()
```

Value

A named character vector: codes, with a one-line description each.

What is deliberately NOT here

Every input-validation failure. A mismatched `psi_soil`/`soil_depth` length, a non-finite driver, a negative water potential, an unusable `R_d_25`, a misconfigured supply path: those stay ordinary errors, and that is the whole safety property of this list. Classifying one of them as infeasible would let it be swallowed by the very `tryCatch()` this exists to enable, and a fit would report a plausible likelihood over whichever rows survived — which is exactly the failure that refusing silent NA rows exists to prevent.

So the test a site has to pass is narrow: it must be reachable from a **well-formed call with in-range data**, because the parameters or the state make the operating point unsolvable. Anything reachable only by handing the model something malformed is not infeasibility, it is a bug in the caller, and it should be loud.

Two judgement calls left ordinary, and why

Of the `stop` sites in the headers, those carrying a code above are classified. Two of the rest are genuinely arguable, and both were left ordinary because that is the direction whose failure mode is a loud error rather than a quiet wrong answer. They are listed here so the decision can be overruled on purpose rather than rediscovered:

- `method = "ift"` **requested at a point where the IFT does not hold** (`leaf_gradient`). The default `method = "auto"` never reaches it, so getting here means the caller overrode the choice and the answer is "not at this point".
- **Trait sign and magnitude checks** (`psi_crit`, `stem_b`, `root_b`, `root_psi_crit` must be positive). Reachable by an unbounded proposal, but a negative `stem_b` is not an infeasible operating point, it is not a stem.

Every other unclassified site is input or configuration validation — vector lengths, non-finite drivers, an unusable supply path — where there is no argument for infeasibility at all.

See Also

`with_phylloptim_conditions()`

Examples

```
leaf_infeasible_codes()
```

`leaf_model` *Build a leaf*

Description

The recommended way to construct a leaf. `Leaf()` is the raw C++ constructor, with all fifteen arguments positional and no defaults; this splits them into traits and numerical settings, defaults both, assigns the five traits the constructor does not take, and initialises the integrator.

Usage

```
leaf_model(
  traits = leaf_traits(),
  control = leaf_control(),
  supply = leaf_supply_singlelayer()
)
```

Arguments

`traits`	a `leaf_traits()` object
`control`	a `leaf_control()` object
`supply`	how water reaches the root collar: `leaf_supply_singlelayer()` (the default) or `leaf_supply_multilayer()`

Details

The result is a stateful R6 object: set drivers with `set_drivers()`, choose a model with `$set_model()` if you want one other than the default, solve with `$optimise()`, then read the operating point off the object or with `operating_point()`. For a single solve you probably want `leaf_solve()` instead, which does all of that in one call.

Value

A `Leaf` R6 object.

See Also

`leaf_traits()`, `leaf_control()`, `leaf_supply_singlelayer()`, `set_drivers()`, `leaf_solve()`

Examples

```
l <- leaf_model()
set_drivers(l, psi_soil = 2.0, PPFD = 900)
l$optimise()
operating_point(l)
```

`leaf_solve` *Solve a leaf, in one call*

Description

Drivers in, operating point out. Vectorised over the scalar drivers, so a response curve is one call rather than a loop.

Usage

