Package: regnans (via r-universe)

November 13, 2026

Title Package for assembling communities with the plant model

Version 0.2

Author Daniel S. Falster and Richard G. FitzJohn

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

Description Assembles ecological communities of competing strategies using the plant model, by iterating demography to equilibrium, computing selection gradients and invasion fitness, and solving for evolutionary attractors.

License BSD_2_clause + file LICENSE

Depends R (>= 4.5.0)

Imports plant, odelia (>= 0.6.0), Rcpp, lhs, mvtnorm, dfoptim, ggplot2 (>= 3.5.0), BB, nleqslv, purrr, dplyr, tidyr, tibble, mlr3, mlr3learners, mlr3mbo, bbotk (>= 1.0.0), paradox, data.table, logger, crayon

LinkingTo Rcpp

Remotes traitecoevo/plant@develop

Suggests testthat (>= 3.0.0), parallel, future, future.apply, DiceKriging, nloptr, lgr, withr, pkgload

Encoding UTF-8

Config/testthat/edition 3

Config/testthat/parallel true

Config/roxygen2/version 8.0.0

Config/pak/sysreqs make libicu-dev

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

Date/Publication 2026-11-10 07:13:07 UTC

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

RemoteRef HEAD

RemoteSha 95257c9c0539ad2551a9cc2aac2c8964ef1f3631

Contents

assembler_control . . . 3
assembler_run . . . 4
assembler_set_traits . . . 5
assembler_start . . . 6
boundary_bisect . . . 7
boundary_control . . . 8
boundary_design . . . 10
boundary_diagram . . . 10
boundary_scout . . . 11
boundary_sweep . . . 12
boundary_switch . . . 14
bounds . . . 15
canonical_community . . . 15
canonical_control . . . 16
community_add . . . 18
community_canonical_equation . . . 19
community_classify_singularity . . . 20
community_continue_boundary . . . 23
community_continue_singularity . . . 26
community_demography . . . 28
community_fitness_gradient . . . 29
community_fitness_hessian . . . 29
community_fitness_landscape . . . 30
community_label . . . 31
community_parameter_map . . . 32
community_parameter_sensitivity . . . 33
community_parameter_values . . . 34
community_pip . . . 35
community_plot_fitness_landscape . . . 36
community_selection_gradient . . . 37
community_selection_gradient_jacobian . . . 37
community_selection_gradient_parameter_jacobian . . . 38
community_solve_boundary . . . 40
community_solve_singularity . . . 42
community_start . . . 44
community_tep . . . 45
community_viable_fitness . . . 46
demographic_step_control . . . 47
derivative_control . . . 48
fitness_landscape_control . . . 49
harness_check_derivatives . . . 50
harness_dd99 . . . 51
harness_dd99_nd . . . 52
harness_explicit . . . 53
harness_fd . . . 54
harness_gk98 . . . 55

harness_gm99 . . . 56
harness_jj12 . . . 57
harness_plant . . . 58
harness_provides . . . 59
length.assembler . . . 59
length.community . . . 60
max_fitness . . . 60
max_growth_rate . . . 61
mutational_vcv_proportion . . . 62
pip_control . . . 62
pip_mutual . . . 63
plant_community_check_for_inviable_strategies . . . 64
plant_default_assembly_control . . . 64
plant_default_assembly_pars . . . 65
plot.canonical_equation . . . 66
plot.pip . . . 67
plot.tep . . . 68
plot_community . . . 69
tidy_assembly . . . 70
trait_matrix . . . 70
util_nlsolve . . . 71

Index **73**

`assembler_control` *Options controllings community assembly process.*

Description

Controls how community assembly works.

Usage

```
assembler_control(control = NULL)
```

Arguments

`control`	A list of values to modify from defaults.

Details

Returns a list. Passing in a list of value via `control` will override the defaults. Options include run_type determines whether population is stepped to demographic equilibrium ("to_equilibrium") or not ("single_step"). "birth_type" determines sampling of new types – "stochastic" or "maximum" (on fitness peak). With "stochastic" births, "n_mutants" and "n_immigrants" determine the frequency of resident mutations and immigrations from global pool. "vcv" is variance-covariance matrix for mutations. If "birth_move_tol" is trait distance within which we attempt to move an

existing resident rather introduce a new type (this helps reduce the number of types). "compute_viable_fitness" asks whether to check bounds of viable trait space. "check_positive" determines whether the fitness of an invader is checked before it is introduced. If "check_inviable" causes dead residents to be removed when birth rate drops below "dead_birth_rate". "eps_too_close" is tolerance in trait values when searching for maxima.

With `evolve` other than `"none"` the residents evolve between births: after each step's deaths (and once at the start) every resident is moved to the singular coalition they form together, so that each maximum-fitness birth is an immigrant into an evolutionarily stationary community and the assembly ends at an evolutionarily stable community (`community_label`). `"newton"` solves for the coalition with `community_solve_singularity` (to `evolve_tol`) and classifies it with `community_classify_singularity`; where that solve fails, or finds a root the dynamics does not reach (not convergence stable, or without protected coexistence), the dynamics are integrated instead from where the residents were. `"canonical"` always integrates: `community_canonical_equation` to its first stationary coalition (`branch = "none"`, polished to `evolve_tol`), which only reaches attractors and drops residents that go extinct or merge, at the cost of an ODE integration. Either way an evolution that fails ends the assembly, reported in `$evolved`. A maximum of fitness within `eps_too_close` of a resident is then that resident at a branching point, and the birth is placed `branch_offset` (a fraction of the bounds' width, on the trait scale) away from it towards the maximum rather than ending the assembly. A birth is refused, ending the assembly with `$refused` saying why, when it would take the community beyond `max_residents`, or (with `edge_tol > 0` and a resident present) when it lies within `edge_tol` of the bounds' edge, that fraction of their width on the trait scale: the box, not the model, is then setting the community. A trait whose bounds are not finite on the trait scale has no edge.

Value

A list with elements run_type, birth_type, birth_move_tol, compute_viable_fitness, n_mutants, n_immigrants, check_positive, vcv, check_inviable, dead_birth_rate, eps_too_close, evolve, evolve_tol, branch_offset, max_residents, edge_tol

Author(s)

Rich FitzJohn, Daniel Falster

---

assembler_run *Take specified number of steps with assembler.*

Description

Take specified number of steps with assembler.

Usage

```
assembler_run(obj, nsteps)
```

Arguments

obj	An assembler object.
nsteps	Number of steps

Value

An assembler object.

Author(s)

Rich FitzJohn, Daniel Falster

---

assembler_set_traits *Set traits on an assembler*

Description

Set the initial traits (and optional birth rates) on an empty assembler.

Usage

```
assembler_set_traits(obj, traits, birth_rate = NULL)
```

Arguments

obj	An assembler object with an empty community.
traits	A trait matrix (see trait_matrix).
birth_rate	Optional birth rate(s); if NULL a default is used.

Value

The updated assembler object.

`assembler_start` *start community assembly process.*

Description

Initialise a community assembly object

Usage

```
assembler_start(
  community,
  control = assembler_control(control),
  filename = NULL,
  prev = NULL
)
```

Arguments

`community`	A `community` object.
`control`	A control object affecting how the assembly proceeds, see `assembler_control` for details,
`filename`	Location to save results.
`prev`	History from previously run assembly, used to restart.

Details

Control options affect how the assembly proceeds.

Value

An `assembler` object.

Author(s)

Rich FitzJohn, Daniel Falster

`boundary_bisect` *Bisect coexistence boundaries along rays*

Description

Find where the coexistence label changes along rays.

Usage

```
boundary_bisect(parameter, from, to, control = boundary_control())

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

Arguments

`parameter`	A function `p -> community`; see `community_parameter_map`.
`from`	The parameter values every ray starts from: typically inside the region of fewest species.
`to`	The far end of each ray, one row per ray (a vector is one ray).
`control`	See `boundary_control`.
`x`	A `boundary_points` object.
`...`	Ignored.

Details

Each ray runs from `from` to one row of `to` in parameter space, $p(t) = from + t(to - from)$ for t in $[0, 1]$, and is labelled at both ends (`community_label`). Where the labels differ the ray is bisected: the label at the midpoint decides which half holds a change, and a midpoint whose label differs from both ends puts a change in each half, until a bracket is shorter than `bisect_tol`. A label that only counts upwards along a ray, as nested regions make it from an interior point, has one change per pair of labels, and bisection cannot then pick the wrong one. A censored label is a label like any other here (its count and status), so a change into or out of one is reported with both labels and is a boundary of the search rather than of the model unless both are `"stable"`; an unresolved label is not known to differ from anything, so a bracket with one at an end, or meeting one at its midpoint, is reported unrefined, with `resolved = FALSE`.

Every label is assembled cold from the community `parameter` builds, so the points are independent: each ray is one job of the parallel map, with the same answer under any `future::plan()`. The rays' residents either side of each change are what a solve for the boundary point starts from.

Value

A data frame of class `boundary_points`, one row per change found: `ray`, the parameters at the bracket's midpoint, `t_lo` and `t_hi` (the bracket along the ray), `n_lo`, `status_lo`, `n_hi`, `status_hi`, `resolved`, `reason`, and the list columns `traits_lo`, `birth_rate_lo`, `traits_hi`, `birth_rate_hi`. `attr(., "rays")` has one row per ray with its labels at the ends, its changes and its equilibrium solves, whose total is `attr(., "evaluations")`.

Author(s)

Daniel Falster

Examples

```
comm <- community_start(bounds(x = c(-2, 2)), trait_scale = "linear",
                        harness = harness_dd99()) |>
  community_add(trait_matrix(0.5, "x"), birth_rate = 100)
by <- community_parameter_map(comm, c("sigma_C", "sigma_K"))
# from wide competition, where one species is stable, towards narrow
boundary_bisect(by, c(1.5, 1), rbind(c(0.5, 1.2), c(1.5, 2)),
                boundary_control(list(bisect_tol = 1e-4)))
```

---

boundary_control *Coexistence boundary settings*

Description

Control how coexistence labels and boundaries are computed.

Usage

```
boundary_control(control = NULL)
```

Arguments

`control` A list of values to modify from the defaults:

- `max_residents` the cap on the community's size; a label that would admit a resident beyond it is `"capped"`.
- `fitness_tol` an immigrant establishes when the maximum of invasion fitness over the box exceeds this: wherever fitness is positive, the default sitting just above the rounding in a resident's own fitness (zero at equilibrium); a model whose fitness is noisier (plant, about 1e-4) needs it raised.
- `edge_tol` an immigrant within this fraction of the bounds' width (on the trait scale) of the box's edge makes the label `"edge"`: the box, not the model, set the count.
- `branch_offset` how far from a resident at a branching point its daughter is placed, as a fraction of the bounds' width (`assembler_control`).
- `eps_too_close` a maximum within this fraction of the bounds' width of a resident is that resident at a branching point.
- `evolve`, `evolve_tol` how the residents evolve between immigrations (`assembler_control`): `"newton"` solves for the coalition and classifies it, integrating the canonical equation only when the root is not an attractor; `"canonical"` always integrates.
- `max_rounds` rounds of evolution and immigration before the label is given up as `"unresolved"`.

`trait_tol` two communities of the same size whose sorted traits differ by less than this fraction of the bounds' width are the same community (a round that returns to one is a cycle, and a sweep whose routes end on different ones has found hysteresis).

`bisect_tol` `boundary_bisect` halves each bracket until it is this short, as a fraction of the ray.

For `community_continue_boundary`:

`ds`, `ds_min`, `ds_max` the first, smallest and largest pseudo-arclength step, in the units of the unknowns (traits and the two free parameters together).

`max_steps` steps in each direction before stopping.

`tangent_tol` a step whose tangent turns so that its dot product with the last falls below this is rejected and the step halved: the corrector has jumped to another branch.

`corrector_tol`, `corrector_maxit` the corrector's tolerance on every residual and its iteration limit.

`density_tol`, `merge_tol` the test functions' thresholds: a resident whose share of the density falls below `density_tol` is going extinct, and two residents (or a resident and the marginal mutant) closer than `merge_tol` of the bounds' width have merged.

`n_landscape` grid points on which the landscape is searched for another peak (one trait).

`event_tol` an event is located by searching the step until it is bracketed this finely in arclength.

`verify_every`, `verify_offset`, `verify_tol` every `verify_every` points (and the last) the branch is verified cold: the residual re-solved from the model's initial densities must be within `verify_tol`, and the labels a relative `verify_offset` either side of the curve, across it in the plane of the free parameters, must be `n` and `n + 1` species, both stable.

Details

Returns the settings read by `community_label` and `boundary_scout`. Pass a list of values to override the defaults.

Value

A list with those elements.

Author(s)

Daniel Falster

boundary_design *Scouting design in a parameter box*

Description

Well-spread points in a parameter box.

Usage

```
boundary_design(box, n)
```

Arguments

box	A matrix with one row per parameter and columns `lower`, `upper`, as `bounds` builds.
n	Number of points.

Details

A deterministic low-discrepancy design (the Kronecker sequence with the generalised golden ratio) over a box of parameter values, for `boundary_scout`: no random numbers, so the same design every time, and each prefix of it is itself well spread.

Value

An `n` by `nrow(box)` matrix, columns named by parameter.

Author(s)

Daniel Falster

Examples

```
boundary_design(bounds(d = c(1, 3), kappa = c(0.5, 3)), 8)
```

boundary_diagram *A diagram of coexistence boundaries*

Description

Collect coexistence boundary branches into a diagram.

Usage

```
boundary_diagram(...)

