Package 'chouca'

May 8, 2026

Title A Stochastic Cellular Automaton Engine

Version 0.1.99

Description An engine for stochastic cellular automata. It provides a high-level interface to declare a model, which can then be simulated by various backends (Genin et al. (2023) <doi:10.1101/2023.11.08.566206>).

License GPL (>= 3)

Encoding UTF-8

RoxygenNote 7.2.3

Depends R (>= 4.0.0)

LinkingTo Rcpp, RcppArmadillo

Imports Rcpp, plyr, digest, stats, graphics, grDevices

Suggests covr, testthat (>= 3.0.0), deSolve, igraph, knitr, rmarkdown, ggplot2, spatialwarnings

Config/testthat/edition 3

VignetteBuilder knitr

URL https://github.com/alexgenin/chouca

BugReports https://github.com/alexgenin/chouca/issues

NeedsCompilation yes

Author Alexandre Genin [aut, cre] (ORCID: <https://orcid.org/0000-0002-3333-1338>), Guillaume Dupont [aut], Daniel Valencia [aut], Mauro Zucconi [aut], M. Isidora Ávila-Thieme [aut], Sergio A. Navarrete [aut], Evie A. Wieters [aut]

Maintainer Alexandre Genin <a.a.h.genin@uu.nl>

Repository CRAN

Date/Publication 2024-03-07 15:00:02 UTC

Contents

`as.camodel_initmat` *Convert a matrix to a CA model landscape*

Description

Convert a matrix to a CA model landscape for later use with run_camodel or run_meanfield.

Usage

```
as.camodel_initmat(m, levels = NULL)
```

Arguments

`m`	The input matrix (numeric, character or factor)
`levels`	The levels to use in the resulting landscape. If `NULL`, the unique values of the input matrix are used as levels. Set this manually if you want the resulting landscape to have extra levels that are not present in the original matrix.

Value

This function returns a matrix containing values as factors, with levels corresponding to the `levels` argument. This matrix has the `class` `camodel_initmat` so that it can be displayed with the `image` generic function and works well with CA-related functions (such as `run_camodel`).

See Also

generate_initmat, run_camodel, run_meanfield

Examples

```
# Simple conversion of a matrix with regular patterns
x <- seq(0, 2 * pi, l = 256)
z <- outer(x, x, function(x, y) as.numeric(sin(10*x) + cos(10*y) > 0.8))
mat <- as.camodel_initmat(z)
summary(mat)
image(mat)

# This is a character matrix. We need to convert it to use it as input to
# run_camodel()
size <- 64
m <- matrix(ifelse(runif(size^2) < .5, "TREE", "EMPTY"), nrow = size, ncol = size)
m <- as.camodel_initmat(m)
summary(m) # this is a landscape object
image(m)

# Start a simulation using this matrix
mod <- ca_library("forestgap")
out <- run_camodel(mod, m, seq(0, 256))
plot(out)


# Run a glider in the game of life
mod <- ca_library("gameoflife")
init <- matrix(c(0, 0, 1, 0, 0, 0, 0,
                 0, 0, 0, 1, 0, 0, 0,
                 0, 1, 1, 1, 0, 0, 0,
                 0, 0, 0, 0, 0, 0, 0,
                 0, 0, 0, 0, 0, 0, 0,
                 0, 0, 0, 0, 0, 0, 0),
               nrow = 6, ncol = 7, byrow = TRUE)
init[] <- ifelse(init == 1, "LIVE", "DEAD")
# image() does not work on init here without conversion by as.camodel_initmat
init <- as.camodel_initmat(init)
image(init)

# Run the model and display simulation output as it is running
ctrl <- list(custom_output_fun = landscape_plotter(mod, fps_cap = 5),
             custom_output_every = 1)
out <- run_camodel(mod, init, times = seq(0, 32), control = ctrl)
```

---

camodel *Definition of a stochastic cellular automaton*

Description

High-level definition of a stochastic cellular automaton

Usage

```
camodel(
  ...,
  neighbors,
  wrap,
  parms = list(),
  all_states = NULL,
  check_model = "quick",
  verbose = FALSE,
  epsilon = sqrt(.Machine[["double.eps"]]),
  fixed_neighborhood = FALSE
)

transition(from, to, prob)
```

Arguments

`...`	A number of transition descriptions, as built by the `transition` function (see Details and Examples)
`neighbors`	The number of neighbors to use in the cellular automaton (4 for 4-way or von-Neumann neigborhood, or 8 for an 8-way or Moore neighborhood)
`wrap`	If `TRUE`, then the 2D grid on which the model is run wraps around at the edges (the top/leftmost cells will be considered neighbors of the bottom/rightmost cells)
`parms`	A named list of parameters, which should be all numeric, single values
`all_states`	The complete set of states of the model (a character vector). If unspecified, it will be guessed from the transition rules, but it is a good idea to pass it here to make sure the model definition is correct.
`check_model`	A check of the model definition is done to make sure there are no issues with it (e.g. probabilities outside the [0,1] interval, or an unsupported model definition). A quick check that should catch most problems is performed if check_model is "quick", an extensive check that tests all possible neighborhood configurations is done with "full", and no check is performed with "none".
`verbose`	Whether information should be printed when parsing the model definition.
`epsilon`	A small value under which the internal model coefficients values are considered to be equal to zero. The default value should work well here, except if you run models that have extremely small transition probabilities (<1e-8).
`fixed_neighborhood`	When not using wrapping around the edges (`wrap = FALSE`), the number of neighbors per cell is variable, which can slow down the simulation. Set this option to `TRUE` to consider that the number of neighbors is always four or eight, regardless of the position of the cell in the landscape, at the cost of approximate dynamics at the edges of the landscape.
`from`	The state from which the transition is defined
`to`	The state to which the transition is defined

`prob` a one-sided formula describing the probability of transition between the two states (see Details section for more information).