```
leaf_solve(
  psi_soil,
  PPFD = 900,
  soil_depth = NULL,
  root_network = NULL,
  leaf_specific_conductance_max = 3.14e-05,
  atm_vpd = 2,
  ca = 40,
  leaf_temp = 25,
  atm_o2_kpa = 21,
  atm_kpa = 101.3,
  traits = leaf_traits(),
  control = leaf_control(),
  supply = leaf_supply_singlelayer(),
  reuse = TRUE,
  model = "TF24_floor",
  route = NULL,
  CF77_lambda = NA_real_,
  TF24_floor_lambda_o = NA_real_,
  LeastCost_beta = NA_real_
)
```

Arguments

`psi_soil`	soil water potential, MPa, **positive magnitude**. Length sets the number of soil layers.
`PPFD`	INTERCEPTED photosynthetic photon flux density per unit leaf area, umol m^-2 s^-1: the light incident on the leaf, before its absorptance (`alpha_PAR_`). On the energy-balance path the leaf also needs `$NIR_int_`, the intercepted near-infrared in W m^-2, set before this call.
`soil_depth`	cumulative depth to the bottom of each layer, m. Defaults to 1 m layers.
`root_network`	the soil-to-collar hydraulic resistances, a `RootNetwork()`. **The same argument on both supply paths**: per rooted layer from `root_network_from_carbon()` on the multi-layer path, or one series resistance from `series_resistance()` on the single-potential path. Defaults to a nominal value on either — 20 kg C m^-2 leaf split evenly, or 1e3 — written out in the body where you can see and replace it. Stand-ins, not recommendations.

`leaf_specific_conductance_max`
: maximum leaf-specific hydraulic conductance, kg m^-2 s^-1 MPa^-1

`atm_vpd`
: atmospheric vapour pressure deficit, kPa. Must lie in `(0, e_s(leaf_temp))`, a relative humidity between 0 and 1; a deficit at or past saturation is refused.

`ca`
: atmospheric CO2 partial pressure, Pa

`leaf_temp`
: leaf temperature, deg C

`atm_o2_kpa`
: atmospheric O2 partial pressure, kPa

`atm_kpa`
: atmospheric pressure, kPa. Note this is not decorative: the ppm-to-Pa conversion is derived from it, and getting it wrong moves assimilation by percent, not by rounding.

`traits`
: a `leaf_traits()` object

`control`
: a `leaf_control()` object

`supply`
: how water reaches the root collar: `leaf_supply_singlelayer()` (the default) or `leaf_supply_multilayer()`. On the single-potential path `soil_depth` must be omitted, `root_network` is a `series_resistance()` (defaulted if omitted), and each `psi_soil` is one value rather than a profile.

`reuse`
: solve every row with one `Leaf` object (`TRUE`) or construct a fresh one per row (`FALSE`, the default). Reuse is faster because the vulnerability tables are built once, and it is safe – every exit from the solve writes its own outputs. `FALSE` exists so that can be checked rather than assumed, and the package's tests do check it.

`model`
: which optimality model to solve, as in `leaf_gradient()`: any name from `cost_curve_names()`, solved on `route`. The default, `"TF24_floor"`, is plant's default model; at its default price, `TF24_floor_lambda_o = 0`, it is TF24.

`route`
: which potential the curve is maximised over: `"leaf"` (the default), `psi_leaf` with the upstream potential pinned at `psi_soil`, which needs `leaf_supply_singlelayer()`; or `"collar"`, the root-collar potential, on either supply path, and the only route on `leaf_supply_multilayer()`. `NULL` is the leaf route.

  The two routes are different models, not two ways of asking one question, even for one curve. They optimise different variables over different supply topologies, and for TF24 disagree on 20 of 30 driver rows.

  On a leaf route the `collar` column comes back non-finite, because the collar is not solved for at all. An absent number is the truthful answer rather than a gap – reading one would be reading whatever the last solve left behind.

  Two curves take a PRESCRIBED price of water rather than deriving one: `CF77` reads `CF77_lambda` and `TF24_floor` reads `TF24_floor_lambda_o`, both below. `CF77`'s default is aligned to TF24 by `align_curves()`; `TF24_floor`'s is 0, where that curve is TF24.

`CF77_lambda`, `TF24_floor_lambda_o`
: the prescribed marginal value of water for the two curves that take one, in umol C (kg H2O)^-1. `NA_real_` (the default) leaves the field as it is:

`CF77`'s aligned default (see `align_curves()`), or `TF24_floor`'s 0, where that curve is TF24.

Their scale is set by the leaf, not chosen freely. At this package's defaults the leaf's own marginal cost of water runs 9e4 to 3e5, and a price far outside that band pins the optimum against a bound — where the answer describes the bracket rather than the model. `operating_point()`'s `lambda` is how you find a value that means something.

`LeastCost_beta`

the `LeastCost` curve's unit-cost ratio (Prentice et al. 2014's beta): the cost of maintaining carboxylation capacity relative to the cost of transpiration, dimensionless. Only `model = "LeastCost"` reads it. `NA_real_` (the default) keeps the leaf's own default, aligned to TF24 by `align_curves()`. Zero is refused, since at zero the capacity term vanishes and what is left is water-use efficiency, which rises without bound as transpiration falls. The published value is 146, which at this package's default leaf pins the leaf optimum at `psi_crit` from the reference `psi_soil` of 1 MPa drier.

Details

This is the entry point for someone who would otherwise reach for `plantecophys::Photosyn()`. The difference in what comes back is the point of the package: `gc` here is not from a fitted conductance model, it is what falls out of maximising profit over an explicit hydraulic path, and `g1_eff` reports the Medlyn `g1` that would have been needed to reproduce it.

Value

A data.frame with one row per driver combination: the drivers, then the columns of `operating_point()`.

`leaf_temp` is a driver, `Tleaf` is an output

Two temperature columns come back and they are not the same thing. `leaf_temp` is the driver you supplied. `Tleaf` is the temperature the reported fluxes were computed at.

By default they are equal, because the default path takes leaf temperature as prescribed. With the Penman-Monteith energy balance on (`$use_energy_balance_ <- TRUE`) they are not: `leaf_temp` is then **air** temperature, and `Tleaf` is solved per operating point from the leaf's own transpiration — so it rises as the stomata close, and it is the column to plot assimilation against.

Performance, and what the lever actually is

About **20 µs per row**, of which the model is **2.8 µs**. The rest is the R boundary: each call into C++ costs ~1.1 µs and a solved row needs several.

Two consequences, both the opposite of what you might expect:

- **This is the fast path, not the convenient-but-slow one.** It is within 6% of building a `Leaf` yourself and looping `set_drivers()` + `$find_root_collar_psi()` + `operating_point()`. Reaching into the object saves about 1 µs a row and is worth doing for access to intermediate state, not for speed.

- **The lever is fewer R calls per row, not a different R function.** Pass all your drivers to one vectorised call rather than looping in R. If you need more than that – a fit's inner loop, say – what has to go is the boundary itself, which means C++.

`reuse = TRUE` is the default because constructing a `Leaf` from R costs about **180x a trivial** `.Call` (~230 µs where this was measured), or 45 solves, and only a small part of that is the vulnerability tables; the rest is R-side object construction. See `set_traits()` for varying traits without reconstructing, and note it costs many solves rather than almost nothing when the trait you change is `stem_c` or `root_c`, which own a table.

Recycling

`PPFD`, `atm_vpd`, `ca`, `leaf_temp`, `atm_o2_kpa`, `atm_kpa` and `leaf_specific_conductance_max` are recycled to a common length, the usual R way. `psi_soil` is recycled too **when it is a plain numeric vector** – each element is then a separate single-layer solve. For multi-layer solves pass a list of numeric vectors, one per row; `soil_depth` follows the same rule. `root_network` is either one `RootNetwork()`, used for every row, or a list of them, one per row.

See Also

`leaf_model()` for the stateful interface.

Examples

```
# one point
leaf_solve(psi_soil = 2.0, PPFD = 900)

# a light response curve
leaf_solve(psi_soil = 1.0, PPFD = seq(100, 1800, length.out = 6))

# a drought response, and the effective g1 it implies
d <- leaf_solve(psi_soil = seq(0.5, 5, length.out = 6), PPFD = 900)
d[, c("psi_soil", "A", "gc", "g1_eff")]

# two three-layer profiles, on a root network and the collar route
leaf_solve(psi_soil = list(c(1, 1.5, 2), c(3, 3.5, 4)), PPFD = 900,
           supply = leaf_supply_multilayer(), route = "collar")

# a different optimality model
leaf_solve(psi_soil = 1.5, PPFD = 900, model = "SOX")
```

`leaf_supply_singlelayer` *Choose how water reaches the root collar*

Description

Two supply paths, and the choice belongs with the model rather than with the drivers, so it is made once in `leaf_model()`.

Usage

```
leaf_supply_singlelayer(gravity_head = 0)