## S3 method for class 'boundary_diagram'
as.data.frame(x, row.names = NULL, optional = FALSE, n = 8L, ...)

## S3 method for class 'boundary_diagram'
plot(x, y, n = 8L, ...)
```

Arguments

`...`	Ignored.
`x`	A `boundary_diagram`.
`row.names, optional`	Ignored.
`n`	Points per step, as for a branch.
`y`	Ignored.

Value

An object of class `boundary_diagram`, a list of branches, with `plot()` and `as.data.frame()` methods.

Author(s)

Daniel Falster

`boundary_scout` *Scout a parameter design for coexistence*

Description

Label the points of a parameter design.

Usage

```
boundary_scout(parameter, design, control = boundary_control())

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

Arguments

parameter	A function p -> community; see community_parameter_map. The community it returns is the start of each assembly.
design	A matrix of parameter values, one row per point, as boundary_design builds; or a vector for one parameter.
control	See boundary_control.
x	A boundary_labels object.
...	Ignored.

Details

The label (community_label) at each row of design, the model at each built by parameter: the scouting pass that shows how many regions a parameter plane holds and roughly where. Every point is labelled from the community parameter builds, apart from the others, so under a future::plan() with several workers the points run in parallel with the same answer (parameter runs on the workers, and under a plan whose workers are separate processes it must not hold a solved plant community).

Value

A data frame of class boundary_labels: the parameters, then n, status, reason, rounds, evaluations (equilibrium solves) and the list columns traits and birth_rate of the community each point ended on. Total solves in attr(., "evaluations").

Author(s)

Daniel Falster

Examples

```
comm <- community_start(bounds(x = c(-2, 2)), trait_scale = "linear",
                        harness = harness_dd99()) |>
  community_add(trait_matrix(0.5, "x"), birth_rate = 100)
by <- community_parameter_map(comm, c("sigma_C", "sigma_K"))
# one species where competition is wider than the resource, else the box
# fills: no finite community resists invasion
boundary_scout(by, boundary_design(bounds(sigma_C = c(0.5, 1.5), sigma_K = c(0.8, 1.2)), 4))
```

---

boundary_sweep *Sweep a path for hysteresis*

Description

Look for alternative attractors along a path in parameter space.

Usage

```
boundary_sweep(parameter, p, control = boundary_control())

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

Arguments

parameter	A function p -> community; see community_parameter_map.
p	The path, one row per point in order (a vector for one parameter).
control	See boundary_control.
x	A boundary_sweep object.
...	Ignored.

Details

Where a model has alternative stable communities the label is not a function of the parameters alone: which community is reached depends on where the assembly started. A sweep labels each point of a path three ways: cold, from the community parameter builds (boundary_scout); and carried, from the community the previous point ended on, along the path forward and back. Each carried label starts from its predecessor's residents and densities, so it follows an attractor for as long as it persists, as a slowly changing environment would. A point whose three labels differ in count or status, or whose communities differ in the residents' sorted traits by more than trait_tol of the bounds' width, is flagged: the history decided. An unresolved label is not known to differ from anything, so it is left out of the comparison, and a point where the routes that did resolve agree is flagged NA. Every route searches the same box: a carried label starts from the box the cold one would search, narrowed to the fundamental niche where the harness can find it (plant), and its predecessor's residents outside that box are dropped.

The cold labels are independent jobs of the parallel map; each carried direction is one job (a chain, sequential by nature). The answer does not depend on the plan.

Value

A data frame of class boundary_sweep, one row per point: the parameters, then n and status for each route (cold_n, cold_status, forward_n, ..., backward_status), hysteresis (NA where an unresolved route leaves it unknown), the list columns cold_traits, forward_traits and backward_traits, and evaluations (equilibrium solves, all three routes), whose total is attr(., "evaluations").

Author(s)

Daniel Falster

Examples

```
comm <- community_start(bounds(x = c(-2, 2)), trait_scale = "linear",
                        harness = harness_dd99()) |>
  community_add(trait_matrix(0.5, "x"), birth_rate = 100)
```

```
# one stable community at every point: nothing to flag
boundary_sweep(community_parameter_map(comm, "sigma_C"), c(1.2, 1.4, 1.6))
```

`boundary_switch` *Start the branch an event seeds*

Description

The branch an event starts.

Usage

```
boundary_switch(branch, parameter, end = c("forward", "backward"))
```

Arguments

`branch`	A `regnans_branch` that ended at an event.
`parameter`	The map the branch was traced with.
`end`	Which end's event: `"forward"` or `"backward"`.

Details

Each event that ends a branch of `community_continue_boundary` is a point where it meets another boundary, and this gives the start of that boundary, to pass on to `community_continue_boundary()`:

`separation` residents (or a resident and y^*) merge: each cluster (single linkage at 2.5 `merge_tol`, so that a pair straddling y^* joins it) becomes one resident at the density-weighted mean of its residents, and the boundary that continues is that resident's branching locus (the pitchfork the merging residents emerged from).

`resident_curvature` a resident stops being an ESS: the boundary that starts is its branching locus, the same residents.

`density` a resident goes extinct: the boundary that starts is the invasion boundary of the community without it, the resident's own trait the marginal mutant.

`other_peak` another peak becomes marginal: the invasion boundary of the same residents at that peak.

The annihilation of the marginal peak (`peak_curvature`) and a crossing branch (`bordered`) start no boundary of these kinds, and are an error.

Value

A list of arguments for `community_continue_boundary`: `community`, `parameter`, `y`, `type`, `which`, `p` and `free`; pass it with `do.call()`.

Author(s)

Daniel Falster

bounds *Trait bounds*

Description

Helper function for making bounds

Usage

```
bounds(...)

check_bounds(bounds, finite = FALSE)

bounds_infinite(trait_names)

check_point(x, bounds)
```

Arguments

...	Named list, each element of which is a 2-element numeric vector of lower and upper bounds.
bounds	A set of bounds
finite	Logical indicating if bounds must be finite
trait_names	Character vector of trait names
x	a point to detect if it lies within bounds

Examples

```
bounds(lma=c(0.01, 10))
bounds(lma=c(0.01, 10), rho=c(1, 1000))
```

canonical_community *Community along a canonical-equation trajectory*

Description

The community at a recorded time of a canonical-equation run.

Usage

```
canonical_community(x, time)
```

Arguments

x	A canonical_equation.
time	Evolutionary time; the nearest recorded time is used.

Details

Rebuilds the community from the record nearest `time` — the residents at their traits then — and solves it to equilibrium, so the fitness landscape, invasibility surface or classification at that moment can be computed with the usual functions.

Value

A solved `community`, with `attr(., "time")` the record used.

Author(s)

Daniel Falster

`canonical_control` *Canonical-equation settings*

Description

Control the canonical equation.

Usage

```
canonical_control(control = NULL)
```

Arguments

`control` A list of values to modify from the defaults:

`rate` the mutation rate μ: mutations per unit evolutionary time in the initial community (the resident's share scales with its relative density when `density` is on). It sets the time scale of the speed and of the waiting times alike.

`vcv` mutational covariance on the trait scale, a k by k matrix. `NULL` uses `diag(mutation_sd^2)`, so the speed and the branching waiting time are driven by the same kernel.

`density` multiply each resident's speed by its equilibrium density relative to the initial total (mutation supply scales with population size).

`stepper` an `odelia` stepper: `"rodas"` (linearly implicit RODAS4(3), the default: trait models are stiff as a rule, and an L-stable step is not held at a stability limit; its Jacobian comes from `community_selection_gradient_jacobian`, central differences across the residents of their gradients and densities, two equilibrium solves per unknown), `"rkck"` (explicit Cash–Karp 4(5), the cheapest on smooth stretches) or `"dopri"` (explicit Dormand–Prince 5(4)). See `OdeSolver`.

`t_max`, `max_steps` limits on evolutionary time (`Inf` by default: a run ends when the coalition is stable or a limit is reached) and accepted steps.

`rtol`, `atol` tolerances of the step-size control, on the trait scale: a step is accepted when its error estimate is within `atol + rtol * abs(z)`. The defaults follow the trajectory to about 1% a step: the end point is found by Newton (`polish`), so a tighter trajectory costs equilibrium solves without moving it, and a looser one, which RODAS would take, holds the explicit pairs at their stability limit.

`dt0`, `dt_max` initial and largest step; `NULL` chooses `dt0` from the initial speed.

`gradient_tol` selection has stopped when every resident's gradient on the trait scale is below this.

`polish`, `polish_tol` once every gradient is below `polish_tol`, finish the approach to the stationary coalition by solving for it (`community_solve_singularity`) instead of integrating it: near a stable coalition the dynamics are stiff, and an explicit stepper held at its stability limit jitters about the point without reaching `gradient_tol`. A solution more than a tenth of the trait range from the current residents is not a local finish and is refused; the integration carries on.

`branch` what happens to a stationary resident at a fitness minimum. `"expected"` (default): a mutant that can invade and coexist arises at the rate $\lambda = \mathrm{rate}\,\tilde{n} \int M(\delta)\, p_{est}(\delta)\, \mathbf{1}[\mathrm{coexist}($ along the branching direction, with M the mutational kernel of width `mutation_sd`, $p_{est} = \min(1, 2s)$ the establishment probability of a rare mutant with invasion fitness s, and coexistence meaning the resident can invade the mutant back; time advances by $1/\lambda$ and the mutant appears at the expected distance of a successful mutation. `"stochastic"`: the waiting time is exponential with that rate and the mutant's distance is drawn. `"immediate"`: split at once, `branch_distance` either side. `"none"`: stop at the first stationary coalition.

`mutation_sd` standard deviation of the mutational kernel on the trait scale, one per trait; `NULL` takes it from `vcv` where given, else 2% of the trait range.

`establishment_factor` a rare mutant or immigrant with invasion fitness $s > 0$ establishes with probability $\min(1, c\,s)$. The default $c = 2$ is Haldane's approximation for a discrete-generation Poisson branching process; fitness is a per-capita rate for DD99, a log ratio for the other reference models and a log net-reproduction ratio for plant, so the right constant is model-dependent and the waiting times are exact only up to it.

`branch_nodes` Gauss–Hermite nodes for the branching-rate integral; each costs one equilibrium solve, and under a `future::plan()` with several workers they are solved in parallel, with the same answer.

`branch_distance` for `"immediate"`: how far, as a fraction of the trait range on the trait scale, the two daughters sit from the parent.

`immigration` `NULL`, or a list with `rate` (arrivals per unit evolutionary time, exponential waiting times), `pool` (where phenotypes come from: `NULL` for uniform on the trait scale within the bounds, a matrix of phenotypes to sample rows from, or a function of `n` returning that many rows) and `establish` (`"deterministic"`: an arrival with positive invasion fitness establishes; `"stochastic"`: with probability $\min(1, 2s)$).

`seed` random seed for the stochastic parts; the caller's RNG state is restored afterwards.

`max_residents` stop rather than branch beyond this many.

`extinct_fraction` a resident whose equilibrium density falls below this fraction of the total is removed.

`classify_tol` eigenvalue magnitude below which the stability test is undecided and the resident is left alone.

Value

A list of the settings above.

Author(s)

Daniel Falster

`community_add` *Add strategies to a community*

Description

Add one or more strategies (residents) to a `community`.

Usage

```
community_add(obj, traits, birth_rate = NULL)
```

Arguments

`obj`	A `community` object (see `community_start`).
`traits`	A trait matrix (see `trait_matrix`) giving the traits of the strategies to add.
`birth_rate`	Optional birth rate(s) for the added strategies; if `NULL` a default is used.

Value

The updated `community` object.

`community_canonical_equation` *Canonical equation with branching*

Description

Integrate the canonical equation of adaptive dynamics.

Usage

```
community_canonical_equation(
  community,
  x0 = NULL,
  control = canonical_control()
)
```

Arguments

`community`	A `community`; its residents (or `x0`) are the starting strategies.
`x0`	Starting trait values, one row per resident, if the community carries none.
`control`	See `canonical_control`.

Details

Every resident's trait moves up its selection gradient,

$$\frac{dz_i}{dt} = \tfrac{1}{2}\,\mu\,\tilde{n}_i\,V\,\nabla_z s(z_i)$$

(Dieckmann & Law 1996), on the community's trait scale z, with the community re-solved to demographic equilibrium at every evaluation; μ is `rate`, $\tilde{n}_i$ the resident's equilibrium density relative to the initial total and V the mutational covariance (see `canonical_control`). A resident whose own gradient has vanished is tested: a positive eigenvalue of its fitness Hessian means it sits at a fitness minimum. By default it then branches after the waiting time for a mutation that can invade and coexist, the parent staying put and the mutant appearing at the kernel's successful distance along that eigenvector; branching and immigration are competing clocks, and whichever comes first happens. When every resident is at a fitness maximum the coalition is evolutionarily stable and the run stops (unless immigration is on, in which case it runs to `t_max`). A resident whose density falls below `extinct_fraction` of the total is removed, and of two residents within two finite-difference steps of each other (`derivative_control`), too close for the resident Jacobian to tell apart, the rarer goes extinct: the converse of a branching.

Value

A `canonical_equation` object: `trajectory` (a tibble of `time`, `lineage`, the trait columns, `density` and the gradient columns `gradient_<trait>` on the trait scale, one row per resident per accepted step), `events` (`time`, `event` — `"branch"`, `"immigrant"`, `"immigrant_refused"` (it could have established but the community is at `max_residents`), `"extinct"`, `"merge"` (the rarer of two merged

residents went extinct), "polish", "stable", "t_max", "max_steps", "max_residents" — and lineage), community (the final solved community), evaluations (how many equilibrium solves it cost), environments (how many runs of the model at given traits and densities its Jacobians cost, under resident_jacobian = "implicit" in derivative_control), steps and rejections (accepted steps and attempts the step-size control rejected), immigration_attempts, outcome and the control. canonical_community rebuilds the community at any recorded time.

Author(s)

Daniel Falster

---

community_classify_singularity

Classify a singular strategy or coalition (1-D and N-D)

Description

Classify a singular strategy or coalition.

Usage

```
community_classify_singularity(
  community,
  birth_rate = NULL,
  tol = 1e-08,
  speeds = "density",
  invasion_tol = tol
)

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

Arguments

community	A community whose residents are at (or very near) a singular strategy or coalition — typically the output of community_solve_singularity.
birth_rate	Birth rates to start each resident equilibrium solve from; see community_solve_singularity. Defaults to the residents' own equilibrium birth rates, which is normally what you want.
tol	Magnitude below which an eigenvalue counts as zero, making the classification degenerate rather than forcing a verdict.
speeds	The residents' relative speeds of evolution, which decide convergence stability for a coalition: "density" (default) their equilibrium densities, as in community_canonical_equation with canonical_control(density = TRUE); "equal"; or a positive vector with one value per resident. Ignored for one resident.