Details

This help page describes in detail technical points related to the definition of models in `chouca`. If this is your first time working with `chouca`, you may like the longer introduction in the vignette, accessible using `vignette("chouca-package")`.

`camodel` allows defining a stochastic cellular automaton model by its set of transition rules. These are defined by a set of calls to the `transition()` function. Each of these calls defines the two states of the transition, and the probability as a one-sided formula involving constants and the special vectors p and q.

`transition()` calls takes three arguments: the state from which the transition is defined, the state to which the transition goes, and a transition probability, defined as a one-sided formula. This formula can include numerical constants, parameters defined in the named list `parms`, and any combination of `p['a']` and `q['b']`, which respectively represent the proportion of cells in a landscape in state 'a', and the proportion of neighbors of a given cell in state 'b' ('a', and 'b' being, of course, any of the possible states defined in the model). Such formula could typically look like `~ 0.2 + 0.3 * p["a"] + q["b"]`. See below for examples of model definitions.

It is important to remember when using this function that `chouca` only supports models where the probabilities depend on constant parameters, the global proportion of each state in the landscape, and the local proportion of cells around a given cell. In other words, all transition probabilities should have the following functional form:

$$a_0 + \sum_{k=1}^{S} g_k(q_k) + s(q,q) + s(p,q) + s(q,q)$$

where a_0 is a constant, g_k are univariate functions of q_k, the proportions of neighbors of a cell in state k, and q is the vector containing all the q_k for k between 1 and S, the total number of states in the model. Similarly, p is the length-S vector containing the proportion of cells in each state in the whole grid. s above is the sum, defined for two vectors $x = (x_1, ..., x_S)$ and $y = (y_1, ..., y_S)$ as

$$a_1 x_1^{\alpha_1} y_1^{\beta_1} + a_2 x_1^{\alpha_2} y_2^{\beta_2} + a_3 x_1^{\alpha_3} y_3^{\beta_3} + a_4 x_2^{\alpha_3} y_1^{\beta_3} + a_4 x_2^{\alpha_3} y_2^{\beta_3} + \cdots + a_K x_S^{\alpha_K} y_S^{\beta_K}$$

where the a_k, α_k and β_k are constants for all k, and K is the total number of terms (equal to S^2). Note that α_K and β_K are capped to 5. This can be overriden using `options(chouca.degmax = n)`, but we do not recommend changing it as higher values typically make the package slow and/or leads to numerical instabilities. The functions g_k above can be any univariate functions of q_k, so `chouca` effectively supports any type of transition rule involving the neighborhood of a cell, including some 'threshold' rules that involve a single state (and only one). For example, a rule such as "more than 5 neighbors in a given state make a cell switch from state A to B" is OK, but combining states may not be supported, such as "more than 5 neighbors in state A *and* 2 in state B means a cell switches from A to B". When in doubt, just write your model, and `chouca` will tell you if it cannot run it accurately by running model checks.

Model checks are controlled by the argument `check_model`. When set to "quick" or "full", a check is performed to make sure the functional form above is able to accurately represent probabilities of transitions in the model, with "full" enabling more extensive testing, and "none" removing it

entirely. Coefficients in the formula above are rounded down to zero when below `epsilon`. This may be an issue if your transition probabilities are close to zero: consider reducing `epsilon` to a smaller value in this case, or adjusting your model parameters.

When space does not wrap around (`wrap = FALSE`), cells in the corners or in the edges will have a lower number of neighbors. The proportions of cells in a given state k, q_k, will thus be computed with a reduced number of cells. For example, a cell in a corner will have only 2 neighbors when using a 4x4 neighborhood, so q_k is computed using only two cells, and can be only equal to 0, 0.5 or 1.

To run a model once it is defined, the function `run_camodel` can be used, or `run_meanfield` for a mean-field approximation. An initial landscape for a simulation can be created using `generate_initmat`.

You can update a model definition with new parameters (all of them or a subset) using the `update` method. The model graph with the different states and transitions can be displayed using the `plot` method (this requires the package igraph).

Value

This function returns a `list` object with class `ca_model`, with the following named components. Please note that most are for internal use and may change with package updates.

`transitions` the list of transitions of the model, as returned by `transition`

`nstates` the number of states of the model

`parms` the parameter values used for the model

`beta_0`,`beta_q`, `beta_pp`, `beta_pq`, `beta_qq` internal tables used to represent probabilities of transitions when running simulations, these tables are for internal use and probably not interesting for end users, but more information is provided in the package source code

`wrap` Whether the model uses a toric space that wraps around the edge

`neighbors` The type of neighborhood (4 or 8)

`epsilon` The `epsilon` values used in the model definition, below which transition probabilities are assumed to be zero

`xpoints` (for internal use only) The number of values used to represent the proportion of neighbors of a cell in each state

`max_error`, `max_rel_error` vector of numeric values containing the maximum error and maximum relative error on each transition probability

`fixed_neighborhood` flag equal to `TRUE` when cells have a fixed number of neighbors

Functions

- `transition()`:

See Also

run_camodel, run_meanfield, generate_initmat, run_meanfield, update.ca_model, ca_library