leaf_supply_multilayer()
```

Arguments

`gravity_head`	head required to lift water to the collar, MPa. Zero by default, which is right for a bare leaf that is not thinking about rooting depth. See the section above for why this, alone, is not a driver.

Details

`leaf_supply_singlelayer()` is the default. `leaf_supply_multilayer()` is what plant uses: a soil profile of one or more layers, each with its own water potential, feeding a root network whose resistances are derived from a root carbon profile.

`leaf_supply_singlelayer()` collapses the whole soil-to-collar path to **one resistance**. This is what a leaf physiologist arriving from `plantecophys` or `tealeaves` actually has: a soil water potential and no root-mass profile. It is also what makes comparison against other optimality models meaningful, since Medlyn, Prentice least-cost and Cowan-Farquhar are all formulated against a single soil potential — comparing them against a multi-layer root network compares two things at once.

Value

A `leaf_supply` object, for the `supply` argument of `leaf_model()` and `leaf_solve()`.

Both paths are configured the same way, and take the same drivers

Neither function takes a resistance. The soil-to-collar resistance is a per-call **driver** on both paths, supplied to `set_drivers()` as `root_network` — `root_network_from_carbon()` on the multi-layer path, `series_resistance()` on this one, so the same quantity arrives the same way on both paths, and setting it resets nothing.

`gravity_head` **is the one asymmetry left, and it is deliberate.** The multi-layer path derives a per-layer head from the depth profile it is handed (`gravity_head * z_soil_mid`); this path has no depth profile to derive one from, and a bare leaf wants **zero** rather than a geometric default — which is precisely the caller `leaf_supply_singlelayer()` exists for. Making it a driver here would mean either inventing a depth for a leaf that has none, or adding a second supply-shaped argument to `set_drivers()` that only one path reads. If you do want the multi-layer rule for a single layer of thickness `d`, pass `gravity_head = 0.00981 * d / 2`.

Why this is not a settable field

The obvious R interface would be `leaf$supply_kind <- "single"`. It is not offered, because flipping the tag on its own leaves the other path's state configured and silently ignored, and flipping back makes it stale rather than absent. Both entry points here reconfigure the object completely, so the tag and the supply state can never disagree. The cost is that

switching clears the solved state and the drivers must be set again — which is honest, since the two paths read different inputs.

The two ends of this path are in different unit bases

Read this before parameterising a whole soil-to-leaf path, which is exactly what this supply path is for.

- `leaf_specific_conductance_max` is **kg** m^-2 s^-1 MPa^-1
- `series_resistance()`'s `resistance` is MPa s **mol**^-1 m^2

So the two quantities the package presents as the two ends of one series are on opposite sides of a kg-per-mol factor, and a caller carrying the whole path has to supply it. The calibration study recorded dropping that factor of 0.018 as its original error, **worth three orders of magnitude** — which is why it ended up named in their code rather than inlined.

The factor is `molar_mass_h2o` = 0.018015 kg/mol. Nothing here can check you applied it: both quantities are just positive numbers, so a path built on the wrong basis returns a plausible operating point, off by ~55x one way or ~0.018x the other.

See Also

`series_resistance()` for the resistance itself, which is a driver.

Examples

```
# a bare leaf: one soil potential, one resistance, no root profile
leaf_solve(psi_soil = 1.5, PPFD = 900, supply = leaf_supply_singlelayer(),
           root_network = series_resistance(1e3))
```

`leaf_traits` *Physiological traits for a leaf*

Description

The traits, and only the traits. Numerical settings live in `leaf_control()`; the C++ constructor mixes the two and this is the split.

Usage

```
leaf_traits(
  vcmax_25 = 96,
  stem_c = 2.680147,
  stem_P50 = 3.4,
  root_c = 2.680147,
  root_P50 = 3.4,
  TF24_beta2 = 1.5,
  jmax_25 = 157.44,
  phi_J = 0.3/0.85,
```

```
  curv_fact_elec_trans = 0.7,
  curv_fact_colim = 0.99,
  TF24_cost_scale = 7.5,
  R_d_25 = 1.44,
  JS22_gamma = 0.4717887793,
  CMax_a = 0.6385596982,
  CMax_b = 0,
  g_min = 0
)
```

Arguments

vcmax_25	maximum carboxylation rate at 25 C (umol m^-2 s^-1)
stem_c	shape parameter of the stem vulnerability curve (unitless)
stem_P50	stem potential at 50% loss of conductivity (MPa, positive magnitude). stem_b and psi_crit are derived from this pair and are readable but not settable on the resulting Leaf.
root_c	shape parameter of the root vulnerability curve (unitless)
root_P50	root potential at 50% loss of conductivity (MPa, positive magnitude), with root_b and root_psi_crit derived from it.
TF24_beta2	exponent for the effect of hydraulic risk (unitless)
jmax_25	maximum electron transport rate at 25 C (umol m^-2 s^-1)
phi_J	quantum yield of electron transport per ABSORBED photon (mol mol^-1). The leaf absorbs alpha_PAR_ (0.85 by default) of the intercepted PPFD, so the default 0.30 / 0.85 is the old incident-basis yield of 0.30 restated. A gas-exchange fit identifies only phi_J * alpha_PAR_; fix alpha_PAR_ to make the fitted value per absorbed photon. An incident-basis yield a is phi_J = a / 0.85.
curv_fact_elec_trans	curvature of the light response curve (unitless)
curv_fact_colim	curvature of the colimited photosynthesis equation
TF24_cost_scale	cost parameter for the TF24 profit model (umol m^-2 s^-1)
R_d_25	dark respiration at 25 C (umol m^-2 s^-1). It is the value at 25 C only: respiration rises from there on Tjoelker's declining-Q10 curve.
JS22_gamma	Joshi & Stocker (2022)'s hydraulic unit cost (umol C m^-2 s^-1 MPa^-2), read only by the JS22 cost curve and returning an exactly zero gradient on every other one. **The default is not a literature value.** It is aligned to TF24 by align_curves(): the value at which the JS22 leaf optimum has TF24's conductance at that function's reference environment. No single value matches more than one soil potential: the JS22_gamma reproducing TF24's cost at its own optimum runs 0.287 at psi_soil 0.5 MPa to 1.761 at 3.0 MPa. TF24's cost tracks the absolute potential and this one tracks the drop, and across a drydown those move in opposite directions.

`CMax_a`, `CMax_b` slope and intercept of the `CMax` marginal cost, `dC/dpsi = CMax_a * psi + CMax_b`, following Wolf et al. (2016) as Anderegg et al. (2018) parameterised it. Read only by the `CMax` curve. `CMax_a`'s default is aligned to TF24 by `align_curves()`, with `CMax_b` at zero; neither default is a literature value.

`CMax_b` **is signed, and negative in the source convention.** Sabot's `TractLSM` stores it with the sign inverted "so as to get a positive parameter value", so a value taken from that code or from a calibration against it must have its sign checked rather than assumed.

Neither default is a literature value. With `CMax_b = 0` the `CMax_a` reproducing TF24's cost at its own optimum runs 0.406 at `psi_soil` 0.5 MPa to 0.807 at 3.0 MPa — a 2.0x range against `JS22_gamma`'s 6.1x, which is the absolute-versus-drop distinction as a number.

`g_min` the residual (cuticular) conductance to water vapour at 25 C (mol H2O m^-2 s^-1). A leaf keeps losing water through its cuticle after its stomata have shut, and the cuticle lets CO2 through as well (at the ratio `$g_min_co2_ratio_`, 0.05 by default). The leaf optimises its STOMATAL flux; the xylem carries the total, `E = E_s + E_r` with `E_r = g_min(T_leaf) D_leaf / P_atm`, and every cost that reads the flux charges the total. Default 0, at which nothing moves.

A measured cuticular conductance, not a fitted `g0`. A `g0` fitted to gas exchange is an order of magnitude larger and is mostly stomata that have not finished closing; measured cuticular conductances run 0.25-0.8 mmol m^-2 s^-1 (2.5e-4 to 8e-4 here). The Medlyn route's `$g0` is the fitted quantity and is untouched by this trait. `g_min` rises with leaf temperature on a smooth curve whose three parameters are fields on the `Leaf` (`$g_min_k_low_`, `$g_min_k_high_`, `$g_min_T_trans_`, `$g_min_T_width_`); their defaults are placeholders, inert at 25 C.

Details

Defaults are *Eucalyptus saligna*, and are the same values `phylloptim::Leaf`'s default constructor uses.

Value

A `leaf_traits` object; a named list.

Two vulnerability curves, not one

`stem_P50`/`stem_c` describe the STEM Weibull curve, which drives the hydraulic cost; `root_P50`/`root_c` describe the ROOT curve, which drives uptake. They are separate parameters with separate meanings, and named so: an unmarked pair is easy to read as the other curve's.

Each curve has exactly TWO settable numbers. `stem_b`, `psi_crit`, `root_b` and `root_psi_crit` are quantiles of the curve those two define – readable on the resulting `Leaf` and not settable anywhere. See `weibull_p50_c()`.

psi_crit is not a free trait

`psi_crit` is the stem curve's **P95** and is derived from `stem_P50`/`stem_c`, not set: a free `psi_crit` would describe a curve it was not derived from. It is a labelled point on a curve that continues past it, and the dry end of the solve only for SOX, JW26, CF77 and LeastCost, whose own forms end there; for the other curves the stem may run past it (#183).

At the defaults:

