

# caspr: Cellular Automata for Spatial Pressure in R

*Florian D. Schneider*

*2015-10-30*

## Contents

<b>General information</b>	<b>2</b>
Contributors . . . . .	2
Version and package home . . . . .	2
Install package . . . . .	2
License . . . . .	2
<b>Package description</b>	<b>3</b>
Landscape objects . . . . .	3
Model objects . . . . .	5
Running models: function <code>ca()</code> . . . . .	7
function <code>ca_array()</code> . . . . .	9
function <code>ca_animate()</code> . . . . .	10
<b>Model descriptions</b>	<b>11</b>
Mussel bed model . . . . .	12
Forest Gap model . . . . .	13
Predator-prey model . . . . .	14
Grazing model . . . . .	16
Conway's Game of Life . . . . .	18

## General information

The caspr package is running spatial disturbance models in a cellular automata framework. This is part of a collaborative project between the group of [Sonia Kéfi](#) (Institut de Sciences d'Evolution, CNRS, IRD, Université Montpellier, France) and [Vishweshha Guttal](#) (Center for Ecological Sciences, Indian Institute of Science, Bangalore, India).

## Contributors

Alain Danet, Alex Genin, Vishweshha Guttal, Sonia Kefi, Sabiha Majumder, Sumithra Sankaran, [Florian Schneider \(Maintainer\)](#)

## Version and package home

The package is developed and distributed on Github. Source code is found at <https://github.com/fdschneider/caspr>. Current release is v0.2.0. Please report bugs on the [package issue tracker](#) and contribute fixes or new model definitions via pull requests!

## Install package

The package can be installed directly from GitHub using the `devtools` package.

```
install.packages("devtools")  
devtools::install_github("fdschneider/caspr")
```

Then, to load the package in your current session, type

```
library(caspr)
```

## License

The MIT License (MIT)

Copyright (c) 2015 the authors

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

## Package description

The R-package is build around objects of a particular class `ca_model` (S3 objects in R) that contain all the model specific information, like the original publication, the update functions, and the cell states. These objects can be handled by a function called `ca()` which runs the cellular automata over time. Other functions allow the plotting of single snapshots or timeseries, the generation of initial lattices/grids/landscapes, the calculation of spatial and temporal indicators as provided by the package ‘[spatialwarnings](#)’.

In the following the objects and functions are described in a (somewhat) logical sequence of usage.

## Landscape objects

Cellular automata are lattice- or grid-based systems, where neighboring cells on the lattice are affecting each others transition rules. Those lattices are termed ‘landscapes’ within the framework of ‘caspr’ and are stored in objects of class “`landscape`”.

## Generate landscape objects

Landscape objects are created using the function `init_landscape()` , e.g.

```
l <- init_landscape(c("1", "0"), c(0.2,0.8), width = 100)
```

Landscapes are squared by default, but a height argument can be specified optionally. However, doing so **might disqualify the output to be spatially analysed!!!**

A landscape object is a list `l` of class "landscape" that contains

- `l$dim` : a named vector of the default form `c(width = 50, height = 50)`
- `l$cells` : a factorial vector of length `prod(l$dims)` that contains the state that each cell of the landscape matrix is in (row-wise from top to bottom).

## Transfer to Matrix

The vectorial storage of the grid is allowing for a more efficient memory use and evaluation than with matrices, using map vectors that are provided as global variables by executing `mapping(l$dims[1], l$dims[2])`.

For compatibility the landscape object can be translated into a matrix using `as.matrix(l)`. Any matrix `L` of *factorial* content can be reverted into a landscape object using `as.landscape(L, states)`. Ensure that the factors occurring in the matrix match the factor levels provided in the argument `states`, which is used to set the order of the states in the resulting landscape object (order is important in the further process!).