Examples

```
# Redefine Kubo's 1996 forest gap model
kubo <- camodel(
  transition(from = "TREE",
             to   = "EMPTY",
             prob = ~ d + delta * q["EMPTY"] ),
  transition(from = "EMPTY",
             to   = "TREE",
             prob = ~ alpha),
  parms = list(d = 0.125,
               delta = 0.5,
               alpha = 0.2),
  all_states = c("EMPTY", "TREE"),
  neighbors = 4,
  wrap = TRUE
)

# Display it as a graph
plot(kubo)

# A fun plant model
mod <- camodel(
  transition("plant", "empty", ~ death * ( 1 - (2*q["plant"]-1)^2) ),
  transition("empty", "plant", ~ q["plant"]^2 ),
  all_states = c("empty", "plant"),
  wrap = TRUE,
  neighbors = 4,
  parms = list(death = 0.2496)
)


# Conway's Game of Life
mod <- camodel(
  transition("LIVE", "DEAD", ~ q["LIVE"] < (2/8) | q["LIVE"] > (3/8)),
  transition("DEAD", "LIVE", ~ q["LIVE"] == (3/8)),
  wrap = TRUE,
  neighbors = 8,
  all_states = c("DEAD", "LIVE")
)

# A spiral-generating rock-paper-scissor model
mod <- camodel(
  transition(from = "r", to = "p", ~ q["p"] > 0.25 ),
  transition(from = "p", to = "c", ~ q["c"] > 0.25 ),
  transition(from = "c", to = "r", ~ q["r"] > 0.25 ),
  parms = list(prob = 1),
  wrap = TRUE,
  neighbors = 8
)

# Display the model as a graph
plot(mod)
```

```
# Running the above model (see also the help files for the relevant functions)
init <- generate_initmat(mod, c(r = 1/3, p = 1/3, c = 1/3), nrow = 128)
out <- run_camodel(mod, init, times = seq(0, 128))
plot(out)

# Update a model definition using update()
mod <- camodel(
  transition("plant", "empty", ~ m),
  transition("empty", "plant", ~ r * q["plant"] * ( 1 - q["plant"] ) ),
  all_states = c("empty", "plant"),
  wrap = TRUE,
  neighbors = 4,
  parms = list(m = 0.35, r = 0.4)
)

mod_updated <- update(mod, parms = list(m = 0.05, r = 1))
init <- generate_initmat(mod_updated, c(plant = 0.8, empty = 0.2), nrow = 128)
out <- run_camodel(mod_updated, init, times = seq(0, 128))
plot(out)
image(out)

# You can also specify only part of the parameters, the others will be
# kept to their original values
mod_updated <- update(mod, parms = list(m = 0.035))
```

`ca_library` *Library of stochastic cellular automata*

Description

Get one of the SCA models included in chouca

Usage

```
ca_library(model, parms = NULL, neighbors = NULL, wrap = TRUE)
```

Arguments

model	The model to return, as a string. See Details for the full list of models included with chouca.
parms	The model parameters to use, as a named list. If unset, the model default parameters will be used.
neighbors	The number of neighbors to use in the cellular automaton (4 for 4-way or von-Neumann neghborhood, or 8 for a Moore neighborhood). If unset, the model default neighborhood will be used.
wrap	Whether the 2D grid should wrap around at the edges. Default it to wrap around the edges of the landscape.

Details

This function gives access to different stochastic cellular automata models. You can provide a named list of parameters, or set the number of neighbor or wrapping options, but default are chosen if left unspecified. This function provides the following models (the string represents the name of the model, as passed using the 'model' argument):

1. `"forestgap"` Kubo's forest gap model (1996), which describes how gaps form in a forest and expand through disturbances.
2. `"musselbed"` A model of mussel beds, in which disturbance by waves occurs at the edge of mussel patches (Guichard et al. 2003)
3. `"aridvege"` A model of arid vegetation, in which facilitation between neighboring plants occur, along with grazing. The original model is to be found in Kéfi et al. (2007), with extensions in Schneider et al. (2016)
4. `"aridvege-danet"` An extension of the previous model to two species with assymetric facilitation (Danet et al. 2021)
5. `"coralreef"` A model of coral reef with local feedbacks of corals and macroalgae (Génin, in prep)
6. `"gameoflife"` The famous Game of Life by Conway, a deterministic model.
7. `"rockpaperscissor"` A rock-paper-scissor model with three states, in which a cell changes state depending on its neighbors according the game rules (e.g. "rock beats scissors"). This deterministic model produces nice spirals.

Value

This function returns a `list` object with class `ca_model`, with the following named components. Please note that most are for internal use and may change with package updates.

`transitions` the list of transitions of the model, as returned by `transition`

`nstates` the number of states of the model

`parms` the parameter values used for the model

`beta_0`,`beta_q`, `beta_pp`, `beta_pq`, `beta_qq` internal tables used to represent probabilities of transitions when running simulations, these tables are for internal use and probably not interesting for end users, but more information is provided in the package source code

`wrap` Whether the model uses a toric space that wraps around the edge

`neighbors` The type of neighborhood (4 or 8)

`epsilon` The `epsilon` values used in the model definition, below which transition probabilities are assumed to be zero

`xpoints` (for internal use only) The number of values used to represent the proportion of neighbors of a cell in each state

`max_error`, `max_rel_error` vector of numeric values containing the maximum error and maximum relative error on each transition probability

`fixed_neighborhood` flag equal to `TRUE` when cells have a fixed number of neighbors

References

Danet, Alain, Florian Dirk Schneider, Fabien Anthelme, and Sonia Kéfi. 2021. "Indirect Facilitation Drives Species Composition and Stability in Drylands." Theoretical Ecology 14 (2): 189–203. doi:10.1007/s12080020004890.

Genin, A., S. A. Navarrete, A. Garcia-Mayor, and E. A. Wieters. in press (2023). Emergent spatial patterns can indicate upcoming regime shifts in a realistic model of coral community. The American Naturalist.