`invasion_tol`	The invasion fitness when rare each resident of a coalition must exceed for protected coexistence. It is in units of invasion fitness, not of an eigenvalue, so worth setting apart from `tol` when the model's fitness carries numerical noise (on plant, its SCM's error).
`x`	A `singularity_classification` object.
`...`	Ignored.

Details

Finding a singular strategy is only half the question: the interesting part is what happens near it. Two independent second-order conditions decide that (Geritz et al. 1998; Leimar 2009):

Evolutionary stability the curvature of invasion fitness in the *mutant* direction, with the residents held where they are. In one trait this is the scalar $\partial^2 s/\partial y^2$; in `k` traits it is the `k` x `k` Hessian H. A resident is uninvadable by nearby mutants when H at its traits is negative definite, i.e. every eigenvalue is negative; a coalition is an ESS when every resident is.

Convergence stability whether selection carries nearby residents towards the point. This is the derivative of the selection gradients with respect to the *resident* traits: the scalar dG/dx in one trait, the Jacobian J in `k`, and for `m` residents the `mk` x `mk` Jacobian of every resident's gradient with respect to every resident's traits, through the environment they share. Under the canonical equation with an isotropic mutational covariance each resident moves at its own speed w_i along its gradient, so the point is convergence stable when the eigenvalues of DJ, with D the speeds on the diagonal, have negative real parts. For one resident the speed only rescales J; for a coalition it does not, and a coalition stable at equal speeds can be unstable at the speeds its densities give (the canonical equation's speed is proportional to the resident's equilibrium density). A negative-definite symmetric part $(J+J^T)/2$ gives *strong* convergence stability, which holds for any mutational covariance matrix and any (positive) speeds of the residents.

Crossing the two gives the standard four-way classification:

	convergence stable	**not convergence stable**
ESS	CSS	Garden of Eden
not ESS	branching point	repeller

For a coalition, "CSS" is an evolutionarily stable coalition that selection reaches, and "branching point" one that selection reaches but where at least one resident sits at a fitness minimum and will branch again (`resident_evolutionarily_stable` says which).

A coalition is first checked for protected coexistence: each resident, made rare, must be able to invade the community the others settle at without it. Positive densities alone do not make a coalition, and one a resident lost by chance cannot return to is not an evolutionary outcome: it is classified `"unprotected"` whatever its second-order conditions. For two Lotka–Volterra residents with positive densities, protection is exactly the stability of their equilibrium (priority effects give positive densities at an unstable one). For three or more it is a stronger condition than stability: a stable coalition can have a resident that, once lost, stays lost, the others settling where it cannot invade. The test is made on invasion fitness, so it means the same whatever the model's time step.

The community the others settle at is their equilibrium with any of them that has no positive density there removed and the rest re-solved (a model's own equilibrium can be infeasible, with

negative densities, once a resident is left out). Where that is not determined — an equilibrium solve fails, no resident persists, or a removed resident could invade the rest back — that resident's `invasion_fitness` is `NA`; `protected_coexistence` is then `NA` unless another resident already fails, with a warning, and the classification is decided by the second-order conditions.

The full eigen-decompositions are returned, not just the verdict: when a resident is not an ESS the leading eigenvector of its H is the direction in trait space along which it disruptively splits, which is itself the scientific result in a multi-trait problem.

Cost: each Hessian (`community_fitness_hessian`) needs `1 + 4k^2` invasion-fitness evaluations, all made in a single vectorised call against the cached resident environment, so it is cheap. The Jacobian (`community_selection_gradient_jacobian`) needs `2mk` *resident* evaluations, each a full demographic equilibrium solve, so it dominates. Under a `future::plan()` with several workers these solves, and those of the coexistence test, run in parallel, with the same answer. Protected coexistence costs `m` more, one per resident left out (and one more for each re-solve after removing a resident without a positive density). Derivative settings come from `derivative_control`.

A community marked as not converged (`attr(., "converged")` `FALSE`, set by `community_solve_singularity` or by the equilibrium solve) is classified with a warning: the verdicts then describe the point the solve stopped at. So is one whose residents are visibly away from a singularity, a Newton step on their selection gradients moving some trait by more than a thousandth of the bounds' width on the trait scale (`newton_reach`): the second-order conditions describe a singularity, and away from one they describe nothing in particular. A coalition with two residents within a thousandth of the bounds' width of each other also warns: at a pitchfork the coalition shrinks to a point, and a near-merged pair can pass as a root whatever its verdict.

Value

An object of class `singularity_classification`: a list with the traits and selection gradient at the point, `hessian` and `jacobian` with their eigen-decompositions (`hessian_eigen`, `jacobian_eigen`, `jacobian_symmetric_eigen`), `speeds` (normalised to mean one) with `jacobian_weighted_eigen`, the eigen-decomposition of DJ on which convergence stability is decided (the same as `jacobian_eigen` for one resident or equal speeds), `invasion_fitness` (each resident's when rare against the others; `NULL` for one resident) with the verdict `protected_coexistence` (`NA` where undetermined), `newton_reach` (how far the point is from a singularity, as a fraction of the bounds' width), the logical verdicts `evolutionarily_stable` (with `resident_evolutionarily_stable`, one per resident), `convergence_stable` and `strongly_convergence_stable`, the `branching_direction` where the point is invadable, the `classification` (the four-way verdict, `"degenerate"`, or `"unprotected"` for a coalition without protected coexistence), and `evaluations` (the equilibrium solves of the Jacobian and the coexistence test). For one resident `traits`, `selection_gradient` and `branching_direction` are vectors named by trait and `hessian` a matrix; for a coalition they are residents-by-traits matrices (a row of `NA` in `branching_direction` for a resident that is not at a fitness minimum) and `hessian` and `hessian_eigen` are lists with one entry per resident.

Author(s)

Daniel Falster

`community_continue_boundary` *Continue a coexistence boundary in two parameters*

Description

Follow a coexistence boundary through a plane of two parameters.

Usage

```
community_continue_boundary(
  community,
  parameter,
  y = NULL,
  type = c("invasion", "branching"),
  free = 1:2,
  which = 1L,
  p = NULL,
  box = NULL,
  direction = c("both", "forward", "backward"),
  control = boundary_control()
)

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

## S3 method for class 'regnans_branch'
as.data.frame(x, row.names = NULL, optional = FALSE, n = 1L, ...)

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

Arguments

`community`	A `community` whose residents start the solve (their densities, if positive, seed the first equilibrium), built by `parameter(p)`.
`parameter`	A function `p -> community`; see `community_parameter_map`.
`y`	Ignored.
`type`	`"invasion"` or `"branching"`.
`free`	The two parameters that vary, by name or position in `p`; the others are held. The start is first corrected onto the curve with the second held.
`which`	For `type = "branching"`, the resident becoming a branching point.
`p`	The parameter values `community` was built at, the free one's starting value among them. Read from the community for a map from `community_parameter_map`.

box — Optional: a two-row matrix of lower and upper limits (columns lower, upper) for the free parameters, in the order of free; the branch ends where it leaves it.

direction — "both" (from the start in both directions), "forward" (the second free parameter increasing at the start) or "backward".

control — See boundary_control.

x — A regnans_branch.

... — Ignored.

row.names, optional — Ignored.

n — Points per step: with n > 1 each step is interpolated by the cubic Hermite polynomial through its ends and their tangents, so a coarse branch plots as a smooth curve.

Details

Starting from a point on a boundary (one from community_solve_boundary, or a bracket from boundary_bisect), follows the curve that the augmented system traces when two parameters are free: $nk + k + 1$ equations in $nk + k + 2$ unknowns for an invasion boundary, $nk + 1$ in $nk + 2$ for a branching one. Each step predicts along the tangent and corrects by Newton on the system bordered by it (pseudo-arclength continuation), with the Jacobian from the derivative layer (2nk + 4 equilibrium solves, through the parallel map). Steps adapt to the corrector's iterations between ds_min and ds_max; a step that does not converge, or whose tangent turns so that its dot product with the last falls below tangent_tol, is halved and retried.

At every accepted point test functions are evaluated, each positive while the boundary describes the coexistence it was found for:

density — the rarest resident's share of the density, less density_tol: a resident going extinct, where the boundary meets the one of a community without it.

resident_curvature — minus the largest eigenvalue of the residents' Hessians (all residents of an invasion boundary, the others of a branching one): a resident ceasing to be an ESS, where the boundary meets that resident's branching locus.

peak_curvature — (invasion) minus the largest eigenvalue of the Hessian at y^*: the marginal peak annihilating with a valley.

separation — the smallest distance between residents (and y^*), as a fraction of the bounds' width on the trait scale, less merge_tol: residents merging, as at a pitchfork.

other_peak — (one trait) fitness_tol less the highest fitness on the landscape farther than twice merge_tol from the residents and y^* (searched on n_landscape points, refined): another peak becoming marginal first, so that the boundary is now that peak's; on a branching locus, also the landscape beside the resident turning upward, where the pitchfork turns subcritical and the locus stops being a coexistence boundary.

bordered — the determinant of the Jacobian bordered by the tangent, which changes sign where another branch crosses this one.

Where one changes sign the step is searched by false position, the trial points corrected without the Jacobian that only bordered needs, until the event is bracketed to event_tol in arclength, the located point ends the branch, and the event is recorded (unless it lies beyond box or the trait

bounds: the branch has left them first, and ends there); boundary_switch gives the start of the branch it seeds. A branch also ends where it closes on itself, leaves box, takes a resident or y^* out of the trait bounds, can no longer step (ds below ds_min) or reaches max_steps.

The two directions from the start are traced as two jobs of the parallel map, with the same answer under any future::plan(). Afterwards every verify_every-th point and both ends (the point before an event, where two boundaries meet and labels cannot tell them apart) are verified cold, also through the map: the residual re-solved from the model's initial densities, with no warm start, and the labels (community_label) either side of the curve: the boundary's n species, stable, on one side and n + 1 on the other, or a count censored at n or more (as where DD99's box fills beyond its branching locus). A point that fails is reported in verification and warned of: systematic drift, or a boundary that has stopped being one, shows here.

Value

An object of class regnans_branch: a list with points (a data frame: the arclength s, the free parameters, the residents' traits, y^*, the densities, the test functions and the corrector's iterations, one row per point in order along the curve), the unknowns and unit tangents in u and tangent (matrices, one row per point), start (the row of the start), events (a data frame: end, type, s, the free parameters, detail), termination (why each direction ended), verification, rejected steps, evaluations (equilibrium solves) and what boundary_switch needs to start a branch from an event.

Author(s)

Daniel Falster

Examples

```
# GK98's three patches, -d, 0 and d, the middle one kappa times the others;
# the map seeds one resident, where every label starts
gk <- function(p) {
  community_start(bounds(x = c(-4, 4)), trait_scale = "linear",
                  harness = harness_gk98(mu = c(-p[1], 0, p[1]), K = c(1, p[2], 1))) |>
    community_add(trait_matrix(0.5, "x"), birth_rate = 1)
}
pair <- community_start(bounds(x = c(-4, 4)), trait_scale = "linear",
                        harness = harness_gk98(mu = c(-1.93, 0, 1.93))) |>
  community_add(trait_matrix(c(-1.27, 1.27), "x"), birth_rate = c(1.5, 1.5))
# where a third species can invade the pair, as the middle patch grows
branch <- community_continue_boundary(pair, gk, y = 0, p = c(1.93, 1),
                                      box = rbind(c(1, 3), c(0.5, 3)),
                                      direction = "forward")
branch
plot(branch)
```

`community_continue_singularity` *Continuation of a singular strategy or coalition along a parameter*

Description

Follow a singular strategy or coalition along one parameter.

Usage

```
community_continue_singularity(
  community,
  parameter,
  p,
  solver = c("nleqslv", "newton", "dfsane"),
  tol = 1e-06,
  maxit = 100,
  classify = list()
)

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

Arguments

`community`	A `community` whose residents (and their densities, if solved) start the path: the singular point at `p[1]`, or near it.
`parameter`	A function from one parameter value to a `community`; see `community_parameter_map` (built with one scalar parameter).
`p`	The parameter values to visit, `p[1]` the starting one.
`solver, tol, maxit`	Passed to `community_solve_singularity` for each correction. The `"bracket"` solver ignores the prediction and is not offered.
`classify`	A list of arguments for `community_classify_singularity` (`tol`, `speeds`, `invasion_tol`, `birth_rate`).
`x`	A `singularity_path` object.
`...`	Ignored.

Details

Natural-parameter continuation of a singular point through the values `p`, in the order given. At each value the point is predicted from the last by its sensitivity (`community_parameter_sensitivity`), $z_i^* \approx z_{i-1}^* + (dz^*/dp)(p_i - p_{i-1})$ on the community's trait scale z (so a log-scale trait stays positive), corrected by `community_solve_singularity` started there with the last equilibrium densities, and classified by `community_classify_singularity`. The classifier's resident Jacobian is also the one

the next prediction uses and the one the next correction starts from, so each point costs the corrector's residual solves, the classification's and two more for $\partial g/\partial p$. Where consecutive classifications differ the path records the change (`changes`): a branching point becoming a CSS, say, as a kernel widens.