## Binary matrix for spatial analysis

The function `as.binary_matrix()` provides compatibility with the package 'spatialwarnings' for spatial analysis of spatial patterns. By default the function assumes the primary cell state to be transferred into binary value `TRUE`, while all other states are set to `FALSE`. The parameter `is` specifies the state or a vector of states that should be set to `TRUE` in the resulting binary matrix.

```
library(spatialwarnings)

L <- as.binary_matrix(1)
L_gap <- as.binary_matrix(1, is = c("0", "-"))

spatialwarnings(L)
```

## Plot and summarize output

Specific methods for functions `plot`, `summary` and `print` do exist, which means the following calls will produce an optimized output:

```
l
summary(l)
plot(l)
```

Also, a method for ggplot2 function `fortify()` exists which allows you to plot landscape objects via

```
require(ggplot2)

ggplot(fortify(l)) +
  geom_raster(aes(x,y,fill=state)) +
  scale_fill_manual(values=grazing$cols)
```

## Model objects

A model object is a list `model` of class `"ca_model"` that contains

- `model$name`: the name of the model
- `model$ref`: the original reference
- `model$states`: the potential cell states. Order does matter: The first entry is considered a 'primary' or 'focus' cell state in downstream functions (e.g. for plotting of timeseries).
- `model$cols`: colors for cell states

- `model$interact` : an interaction matrix, defaults to 4-cell neighborhood. This affects behaviour of the `count()` function, if this is required in the update function.
- `model$params`: a set of default parameters that also serves as a template to check for completeness of custom parameters.
- `model$update`: the update function, which takes a landscape object `x_old` and returns a landscape object `x_new`, representing the updating of one single timestep (e.g. one year, one season).

Other functions or static parameters can be provided as further objects to the list. But those listed above are mandatory. In the current version of *caspr*, the objects of class `ca_model` are not created by a function, but provided manually as custom R objects. If you add a model then start development from `model_template.R`! Before running `ca(1, yourmodel)` you should test the update function for valid output and optimize for evaluation speed!

A method `print.ca_model` exists which allows to review a model's specifications simply by calling the model object in R, e.g. `grazing` returns.

```
##
## Spatial Grazing Model
## -----
##
## Possible cell states: +, 0, -
##
## Required parameters and default values:
##   del = 0.9
##   b = 0.5
##   c_ = 0.2
##   m0 = 0.05
##   g = 0.2
##   r = 0.01
##   f = 0.9
##   d = 0.1
##   p = 1
##
## Original reference: Schneider and Kefi 2015, in review
```

It is envisioned to provide a function that helps generating the backbone of a new model object.

## Running models: function `ca()`

The function `ca(x, model, parms)` runs a cellular automata simulation starting from the landscape object provided in `x` and using the update function provided by `model` with the parameter set `parms` (a list of parameters). Before running the model, the function validates the parameter set provided against the template stored in `model$parms`.

Further parameters are available to adjust the timespan run and the snapshots of the landscape saved.

```
l <- init_landscape(c("+", "0", "-"), c(0.2,0.7,0.1))
p <- list(d = 0.4, r = 0.2, delta = 0.4)
r <- ca(l, musselbed, p)
```

By default, simulations run for 200 timesteps and then terminate. This can be modified by setting the `t_max` parameter to a lower or higher value.

```
r <- ca(l, musselbed, p, t_max = 500)
```

After simulation, the function returns an object of class `ca_results`. The result object `r` would have the structure:

- `r$model`: the `ca_model` object used to run the simulation, including the provided parameter set.
- `r$time`: a vector of the distinct timesteps of the simulation
- `r$issteady`: Binary vector reporting for each timestep if the criterion for steady state was fulfilled. The criterion can be customized by adjusting parameter `steady` (see below).
- `r$cover`: a data frame reporting the global cover with one column for each state of the model and one row for each timestep. Thus, `r$cover[1]` returns the timeseries of the primary cell state.
- `r$local`: a data frame reporting the average local cover of the cell states, i.e. the average probability that a state  $i$  is found in the neighborhood given that the focal cell is in state  $i$ . Thus, `r$local[1]` returns the timeseries of the primary cell state.
- `r$snaps`: a vector of times at which snapshots were taken.
- `r$landscapes`: a list of landscape objects for the given snapshots.

Thus the result of a simulation contains two major types of output:

1. the full timeseries of cover of each state (`r$cover`), as well as a timeseries of the average local cover (`r$local`).
2. selected snapshots of the full landscape (`r$landscapes`). By default each timestep is stored (see below).

### define frequency of snapshots

The output object contains a list object `r$landscapes` which contains individual snapshots of the landscapes, by default taken each full timestep. The parameter `saveeach` reduces the frequency of snapshots to be saved in the output object. This is useful to reduce size of the output objects or to produce time-lapse videos with the `ca_animate()` function.