Guichard, F., Halpin, P.M., Allison, G.W., Lubchenco, J. & Menge, B.A. (2003). Mussel disturbance dynamics: signatures of oceanographic forcing from local interactions. The American Naturalist, 161, 889–904. doi:10.1086/375300

Kefi, Sonia, Max Rietkerk, Concepción L. Alados, Yolanda Pueyo, Vasilios P. Papanastasis, Ahmed ElAich, and Peter C. de Ruiter. 2007. "Spatial Vegetation Patterns and Imminent Desertification in Mediterranean Arid Ecosystems." Nature 449 (7159): 213–17. doi:10.1038/nature06111.

Kubo, Takuya, Yoh Iwasa, and Naoki Furumoto. 1996. "Forest Spatial Dynamics with Gap Expansion: Total Gap Area and Gap Size Distribution." Journal of Theoretical Biology 180 (3): 229–46.

Schneider, Florian D., and Sonia Kefi. 2016. "Spatially Heterogeneous Pressure Raises Risk of Catastrophic Shifts." Theoretical Ecology 9 (2): 207-17. doi:10.1007/s1208001502891.

Examples

```
# Import a model, create an initial landscape and run it for ten iterations
forestgap_model <- ca_library("forestgap")
im <- generate_initmat(forestgap_model, c(0.5, 0.5), nrow = 64, ncol = 100)
run_camodel(forestgap_model, im, times = seq(0,100))
```

---

chouca *chouca: a package for stochastic cellular automata*

Description

chouca is a package that can be used to implement and run stochastic cellular automata (SCA). SCA are model that describe dynamics over a grid of cells. Each cell can be in one of a set of states, and at each time step, can switch to another one with a given transition probabilities. These transitions probabilities typically depend on the neighborhood of a focal cell.

The chouca package is able to simulate SCA models efficiently, including by emitting and compiling the required C++ code at runtime. It does not support all cellular automata, but typically only those where transition probabilities only depend on (1) constant parameters, (2) the proportion of cells in a given state in the landscape, and (3) the proportion of neighboring cells in a given state. More information on the types of supported model can be found in our publication, but in any case chouca is able to identify and warn you if your model is unsupported.

The package workflow has typically four steps (1) define the model, (2) create an initial landscape (grid of cells), (3) run the model, and (4) display/extract the results. We describe each step below, and give examples at the end of this document. A more complete description of the workflow and the package is available in the vignette `vignette("chouca-package", package = "chouca")`.

(1) Model definition

Models can be defined using the camodel function. A typical call would be something looking like this, for a simple model of plants growing over space:

```
mod <- camodel(
 transition(from = "bare", to = "plant", ~ r1 * p["plant"] + r2 * q["plant"]),
  transition(from = "plant", to = "bare", ~ m),
  parms = list(r1 = 0.1, r2 = 0.1, m = 0.05),
  wrap = TRUE,
  neighbors = 8
)
```

This model defines two transitions (between the "bare" and the "plant" state and vice versa). These transitions are defined using the transition() function, which arguments define where the transition goes from, to, and an expression on how to compute its probability. In this model, the probability that the first transition occurs depends on the proportion of "plant" cells in the landscape, `p["plant"]`, and the proportion of neighbors in the "plant" state, `q["plant"]`. The model has three parameters, `r1`, `r2` and `m` whose value is passed through the named list `parms`. The `neighbors` argument defines the type of neighborhood (here 8-way, or Moore neighborhood), and we specify that the model should run over a toric space that wraps around the edges of the grid (`wrap = TRUE`).

More information about the creation of models is available at camodel.

(2) Creation of the initial landscape

An initial grid of cell (or landscape) can be created using generate_initmat, which will fill a grid with the states provided by a model object created using camodel above, and respecting the specified proportions:

```
init_grid <- generate_initmat(mod, c(bare = 0.4, plant = 0.6), nrow = 128, ncol = 90)
```

Here, we create a 128x90 rectangular grid that contains 40 distributed randomly through space.

If you already have a specific grid of cells (as an R matrix) you would like to use, we recommend to process it first through as.camodel_initmat so it will play nicely with the rest of the package functions.

(3) Running the model

You can feed the model and the initial landscape to run_camodel, which will run the model and output the results at the time step specified by the `times` argument:

```
out <- run_camodel(mod, init_grid, times = seq(0, 1024))
```

run_camodel has many options to control how the simulation is run and the model outputs are saved, and are documented in the function help page.

(4) Extracting results

The results of the simulation run can be extracted from the resulting object using the '[[' operators, typically using `out[["output"]][["covers"]]` or `out[["output"]][["snapshots"]]` to extract the global proportions of cells in each state and the landscapes, respectively. These data can then be used for further analyses. Standard methods can be used to display the results (e.g. `plot()` or `image()`).

Each step of this workflow can be adjusted as necessary, either to display the results as the simulation is run (see e.g. landscape_plotter), or use different simulation backends (see options in run_camodel). You can also run the equivalent mean-field model of the SCA using run_meanfield, which assumes that local and global proportion of states are equal.

chouca comes with a set of pre-implemented models, which you can access using ca_library.