A prediction outside the bounds is clamped to them, as the corrector clamps every candidate. The path stops at the first value it cannot reach — the corrector does not converge, a coalition loses or merges residents, or the corrected point cannot be classified or differentiated (a singular Jacobian) — with one warning naming the value and the reason, and keeps the points before it. Such a stop usually brackets a fold or a bifurcation; a finer `p` near it, or a path restarted from `community` at the last point, says more. Failing at `p[1]` is an error.

Two limits of natural-parameter continuation. Near a fold or bifurcation the resident Jacobian approaches singularity and `sensitivity` grows without bound; the classification of a point there is not reliable, and a point exactly at a pitchfork can be a near-merged coalition that passes as a root (the classifier warns of residents that close). And the corrector is not held to the prediction: where singular points lie close together it can land on another branch, which `traits` against `predicted` shows.

Value

An object of class `singularity_path`: a list with `p` (the values reached), `traits`, `predicted` (the corrector's starting points, the community's residents at `p[1]`) and `sensitivity` (dx^*/dp), each a matrix with one row per value and one column per resident trait, stacked as in `community_selection_gradient_jacobian`; `birth_rate` (one row per value, one column per resident); `classification` and the full `classifications`; `changes`, a data frame of `p_before`, `p_after`, `from`, `to` for each change of classification; `evaluations`, the equilibrium solves at each value (their sum in `attr(., "evaluations")`); `stopped`, `NULL` or the value and reason the path stopped at; and `community`, the community at the last point, from which the path can be continued.

Author(s)

Daniel Falster

Examples

```
comm <- community_start(bounds(x = c(-2, 2)), trait_scale = "linear",
                        harness = harness_dd99(sigma_C = 0.7))
by_sigma_C <- community_parameter_map(comm, "sigma_C")
# a branching point becomes a CSS as the competition kernel outgrows the
# resource kernel (sigma_K = 1)
path <- community_continue_singularity(community_solve_singularity(comm),
                                       by_sigma_C, seq(0.7, 1.3, by = 0.2))
path
path$changes
```

`community_demography` *Update demography of community*

Description

Solve a community to demographic equilibrium.

Usage

```
community_demography(community)
```

Arguments

`community`	A `community` object.

Details

Dispatches on `community$demography_control$equilibrium_solver_name` (see `demographic_step_control`). Every solver except `"model"` works from the harness's demography runner, the map from resident birth rates to next-generation offspring production: a single application, fixed-point iteration, root-finding on $n - f(n)$ with the package's Newton–Broyden iteration (`"equilibrium_solve_newton"`), `nleqslv` or `dfsane`, or the hybrid. Fixed-point iteration slows as the map's multiplier approaches one, which is where strongly interacting communities live; Newton converges quadratically there, and carries its Jacobian on the community so the next solve of a nearby community starts from it (`community$demography_state`). `"model"` asks the harness for its equilibrium directly — analytic or from the model's own internal solver — which only the reference models can answer; it is their default; plant's is iteration for FF16 and Newton for TF24, whose map overcompensates so that iteration settles on a two-cycle. Afterwards the invasion-fitness closure is rebuilt.

Value

A `community` object with `birth_rate` at equilibrium and `attr(., "converged")`, `attr(., "progress")` describing the solve.

Author(s)

Rich FitzJohn, Daniel Falster

community_fitness_gradient

Fitness gradient in the mutant direction

Description

Gradient of invasion fitness in the mutant direction.

Usage

```
community_fitness_gradient(community, y = community$traits)
```

Arguments

community	A community solved to demographic equilibrium.
y	Trait points to differentiate at, one per row (a bare vector is one point per element for a single trait, or a single point otherwise). Defaults to the residents' traits.

Details

The derivative of invasion fitness $s(y)$ with respect to the mutant trait y, with the residents and their environment held fixed. Evaluated at the residents' own traits this is the selection gradient (community_selection_gradient); evaluated elsewhere it is the slope of the fitness landscape. All stencil points are evaluated in a single call to the community's vectorised fitness function.

Value

A nrow(y) by k matrix of gradients with the fitness at each point in attr(., "value").

Author(s)

Daniel Falster

community_fitness_hessian

Fitness Hessian in the mutant direction

Description

Hessian of invasion fitness in the mutant direction.

Usage

```
community_fitness_hessian(community, y = community$traits)
```

Arguments

community	A community solved to demographic equilibrium.
y	A single trait point; defaults to the sole resident's traits.

Details

The curvature of invasion fitness $s(y)$ with respect to the mutant trait, residents held fixed. At a singular strategy a negative-definite Hessian is the condition for evolutionary stability (community_classify_singularity). All stencil points are evaluated in one call to the fitness function.

Value

A symmetric k by k matrix.

Author(s)

Daniel Falster

---

community_fitness_landscape

Fitness Landscape

Description

Construct a fitness landscape.

Usage

```
community_fitness_landscape(community, method = NULL, ...)
```

Arguments

community	A community object
method	used to construct landscape; defaults to community$fitness_control$method and falls back to "grid" (see fitness_landscape_control).
...	additional arguments passed to the chosen landscape method (e.g. bounds, n_evals).

Author(s)

Daniel Falster

community_label *Label a community by the species its parameters support*

Description

The number of species a community's parameters support.

Usage

```
community_label(community, control = boundary_control())
```

Arguments

community	A community, the start of the assembly.
control	See boundary_control.

Details

Assembles the community by immigration and evolution until it is evolutionarily stable, and reports its size. Each round the residents evolve to the singular coalition they form together and one immigrant arrives at the maximum of invasion fitness over the trait box, if its fitness there exceeds fitness_tol; the rounds are the assembler's (assembler_run with assembler_control's evolve), started from community as given: empty, or seeded with a resident for a model whose empty community cannot be invaded (soft selection) or whose fundamental fitness has no interior maximum (DD99's is flat). The box is the community's bounds, narrowed to the fundamental niche where the harness can find it (plant, from an empty start).

The status says whether the count is the model's:

"stable" no immigrant can establish and every resident is at a singular coalition the dynamics reaches: an evolutionarily stable community, and the only genuine label.

"capped" an immigrant would take the community beyond max_residents; it is refused. The count is at least n.

"edge" the immigrant's maximum lies on the box's edge, so the box set the count; it is refused (a first immigrant into an empty community is exempt, since evolution moves it). At least n.

"unresolved" the residents' evolution fails (the coalition cannot be solved, or the dynamics integrated from it do not settle), or the rounds cycle or run out: the count is the search's, with reason saying what happened.

A censored label is never a boundary of the model; a bisection (boundary_scout) treats it as "more than n" and an unresolved one as unknown.

Value

The community the assembly ends on (solved to demographic equilibrium, at its singular coalition), with attr(., "label"): a list of n, status, reason, rounds and evaluations, the equilibrium solves it cost.

Author(s)

Daniel Falster

Examples

```
gk <- function(d) {
  community_start(bounds(x = c(-4, 4)), trait_scale = "linear",
                  harness = harness_gk98(d = d, sigma = 1)) |>
    community_add(trait_matrix(0.5, "x"), birth_rate = 1)
}
# three patches, farther apart: one generalist, then a pair of
# specialists, then one per patch
for (d in c(1, 1.5, 2.5)) print(attr(community_label(gk(d)), "label")[c("n", "status")])
```

---

community_parameter_map

Parameter map for an explicit harness

Description

Vary the parameters of a model with an explicit harness.

Usage

```
community_parameter_map(community, names)
```

Arguments

community	A community with an explicit harness.
names	Names of the harness parameters (names(harness$pars)) to vary.

Details

Builds the function p -> community that community_parameter_sensitivity, community_continue_singularity and community_selection_gradient_parameter_jacobian take, for a community whose harness was built by harness_explicit (the reference models harness_dd99(), harness_jj12(), ...). Editing community$harness$pars does nothing, since the parameters are closed into the model's functions when the harness is built; the map rebuilds the harness instead. For any other model, or a parameter that is not one of the harness's, write the function directly: it takes the parameter values and returns a community of the same traits.

Value

A function taking a numeric vector p, the named parameters concatenated in order (a vector-valued parameter gives one entry per element), and returning community with those parameters set to p: same residents, bounds and controls, reset so that nothing solved under the old parameters is carried over. The names it varies are in attr(., "pars"); the functions that take a map read a community's own values of them (community_parameter_values()) as their default p, so a community the map built always answers at the parameters it was built at.

Author(s)

Daniel Falster

Examples

```
comm <- community_start(bounds(x = c(-2, 2)), trait_scale = "linear",
                        harness = harness_dd99())
by_x0 <- community_parameter_map(comm, "x0")
community_parameter_values(by_x0(0.5), by_x0)
community_solve_singularity(by_x0(0.5))$traits
```

---

community_parameter_sensitivity

Parameter sensitivity of a singular strategy or coalition

Description

Sensitivity of a singular strategy or coalition to model parameters.

Usage

```
community_parameter_sensitivity(
  community,
  parameter,
  p = NULL,
  birth_rate = NULL
)
```

Arguments

community	A community at a singular strategy or coalition, typically from community_solve_singularity on parameter(p).
parameter	A function p -> community; see community_parameter_map.
p	The parameter values community was built at. Needed for a function written by the caller; for a map from community_parameter_map they are read from the community, and a p given must agree with them.
birth_rate	Birth rates to start each equilibrium solve from, one per resident; defaults to the residents' own.

Details

A singular point x^* is a root of the residents' selection gradients $g(x, p) = 0$, so where the resident Jacobian $J = \partial g/\partial x$ is non-singular it moves smoothly with the parameters, at

$$dx^*/dp = -J^{-1}\partial g/\partial p.$$

J is community_selection_gradient_jacobian (the mk x mk Jacobian for a coalition of m residents with k traits, through the environment they share, so a coalition's residents move together) and $\partial g/\partial p$ is community_selection_gradient_parameter_jacobian. A singular J is a fold or bifurcation of the singularity, where it does not move smoothly with p, and is an error.

Cost: 2mk equilibrium solves for J and two per parameter for $\partial g/\partial p$, all warm-started from the community.

The sensitivity is in raw trait units whatever the community's trait scale (on a log scale, $d \log x^*/dp$ is it divided by x^*). community is taken to be the model at p, as parameter(p) builds it; it is not re-solved to check, but with a map from community_parameter_map p is read from it. As in community_classify_singularity, a community marked as not converged, or whose residents a Newton step on their gradients would move by more than a thousandth of the bounds' width, is answered with a warning.

Value

An mk by np matrix of dx^*/dp, rows named as in community_selection_gradient_jacobian and columns by parameter, with attr(., "jacobian"), attr(.,"parameter_jacobian") and the equilibrium solves in attr(., "evaluations").

Author(s)

Daniel Falster

Examples

```
comm <- community_start(bounds(x = c(-2, 2)), trait_scale = "linear",
                        harness = harness_dd99(sigma_C = 0.4))
by <- community_parameter_map(comm, c("sigma_C", "sigma_K"))
pair <- community_solve_singularity(comm, x0 = trait_matrix(c(-0.3, 0.6), "x"))
# the dimorphic pair spreads apart as either kernel widens
community_parameter_sensitivity(pair, by)
```

---

community_parameter_values

Parameter values of a community

Description

The values of the parameters a parameter map varies, as a community carries them.

Usage

```
community_parameter_values(community, parameter)
```

Arguments

community	A community, typically built by parameter.
parameter	A map from community_parameter_map.

Value

A named numeric vector, the parameters in the order `parameter` takes them.

Author(s)

Daniel Falster

`community_pip` *Pairwise invasibility plot*

Description

Pairwise invasibility surface and its zero contours.

Usage

```
community_pip(
  community,
  resident = NULL,
  mutant = NULL,
  bounds = community$bounds,
  control = pip_control()
)
```

Arguments

`community`	A single-trait `community`; any residents it carries are discarded.
`resident`	Resident trait values for the initial grid. Defaults to `control$n_resident` values spaced on the trait scale across `bounds`.
`mutant`	Mutant trait values. Defaults to `control$n_mutant` values across `bounds`.
`bounds`	Trait bounds for the default grids; defaults to the community's.
`control`	See `pip_control`.

Details

For each resident trait value the community is reduced to that single resident, solved to demographic equilibrium, and the invasion fitness of every mutant value is evaluated against it. The sign of the resulting surface $s(y; x)$ is the pairwise invasibility plot: it shows where selection points, where the singular strategies are, and whether each is an ESS or a branching point.

Residents are the expensive axis (one equilibrium solve each) and are handled accordingly: a coarse subset is solved first, through the active `future::plan()`, and every other resident starts its solve from the equilibrium density interpolated between them, so each lands near its answer and all of them run in parallel; after the initial grid the resident axis is refined where neighbouring residents disagree about their zero contours (see `pip_control`). `residents$n_evals` records what each solve cost. Along the cheap mutant axis the contours $s(y; x) = 0$ are located exactly by Newton's method on the model's fitness gradient, with the diagonal root $y = x$ divided out so that the second contour is found even where it hugs the diagonal near a singular strategy.

Value

A pip object: a list with surface (a tibble of resident, mutant, fitness), contours (a tibble of resident, mutant on the zero set, including the diagonal), residents (each resident's equilibrium birth_rate, the number of demography evaluations its solve took, and its number of crossings), the trait name and scale, the control used, and elapsed seconds. as.data.frame() returns the surface. See plot.pip, pip_mutual, community_tep.

Author(s)

Daniel Falster

---

community_plot_fitness_landscape

Plot an invasion-fitness landscape

Description

Plots the sampled fitness landscape held on a community (community$fitness_points, from community_fitness_landscape), with the residents marked and the zero-fitness line that separates trait values that can invade from those that cannot. If the landscape has not been computed yet it is computed here.