The following would only save the initial and final landscape.

```
r <- ca(1, musselbed, p, t_max = 500, saveeach = 500)
```

### define auto-stop function

Alternatively to running the simulations until `t_max` is reached, the function `ca()` can automatically terminate the simulation if steady state is reached. The switch `stopifsteady = TRUE` activates this option and applies the default criterion for steady state, which is comparing the mean cover of the primary cell state in two subsequent periods of time. The simulation terminates if the difference in means is below a threshold value.

**(This method is still problematic and needs to be refined!)**

### realised timesteps

The update functions of the models are defined to represent a full timestep. What that means depends on the model specifications. It can be one year, a day or any particular timespan that the model parameters refer to. However, the computational implementation of one update cycle is model specific.

For instance the predator-prey model applies an asynchronous updating, i.e. within each annual timestep, `width × height` random cells are updated,

allowing for repeated draws. Each cell has the fair chance to be updated, but no guarantee. Thus, no substeps in that sense are implemented, but they could be.

The grazing model has synchronous updating and each timestep is implemented as `subs` substeps. Within each substep, only a fraction of the annual transition probability is applied. The default value here is set in the model specifications, but it can be provided by the `ca()` function as an argument that is handed over to the update function call. For instance, you can run the grazing model with 12 instead of the default 10 substeps by calling:

```
ca(1 , model = grazing, parms = p, subs = 12)
```

In any case only the full annual timestep is returned to the `ca()` function.

## function `ca_array()`

The package includes a wrapper function `ca_array()` that runs repeated model simulations along a gradient of one parameter value or an array of parameter values. This function makes use of a parallel backend, e.g. provided by the `foreach` package. The initial landscape is drawn for each replicate using the numerical vector of initial cover provided in `init`.

Make sure to adapt the type of parallel backend to your computer infrastructure (see [package vignette of the foreach -package](#)).

```
# provides parallel backend
library(foreach)
library(doSNOW)

cl <- makeCluster(rep("localhost", 10), type = "SOCK")

p <- list(
  r = 0.4, # recolonisation of empty sites dependent on local density
  d = seq(0,1,0.1), # wave disturbance
  delta = 0.01, # intrinsic disturbance rate
  replicates = 1:5
)
r <- ca_array(musselbed, init = c(0.7,0.15,0.15), parms = p)
```

## `stopCluster(c1)`

The function returns a dataframe with global and local cover for each state for each parameter value or combination of parameter values given in `parms`.

Parameters specifying the behaviour of each model run can be provided as optional parameters and will be handed to function `ca()` (e.g. `t_max` or a custom stability criterion).

## Save output to files

Optionally, the individual model run outputs will be saved to ‘.Rd’ files if parameter `save = TRUE` is set. The `filename` parameter can be used to specify a relative path and filename root, which will be extended by an individual counter.

```
r <- ca_array(musselbed, init = c(0.7,0.15,0.15), parms = p,  
             save = TRUE, filename = "output/musselbed")
```

**Note:** If running in parallel on a cluster, output files will be saved in the workers home directory!

## Random number seeds

The function `ca_array()` sets the seed for random number generation for each iteration in the parameter array. The seeds of each initial run will be reported in the output table. To make things reproducible, it initiates the seeds of the array using a random number generator that is seeded to a value provided in the parameter `salt`. Thus, two runs of `ca_array()` that are supposed to differ from one another run must receive their own grain of salt.

```
r1 <- ca_array(musselbed, init = c(0.7,0.15,0.15), parms = p, salt = 378539)  
r2 <- ca_array(musselbed, init = c(0.7,0.15,0.15), parms = p, salt = 745283)
```

## function `ca_animate()`

To visualize simulation runs the function `ca_animate()` provides a wrapper that generates an animated gif (default) or video of the landscapes development over

time. The function requires the `'animation'` package, which uses `'ImageMagick'` to convert the multiples into an animated gif or `'ffmpeg'` to convert into a video. Make sure that `ImageMagick` and/or `ffmpeg` are installed and properly found by the animation package functions (see `?saveGIF`). The animated gifs are tiny and represent one landscape cell per pixel. Use an image viewer that scales images without aliasing to view the animated gif at larger size.