```
stem_b = 3.898245, stem_c = 2.680147
3.898245 * log(1/0.05)^(1/2.680147) = 5.870283 = psi_crit
```

to six decimal places. It moves with the pair, so there is nothing left to set inconsistently. `vignette("fitting")` derives the pair from a published P50/P88 pair, which is the right way round.

The root-resistance constants are not traits

`beta_R_H` and `beta_R_V` parameterise the root-architecture model, which the leaf does not run – it takes the resistances. They are arguments to `root_network_from_carbon()`.

So `leaf_gradient()` does not differentiate with respect to either. Perturbing them is cheap, because `root_network_from_carbon()` is homogeneous of degree 1 in each, but the derivative of a solved output still needs two solves either side.

See Also

`leaf_control()`, `leaf_model()`, `leaf_solve()`

Examples

```
leaf_traits()
# A more brittle stem. Only the curve's own two parameters are set: the scale
# `stem_b` and the critical potential `psi_crit` are quantiles of that curve
# and are derived from them, so they cannot be set inconsistently.
leaf_traits(vcmax_25 = 120, stem_P50 = 2.2)
```

`model_registry` *The model registry*

Description

phylloptim's leaf as one record: every equation (its LaTeX, the C++ symbol that evaluates it, the parameters it reads, the bound fields it writes) and every parameter (symbol, units, class, meaning, source, placeholder flag, the symbol that holds it). The file is `inst/model-registry.yml`; documents are generated from it rather than written beside it, and CI checks it (`tests/testthat/test-registry.R`, `docs.yml`). Its header states the three rules it is kept by: no stored line numbers or defaults, validated in CI, one registry per package.

Usage

```
model_registry(
  path = system.file("model-registry.yml", package = "phylloptim")
)
```

Arguments

`path`	the YAML file; the package's own by default

Value

an object of class `model_registry`: `package`, `sections` (a data frame of `id` and `title`), `parameters` and `equations` (data frames, one row each, with list-columns for `reads`, `writes`, `also` and `aliases`), `diagnostics` (the bound fields that are neither), `raw` (the validated list) and `path`

See Also

`registry_values()` for the values at render time, `registry_resolve()` for file, line and API link from Doxygen's XML

Examples

```
reg <- model_registry()
reg
subset(reg$equations, section == "respiration", c(id, title, code))
```

`operating_point` *The solved operating point, as one row*

Description

Reads the outputs off a leaf that has been solved. Call after `$optimise()`; before that the values are the missing-value sentinels the object was constructed with.

Usage

```
operating_point(x)
```

Arguments

`x`	a solved `Leaf`

Details

Cheap enough to use in a loop: one call into C++ for every output, and no `data.frame()` per row (the README's "Performance from R" has the figure).

Value

A one-row data.frame.

What the cost columns mean

`hydraulic_cost` is the whole cost a difference objective subtracted. Each curve states which part of its cost is carbon the plant spent; `shadow_cost` is the rest, a *price*: the value of water, or of a drop in potential, in its best alternative use. `carbon` is `A` less the carbon part, the carbon the leaf kept:

```
carbon = A - (hydraulic_cost - shadow_cost)
       = profit + shadow_cost       on a difference curve
```

A consumer that grows a plant on this leaf wants `carbon`, never `profit`: deducting a price from a carbon budget taxes growth by something the plant never spent.

Only TF24's hydraulic cost is carbon spent. So `shadow_cost` is zero on `TF24`, whose `carbon` is its `profit`; `TF24_floor_lambda_o * E` on `TF24_floor`; and the whole cost on `CF77`, `JS22` and `CMax`, whose costs are prices. `SOX`, `JW26`, `LeastCost` and `ProfitMax` have no cost in carbon units to split, so their `shadow_cost` is `NA` and their `carbon` is `A`.

See Also

`leaf_solve()`, which does the whole thing in one call.

Examples

```
l <- leaf_model()
set_drivers(l, psi_soil = 2.0, PPFD = 900)
l$optimise()
operating_point(l)
```

---

plot_model_overview *Overview figure: how the parts of the model fit together*

Description

Draws the four zones of the model in one figure: the profit panel that everything else serves, the supply-equals-demand root-find inside the benefit, the causal wiring between the parts, and the soil-to-leaf path the equations describe.

Usage