If you use and like chouca, we would appreciate you to cite our corresponding publication:

Genin A, Dupont G, Valencia D, Zucconi M, Avila-Thieme M, Navarrete S, Wieters E (2023). "Easy, fast and reproducible Stochastic Cellular Automata with 'chouca'." doi:10.1101/2023.11.08.566206

See Also

camodel, generate_initmat, run_camodel, run_meanfield, ca_library

Examples

```
# The above example in full
mod <- camodel(
  transition(from = "bare", to = "plant", ~ r1 * p["plant"] + r2 * q["plant"]),
  transition(from = "plant", to = "bare", ~ m),
  parms = list(r1 = 0.1, r2 = 0.1, m = 0.05),
  wrap = TRUE,
  neighbors = 8
)

# Display the structure of the model
plot(mod)

init_grid <- generate_initmat(mod, c(bare = 0.4, plant = 0.6),
                              nrow = 128, ncol = 90)
out <- run_camodel(mod, init_grid, times = seq(0, 128))

# Display results
plot(out)
image(out)

# Run the meanfield model (uses deSolve internally)
if ( requireNamespace("deSolve", quietly = TRUE) ) {
  out <- run_meanfield(mod, init_grid, times = seq(0, 1024))
  plot(out)
}
```

---

generate_initmat *Generate an initial matrix for a* chouca *model*

Description

Helper function to create a spatially-random initial landscape (matrix) with specified covers for a cellular automaton

Usage

```
generate_initmat(mod, pvec, nrow, ncol = nrow)
```

Arguments

mod	A stochastic cellular automaton model created by camodel
pvec	A numeric vector of covers for each state in the initial configuration, possibly with named elements.
nrow	The number of rows of the output matrix
ncol	The number of columns of the output matrix

Details

This function is a helper to build a starting configuration (matrix) for a stochastic cellular automaton based on the definition of the model and the specified starting covers (in pvec). It will produce a landscape with expected global cover of each state equal to the covers in pvec, and a completely random spatial structure.

The length of the pvec vector must match the number of possible cell states in the model. If present, the names of pvec must match the states defined in the model. In this case, they will be used to determine which state gets which starting cover instead of the order of the values.

The pvec will be normalized to sum to one, and the function will produce a warning if this produces a meaningful change in covers.

If you already have a matrix you want to use as a starting configuration, we recommend you to use as.camodel_initmat to convert it to an object that run_camodel can use.

Value

This function returns a matrix containing values as factors, with levels corresponding to the model states (defined in the mod argument) and dimensions set by nrow and ncol. This matrix has the class camodel_initmat so that it can be displayed with the image generic function.

See Also

as.camodel_initmat

Examples

```
# Run the Game of Life starting from a random grid
game_of_life <- ca_library("gameoflife")
grid <- generate_initmat(game_of_life, c(LIVE = .1, DEAD = .9), nrow = 64)
out <- run_camodel(game_of_life, grid, times = seq(0, 128))
image(out) # final configuration

# Logistic growth of plants
mod <- camodel(
  transition(from = "empty", to = "plant", ~ r * p["plant"]),
  transition(from = "plant", to = "empty", ~ m),
  parms = list(r = 1, m = .03),
```

```
  wrap = TRUE,
  neighbors = 8
)
grid <- generate_initmat(mod, c(empty = .99, plant = .01), nrow = 128)
image(grid) # initial state
out <- run_camodel(mod, grid, times = seq(0, 30))
image(out) # final state
plot(out) #
```

`landscape_plotter` *Plot simulation landscapes*

Description

This function creates an internal function to plot the model landscape during the simulation of a stochastic cellular automaton.

Usage

```
landscape_plotter(
  mod,
  col = NULL,
  fps_cap = 24,
  burn_in = 0,
  transpose = TRUE,
  new_window = TRUE,
  ...
)
```

Arguments

`mod`	The model being used (created by `link{camodel}`)
`col`	a set of colors (character vector) of length equal to the number of states in the model.
`fps_cap`	The maximum number of frame displayed per seconds. Simulation will be slowed down if necessary so that plot updates will not be more frequent than this value
`burn_in`	Do not display anything before this time step has been reached
`transpose`	Set to `TRUE` to transpose the landscape matrix before displaying it `image`
`new_window`	Controls whether the plots are displayed in a new window, or in the default device (typically the plot panel in Rstudio)
`...`	other arguments are passed to `image`

Details

This function creates another function that is suitable for use with `run_camodel`. It allows plotting the landscape as it is being simulated, using the base function `image`. You can set colors using the argument `col`, or tranpose the landscape before plotting using `transpose`. The resulting function must by passed to `run_camodel` as the control argument `custom_output_fun`. Typically, this function is not used by itself, but is being used when specifying simulation options before calling `run_camodel`, see examples below.

`image` is used internally, and tends to be quite slow at displaying results, but if it is still too fast for your taste, you can cap the refresh rate at a value given by the argument `fps_cap`.

It is important to note that this function will probably massively slow down a simulation, so this is most useful for exploratory analyses.

Value

This function returns another function, which will be called internally when simulating the model using `run_camodel`, and has probably not much use outside of this context. The return function will display the simulation and returns NULL.

See Also

trace_plotter, run_camodel

Examples

```
# Display the psychedelic spirals of the rock-paper-scissor model as the model is
# being run
mod <- ca_library("rock-paper-scissor")
colors <- c("#fb8500", "#023047", "#8ecae6")
ctrl <- list(custom_output_every = 1,
             custom_output_fun = landscape_plotter(mod, col = colors))
init <- generate_initmat(mod, rep(1, 3)/3, nrow = 100, ncol = 178)
run_camodel(mod, init, times = seq(0, 128), control = ctrl)

# Arid vegetation model
mod <- ca_library("aridvege")
init <- generate_initmat(mod, rep(1, 3)/3, nrow = 100, ncol = 178)
colors <- c("gray80", "white", "darkgreen")
ctrl <- list(custom_output_every = 1,
             custom_output_fun = landscape_plotter(mod, col = colors, xaxt = "n",
                                                   yaxt = "n"))
run_camodel(mod, init, times = seq(0, 128), control = ctrl)

# Game of life, set plot margins to zero so that the landscape takes all
# of the plot window
mod <- ca_library("gameoflife")
init <- generate_initmat(mod, c(0.5, 0.5), nrow = 100, ncol = 178)
colors <- c("white", "black")
ctrl <- list(custom_output_every = 1,
```

```
            custom_output_fun = landscape_plotter(mod, col = colors,
                                                  mar = c(0, 0, 0, 0)))
run_camodel(mod, init, times = seq(0, 128), control = ctrl)
```