The type of video generated can be specified by the `type` argument which can be `mp4`, `wmv` or `avi`. So far, `mp4` produces the nicest output.

```
l <- init_landscape(musselbed$states, cover = c(0.3,0.2,0.5), width = 100)
r <- ca(1, musselbed, t_max = 40)

ca_animate(r, filename = "musselbed")
ca_animate(r, filename = "musselbed", type = "mp4")
```

Note that the parameter `saveeach` in `ca()` can be used to produce timelapse effects.

```
r <- ca(1, musselbed, t_max = 200, saveeach = 3)
ca_animate(r, filename = "musselbed", speed = 0.8)
```

Additionally, the speed of the output video or animation can be adjusted by setting the relative `speed` parameters (defaults to `speed = 1`).

By default, output is saved in current working directory. The parameter `directory` allows specifying any relative or absolute target path.

## Model descriptions

A set of spatially-explicit cellular automata is provided with `caspr`. They have in common that a spatially explicit stress is acting on the focal cell state, for instance by increasing mortality if predators or gaps in the canopy are present or by reducing mortality by associational growth.

We also added Conway's Game of Life as a test case for animation functions – and for fun.

## Mussel bed model

The model represents the spatial dynamics in mussel cover of rock substrate in intertidal systems. The stochastic wave disturbances will most likely remove mussels that are located next to a gap because of the loosened byssal threads in their proximity. This causes a dynamic gap growth.

The model describes the process by simplifying the system into three potential cell states: occupied by mussel (“+”), empty but undisturbed (“0”), and disturbed, bare rock with loose byssal threads (“-”).

Mussel growth on empty cells is defined by parameter  $r$  multiplied by the local density of mussels in the direct 4-cell neighborhood.

Any cell occupied by mussels has an intrinsic chance of  $\delta$  to be disturbed from intrinsic cause, e.g. natural death or predation. Additionally, wave disturbance will remove mussels and leave only bare rock, i.e. disturbed sites, with probability  $d$  if at least one disturbed cell is in the direct 4-cell neighborhood. This causes disturbances to cascade through colonies of mussels.

Disturbed sites will recover into empty sites with a constant rate of 1 per year, i.e. on average a disturbed site becomes recolonisable within one year after the disturbance happened.

### Example

```
# create initial landscape:
l <- init_landscape(c("+","0","-"), c(0.6,0.2,0.2), width = 50)
# set parameters:
p <- list(delta = 0.01, d = 0.9, r = 0.4)
# run simulation:
r <- ca(l, musselbed, p, t_max = 100)
```

### Original reference

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](https://doi.org/10.1086/375300)

## Forest Gap model

Dynamic model for forest pattern after recurring wind disturbance with two cell states: empty ("0") and vegetated ("+").

This model can use either an explicit height for trees, in which case states can be anywhere in a range [*S<sub>m</sub>in*...*S<sub>m</sub>ax*] (Solé et al., 1995), or use only two states, vegetated (non-gap) and empty (gap) (Kubo et al., 1996). Here we focus on the version that uses only two states: gap ("0") and non-gap ("+"). Without spatial spreading of disturbance (all cells are independent), a cell transitions from empty to vegetated with a birth probability  $b$  and from vegetated to empty with death probability  $d$ .

However, gap expansion occurs in nature as trees having empty (non-vegetated) surroundings are more likely to fall due to disturbance (e.g. wind blows). Let  $p(0)$  be the proportion of neighbouring sites that are gaps. We can implement this expansion effect by substituting the death rate

$$d = d_0 + \delta p_0 .$$

Since  $0 \leq p(0) \leq 1$ ,  $\delta$  represents the maximal added death rate due to gap expansion (i.e. the spatial component intensity).

The authors consider two cases: one in which the recovery of trees is proportional to the global density of vegetated sites, and one where the recovery is proportional to the local density of vegetation. We use only the first case as it the only one producing bistability.

The birth rate  $b$  is substituted with

$$b = \alpha \rho_+ ,$$

where  $\rho_+$  represents the global density of non-gap sites and  $\alpha$  is a positive constant. This can produce alternative stable states over a range of  $\delta$  values within 0.15–0.2 ( $\alpha$  is fixed to 0.20 and  $d$  to 0.01).

In their original simulations, the authors of the model (Kubo et al. 1996) use a  $100 \times 100$ -torus-type lattice (with random initial covers?).