```
plot_model_overview(model = NULL, cex = 1)
```

Arguments

`model`	A `Leaf`, already driven. Defaults to a leaf at `psi_soil = 1` MPa and `PPFD = 1500`, which is the reference state used throughout `vignette("the-models")`.
`cex`	Overall text scaling, applied on top of the per-element sizes. Raise it for a slide, lower it for a two-column figure.

Details

The two left-hand panels are computed from `model`, so they move if the model does. The two right-hand zones are schematic.

The layout follows figure 1 of Westoby et al. (2012), which pairs a data panel with a box-and-arrow flow and a physical cartoon so that the reader can see the same object three ways.

Value

`NULL`, invisibly. Called for the plot.

References

Westoby, M., Falster, D.S. and Moles, A.T. (2012) Trade-offs between height growth rate, stem persistence and maximum height among plant species in a post-fire succession. *Journal of Theoretical Biology*.

Examples

```
plot_model_overview()

# a vector figure for a manuscript
## Not run:
pdf("overview.pdf", width = 11, height = 6.6)
plot_model_overview()
dev.off()

## End(Not run)
```

`plot_solver_convergence` *Convergence of the two ways to locate an optimum*

Description

Draws two panels for `vignette("the-models")`: the error in the located operating point, and the first-order residual at it, both against the number of model evaluations spent. One series solves `dJ/dpsi = 0` by Brent's method; the other searches `J` for its maximum by golden section.

Usage

```
plot_solver_convergence(psi_soil = 1, PPFD = 1500, atm_vpd = 1.5, n_eval = 60)
```

Arguments

psi_soil, PPFD, atm_vpd	Drivers giving an interior optimum.
n_eval	Evaluations to trace for the search.

Details

The iterations run in R against the package's own objective and derivative; only the outer loop is reproduced, because the C++ solvers do not return their iterates. See the file's header comment.

Value

Invisibly, a list with the two traces and the reference optimum.

---

psi_at_plc *The water potential at a given loss of conductivity*

Description

Inverts `f(psi) = 2^-(psi/P50)^c` for the potential at which a given fraction of conductivity has been lost.

Usage

```
psi_at_plc(P50, c, plc)
```

Arguments

P50	potential at 50% loss, MPa as a positive magnitude
c	shape parameter of the Weibull curve (unitless)
plc	proportion of conductivity LOST, in `(0, 1)`. `plc = 0.5` returns `P50` by construction; `0.88`, `0.95` and `0.99` return P88, P95 and P99.

Value

The potential, MPa as a positive magnitude.

See Also

`weibull_p50_c()` for the reverse, and `leaf_traits()` for where the result goes.

Examples

```
psi_at_plc(3.4, 2.680147, 0.5)   # returns P50
psi_at_plc(3.4, 2.680147, 0.95)  # P95, which is also `psi_crit`
```

`registry_locations` *Where every symbol is, from Doxygen's XML*

Description

Walks the compounds Doxygen's `index.xml` lists and returns one row per member: its qualified name, kind, file and line (declaration), body span, the anchor of its page in the HTML render, its initializer where it has one (a constexpr's value), and the symbols it references. Lines are the source's, through `registry_line_map()`. A function's references include the `Leaf` members it reads through a `Leaf&` parameter (`l.TF24_cost_scale`, as each curve's `Curve<K>` does), read from its body: Doxygen 1.9.8, which CI runs, records none of them.

Usage

```
registry_locations(
  xml_dir,
  include_dir = file.path(dirname(dirname(xml_dir)), "inst", "include"),
  filter = file.path(dirname(dirname(xml_dir)), "tools", "doxygen_filter.awk")
)
```

Arguments

`xml_dir`	Doxygen's XML output directory
`include_dir`	the directory the XML's file paths are relative to (`STRIP_FROM_PATH` in the Doxyfile): the package's `inst/include`
`filter`	the awk filter Doxygen read the headers through

Details

Needs a render with `GENERATE_XML = YES` and `REFERENCES_RELATION = YES`, which the package's `Doxyfile` sets: `doxygen` at the package root writes `docs/xml`.

Value

a data frame with columns `symbol`, `kind`, `id`, `compound`, `file`, `line`, `body_file`, `body_start`, `body_end`, `url`, `initializer` and the list-column `references`; the body is in `body_file`, which differs from `file` for a member defined out of line

`registry_resolve` *Resolve the registry's symbols against Doxygen's XML*

Description

Joins every parameter and equation to the file, line, body span, page anchor and initializer of the symbol its `code` names (and, for equations, of each symbol in `also`). Fails naming every symbol that does not resolve: a rename in the headers that the registry did not follow.

Usage

```
registry_resolve(
  raw,
  xml_dir = NULL,
  locations = NULL,
  unreferenced = c("warn", "error", "none"),
  ...
)
```

Arguments

`raw`	a registry list from `registry_read()`, or a `model_registry`
`xml_dir`	Doxygen's XML output directory
`locations`	the output of `registry_locations()`, if already in hand
`unreferenced`	what to do about a read the function does not reference: `"warn"`, `"error"` or `"none"`
`...`	passed to `registry_locations()`

Details

It also checks what each equation reads against what its function references, which Doxygen records: a `reads` entry whose parameter symbol (or one of its `aliases`) is referenced by none of the equation's functions is reported, as a warning by default. Literals are exempt, being written inside the function rather than referenced by it.

Value

a list: `parameters` and `equations` (data frames, with the location columns added), `locations` (every symbol in the XML), `unresolved` (character, empty on success) and `unreferenced` (a data frame of equation id and parameter id)

registry_values *The parameters' values, read at render time*

Description

Adds `value` and `value_source` to the registry's parameters: a bound field is read off `leaf`, a driver off `set_drivers()`'s default, a constexpr off its initializer in Doxygen's XML when `locations` is given, and a literal off the registry (the one kind of number it stores). Anything else is `NA`. Nothing is read from a comment or a document, which is rule 1 of the registry's header.

Usage

```
registry_values(registry, leaf = leaf_model(), locations = NULL)
```

Arguments

`registry`	a `model_registry()`
`leaf`	the `Leaf` to read bound fields off; `NULL` reads none
`locations`	the output of `registry_locations()`, for the constexprs

Value

the registry's `parameters` data frame with `value` and `value_source` (`"binding"`, `"binding (vector)"`, `"set_drivers()"`, `"initializer"`, `"literal"` or `NA`), and the list-column `value_vector` holding a per-layer field whole (its `value` is then `NA`)

Examples

```
v <- registry_values(model_registry())
v[v$section == "respiration", c("id", "symbol", "value", "value_source")]
```

root_network_from_carbon *Root resistances from a root carbon profile*

Description

The root-architecture model, which the caller runs: `set_physiology()` takes the resistances it returns. Each layer's root carbon is split 1/3 vertical : 2/3 horizontal; the minimum horizontal resistance is `beta_R_H / c_r_h`, and the vertical resistance is `beta_R_V * dz[i]^2 / c_r_v`, accumulated from the surface down, where `dz[i]` is the thickness of layer `i`.

Usage