Usage

```
community_plot_fitness_landscape(
  community,
  label = NA,
  xlim = NULL,
  ylim = NULL
)
```

Arguments

community	A community object.
label	Optional label drawn in the top-left corner (e.g. an assembly step number).
xlim	Length-2 trait range for the x axis. Defaults to the community's bounds.
ylim	Optional length-2 fitness range. Applied with coord_cartesian() so points outside it are clipped rather than dropped; NULL (the default) lets the data set the range.

Details

Where the community carries a Gaussian-process surrogate (the "bayesopt" landscape method) the surrogate's smooth prediction is drawn over the sampled points; otherwise the sampled points are joined directly. The x axis follows the community's trait scale — log for the strictly positive plant traits, linear for traits spanning zero.

Value

A ggplot object.

`community_selection_gradient`
Selection gradient of the residents

Description

The derivative of invasion fitness with respect to the mutant trait, evaluated at each resident's own traits: the direction in which selection pushes that resident. Solve the community to demographic equilibrium first (`community_demography`). Derivative settings come from `derivative_control`.

Usage

```
community_selection_gradient(community)
```

Arguments

`community`	A `community` solved to demographic equilibrium.

Value

The community with `selection_gradient` (a vector for a single resident, otherwise a residents-by-traits matrix) and `resident_fitness` set.

Author(s)

Daniel Falster

`community_selection_gradient_jacobian`
Resident Jacobian of the selection gradient

Description

Jacobian of the selection gradient with respect to the resident traits.

Usage

```
community_selection_gradient_jacobian(community, birth_rate = NULL)
```

Arguments

`community`	A `community` with one or more residents, no two of them within the finite-difference step of each other (an error: the stencil cannot tell them apart).
`birth_rate`	Birth rates to start each equilibrium solve from, one per resident; defaults to the residents' own. Every stencil point starts from the solve at the residents.

Details

How the selection gradients of the residents change as the residents move, each re-solved to demographic equilibrium: for one resident the `k` by `k` matrix $J_{ij} = dg_i/dx_j$; for `m` residents the `mk` by `mk` matrix of every resident's gradient with respect to every resident's traits, its gradients and traits stacked as the trait matrix is stored (trait by trait, residents within each). The off-diagonal blocks are how a resident's selection responds to the others through the environment they make. Eigenvalues with negative real parts mean selection carries nearby residents towards the point — convergence stability. By default each stencil point solves the perturbed community to demographic equilibrium, so this is the expensive derivative: `2mk` equilibrium solves per Richardson level. Each is solved from the densities and solver state at the residents, so under a `future::plan()` with several workers they run in parallel, with the same answer, and each to the cube of the relative step (see `derivative_control`). With `resident_jacobian = "implicit"` in `derivative_control` it differentiates the equilibrium condition instead: `2m(k + 1)` runs of the model at given traits and densities per Richardson level, and no equilibrium solve beyond the one at the residents.

Value

An `mk` by `mk` matrix, rows and columns named by trait (and `[i]` for resident `i` when there are several), with the stacked selection gradient at the residents in `attr(., "selection_gradient")`, the `m` by `mk` Jacobian of the residents' equilibrium birth rates (densities, for the reference models) from the same solves in `attr(., "birth_rate_jacobian")`, the equilibrium solves it cost in `attr(., "evaluations")` and the runs at given traits and densities in `attr(., "environments")`.

Author(s)

Daniel Falster

`community_selection_gradient_parameter_jacobian` *Parameter Jacobian of the selection gradient*

Description

Jacobian of the selection gradient with respect to model parameters.

Usage

```
community_selection_gradient_parameter_jacobian(
  community,
  parameter,
  p = NULL,
  birth_rate = NULL
)
```

Arguments

community	A community with one or more residents, built by parameter(p).
parameter	A function taking a numeric vector like p and returning a community with the same traits (its residents are replaced by those of community).
p	The parameter values at which community was built. Needed for a function written by the caller; for a map from community_parameter_map they are read from the community (community_parameter_values), and a p given must agree with them.
birth_rate	Birth rates to start each equilibrium solve from, one per resident; defaults to the residents' own.

Details

How the selection gradients of the residents change as model parameters move, the residents held at their traits and re-solved to demographic equilibrium under the changed model: the mk by np matrix $\partial g/\partial p$, its rows stacked as in community_selection_gradient_jacobian. With that Jacobian it gives the sensitivity of a singular strategy or coalition (community_parameter_sensitivity).

A parameter enters through parameter, a function from a parameter vector to a community: community_parameter_map builds one for a model with an explicit harness, and any function that rebuilds the community from p will do for another model or for an input that is not a harness parameter. Each column is a central difference, two equilibrium solves per Richardson level, warm-started from the residents' densities and solved to the cube of the relative step, with steps from derivative_control (d_parameter, eps_parameter, r_parameter) of the community. Under a future::plan() with several workers the solves run in parallel, with the same answer, and parameter runs on the workers: under a plan whose workers are separate processes (anything but multicore) it must not hold a solved plant community, whose fitness function holds the SCM; that is an error.

Value

An mk by np matrix, rows named as in community_selection_gradient_jacobian and columns by names(p), with the equilibrium solves it cost in attr(., "evaluations").

Author(s)

Daniel Falster

Examples

```
comm <- community_start(bounds(x = c(-2, 2)), trait_scale = "linear",
                        harness = harness_dd99(x0 = 0.3)) |>
  community_add(trait_matrix(0, "x"), birth_rate = 100) |>
  community_demography()
# away from the singular strategy, moving its optimum x0 tilts selection
community_selection_gradient_parameter_jacobian(
  comm, community_parameter_map(comm, c("x0", "sigma_K")))
```

`community_solve_boundary`
Solve for a coexistence boundary point

Description

Solve for a point on a coexistence boundary.

Usage

```
community_solve_boundary(
  community,
  parameter,
  y = NULL,
  type = c("invasion", "branching"),
  free = 1L,
  which = 1L,
  p = NULL,
  tol = 1e-08,
  maxit = 50
)
```

Arguments

`community`	A `community` whose residents start the solve (their densities, if positive, seed the first equilibrium), built by `parameter(p)`.
`parameter`	A function `p -> community`; see `community_parameter_map`.
`y`	For `type = "invasion"`, the marginal mutant's starting traits, one value per trait.
`type`	`"invasion"` or `"branching"`.
`free`	The parameter solved for, by name or position in `p`; the others are held.
`which`	For `type = "branching"`, the resident becoming a branching point.
`p`	The parameter values `community` was built at, the free one's starting value among them. Read from the community for a map from `community_parameter_map`.
`tol`, `maxit`	Passed to `util_nlsolve`: the tolerance on every residual and the iteration limit.

Details

A boundary between regions supporting n and $n + 1$ species is the solution set of an augmented system: the n residents at their singular coalition, each at demographic equilibrium, and a condition that makes the community marginal. Two kinds of boundary:

`"invasion"` a new peak of the fitness landscape, away from the residents, crossing zero: a mutant y^* at a critical point of invasion fitness, $\partial s/\partial y(y^*) = 0$, with $s(y^*) = 0$. Unknowns: the residents' traits, y^* and one parameter; $nk + k + 1$ equations in as many unknowns.

`"branching"` a resident becoming a branching point, where the pair it branches into emerges from it (a pitchfork): the determinant of resident `which`'s Hessian in the mutant direction is zero. Unknowns: the residents' traits and one parameter.

With the other parameters fixed the system is square and is solved by the package's Newton iteration (`util_nlsolve`, `solver = "newton"`), the densities solved to equilibrium inside each evaluation (`community_demography`). Its Jacobian comes from the derivative layer: the residents' columns by finite differences across the residents, each point solved to the cube of its step and the points through the parallel map, the marginal mutant's from the exact mutant-direction derivatives, the free parameter's by finite differences across the parameter; so `2nk + 2` equilibrium solves per Jacobian until the model supplies equilibrium sensitivities.

The start is a boundary point found by `boundary_bisect`: the community at the side with fewer species, `y` the newcomer's trait on the other side, and the bracket's midpoint as `p`. y^* must stay away from the residents: at a resident the conditions hold for any parameter (its own fitness is zero, and its gradient too at a singular coalition), so a start or an answer there is refused. An answer at which y^* is not a strict maximum of fitness warns: there the boundary is not where a new peak crosses zero.

Traits and parameters are solved in raw units, whatever the community's trait scale.

Value

The community at the boundary point, solved to demographic equilibrium at the parameters found, with `attr(., "boundary")`: a list of `type`, `p` (all the parameters, named), `free`, `y` (`NULL` for a branching boundary), `residual`, `converged`, `message`, `iterations` and `evaluations` (equilibrium solves).

Author(s)

Daniel Falster

Examples

```
gk <- function(p) {
  community_start(bounds(x = c(-4, 4)), trait_scale = "linear",
                  harness = harness_gk98(mu = c(-p[1], 0, p[1]))) |>
    community_add(trait_matrix(0.5, "x"), birth_rate = 1)
}
# the pair of specialists on the outer patches, and the middle patch's
# peak, near the spacing at which a third species invades
pair <- community_start(bounds(x = c(-4, 4)), trait_scale = "linear",
                        harness = harness_gk98(mu = c(-1.95, 0, 1.95))) |>
```

```
  community_add(trait_matrix(c(-1.3, 1.3), "x"), birth_rate = c(1.5, 1.5))
at <- community_solve_boundary(pair, gk, y = 0, p = 1.95)
attr(at, "boundary")$p
```

`community_solve_singularity` *Solve for a singular strategy or coalition (N-dimensional)*

Description

Find a singular strategy or a singular coalition.

Usage

```
community_solve_singularity(
  community,
  x0 = NULL,
  bounds = NULL,
  solver = c("nleqslv", "newton", "dfsane", "bracket"),
  tol = 1e-06,
  maxit = 100,
  birth_rate = NULL,
  edge_ok = TRUE,
  jacobian = NULL
)
```

Arguments

`community`	A `community` object to search within.
`x0`	Starting traits, one row per resident (a bare vector is one resident with one value per trait, whatever the number of traits; several residents need a matrix, e.g. `trait_matrix(c(-0.3, 0.6), "x")`). Defaults to the community's residents, or, if it has none, the midpoint of `bounds` on the community's trait scale.
`bounds`	A k by 2 matrix of lower/upper bounds (a length-2 vector is accepted when there is a single trait). Defaults to the community's bounds.
`solver`	Root finder: `"nleqslv"` (default), `"newton"`, `"dfsane"`, or `"bracket"` (one resident, one trait).
`tol`	Convergence tolerance passed to the solver: on the residual for `"nleqslv"`, `"newton"` and `"dfsane"`, on the root (on the trait scale) for `"bracket"`.
`maxit`	Maximum solver iterations.
`birth_rate`	Birth rates to start the first equilibrium solve from, one per resident. Defaults to the residents' own if the community's residents are being searched, otherwise `birth_rate_initial`; thereafter each solve warm-starts from the previous one. Worth setting when the model's equilibrium birth rates are far from `birth_rate_initial`.

edge_ok — Is it (not) an error if the solution lands on the edge of bounds?

jacobian — A resident Jacobian of the selection gradients, mk by mk in raw trait units as community_selection_gradient_jacobian returns it, to start "nleqslv" or "newton" from in place of computing one at x0: typically that of a nearby singular point. Later Jacobians are computed as usual.

Details

A singular strategy is a resident trait combination at which the selection gradient vanishes; a singular coalition is a set of coexisting residents at each of which its own selection gradient vanishes, in the environment they make together (the residents a population reaches after evolutionary branching). The solver finds either: the stacked selection gradients of all residents are handed to a root finder, the community being re-solved to demographic equilibrium at every evaluation. One resident is the monomorphic singular strategy.

Root finders: nleqslv (Broyden with Newton restarts), newton (the package's Newton–Broyden iteration with a backtracking line search, see util_nlsolve) and dfsane (derivative-free), all local searches from x0; or, for one resident with a single trait, "bracket", which brackets the scalar gradient between the bounds with uniroot and so needs a sign change across them but then cannot miss the root inside. nleqslv and newton take the Jacobian of the gradients with respect to the residents from community_selection_gradient_jacobian (finite differences across the residents, 2mk equilibrium solves, until the model supplies equilibrium sensitivities), refreshing it only when their rank-one updates fail; under a future::plan() its solves run in parallel (derivative_control). A Jacobian already in hand for a nearby point (jacobian, as in a continuation) answers their first request instead, saving those solves.

Each residual evaluation costs one demographic equilibrium solve, warm started from the last, plus a gradient of invasion fitness at each resident. The residents searched are those of x0, by default the community's own; a community without residents is searched from the midpoint of bounds for a monomorphic singular strategy.

The search runs on the community's trait scale (see community_start(trait_scale =)): for "log" traits the root is sought in log(x) and the residual is the gradient with respect to log(x), which is far better conditioned for strictly positive biological traits. The root is the same either way. Candidate points are clamped to bounds, and hitting a bound produces a warning: the search region did not contain a singularity. With "bracket" the same happens when the gradient has one sign across the bounds, and the returned point is the bound selection pushes towards. A coalition that loses a resident on the way — one whose equilibrium density is no longer positive, or two that merge — is not a coalition of that size: the solve is reported as not converged, with a warning naming the residents.

Value

The community at the singular strategy or coalition, solved to demographic equilibrium, with selection_gradient set. Attributes record the solve: converged, solver, evaluations (the equilibrium solves it cost) and singularity (the root: named by trait for one resident, a residents-by-traits matrix for several).

Author(s)

Daniel Falster

See Also

community_classify_singularity to determine whether the point found is a CSS, a branching point, a repeller or a Garden of Eden.