`run_camodel` *Run a cellular automata*

Description

Run a pre-defined stochastic cellular automaton

Usage

```
run_camodel(mod, initmat, times, control = list())
```

Arguments

`mod`	A stochastic cellular automaton model defined using `camodel`
`initmat`	An initial matrix to use for the simulation, possibly created using `generate_initmat`
`times`	A numeric vector describing the time sequence for which output is wanted. Time will always start at zero but output will only be saved at the time steps specified in this vector.
`control`	a named list with settings to alter the way the simulation is run (see full list of settings in 'Details' section)

Details

`run_camodel()` is the workhorse function to run cellular automata. It runs the simulation and outputs the results at the time steps specified by the `times` argument, starting from the initial landscape `initmat` (a matrix typically created by `generate_initmat`).

Note that the simulation is run for all time steps, but output is only provided for the time steps specified in `times`.

The `control` list must have named elements, and allows altering the way the simulation is run, including the live display of covers or landscapes (see `trace_plotter` or `landscape_plotter`).

Possible options are the following:

1. `save_covers_every` By default, global covers are saved for each time step specified in `times`. Setting this argument to values higher than one will skip some time steps (thinning). For example, setting it to 2 will make `run_camodel` save covers only every two values specified in `times`. Set to 0 to skip saving covers. This value must be an integer.

2. `save_snapshots_every` In the same way as covers, landscape snapshots can be saved every set number of values in `times`. By default, only the initial and final landscape are saved. Set to one to save the landscape for each value specified in `times`. Higher values will skip elements in `times` by the set number. Set to zero to turn off the saving of snapshots. This value must be an integer.

3. `console_output_every` Set the number of iterations between which progress report is printed on the console. Set to zero to turn off progress report. The default option is to print progress five times during the simulation.

4. `custom_output_fun` A custom function can be passed using this argument to compute something on the landscape as the simulation is being run. This function can return anything, but needs to take two arguments, the first one being the current time in the simulation (single numeric value), and the other one the current landscape (a matrix). This can be used to plot the simulation results as it is being run, see `landscape_plotter` and `trace_plotter` for such use case.

5. `custom_output_every` If `custom_output_fun` is specified, then it will be called for every time step specified in the `times` vector. Increase this value to skip some time points, in a similar way to covers and snapshots above.

6. `substeps` Stochastic CA can run into issues where the probabilities of transitions are above one. A possible solution to this is to run the model in 'substeps', i.e. an iteration is divided in several substeps, and the substeps are run subsequently with probabilities divided by this amount. For example, a model run with 4 substeps means that each iteration will be divided in 4 'sub-iterations', and probabilities of transitions are divided by 4 for each of those sub-iterations.

7. `engine` The engine used to run the simulations. Accepted values are 'cpp' to use the C++ engine, or 'compiled', to emit and compile the model code on the fly. Default is to use the C++ engine. Note that the 'compiled' engine uses its own random number generator, and for this reason may produce simulations that are different from the C++ engine (it does respect the R seed however). You will need a compiler to use the 'compiled' engine, which may require you to install Rtools on Windows systems.

8. `precompute_probas` (Compiled engine only) Set to `TRUE` to precompute probabilities of transitions for all possible combinations of neighborhood. When working with a model with a low number of states (typically 3 or 4), this can increase simulation speed dramatically. By default, a heuristic is used to decide whether to enable precomputation or not.

9. `verbose_compilation` (Compiled engine only) Set to `TRUE` to print Rcpp messages when compiling the model. Default is `FALSE`.

10. `force_compilation` (Compiled engine only) `chouca` has a cache system to avoid recompiling similar models. Set this argument to `TRUE` to force compilation every time the model is run.

11. `write_source` (Compiled engine only) A file name to which the C++ code used to run the model will be written (mostly for debugging purposes).

12. `cores` (Compiled engine only) The number of threads to use to run the model. This provides a moderate speedup in most cases, and is sometimes counter-productive on small landscapes. If you plan on running multiple simulations, you are probably better off parallelizing at a higher level. See also the 'Performance' section in the vignette, accessible using the command `vignette("chouca-package")`.

Value

A `ca_model_result` objects, which is a list with the following components:

1. `model` The original model used for the model run (see return value of `camodel` for more details about these objects).
2. `initmat` The initial landscape (matrix) used for the model run, such as what is returned by `generate_initmat`.
3. `times` The time points vector at which output is saved
4. `control` The control list used for the model run, containing the options used for the run
5. `output` A named list containing the simulation outputs. The 'covers' component contains a matrix with the first column containing the time step, and the other columns the proportions of cells in a given state. The 'snapshots' component contains the landscapes recorded as matrices(`camodel_initmat` objects), with a 't' attribute indicating the corresponding time step of the model run. The 'custom' component contains the results from calling a custom function provided as 'custom_output_fun' in the control list (see examples below).