```
root_network_from_carbon(
  root_carbon_per_leaf_area,
  soil_depth,
  beta_R_H = 340,
  beta_R_V = 9400
)
```

Arguments

`root_carbon_per_leaf_area`	root carbon **per unit leaf area**, kg C m^-2 leaf, one entry per soil layer. Per unit leaf area because the leaf is purely intensive; passing absolute carbon gives resistances that are wrong by the leaf area and no error anywhere will say so.
`soil_depth`	cumulative depth to the bottom of each layer, m, strictly increasing. Layer thicknesses are the differences between consecutive entries, with an implicit 0 at the surface, so the layers need not be equal: `soil_depth = c(0.02, 0.3, 1.5)` is a 2 cm surface layer over two thicker ones. This takes the profile rather than the widths so that a caller reading a depth profile cannot get the differencing wrong.
`beta_R_H`	proportionality constant between minimum horizontal (intralayer) root hydraulic resistance and `C_r^-1` (MPa s mol C / mol H2O).
`beta_R_V`	proportionality constant between minimum vertical (interlayer) root hydraulic resistance and `dz[i]^2/C_r` (MPa mol C s / mol H2O / m^2).

Details

The result is sized to the deepest layer with non-zero root carbon, so it is shorter than `soil_depth` for a plant that is shallower than the soil profile. That is what `max_soil_layer` reports after a solve.

Value

A `RootNetwork()`.

This is a model, not a conversion

The 1/3 : 2/3 split, the `dz[i]^2` scaling and both `beta_R_*` constants are choices, made in plant's TF24 strategy and calibrated there for *Eucalyptus saligna*. Nothing in the leaf solve depends on them – it reads the resistances. If you have measured root resistances, or are fitting them, build a `RootNetwork()` directly and do not come through here.

See Also

`RootNetwork()`, `set_drivers()`

Examples

```
root_network_from_carbon(rep(20 / 3, 3), soil_depth = 1:3)
```

RootNetwork *Root resistance network*

Description

Root hydraulic resistances, per layer and per unit leaf area

Usage

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

Arguments

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

Details

What `set_physiology()` takes in place of a root carbon profile (#33). Which root-architecture model turned carbon into these resistances is not this package's business, exactly as which conductance-versus-height model produced `leaf_specific_conductance_max` never was. If you do have a carbon profile, `root_network_from_carbon()` is the model that used to run inside the leaf.

`r_R_H_min` and `r_R_V_sum` are the two fields the solve reads. Their common length is the number of ROOTED layers, which may be fewer than the number of soil layers but never more. `c_r_V`, `c_r_H` and `r_R_V` are diagnostics and may be left empty.

---

series_resistance *A single soil-to-collar series resistance, as a supply driver*

Description

The single-potential path's counterpart to `root_network_from_carbon()`: it packages one series resistance into the `RootNetwork()` that `set_drivers()` takes, so the two supply paths are driven through the same argument.

Usage

```
series_resistance(resistance)
```

Arguments

`resistance`	the whole soil-to-collar path as one series resistance, **per unit leaf area**, MPa s (mol H2O)^-1 m^2 leaf. Must be positive: a zero resistance is an infinite flux. Per unit leaf area because the leaf is purely intensive — nothing here may scale with plant size. `set_drivers()` refuses one below 1e-2, where the collar route can no longer resolve the flux; the leaf routes are the zero-resistance model.

Details

It goes in `r_R_V_sum` rather than `r_R_H_min`, and that is exact rather than a convention: `r_R_V_sum` already means "series resistance from the surface down to this layer, with no vulnerability weighting", and this is that quantity with one layer. `r_R_H_min` is the vulnerability-weighted horizontal term, which this path does not have — supplying a non-zero one is an error rather than ignored, since it would otherwise silently drop a resistance you meant to use.

Value

A `RootNetwork()` with one entry in `r_R_V_sum`.

See Also

`leaf_supply_singlelayer()`, `root_network_from_carbon()`, `set_drivers()`

Examples

```
series_resistance(1500)
leaf_solve(psi_soil = 1.5, supply = leaf_supply_singlelayer(),
           root_network = series_resistance(1500))
```

`set_drivers` *Set the drivers for a leaf*

Description

Named and defaulted, over the C++ `set_physiology()`'s ten positional arguments. Modifies `x` in place and returns it invisibly, so it can be piped.

Usage

```
set_drivers(
  x,
  psi_soil,
  PPFD = 900,
  soil_depth = NULL,
  root_network = NULL,
  leaf_specific_conductance_max = 3.14e-05,
```