Examples

```
comm <- community_start(bounds(x = c(-2, 2)), trait_scale = "linear",
                        harness = harness_dd99(sigma_C = 0.4))
# the monomorphic singular strategy, then the dimorphic coalition it
# branches into
one <- community_solve_singularity(comm, x0 = 0.5)
pair <- community_solve_singularity(comm, x0 = trait_matrix(c(-0.3, 0.6), "x"))
pair$traits
community_classify_singularity(pair)
```

---

community_start *Initialise a community object*

Description

Initialise a community object

Usage

```
community_start(
  bounds,
  birth_rate_initial = 0.001,
  demography_control = demographic_step_control(),
  fitness_control = NULL,
  derivative_control = NULL,
  model_support = NULL,
  harness = NULL,
  trait_scale = c("log", "linear")
)
```

Arguments

bounds	A set of bounds, as specified in plant.
birth_rate_initial	A vector of birth rates.
demography_control	List controlling the demography/equilibrium solver. See demographic_step_control.
fitness_control	List of parameters controlling how invasion-fitness landscapes are generated. See fitness_landscape_control. Values given here override the defaults; NULL uses them all.

`derivative_control`
: List controlling how derivatives of fitness are computed. See `derivative_control`.

`model_support`
: Model support data passed to the harness (e.g. the `plant` parameters and control objects).

`harness`
: A model harness wiring the community to a demographic backend (see `harness_plant`); defaults to the `plant` SCM.

`trait_scale`
: How trait space is spaced/searched during assembly: `"log"` for strictly positive traits or `"linear"` for traits spanning zero.

Details

Used to store full description of community. Is a list with elements parameters, bounds, birth_rate_initial, trait_names, traits, birth_rate, fitness_control.

Value

An `community` object.

Author(s)

Rich FitzJohn, Daniel Falster

`community_tep` *Trait-evolution plot*

Description

Trait-evolution plot: selection on protected dimorphisms.

Usage

```
community_tep(community, pip)
```

Arguments

`community`
: The `community` the surface was computed from.

`pip`
: A `pip` from `community_pip`.

Details

For every mutually invasible pair of residents the two are introduced together, the dimorphic community is solved to demographic equilibrium (seeded from the pair's monomorphic densities), and the selection gradient on each resident is recorded. Drawn as arrows over the coexistence region this is the trait-evolution plot of Geritz et al. (1998): it shows where a dimorphism formed by branching is carried next, and whether a dimorphic singular coalition (both gradients zero) exists. Pairs where one strategy is driven out at the dimorphic equilibrium are dropped and counted in `attr(., "excluded")`.

Cost: one two-resident equilibrium solve per mutually invasible pair above the diagonal, through the same parallel map as `community_pip`.

Value

A `tep` object: a tibble with one row per coexisting pair (`x1 < x2`), the equilibrium state `n1`, `n2` and selection gradients `g1`, `g2`. See `plot.tep`.

Author(s)

Daniel Falster

`community_viable_fitness` *Compute region of positive fitness for a community*

Description

Compute the region of positive (fundamental) fitness for a community.

Usage

```
community_viable_fitness(community, x = NULL, dx = 1)
```

Arguments

`community`	A `community` object (usually empty).
`x`	Trait values inside the region to search out from. Defaults to the fitness maximum within the community's bounds (`max_fitness`), which is also used when fitness at `x` is negative.
`dx`	Initial step outwards when bracketing each edge, on the trait scale; it doubles until fitness turns negative or a bound is reached.

Details

Finds the trait region over which a strategy has positive invasion fitness into the community's environment, as the smallest box of trait bounds that contains it. This is normally run on an *empty* community, in which case it is the fundamental niche: where a lone strategy can persist with no competitors.

Along each trait the box runs as far as the region does: it is the interval over which the *profile* of fitness, its maximum over the other traits with this one held fixed, stays positive. That interval is bracketed and refined by root-finding outwards from the fitness maximum, on the community's trait scale. With one trait the profile is the fitness itself; with `k` traits each profile value is a `k - 1` trait maximisation, which dominates the cost.

Value

A bounds matrix (`lower`/`upper` columns, one row per trait), or `NULL` if fitness is nowhere positive within the bounds. An edge that reaches the community's bounds is returned as that bound.

Author(s)

Rich FitzJohn, Daniel Falster

`demographic_step_control` *Options controllings community assembly process.*

Description

Controls how community assembly works.

Usage

```
demographic_step_control(control = NULL)
```

Arguments

`control`	A list of values to modify from defaults.

Details

Returns a list. Passing in a list of value via `control` will override the defaults. Options include run_type determines whether population is stepped to demographic equilibrium ("to_equilibrium") or not ("single_step"). "birth_type" determines sampling of new types – "stochastic" or "maximum" (on fitness peak). With "stochastic" births, "n_mutants" and "n_immigrants" determine the frequency of resident mutations and immigrations from global pool. "vcv" is variance-covariance matrix for mutations. If "birth_move_tol" is trait distance within which we attempt to move an existing resident rather introduce a new type (this helps reduce the number of types). "compute_viable_fitness" asks whether to check bounds of viable trait space. "check_positive" determines whether the fitness of an invader is checked before it is introduced. If "check_inviable" causes dead residents to be removed when birth rate drops below "dead_birth_rate". "eps_too_close" is tolerance in trait values when searching for maxima.

Value

A list with elements run_type, birth_type, birth_move_tol, compute_viable_fitness, n_mutants, n_immigrants, check_positive, vcv, check_inviable, dead_birth_rate, eps_too_close

Author(s)

Rich FitzJohn, Daniel Falster

`derivative_control` *Derivative settings*

Description

Control how derivatives are computed.

Usage

```
derivative_control(control = NULL)
```

Arguments

`control`	A list of values to modify from the defaults.

Details

Returns the list of settings read by the derivative functions (`community_fitness_gradient`, `community_fitness_hessian`, `community_selection_gradient_jacobian`). Pass a list of values to override the defaults; the result lives at `community$derivative_control` (see `community_start`).

Finite-difference steps are relative (`d_*`) with an absolute fallback (`eps_*`) for trait values that are numerically zero, as the singular strategies of the reference models are. First derivatives of fitness use the `*_gradient` settings; the Hessian and the resident Jacobian use `*_second`; the derivative of the selection gradient with respect to model parameters (`community_selection_gradient_parameter_jacobian`) uses `*_parameter`. The `r_*` settings are the number of successively halved steps combined by Richardson extrapolation; each level of a resident or parameter Jacobian costs two demographic equilibrium solves per column, so they stay at one by default.

Every point of a resident Jacobian is solved from the densities and solver state at the centre of the stencil, and every point of a parameter Jacobian from the residents' densities, so the solves are independent: a `future::plan()` with several workers solves them in parallel, with the same answer as without one. An error in a solve enters a finite difference divided by the step, and a central difference is itself only good to about the square of its step, so each point is solved to the cube of the stencil's relative step (`d_*`, or `eps_*` at a trait or parameter of zero, halved at each further Richardson level), or to the community's `equilibrium_eps` (`demographic_step_control`) where that is tighter: the derivative's error is then set by its step, not by the equilibrium tolerance. With Richardson levels that is the error of a single difference at the finest step; a tighter `equilibrium_eps` lets the extrapolation go further. A larger step relaxes the tolerance; a model whose demography cannot be solved that finely warns that the derivative carries the shortfall.

`resident_jacobian` chooses how the resident Jacobian differentiates through the demographic equilibrium. `"stencil"` differences the selection gradient across equilibrium solves, as above. `"implicit"` differentiates the equilibrium condition instead: every resident's fitness is zero there, $s(x, n^*) = 0$, so $dn^*/dx = -S_n^{-1} S_x$ and $dg/dx = G_x + G_n\, dn^*/dx$, with S the residents' fitness and G their selection gradients, both differentiated at a fixed resident environment, at the `*_second` steps. Each point is then one run of the model at given traits and densities, with no equilibrium solve, and its accuracy does not depend on solving anything to the cube of the step.

Value

A list with elements d_gradient, eps_gradient, r_gradient, d_second, eps_second, r_hessian, r_jacobian, d_parameter, eps_parameter, r_parameter, resident_jacobian.

Author(s)

Daniel Falster

`fitness_landscape_control` *Options controlling fitness-landscape construction*

Description

Controls how invasion-fitness landscapes are sampled.

Usage

```
fitness_landscape_control(control = NULL)
```

Arguments

control	A list of values to modify from the defaults.

Details

Returns a list of options read by community_fitness_landscape and its methods. Passing a list via control overrides the defaults; unknown names are an error. This is the fitness_control slot of a community (see community_start), and is the fitness analogue of demographic_step_control.

Options:

- method — "grid" (the default: evenly spaced points on the community's trait scale, augmented with the residents) or "bayesopt" (Gaussian-process surrogate via **mlr3mbo**, fitted on the community's trait scale; it draws random numbers when fitting the surrogate, so set a seed beforehand for a reproducible landscape).
- n_evals — number of fitness evaluations.
- n_init — size of the initial design for "bayesopt".

Value

A list with elements method, n_evals, n_init.

Author(s)

Daniel Falster

`harness_check_derivatives` *Verify model-supplied derivatives*

Description

Check a harness's derivatives against finite differences.

Usage

```
harness_check_derivatives(
  community,
  n_points = 5,
  tol_rel = 1e-04,
  tol_abs = 1e-06
)
```

Arguments

`community`	A `community`; solved to demographic equilibrium if it has residents (an empty community is solved here).
`n_points`	Number of trait points to test at.
`tol_rel`, `tol_abs`	An entry passes when `abs(model - fd) <= tol_abs + tol_rel * abs(fd)`.

Details

Every derivative a model supplies must agree with a finite difference of its own fitness function; this is the contract a harness author signs by advertising a derivative (`harness$provides`). The check evaluates each advertised quantity at `n_points` well-spread points inside the community's bounds (on its trait scale) and compares every entry with a three-level Richardson finite difference. A harness that advertises nothing has nothing to check and returns an empty table.

Call it from a package's own tests with a tolerance suited to the model: the reference models agree to machine precision; a model whose fitness comes from an adaptive ODE integration needs something like `tol_rel = 1e-2`. The absolute tolerance exists for entries that are close to zero, where the finite-difference reference itself is only good to its roundoff floor — about `1e-7` for a second difference of an order-one fitness with the default steps.

Value

Invisibly, a data frame with one row per entry checked: `quantity`, `point`, `entry`, `model`, `fd`, `abs_err`, `rel_err`, `pass`. Any failure is an error of class `derivative_check_error` carrying the table as `$results`.

Author(s)

Daniel Falster

`harness_dd99` *DD99 (Dieckmann & Doebeli 1999) harness*

Description

DD99: Dieckmann & Doebeli 1999 competition model (Nature 400:354-357).

Usage

```
harness_dd99(
  r = 1,
  K0 = 500,
  x0 = 0,
  sigma_K = 1,
  sigma_C = 0.4,
  trait_name = "x"
)
```

Arguments

`r`	intrinsic growth rate
`K0`	maximum carrying capacity
`x0`	trait optimising the resource (singular strategy)
`sigma_K`	width of the resource/carrying-capacity kernel
`sigma_C`	width of the competition kernel
`trait_name`	name of the evolving trait

Details

Continuous-time logistic competition for a Gaussian resource: carrying capacity `K(x) = K0 exp(-(x-x0)^2/(2 sigma_K^2))` and competition kernel `C(d) = exp(-d^2/(2 sigma_C^2))`. Invasion fitness `s(y) = r (1 - sum_i N_i C(y-x_i)/K(y))`. The singular strategy is `x* = x0`; it is an evolutionary *branching point* iff `sigma_C < sigma_K` and an ESS iff `sigma_C > sigma_K` – the classic disruptive-selection oracle.

Value

a 'harness' object

Author(s)

Daniel Falster

`harness_dd99_nd` *DD99 multi-trait harness*

Description

DD99 in two (or more) trait dimensions (cf. Ito & Dieckmann 2007).

Usage

```
harness_dd99_nd(
  r = 1,
  K0 = 500,
  x0 = c(0, 0),
  sigma_K = c(1, 1),
  sigma_C = c(0.4, 0.4),
  trait_names = c("x1", "x2")
)
```

Arguments

`r`	intrinsic growth rate
`K0`	maximum carrying capacity
`x0`	length-k vector of per-dimension optima (the singular strategy)
`sigma_K`	length-k vector of resource-kernel widths
`sigma_C`	length-k vector of competition-kernel widths
`trait_names`	length-k character vector naming the traits

Details

The multi-trait extension of `harness_dd99`: the resource and competition kernels are products of per-dimension Gaussians, so each trait dimension ‘d‘ has its own optimum `x0[d]` and widths `sigma_K[d]`, `sigma_C[d]`. The singular strategy is `x* = x0` and dimension ‘d‘ is a branching direction iff `sigma_C[d] < sigma_K[d]`. Use `trait_scale = "linear"` in `community_start()` since the optima sit at 0.

Value

a ‘harness‘ object

Author(s)

Daniel Falster

`harness_explicit` *Explicit (toy-model) harness*

Description

A harness for fast toy models with explicitly-supplied fitness/equilibrium.

Usage

```
harness_explicit(
  fitness,
  equilibrium,
  pars,
  trait_names,
  label = "explicit",
  fitness_gradient = NULL,
  fitness_hessian = NULL
)
```

Arguments

`fitness`	function(x_mut, x_res, n_res, pars) -> numeric vector of invasion fitness, one per mutant in `x_mut`.
`equilibrium`	function(x_res, pars) -> numeric vector of resident equilibrium densities, one per resident.
`pars`	named list of model parameters.
`trait_names`	character vector naming the evolving trait(s).
`label`	short model label (used in the harness class).
`fitness_gradient`	optional function(x_mut, x_res, n_res, pars) -> the gradient of invasion fitness with respect to the mutant trait(s), one row per mutant and one column per trait.
`fitness_hessian`	optional function(x_mut, x_res, n_res, pars) -> the k by k Hessian of invasion fitness with respect to the mutant trait(s) for a single mutant.