See Also

camodel, generate_initmat, trace_plotter, landscape_plotter, run_meanfield

Examples

```
# Run a model with default parameters
mod <- ca_library("musselbed")
im  <- generate_initmat(mod, c(0.4, 0.6, 0), nrow = 100, ncol = 50)
out <- run_camodel(mod, im, times = seq(0, 100))
plot(out)

# Disable console output
opts <- list(console_output_every = 0)
out <- run_camodel(mod, im, times = seq(0, 100), control = opts)

#



# Run the same model with the 'compiled' engine, and save snapshots. This
# requires a compiler on your computer (typically available by installing
# 'Rtools' on Windows)
ctrl <- list(engine = "compiled", save_covers_every = 1, save_snapshots_every = 100)
run <- run_camodel(mod, im, times = seq(0, 100), control = ctrl)
plot(run)

#
oldpar <- par(mfrow = c(1, 2))
image(run, snapshot_time = 0)
image(run, snapshot_time = 100)
par(oldpar)

# Disable console output
```

```
ctrl <- list(console_output_every = 0)
run <- run_camodel(mod, im, times = seq(0, 100), control = ctrl)
plot(run)

# Very verbose console output (display compilation information, etc.)
ctrl <- list(console_output_every = 1,
             verbose_compilation = TRUE,
             engine = "compiled",
             force_compilation = TRUE)
run <- run_camodel(mod, im, times = seq(0, 100), control = ctrl)

# Turn on or off the memoisation of transition probabilities (mind the speed
# difference)
ctrl <- list(engine = "compiled", precompute_probas = FALSE)
run <- run_camodel(mod, im, times = seq(0, 256), control = ctrl)
ctrl2 <- list(engine = "compiled", precompute_probas = TRUE)
run2 <- run_camodel(mod, im, times = seq(0, 256), control = ctrl2)

# Use a custom function to compute statistics while the simulation is running
fun <- function(t, mat) {
  # Disturbed cell to mussel cell ratio
  ratio <- mean(mat == "DISTURB") / mean(mat == "MUSSEL")
  data.frame(t = t, ratio = ratio)
}
ctrl <- list(custom_output_fun = fun, custom_output_every = 1)

run <- run_camodel(mod, im, times = seq(0, 256), control = ctrl)
stats <- do.call(rbind, run[["output"]][["custom"]])
plot(stats[ ,1], stats[ ,2], ylab = "DISTURB/MUSSEL ratio", xlab = "time", type = "l")
```

`run_meanfield` *Mean field model*

Description

Run the mean field model corresponding to a cellular automaton

Usage

```
run_meanfield(mod, init, times, ...)
```

Arguments

`mod`	The cellular automaton model (produced by `camodel`
`init`	The initial landscape, or a vector of covers summing to one, whose length is equal to the number of states in the model.
`times`	The points in time for which output is wanted
`...`	other arguments are passed to `ode`

Details

The mean field approximation to a cellular automaton simply describes the dynamics of the global covers using differential equations, assumming that both global and local covers are equal (in chouca model specifications, this assumes p = q).

For example, if we consider a model with two states 'a' and 'b' and transitions from and to each other, then the following system of equation is used to describe the variations of the proportions of cells in each state:

$$\frac{da}{dt} = p_b P(b \to a) - p_a P(a \to b)$$

$$\frac{db}{dt} = p_a P(a \to b) - p_b P(b \to a)$$

Running mean-field approximations is useful to understand general dynamics in the absence of neighborhood interactions between cells, or simply to obtain an fast but approximate simulation of the model.

Note that this function uses directly the expressions of the probabilities, so any cellular automaton is supported, regardless of whether or not it can be simulated exactly by run_camodel.

Value

This function returns the results of the ode function, which is a matrix with class 'deSolve'

Examples

```
if ( requireNamespace("deSolve") ) {
  # Get the mean-field approximation to the arid vegetation model
  arid <- ca_library("aridvege")
  mod <- ca_library("aridvege")
  init <- generate_initmat(mod, rep(1, 3)/3, nrow = 100, ncol = 100)
  times <- seq(0, 128)
  out <- run_meanfield(mod, init, times)
  # This uses the default plot method in deSolve
  plot(out)

  # A different model and way to specifiy initial conditions.
  coralmod <- ca_library("coralreef")
  init <- c(ALGAE = 0.2, CORAL = 0.5, BARE = 0.3)
  times <- 10^seq(0, 4, length.out = 64)
  out <- run_meanfield(coralmod, init, times, method = "lsoda")
  plot(out, ylim = c(0, 1))
}
```

`trace_plotter` *Plot simulation covers*

Description

This function creates an internal function to plot the model landscape during the simulation of a stochastic cellular automaton.

Usage

```
trace_plotter(
  mod,
  initmat,
  fun = NULL,
  col = NULL,
  max_samples = 256,
  fps_cap = 24,
  burn_in = 0,
  new_window = TRUE,
  ...
)
```

Arguments

`mod`	The model being used (created by `link{camodel}`)
`initmat`	The initial landscape given to `run_camodel`
`fun`	The function used to summarise the landscape into summary metrics. By default, gloal covers of each state are computed. It must return a vector of numeric values.
`col`	a set of colors (character vector) of length equal to the number of values returned by fun.
`max_samples`	The maximum number of samples to display in the plot
`fps_cap`	The maximum number of frame displayed per seconds. Simulation will be slowed down if necessary so that plot updates will not be more frequent than this value
`burn_in`	Do not display anything before this time step has been reached
`new_window`	Controls whether the plots are displayed in a new window, or in the default device (typically the plot panel in Rstudio)
`...`	other arguments passed to `matplot`, which is used to display the trends.

Details

This function creates another function that is suitable for use with `run_camodel`. It can plot any quantity computed on the landscape as it is being simulated, using the base function `matplot`. The resulting function must by passed to `run_camodel` as the control argument `custom_output_fun`.

Typically, this function is not used by itself, but is being used when specifying simulation options before calling run_camodel, see examples below.

By default, the global covers of each state in the landscape will be displayed, but you can pass any function as argument fun to compute something else, as long as fun returns a numeric vector of length at least 1.

matplot is used internally, and tends to be quite slow at displaying results, but if it is still too fast for your taste, you can cap the refresh rate at a value given by the argument fps_cap.