```
  atm_vpd = 2,
  ca = 40,
  leaf_temp = 25,
  atm_o2_kpa = 21,
  atm_kpa = 101.3
)
```

Arguments

x	a Leaf, from leaf_model()
psi_soil	soil water potential, MPa, **positive magnitude**. Length sets the number of soil layers.
PPFD	INTERCEPTED photosynthetic photon flux density per unit leaf area, umol m^-2 s^-1: the light incident on the leaf, before its absorptance (alpha_PAR_). On the energy-balance path the leaf also needs $NIR_int_, the intercepted near-infrared in W m^-2, set before this call.
soil_depth	cumulative depth to the bottom of each layer, m. Defaults to 1 m layers.
root_network	the soil-to-collar hydraulic resistances, a RootNetwork(). **The same argument on both supply paths**: per rooted layer from root_network_from_carbon() on the multi-layer path, or one series resistance from series_resistance() on the single-potential path. Defaults to a nominal value on either — 20 kg C m^-2 leaf split evenly, or 1e3 — written out in the body where you can see and replace it. Stand-ins, not recommendations.
leaf_specific_conductance_max	maximum leaf-specific hydraulic conductance, kg m^-2 s^-1 MPa^-1
atm_vpd	atmospheric vapour pressure deficit, kPa. Must lie in (0, e_s(leaf_temp)), a relative humidity between 0 and 1; a deficit at or past saturation is refused.
ca	atmospheric CO2 partial pressure, Pa
leaf_temp	leaf temperature, deg C
atm_o2_kpa	atmospheric O2 partial pressure, kPa
atm_kpa	atmospheric pressure, kPa. Note this is not decorative: the ppm-to-Pa conversion is derived from it, and getting it wrong moves assimilation by percent, not by rounding.

Value

x, invisibly.

Water potentials are positive magnitudes

psi_soil is a positive number of MPa. There is one representation for water potential throughout this package and it is asserted at the input boundary rather than documented: a negative entry is an error. That check is the only thing standing between a script written against the old signed convention and a plausible wrong number, so this wrapper surfaces it rather than smoothing it over – it does not take abs().

Multiple soil layers

Pass vectors for `psi_soil`, and optionally for `soil_depth`. Both must end up the same length; by default `soil_depth` becomes 1 m layers.

The supply resistances, on either path

The leaf takes the soil-to-collar hydraulic RESISTANCES, not the root carbon they may have been derived from. The solve reads two vectors of them, `r_R_H_min` and `r_R_V_sum`, and nothing in it knows about root carbon, layer thickness, the 1/3 : 2/3 vertical/horizontal split or either `beta_R_*` constant, so a caller without a root carbon profile needs none.

`root_network` **is the same argument on both supply paths**, which is the point. On the multi-layer path it carries one `r_R_H_min` and one `r_R_V_sum` per rooted layer; on the single-potential path it carries one series resistance in `r_R_V_sum` and nothing else, which is that field's own meaning with one layer and no vulnerability-weighted term. `root_network_from_carbon()` builds the first, `series_resistance()` the second.

Both paths default it, because neither should force a caller to own a supply model they do not have, and both defaults are written out in this function's body rather than buried in a signature: a nominal 20 kg C m^-2 leaf through `root_network_from_carbon()` on the multi-layer path, and a nominal 1e3 MPa s (mol H2O)^-1 m^2 leaf series resistance on the single-potential one. Both are stand-ins, not recommendations.

`soil_depth` is the one argument that is genuinely multi-layer-only, and passing it on the single-potential path is an error rather than ignored: there is no depth profile there for anything to read. That is also why `leaf_supply_singlelayer()` keeps `gravity_head` — see its documentation for the one remaining asymmetry between the two paths, and why it is left in place.

See Also

`leaf_model()`, `leaf_solve()`

Examples

```
l <- leaf_model()
set_drivers(l, psi_soil = 2.0, PPFD = 900, atm_vpd = 1.5)
# three drying layers, on a root network
ml <- leaf_model(supply = leaf_supply_multilayer())
set_drivers(ml, psi_soil = c(1.0, 1.5, 2.0), PPFD = 900)
```

---

set_traits *Replace the traits on an existing leaf*

Description

Changes the traits of a `Leaf` in place, leaving `leaf_control()`'s numerical settings alone. This is the fast path for anything that varies traits – calibration, sensitivity analysis, a trait response curve – because constructing a `Leaf` from R costs about 45 times as much as solving one.

Usage

```
set_traits(x, traits)
```

Arguments

x	a Leaf, from leaf_model()
traits	a leaf_traits() object

Value

x, invisibly.

You must set the drivers again afterwards

This returns the leaf to its just-constructed state, so set_drivers() has to be called before the next solve. That is not caution: vcmax_, jmax_ and R_d_ are derived from the traits inside set_physiology(), so they are genuinely unknown until the drivers are re-supplied.

The reason it is a function rather than sixteen assignable fields is that leaf$vcmax_25 <- x could not be made correct. Three separate pieces of derived state go stale on a bare trait write: the two pre-integrated vulnerability tables, the solved operating point, and – least visibly – vcmax_/jmax_/R_d_, which set_physiology() computes behind a cache. The cache is keyed on the traits it reads, so setting the drivers again refreshes it, but nothing short of set_traits() refreshes the tables or the solved point.

See Also

leaf_model(), set_drivers(), leaf_gradient()

Examples

```
l <- leaf_model()
set_traits(l, leaf_traits(vcmax_25 = 120))
set_drivers(l, psi_soil = 2.0, PPFD = 900)
l$optimise()
operating_point(l)
```

---

weibull_p50_c *Recover (P50, c) from any two published quantities*

Description

A vulnerability curve has two parameters, so **any two** of the quantities below determine it. This solves for (P50, c) from whichever pair you have.

Usage

```
weibull_p50_c(
  P50 = NULL,
  P88 = NULL,
  P95 = NULL,
  P99 = NULL,
  px = NULL,
  plc = NULL,
  S50 = NULL,
  c = NULL
)
```

Arguments

`P50`, `P88`, `P95`, `P99`	potentials at 50%, 88%, 95% and 99% loss of conductivity, MPa as positive magnitudes
`px`, `plc`	an arbitrary quantile: the potential `px` at which the proportion `plc` of conductivity is lost. Supply both or neither; the pair counts as ONE of the two quantities.
`S50`	slope at P50, % PLC per MPa. See `weibull_s50()`.
`c`	shape parameter, if you already have it.

Value

A list with `P50` and `c`, suitable for splicing into `leaf_traits()`.

Which pairs are closed-form

All of them except `S50` paired with a quantile other than P50, which is transcendental in `c` and is solved by a root-find over `c` in `(0, 20]`. Everything else follows from

$$P_f = P_{50}\left(\log_2(1/f)\right)^{1/c}$$

with `f` the remaining fraction (`f = 1 - PLC`), so `log2(1/f)` is 1, 3.0589, 4.3219 and 6.6439 for P50, P88, P95 and P99. From two quantiles, `c = ln(L2/L1) / ln(P_f2/P_f1)`.

Guards

`c > 0` is required. A `c` below 1 is **warned about rather than refused**: it is possible, but across measured angiosperm stems it usually flags a measurement artefact — native embolism already present at low tension flattens the curve — rather than a real trait. For reference, `c = 2.0` is a reasonable default for angiosperm stems, measured 1.8–2.6 across three species whose P50 spanned 2.9–4.3 MPa, i.e. shape roughly conserved while scale varied freely.

See Also

`psi_at_plc()`, `weibull_s50()`

Examples

```
# From a published P50 and P88 pair
weibull_p50_c(P50 = 3.4, P88 = 5.0)

# From P50 and a slope
weibull_p50_c(P50 = 3.4, S50 = 39.4)

# From an arbitrary quantile plus a shape
weibull_p50_c(px = 5.87, plc = 0.95, c = 2.680147)
```

---

weibull_p50_from_b *Convert a Weibull scale parameter to P50*

Description

`b` is the scale of the conventional form `exp(-(psi/b)^c)`; this package is parameterised on `P50` instead. The two are one substitution apart, `b = P50 / (ln 2)^(1/c)`, so

$$P_{50} = b\,(\ln 2)^{1/c}$$

Exported because published fits and older datasets carry `b`, and the alternative is that conversion being pasted at each call site. `Leaf` reports `stem_b` and `root_b` as derived read-only accessors, which is the forward direction of the same identity.

`P50` **is always SMALLER than** `b`, by a factor of `(ln 2)^(1/c)` — 0.87 at `c = 2.68`. So passing a `b` where a `P50` is wanted, or the reverse, gives a plausible number roughly 15% out rather than an error. Neither function can detect it: both arguments are positive potentials in MPa.

Usage