Details

Implements the connectors generically in terms of two model primitives: a vectorised invasion-fitness function and the model's own resident equilibrium — analytic where the model has one, otherwise from the model's own internal solver (both typically backed by C++) — plus optional derivatives of fitness in the mutant direction. Concrete instances — `harness_dd99`, `harness_gk98`, `harness_gm99`, `harness_jj12` — supply all of them.

The demography runner is the model's own one-generation map $n_{t+1} = n_t \exp(s(x; x, n_t))$ — for GK98 its soft-selection recursion (Eq. 19), for JJ12 the territory recursion, for GM99 the seeds-per-seed return, for DD99 (a per-capita rate) a unit time step — so every equilibrium solver in

demographic_step_control can be run and timed on these models against the model's own answer. That answer is the "model" solver and the default these harnesses recommend; community_start() adopts it unless the control names another. "Explicit" refers to the mechanism (the fitness/equilibrium are computed directly, not by running the plant SCM), NOT a claim that every quantity is closed-form.

Conventions:

- community$birth_rate stores resident abundance/density (the state the equilibrium solver returns), not a plant offspring rate.
- fitness returns invasion fitness with the resident value ~0 (a log ratio for jj12/gk98/gm99; a per-capita rate for dd99).
- the "model" solver returns the model's own equilibrium in one evaluation; every other solver iterates the one-generation map.

Value

a 'harness' object

Author(s)

Daniel Falster

---

harness_fd *Finite-difference derivatives for a harness*

Description

Supply finite-difference derivatives for whatever a harness lacks.

Usage

```
harness_fd(harness)
```

Arguments

harness	A harness.

Details

The methods in this package are written against exact derivatives of invasion fitness (community_fitness_gradient, community_fitness_hessian). A model that cannot differentiate itself still works, at the cost and accuracy of finite differences, once its harness has been through this function: it wraps the harness's fitness-function connector so that any derivative not provided by the model is computed by the finite-difference stencils in R/util_gradient.R, with the settings in derivative_control. Derivatives the model does provide are left alone, and the source of each is recorded in community$fitness_derivatives$source.

community_start applies this to every harness, so it is only needed when building a community by hand. Applying it twice is harmless.

Value

The harness, with its fitness-function connector wrapped.

Author(s)

Daniel Falster

`harness_gk98` *GK98 (Geritz et al. 1998 soft-selection) harness*

Description

GK98: Geritz, Kisdi, Meszena & Metz 1998 soft-selection model (Evol. Ecol. 12:35-57; the worked Levene example).

Usage

```
harness_gk98(
  d = 1.5,
  sigma = 1,
  mu = c(-d, 0, d),
  K = rep(1, length(mu)),
  trait_name = "x"
)
```

Arguments

`d`	patch-optimum spacing for the symmetric default (optima -d, 0, d)
`sigma`	within-patch survival width
`mu`	patch optima (overrides the symmetric default built from `d`)
`K`	patch capacities (defaults to equal capacities)
`trait_name`	name of the evolving trait

Details

`m` patches with optima `mu` and capacities `K`; Gaussian within-patch survival of width `sigma`. Invasion fitness reduces, for a single resident, to `S(y) = log sum_j c_j f_j(y)/f_j(x)` with `c_j = K_j/sum K`. For the symmetric three-patch default (`mu = (-d, 0, d)`, equal `K`) the singular strategy is `x* = 0`: convergence stable for all `d/sigma`, and an evolutionary branching point iff `d/sigma > sqrt(3/2) ~= 1.2247` (a CSS below that).

Value

a 'harness' object

Author(s)

Daniel Falster

`harness_gm99` *GM99 (Geritz et al. 1999 seed-size) harness*

Description

GM99: Geritz, van der Meijden & Metz 1999 seed-size safe-site model.

Usage

```
harness_gm99(alpha = 6, beta = 25, R = 1, trait_name = "x")
```

Arguments

`alpha`	competitive asymmetry (larger = more size-asymmetric)
`beta`	habitat parameter controlling pre-competitive survival
`R`	resources per safe site (only alpha*R and beta*R matter)
`trait_name`	name of the evolving trait (seed size)

Details

Plants compete for safe sites; seeds arrive by a Poisson process and seedlings undergo size-asymmetric lottery competition. Pre-competitive survival `s(x) = max(0, 1 - 2 exp(-beta x))`, competitive ability `c(x) = exp(alpha x)`, invasion fitness (a lifetime reproductive ratio) `W = (R/x) s(x) g(x_mut, x, N)` (mutant trait `x_mut` against resident `x`) averaged over the Poisson number of competitors. Only the products `alpha*R` and `beta*R` matter. There is no closed-form singular strategy; size-asymmetric competition (large `alpha`) drives evolutionary branching in seed size (see Fig. 5 of the paper: e.g. `alpha*R = 4.5` vs `7.0` at `beta*R = 15`).

This is the Geritz *1999* seed-size model (the one in the original MATLAB code), distinct from the GK98 1998 soft-selection model (`harness_gk98`).

Value

a 'harness' object

Author(s)

Daniel Falster

`harness_jj12` *JJ12 (Johansson & Jonzen 2012 bird arrival) harness*

Description

JJ12: migratory-bird arrival-time model (Johansson & Jonzen 2012; the simplified analytic form of Brannstrom, Johansson & von Festenberg 2013, Games 4:304-328, section 4).

Usage

```
harness_jj12(
  a = 0.1,
  x_opt = 0,
  sigma = 1,
  R0 = 1,
  K = 1,
  p = 0.5,
  trait_name = "x"
)
```

Arguments

`a`	competitive advantage of early arrival (>= 0)
`x_opt`	seasonal optimum arrival time
`sigma`	width of the benign season
`R0`	maximum reproductive output
`K`	number of territories
`p`	year-to-year survival, in (0, 1)
`trait_name`	name of the evolving trait

Details

Trait = arrival time. Early arrival raises competitive ability `C(x) = exp(-a x)` for a fixed number `K` of territories; reproduction `R(x) = R0 exp(-(x - x_opt)^2 / (2 sigma^2))` peaks at the seasonal optimum; survival `p`. The model has a single continuously stable strategy (no branching) with the closed form

$$x^* = x_{opt} - a\,\sigma^2,$$

which arrives *before* the population optimum (a tragedy of the commons). This exact answer makes the model a benchmark for numerical selection-gradient / singular-strategy solvers.

Value

a 'harness' object

Author(s)

Daniel Falster

`harness_plant` *plant model harness*

Description

The default model harness: the full ‘plant‘ SCM.

Usage

```
harness_plant(
  model = c("FF16", "TF24"),
  version = as.character(utils::packageVersion("plant")),
  fitness_gradient = NULL
)
```

Arguments

`model`	plant physiological model: "FF16" or "TF24"
`version`	plant package version (defaults to the installed version)
`fitness_gradient`	`"fd"` or `"model"`: where the fitness gradient comes from. By default `"fd"` for FF16 and `"model"` for TF24.

Details

Forwards each connector to the existing ‘plant_community_*‘ implementation, so the working plant path is unchanged. It is identified by the plant physiological model (‘FF16‘, the current default, or ‘TF24‘) and the plant package ‘version‘, mirroring how the toy harnesses are named by author/year.

The actual ‘plant‘ ‘Parameters‘ are still supplied via ‘community_start(model_support = ...)‘, built for the same model (`plant_default_assembly_pars(model = )`).

The resident equilibrium is found by fixed-point iteration for FF16 and by Newton's method for TF24, whose one-generation map overcompensates: from a sparse stand it overshoots to a dense one, which crashes, and iteration settles on that two-cycle rather than the equilibrium.

The fitness gradient in the mutant direction is either a finite difference of the fitness function (`fitness_gradient = "fd"`) or plant's own (`"model"`: `plant::stand_gradient()` on the mutants' replay in the recorded resident canopy, through the hyperparameter function), exact on the replay's discretisation, with a finite difference wherever plant refuses a point; it needs the birth-date coordinate (`plant_default_assembly_control`). Each model defaults to the cheaper: on FF16 a sweep costs about three times the finite difference at one or two traits, and on TF24 it costs the same at one trait and less from two.

Value

a ‘harness‘ object

Author(s)

Daniel Falster

`harness_provides` *Query a harness's derivatives*

Description

Which derivatives a harness supplies itself.

Usage

```
harness_provides(harness, what = NULL)
```

Arguments

`harness`	A `harness`.
`what`	`"fitness_gradient"`, `"fitness_hessian"` or `"equilibrium"` (the model's own resident equilibrium, analytic or from its own internal solver). With `NULL`, the names of all it provides.

Value

A logical, or with `what = NULL` a character vector.

Author(s)

Daniel Falster

`length.assembler` *Returns number taken by an* `assembler` *object*

Description

Returns number taken by an assembler object

Usage

```
## S3 method for class 'assembler'
length(x)
```

Arguments

`x`	An `assembler` object.

Value

Length of history in assembler object.

Author(s)

Rich FitzJohn, Daniel Falster

`length.community` *Returns number of types in* `community`

Description

Returns number of types in community

Usage

```
## S3 method for class 'community'
length(x)
```

Arguments

`x`	A `community` object.

Value

number of rows in trait matrix.

Author(s)

Rich FitzJohn, Daniel Falster

`max_fitness` *Find point of maximum fitness within some range*

Description

Find the trait value of maximum fitness within some bounds.

Usage

```
max_fitness(community, bounds = NULL, tol = 0.001)
```

Arguments

community	A community object.
bounds	Bounds matrix (lower/upper per trait). Defaults to community$bounds.
tol	Tolerance on the trait scale for the one-trait search; with more traits L-BFGS-B uses its own convergence test.

Details

Searches bounds for the trait value(s) that maximise invasion fitness into the community's environment (the fundamental niche peak for an empty community), on the community's trait scale (see community_start(trait_scale =)). One trait is searched by golden section over the whole interval; more traits by L-BFGS-B from the centre of the bounds, which finds the maximum nearest it.

Value

The maximising trait value(s), named by trait, with the achieved fitness in attribute "fitness".

Author(s)

Daniel Falster, Rich FitzJohn

---

max_growth_rate *Fitness of trait values against a community*

Description

Fitness of trait value(s) evaluated against a community.

Usage

```
max_growth_rate(community, values)
```

Arguments

community	A community object.
values	A vector (1D) or matrix (multi-trait) of trait values, as for trait_matrix.

Details

Returns the invasion fitness (log net reproduction ratio) of one or more mutant trait values in the community's environment. For an *empty* community this is the fundamental fitness (growth with no competitors).

Reimplemented on the community machinery: it evaluates community$fitness_function (built by plant_community_update_fitness_function), replacing plant's removed fundamental_fitness() / fitness_landscape().

Value

A numeric vector of fitnesses, one per trait value.

Author(s)

Daniel Falster, Rich FitzJohn

`mutational_vcv_proportion` *Generate mutational covariance matrices*

Description

Generate a mutational variance covariance matrix where each trait mutates by a fraction of its total range, and where there is no covariance among traits. This is a helper function for use with the stochastic mutation kernel `make_mutation_stochastic_naive`.

Usage

```
mutational_vcv_proportion(x, p = 0.001)
```

Arguments

`x`	A community object, or appropriate matrix of bounds
`p`	Fraction of trait range to mutate (vectors will be recycled according R's usual rules).

Details

The trait range is measured on the community's trait scale: the log scale by default (so the variance is a proportion of the log range, as for strictly-positive traits), or the linear scale when the community was created with `trait_scale = "linear"`. A bare bounds matrix has no scale attached and is treated as log, preserving the historical behaviour.

`pip_control` *Pairwise-invasibility settings*

Description

Control how a pairwise invasibility surface is computed.

Usage

```
pip_control(control = NULL)
```

Arguments

`control` A list of values to modify from the defaults:

`n_resident` initial number of residents across the bounds, spaced on the trait scale. Each is an equilibrium solve.

`n_mutant` number of mutant values per resident. Cheap; this sets the resolution of the shaded surface and of contour detection.

`seed` how each resident's equilibrium solve is started. `"interpolate"` (default): `n_coarse` residents are solved first and the rest start from the density interpolated between them, on the trait scale and in log density. `"neighbour"`: residents are solved in order, each from the previous one's density, in one contiguous chunk per worker. `"cold"`: every solve starts from the community's default birth rate.

`n_coarse` number of residents in the first pass of `seed = "interpolate"`.

`refine` rounds of resident refinement: a resident is inserted between neighbours where the zero contours bend — a contour appears or disappears between them, or its second difference across three consecutive residents exceeds `refine_tol` of the mutant range.

`refine_tol` contour curvature (second difference of a crossing across three residents, as a fraction of the mutant range on the trait scale) that triggers refinement.

`tol` convergence tolerance of the contour root-finding, on the trait scale.

`parallel` evaluate residents through the active `future::plan()` when one is set.

Value

A list of the settings above.

Author(s)

Daniel Falster

`pip_mutual` *Mutual invasibility*

Description

Mutually invasible pairs of residents.

Usage

```
pip_mutual(pip)
```

Arguments

`pip` A `pip` from `community_pip`.

Details

Two strategies can form a protected dimorphism when each invades the other's equilibrium: $s(x_2; x_1) > 0$ and $s(x_1; x_2) > 0$. Both are read from the surface, so no further model evaluations are needed.

Value

A tibble with one row per ordered pair of residents: `x1`, `x2`, `s12` (fitness of `x2` invading `x1`), `s21`, and `mutual`.

Author(s)

Daniel Falster

`plant_community_check_for_inviable_strategies` *Check low-abundance strategies for viability*

Description

Check low-abundance strategies for viability.