## Example

```
# create initial landscape:
l <- init_landscape(c("+","0"), c(0.6,0.4), width = 100)
# set parameters:
p <- list(alpha = 0.2, delta = 0.2, d = 0.01)
# run simulation:
r <- ca(1, model = forestgap, parms = p, t_max = 50)
r
plot(r)
```

## Original reference

Kubo, T. et al. (1996) Forest spatial dynamics with gap expansion: total gap area and gap size distribution. *J. Theor. Biol.* 180, 229–246, doi: [10.1006/jtbi.1996.0099](https://doi.org/10.1006/jtbi.1996.0099)

## Predator-prey model

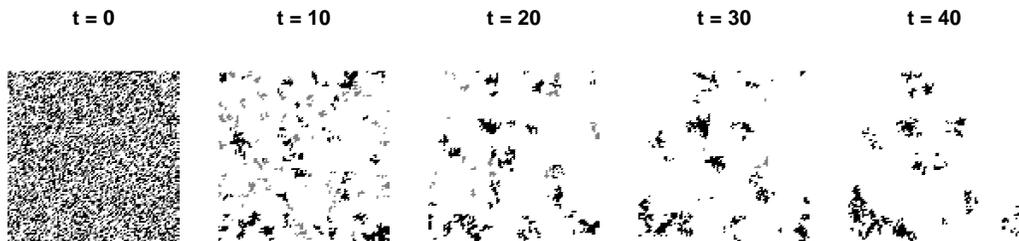
A spatially explicit predator prey model with three potential cell states: Occupied by prey (fish, "f"), occupied by predators (sharks, "s") and empty ("0").

The particular neighbourhood considered in the simulations consists of the four nearest sites. Prey growth occurs as a contact process: a prey chooses a neighbouring site at random and gives birth to another prey if this site is empty at a birth rate  $\beta_f$ . Predators hunt for prey by inspecting their neighbourhood for the presence of prey at rate  $\beta_s = 1$ . If prey are present in the nearest neighborhood, the predator selects one at random and eats it, moving to this neighbouring site. Only predators that find a prey can reproduce, and do so with a specified probability,  $\beta_s$ . The offspring is placed in the original site of the predator. Predators that do not find prey are susceptible to starvation and die with a probability  $\delta$ . Random movement occurs through mixing: neighbouring sites exchange state at a constant rate  $\nu$ .

## Example

```
# create initial landscape:
l <- init_landscape(c("f","s","0"), c(0.3,0.2,0.5), width = 100)
# set parameters:
p <- list(betaf = 0.01, betas = 0.1, delta =0.2)
# run simulation:
r <- ca(l, model = predprey, parms = p, t_max = 50)
r

##
## Model run of Predator-prey Gap Model over 50 timesteps.
##
## average cover:
##      f      s      0
## 0.052    0 0.948
##
## for more details call 'summary(x)' or 'plot(x)!'
## access simulation results:
## 'x$model$parms' : simulation parameters
## 'x$time' : a vector of timesteps
## 'x$cover' : a dataframe of the states' timeseries
## 'x$ini_landscape' : the initial landscape object
## 'x$snapshots' : an index table of saved snapshots
## 'x$landscapes[[i]]' : extract snapshot from list of snapshots
##
##
```



## Original reference

Pascual, M., Roy, M., Guichard, F., & Flierl, G. (2002). Cluster size distributions: signatures of self-organization in spatial ecologies. *Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences*, 357(1421), 657–666. doi: [10.1098/rstb.2001.0983](https://doi.org/10.1098/rstb.2001.0983)

## Grazing model

The model builds upon a published model by Kefi et al. (2007). Spatial models of vegetation cover so far have considered grazing mortality a rather constant pressure, affecting all plants equally, regardless of their position in space. In the known models it usually adds as a constant to the individual plant risk (Kefi et al. 2007). However, grazing has a strong spatial component: Many plants in rangelands invest in protective structures such as thorns or spines, or develop growth forms that reduce their vulnerability to grazing. Therefore, plants growing next to each other benefit from the protection of their neighbors.

Such **associational resistance** is widely acknowledged in vegetation ecology but hardly integrated in models as a cause for spatially heterogeneous grazing pressure. It also renders the plant mortality density dependent, which has important impacts on the bistability of the system.