```
weibull_p50_from_b(b, c)
```

Arguments

`b`	scale parameter of the Weibull curve, MPa as a positive magnitude
`c`	shape parameter of the Weibull curve (unitless)

Value

`P50`, MPa as a positive magnitude.

See Also

`weibull_p50_c()`, `psi_at_plc()`

Examples

```
weibull_p50_from_b(3.898245, 2.680147)   # the package default, 3.4
```

`weibull_s50` *The slope of the vulnerability curve at P50*

Description

`S50 = 50*c*ln(2)/|P50|`, in % PLC per MPa, which is the other form vulnerability curves are commonly published in.

`S50` **is not an independent shape parameter** — it has `P50` built into it by construction, so a curve is not determined by `S50` alone. It is exported rather than kept internal for exactly that reason: a caller reading a published `S50` needs to see what it depends on.

Usage

```
weibull_s50(P50, c)
```

Arguments

`P50`	potential at 50% loss, MPa as a positive magnitude
`c`	shape parameter of the Weibull curve (unitless)

Value

The slope at P50, % PLC per MPa.

See Also

`weibull_p50_c()`, which accepts `S50` as one of its two inputs.

Examples

```
weibull_s50(3.4, 2.680147)
```

`with_phylloptim_conditions` *Run leaf code with classified error conditions*

Description

Evaluates `expr` and, if it fails with an error the model has tagged as infeasible, re-signals it as a `phylloptim_infeasible` condition carrying a machine-readable `code`. Anything else propagates unchanged.

Usage

```
with_phylloptim_conditions(expr)
```

Arguments

`expr`	An expression to evaluate.

Details

`leaf_solve()`, `leaf_gradient()` and `leaf_gradient_batch()` already do this, so wrap `expr` yourself when you are **driving a `Leaf` directly** — `$find_root_collar_psi()` is generated glue and cannot signal a classified condition on its own.

Value

The value of `expr`.

Why this and not a per-row NA mode

Returning `NA` rows instead of throwing is not built, deliberately: it must not be built before the classification in `leaf_infeasible_codes()` has been reviewed, or the opt-in would return `NA` for programming errors too and a guard written against "most rows failed" would be applied to bugs as well as to infeasibility. `leaf_gradient_batch()` already reports per-row `status`, which is the same idea where it was safe to have it.

See Also

`leaf_infeasible_codes()`

Examples

```
l <- leaf_model()
set_drivers(l, psi_soil = 2.0, PPFD = 900)
# An ordinary solve: nothing to catch.
with_phylloptim_conditions(l$find_root_collar_psi())

# A caller's per-row guard, written against the code rather than the message.
tryCatch(
  with_phylloptim_conditions(l$find_root_collar_psi()),
  phylloptim_infeasible = function(e) message("infeasible: ", e$code))
```

Index

∗ **package**
phylloptim-package, 3

align_curves, 4
align_curves(), *7*, *12*, *30*, *31*, *35*, *36*

cost_curve_names, 6
cost_curve_names(), *3*, *5*, *7*, *10*, *19*, *25*, *30*
cost_curve_parameters, 7
cost_curve_parameters(), *4*, *8*

gradient_output_names, 7
gradient_par_names, 8
gradient_par_names(), *7*, *26*

Leaf, 9
Leaf(), *28*
leaf_batch, 11
leaf_batch(), *3*, *5*, *7*, *24–26*
leaf_control, 13
leaf_control(), *3*, *5*, *12*, *18*, *28*, *30*, *34*, *37*, *49*
leaf_damage_gradient, 15
leaf_damage_gradient(), *3*, *19*
leaf_damage_trajectory, 16
leaf_damage_trajectory(), *3*, *15*
leaf_gradient, 17
leaf_gradient(), *3*, *6*, *8*, *13*, *24–26*, *30*, *37*, *50*, *54*
leaf_gradient_batch, 24
leaf_gradient_batch(), *3*, *6*, *8*, *11–13*, *54*
leaf_infeasible_codes, 26
leaf_infeasible_codes(), *54*
leaf_model, 28
leaf_model(), *3*, *14–16*, *18*, *32*, *33*, *37*, *48–50*
leaf_solve, 29
leaf_solve(), *3*, *5*, *7*, *12*, *13*, *24*, *28*, *33*, *37*, *39*, *49*, *54*
leaf_supply_multilayer *(leaf_supply_singlelayer)*, 32
leaf_supply_multilayer(), *12*, *18*, *28*, *30*
leaf_supply_singlelayer, 32
leaf_supply_singlelayer(), *5*, *12*, *18*, *28*, *30*, *47*, *49*
leaf_traits, 34
leaf_traits(), *3*, *5*, *7*, *8*, *12–14*, *18*, *23*, *25*, *28*, *30*, *41*, *50*, *51*

model_registry, 37
model_registry(), *44*

operating_point, 38
operating_point(), *3*, *5*, *28*, *31*

phylloptim *(phylloptim-package)*, 3
phylloptim-package, 3
plot_model_overview, 39
plot_solver_convergence, 40
psi_at_plc, 41
psi_at_plc(), *3*, *51*, *52*

registry_line_map(), *42*
registry_locations, 42
registry_locations(), *43*, *44*
registry_read(), *43*
registry_resolve, 43
registry_resolve(), *38*
registry_values, 44
registry_values(), *38*
root_network_from_carbon, 44
root_network_from_carbon(), *12*, *18*, *19*, *29*, *33*, *37*, *46–49*
RootNetwork, 46
RootNetwork(), *12*, *13*, *18*, *29*, *32*, *45–48*

series_resistance, 46
series_resistance(), *12*, *18*, *29*, *30*, *33*, *34*, *48*, *49*

set_drivers, 47
set_drivers(), *3*, *14*, *28*, *33*, *44–47*, *50*
set_traits, 49
set_traits(), *3*, *24*, *32*
stats::uniroot(), *5*

weibull_p50_c, 50
weibull_p50_c(), *3*, *36*, *41*, *52, 53*
weibull_p50_from_b, 52
weibull_s50, 53
weibull_s50(), *51*
with_phylloptim_conditions, 53
with_phylloptim_conditions(), *27*