Usage

```
plant_community_check_for_inviable_strategies(community)
```

Arguments

`community`	A community object

`plant_default_assembly_control` *Default plant control for community assembly*

Description

Default plant SCM control for community assembly.

Usage

```
plant_default_assembly_control(...)
```

Arguments

`...`	Passed to `plant::control()` to override defaults.

Details

Thin wrapper around `plant::control()`. Kept as a named helper so the plant control used for assembly lives in one place and can be tuned in future (e.g. schedule tolerances) without touching every call site.

The size density is carried on the birth-date coordinate (`node_density_coordinate = "birth_date"`) unless overridden: plant's gradients (`plant::stand_gradient()`) are defined on it alone.

Value

A plant `Control` object.

Author(s)

Daniel Falster

`plant_default_assembly_pars` *Default plant parameters for assembly*

Description

Default `plant` parameters tuned for community-assembly runs.

Usage

```
plant_default_assembly_pars(
  hmat = 10,
  max_patch_lifetime = 60,
  fixed_RA = FALSE,
  model = c("FF16", "TF24")
)
```

Arguments

`hmat`	Height at maturation.
`max_patch_lifetime`	Maximum patch lifetime.
`fixed_RA`	If `TRUE`, use a fixed reproductive allocation.
`model`	plant physiological model: `"FF16"` or `"TF24"`.

Details

For FF16 the leaf-area allometry is set to the assembly values (`a_l1 = 2.17`, `a_l2 = 0.5`); TF24 keeps its own. Pass the same `model` to `harness_plant`.

Value

A `plant` `Parameters` object.

`plot.canonical_equation` *Plot a canonical-equation run*

Description

Plot trait trajectories from the canonical equation.

Usage

```
## S3 method for class 'canonical_equation'
plot(
  x,
  type = c("trajectory", "landscapes"),
  times = NULL,
  n_evals = 101L,
  ...
)

## S3 method for class 'canonical_equation'
autoplot(object, ...)
```

Arguments

`x`	A `canonical_equation`.
`type`	`"trajectory"` or `"landscapes"`.
`times`	For `"landscapes"`: evolutionary times to show.
`n_evals`	For `"landscapes"`: mutant values per landscape.
`...`	Ignored.
`object`	A `canonical_equation`, for `autoplot`.

Details

`type = "trajectory"`: each lineage is a line of trait value against evolutionary time; a branching event is where one line becomes two, an immigrant a line that starts in mid-air. With several traits, one panel per trait. `type = "landscapes"`: the invasion-fitness landscape of the community at chosen `times` (by default the record just before, at and after each branching), residents marked, one panel per time — the fitness minimum forming and the daughters moving apart. Single-trait communities only.

Value

A `ggplot` object.

Author(s)

Daniel Falster

plot.pip *Plot a pairwise or mutual invasibility surface*

Description

Plot a pairwise or mutual invasibility surface.

Usage

```
## S3 method for class 'pip'
plot(
  x,
  type = c("pip", "mip"),
  smooth = TRUE,
  fill = c("sign", "fitness"),
  contours = TRUE,
  n_display = 300L,
  ...
)

## S3 method for class 'pip'
autoplot(object, ...)
```

Arguments

x	A pip from community_pip.
type	"pip" or "mip".
smooth	Shade from the interpolated contour branches (default) or from the raw cells.
fill	For the raw "pip" view: "sign" (default) or "fitness".
contours	Draw the zero contours ("pip" only): as lines when smooth, as the located points otherwise.
n_display	Number of columns the smooth polygons are sampled at.
...	Ignored.
object	A pip, for autoplot.

Details

type = "pip" shades the (resident, mutant) region where the mutant can invade; type = "mip" shades the resident pairs that invade each other (see pip_mutual). The diagonal, where the mutant is the resident, is drawn through.

By default (smooth = TRUE) the exactly located zero crossings are linked across residents into contour branches, each interpolated along the trait axis, and the shading is drawn as polygons bounded by those curves — between consecutive contours in a column the sign of fitness is constant, so this adds no model evaluations and the fill meets the contour lines exactly rather than approximating

them with cells. `smooth = FALSE` shows the data as computed: one cell per (resident, mutant), following the possibly refined resident grid, with `fill = "fitness"` available to show the value on a diverging scale instead of its sign.

Value

A `ggplot` object, returned rather than drawn, so it can be extended with further layers, labels, themes or coordinates before printing; `autoplot()` is the same.

Author(s)

Daniel Falster

`plot.tep` *Plot a trait-evolution plot*

Description

Plot a trait-evolution plot.

Usage

```
## S3 method for class 'tep'
plot(x, arrow = 0.08, n_arrows = 15L, n_display = 300L, ...)

## S3 method for class 'tep'
autoplot(object, ...)
```

Arguments

`x`	A `tep` from `community_tep`.
`arrow`	Length of the longest arrow as a fraction of the axis range.
`n_arrows`	Most arrows along each axis.
`n_display`	Number of columns the shaded region is sampled at.
`...`	Ignored.
`object`	A `tep`, for `autoplot`.

Details

The region where two strategies can coexist (mutual invasibility, from the surface the plot was built on) is shaded; inside it, a lattice of at most `n_arrows` coexisting pairs per axis (one per occupied cell, the one nearest its centre) carries an arrow in the direction of the selection gradient on the two residents, scaled so the longest spans `arrow` of the axis range on the plotting scale. A dimorphic singular coalition sits where the arrows vanish. The picture is symmetric about the diagonal $x_1 = x_2$, where the two residents are one strategy: the coexistence region touches that line only at branching points, so arrows leading away from it there are evolutionary branching in progress.

Value

A ggplot object, returned rather than drawn, to extend before printing; autoplot() is the same.

Author(s)

Daniel Falster

---

plot_community *Plot the residents of an assembled community*

Description

Plot the residents of an assembled community

Usage

```
plot_community(
  tidy,
  step = NULL,
  trait_scale = attr(tidy, "trait_scale"),
  xlim = NULL,
  ylim = NULL
)
```

Arguments

tidy	Output of tidy_assembly.
step	The assembly step to plot. Defaults to the last.
trait_scale	"log" or "linear" trait axes. Defaults to the scale of the assembled community, which tidy_assembly records.
xlim, ylim	Optional length-2 axis ranges, replacing those taken from the whole assembly. Applied with coord_cartesian(), so points outside them are clipped rather than dropped.

Details

Plots the resident strategies at one step of an assembly, from the output of tidy_assembly. With one trait, each resident's birth rate is plotted against its trait value; with two, the residents are placed in the trait plane and coloured by birth rate.

Axis limits are taken over every step in tidy, so plots of successive steps share their axes and can be set side by side or animated: lapply(unique(tidy$step), function(s) plot_community(tidy, step = s)).

Value

A ggplot object.

tidy_assembly *Tidies output from assembler_run into tidy dataframe.*

Description

Tidies output from assembler_run into tidy dataframe.

Usage

```
tidy_assembly(obj)
```

Arguments

obj	output of an assembly run

Value

A tibble of the assembly, one row per resident per step. The community's trait scale is kept in `attr(, "trait_scale")` for `plot_community`.

trait_matrix *Create trait matrix*

Description

Helper function to create trait matrices suitable for `generate_strategy`.

Usage

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

Arguments

x	Values
trait_name	Name of a single trait

Author(s)

Rich FitzJohn

`util_nlsolve` *Solve a nonlinear system*

Description

Solve a nonlinear system.

Usage

```
util_nlsolve(
  x,
  fn,
  tol = 1e-06,
  maxit = 100,
  solver = "nleqslv",
  require_converged = TRUE,
  jac = NULL,
  J0 = NULL,
  refresh = 10L,
  max_step = Inf,
  xtol = tol * 0.001
)
```

Arguments

`x`	Starting point.
`fn`	Function to solve: returns a residual vector of the same length.
`tol`	Tolerance on the residual: a solve has converged when every component of the residual is within `tol` of zero (for `dfsane`, when its root-mean-square residual is).
`maxit`	Maximum number of iterations. The number of function evaluations will likely exceed this.
`solver`	`"nleqslv"`, `"dfsane"` or `"newton"`.
`require_converged`	If `TRUE` (the default) a solver that fails to converge is an error. Set `FALSE` to get the best point found back with `attr(., "converged") == FALSE` and decide what to do with it.
`jac`	Optional exact Jacobian function of `fn`, used by `newton` and `nleqslv`.
`J0`	Optional starting Jacobian approximation for `newton` and `nleqslv`, e.g. the one a previous nearby solve returned. It is only a hint. For `newton`, if the first step from it fails its line search, the Jacobian is recomputed by finite differences (or `jac`) and the solve continues, so a poor `J0` costs one wasted step rather than the solve. `nleqslv` never recomputes a Jacobian once it has one, so if the solve from `J0` does not converge it is run again from `x` with the Jacobian computed, at the cost of the first attempt.

refresh	For newton: recompute the Jacobian by finite differences every this many steps (Broyden updates in between).
max_step	For newton: the largest change in any coordinate a single step may make, before the line search. Far from the root a Newton step can be huge (in log density, a factor of e^{1000}); capping it keeps the iterates where the residual is well behaved.
xtol	A search whose steps fall below this (relative to the point for nleqslv, absolute for newton) stops, so a problem whose residual cannot reach tol does not run to maxit; it has converged only if the residual is within tol. Far smaller than tol by default, so that a steep residual is not stopped short of it.

Details

One wrapper over three root finders, so the equilibrium code chooses by name: nleqslv (Broyden or Newton with its own line search), dfsane (derivative-free spectral method) and newton, the package's own Newton–Broyden iteration. newton evaluates a finite-difference Jacobian once (or takes an exact one from jac, or a starting approximation from J0), then updates it by Broyden's rank-one formula after every step and refreshes it by finite differences only when a line search fails or every refresh steps. Each Newton step is damped by backtracking on the residual norm. Its final Jacobian is returned in attr(., "jacobian"), which lets the next solve of a nearby problem start from it and skip the finite-difference pass.

Value

The solution with attributes y (residual), iter, feval, code, message, converged, solver, and for newton also jacobian.

Index

as.data.frame.boundary_diagram *(boundary_diagram)*, 10
as.data.frame.regnans_branch *(community_continue_boundary)*, 23
assembler_control, 3, *8*, *31*
assembler_run, 4, *31*
assembler_set_traits, 5
assembler_start, 6
autoplot.canonical_equation *(plot.canonical_equation)*, 66
autoplot.pip *(plot.pip)*, 67
autoplot.tep *(plot.tep)*, 68

boundary_bisect, 7, *9*, *24*, *41*
boundary_control, *7*, 8, *12*, *13*, *24*, *31*
boundary_design, 10, *12*
boundary_diagram, 10
boundary_scout, *9*, *10*, 11, *13*, *31*
boundary_sweep, 12
boundary_switch, 14, *25*
bounds, *10*, 15
bounds_infinite *(bounds)*, 15

canonical_community, 15, *20*
canonical_control, 16, *19*
check_bounds *(bounds)*, 15
check_point *(bounds)*, 15
community_add, 18
community_canonical_equation, *4*, 19, *20*
community_classify_singularity, *4*, 20, *26*, *30*, *34*, *44*
community_continue_boundary, *9*, *14*, 23
community_continue_singularity, 26, *32*
community_demography, 28, *37*, *41*
community_fitness_gradient, 29, *48*, *54*
community_fitness_hessian, *22*, 29, *48*, *54*
community_fitness_landscape, 30, *36*, *49*
community_label, *4*, *7*, *9*, *12*, *25*, 31
community_parameter_map, *7*, *12*, *13*, *23*, *26*, *32*, *33*, *34*, *39*, *40*
community_parameter_sensitivity, *26*, *32*, 33, *39*
community_parameter_values, 34, *39*
community_pip, 35, *45*, *63*, *67*
community_plot_fitness_landscape, 36
community_selection_gradient, *29*, 37
community_selection_gradient_jacobian, *16*, *22*, *27*, *34*, 37, *39*, *43*, *48*
community_selection_gradient_parameter_jacobian, *32*, *34*, 38, *48*
community_solve_boundary, *24*, 40
community_solve_singularity, *4*, *17*, *20*, *26*, *33*, 42
community_start, *18*, 44, *48*, *49*, *54*
community_tep, *36*, 45, *68*
community_viable_fitness, 46

demographic_step_control, *28*, *44*, 47, *48*, *49*, *54*
derivative_control, *19*, *20*, *22*, *37–39*, *43*, *45*, 48, *54*

fitness_landscape_control, *30*, *44*, 49

generate_strategy, *70*

harness_check_derivatives, 50
harness_dd99, 51, *52*
harness_dd99_nd, 52
harness_explicit, *32*, 53
harness_fd, 54
harness_gk98, 55, *56*
harness_gm99, 56
harness_jj12, 57
harness_plant, *45*, 58, *65*
harness_provides, 59

length.assembler, 59
length.community, 60

max_fitness, *46*, 60
max_growth_rate, 61
mutational_vcv_proportion, 62

OdeSolver, *16*

pip_control, *35*, 62
pip_mutual, *36*, 63, *67*
plant_community_check_for_inviable_strategies, 64
plant_default_assembly_control, *58*, 64
plant_default_assembly_pars, *58*, 65
plot.boundary_diagram (boundary_diagram), 10
plot.canonical_equation, 66
plot.pip, *36*, 67
plot.regnans_branch (community_continue_boundary), 23
plot.tep, *46*, 68
plot_community, 69, *70*
print.boundary_labels (boundary_scout), 11
print.boundary_points (boundary_bisect), 7
print.boundary_sweep (boundary_sweep), 12
print.regnans_branch (community_continue_boundary), 23
print.singularity_classification (community_classify_singularity), 20
print.singularity_path (community_continue_singularity), 26

tidy_assembly, *69*, 70
trait_matrix, *5*, *18*, *61*, 70

util_nlsolve, *40, 41*, *43*, 71