The model investigates how the assumption of spatially heterogeneous pressure alters the bistability properties and the response of spatial indicators of catastrophic shifts.

The model knows three different cell states: occupied by vegetation "+", empty but fertile "0" and degraded "-". Transitions between cell states are only possible between vegetated and empty (by the processes of plant ‘death’ and ‘recolonization’) and between empty and degraded (by ‘degradation’ and ‘regeneration’).

To account for the spatially heterogeneous impacts of grazing due to associational resistance, we assumed that a plant’s vulnerability to grazers decreases with the proportion of occupied neighbors,  $q_{++}$ . The individual probability of dying is therefore defined as

$$w_{\{+,0\}} = m_0 + g_0 (1 - q_{++})$$

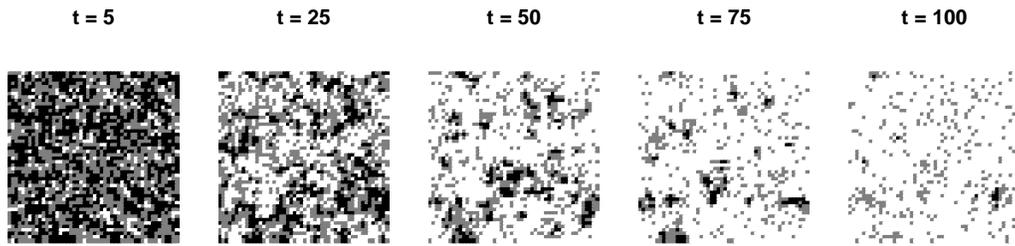
where the additional mortality due to grazing is maximized to  $g_0$  if a plant

has no vegetated neighbor (i.e.,  $q_{+|+} = 0$ ) and gradually reduces to 0 with an increasing fraction of occupied neighbors,  $q_{+|+}$ .

## Example

```
# create initial landscape:
l <- init_landscape(c("+","0","-"), c(0.6,0.2,0.2), width = 50)
# set parameters:
p <- list(del = 0.1, # global proportion of seed dispersal
          b = 0.4, # environmental quality
          c_ = 0.2, # global competition
          m0 = 0.05, # intrinsic plant mortality
          g = 0.25, # grazing rate
          r = 0.01, # intrinsic regeneration rate
          f = 0.9, # local facilitation
          d = 0.1, #
          p = 1 # intensity of associational protection
          )
r <- ca(1, model = grazing, parms = p, t_max = 100) # run simulation
r
```

```
##
## Model run of Spatial Grazing Model over 100 timesteps.
##
## average cover:
##      +      0      -
##    0.008  0.137  0.855
##
## for more details call 'summary(x)' or 'plot(x)!'
## access simulation results:
##   'x$model$parms' : simulation parameters
##   'x$time'       : a vector of timesteps
##   'x$cover'      : a dataframe of the states' timeseries
##   'x$ini_landscape' : the initial landscape object
##   'x$snapshots'  : an index table of saved snapshots
##   'x$landscapes[[i]]' : extract snapshot from list of snapshots
##
##
```



### Original reference

This model is unpublished yet, but inherits most assumptions from:

Kéfi, S., Rietkerk, M., van Baalen, M. & Loreau, M. (2007). Local facilitation, bistability and transitions in arid ecosystems. *Theoretical Population Biology*, 71, 367–379. doi: [10.1016/j.tpb.2006.09.003](https://doi.org/10.1016/j.tpb.2006.09.003)

### Conway's Game of Life

This is an implementation of the most popular cellular automata model 'Life'. We added it to caspr for testing purposes.

The possible cell states are alive, "1", and dead, "0". The transition rules are deterministic:

- Any live cell with fewer than two live neighbours dies, as if caused by under-population.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any live cell with more than three live neighbours dies, as if by over-crowding.
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

### Example

```
l <- init_landscape(c("1","0"), c(0.35,0.65), 150)
r <- ca(l, life, t_max = 100 )
ca_animate(r, filename = "life")
```

## Original reference

A detailed description can be found on [Wikipedia](#). The first publication is:

Gardner, Martin (1970). Mathematical Games – The fantastic combinations of John Conway’s new solitaire game “life”. *Scientific American* 223. pp. 120–123. ISBN 0-89454-001-7.