It is important to note that this function will probably massively slow down a simulation, so it is mostly here for exploratory analyses, or just to have a good look of what is happening in a model.

Value

This function returns another function, which will be called internally when simulating the model using run_camodel, and has probably not much use outside of this context. The return function will display the simulation and returns NULL.

See Also

landscape_plotter, run_camodel

Examples

```
# Display covers of the rock/paper/scissor model as it is running
mod <- ca_library("rock-paper-scissor")
init <- generate_initmat(mod, rep(1, 3)/3, nrow = 100, ncol = 178)
ctrl <- list(custom_output_every = 1,
             custom_output_fun = trace_plotter(mod, init))
run_camodel(mod, init, times = seq(0, 256), control = ctrl)

# Adjust colors of the previous example and increase speed
colors <- c("#fb8500", "#023047", "#8ecae6")
ctrl <- list(custom_output_every = 1,
             custom_output_fun = trace_plotter(mod, init, fps_cap = 60, col = colors))
run_camodel(mod, init, times = seq(0, 600), control = ctrl)

# Display the vegetated to degraded cover ratio for the arid
# vegetation model.
mod <- ca_library("aridvege")
init <- generate_initmat(mod, rep(1, 3)/3, nrow = 100, ncol = 178)
ratio <- function(mat) {
  mean(mat == "VEGE") / mean(mat == "DEGR")
}
ctrl <- list(custom_output_every = 1,
             custom_output_fun = trace_plotter(mod, init,
                                               fun = ratio))
run_camodel(mod, init, times = seq(0, 512), control = ctrl)

# Display ratios of cell pairs in the rock-paper-scissor model
mod <- ca_library("rock-paper-scissor")
```

```
init <- generate_initmat(mod, rep(1, 3)/3, nrow = 100, ncol = 178)
ratio <- function(mat) {
  c(mean(mat == "r") / mean(mat == "p"),
    mean(mat == "p") / mean(mat == "c"),
    mean(mat == "c") / mean(mat == "r"))
}
ctrl <- list(custom_output_every = 1,
             custom_output_fun = trace_plotter(mod, init,
                                               fun = ratio))
run_camodel(mod, init, times = seq(0, 512), control = ctrl)
```

`update.ca_model` *Update a cellular automaton*

Description

Update the definition of a stochastic cellular automaton (SCA), using new parameters, type of wrapping, or any other parameters entering in the definition of the model.

Usage

```
## S3 method for class 'ca_model'
update(
  object,
  parms = NULL,
  neighbors = NULL,
  wrap = NULL,
  fixed_neighborhood = NULL,
  check_model = "quick",
  verbose = FALSE,
  ...
)
```

Arguments

`object`	The SCA object (returned by `camodel`)
`parms`	a named list of parameters, which should be all numeric, single values. If this list contains only a subset of model parameters, the old parameter values will be reused for those not provided.
`neighbors`	The number of neighbors to use in the cellular automaton ('4' for 4-way or von-Neumann neghborhood, or '8' for an 8-way or Moore neighborhood)
`wrap`	If `TRUE`, then the 2D grid on which the model is run wraps around at the edges (the top/leftmost cells will be considered neighbors of the bottom/rightmost cells)

`fixed_neighborhood`
: When not using wrapping around the edges (`wrap = TRUE`, the number of neighbors per cell is variable, which can slow down the simulation. Set this option to `TRUE` to consider that the number of neighbors is always four or eight, regardless of the position of the cell in the landscape, at the cost of approximate dynamics on the edge of the landscape.

`check_model`
: A check of the model definition is done to make sure there are no issues with it (e.g. probabilities outside the [1,0] interval, or an unsupported model definition). A quick check that should catch most problems is performed if check_model is "quick", an extensive check that tests all neighborhood configurations is done with "full", and no check is performed with "none".

`verbose`
: whether information should be printed when parsing the model definition.

`...`
: extra arguments are ignored

Details

This function updates some aspects of a pre-defined stochastic celullar automaton, such as parameter values, the type of neighborhood, whether to wrap around the edge of space, etc. It is handy when running multiple simulations, and only a few aspects of the model needs to be changed, such as parameter values. Note that this function cannot add or remove states to a model.

Note that the `parms` list may only specify a subset of the model parameters to update. In this case, old parameter values not specified in the call to `update` will be re-used.

Value

This function returns a list with class `ca_model` with the changes applied to the original model (see `camodel` for details about this type of object).

See Also

camodel, run_camodel...

Examples

```
# Update the parameters of a model
mussels <- ca_library("musselbed")
mussels[["parms"]] # old parameters
mussels_new <- update(mussels, parms = list(d = 0.2, delta = 0.1, r = 0.8))
mussels_new[["parms"]] # updated parameters

# Update the type of neighborhood, wrapping around the edges, and
# the parameters
mussels_new <- update(mussels,
                      parms = list(d = 0.2, delta = 0.1, r = 0.8),
                      wrap = TRUE,
                      neighbors = 8)
mussels_new
```

```
# Run the model for different values of d, the death rate of mussels
ds <- seq(0, 0.25, length.out = 12)
initmat <- generate_initmat(mussels, c(0.5, 0.5, 0), nrow = 64, ncol = 64)
results <- lapply(ds, function(this_dvalue) {
  musselmod <- update(mussels, parms = list(d = this_dvalue))
  run <- run_camodel(musselmod, initmat, times = seq(0, 128))
  data.frame(d = this_dvalue,
             as.data.frame(tail(run[["output"]][["covers"]], 1)))
})

results <- do.call(rbind, results)
plot(results[ ,"d"], results[ ,"MUSSEL"], type = "b",
     xlab = "d", ylab = "Mussel cover")
```

Index