Methods in Ecology and Evolution

The *nlrx* R package: A next-generation framework for reproducible NetLogo model analyses

Jan Salecker[1*], Marco Sciaini[1], Katrin M. Meyer[1], and Kerstin Wiegand[1,2]

[1] *Department of Ecosystem Modelling, University of Göttingen, Büsgenweg 4, 37077, Göttingen, Germany*

[2] *Centre of Biodiversity and Sustainable Land Use (CBL), University of Goettingen, Germany*

*Corresponding author:

E-mail: jsaleck@gwdg.de

Phone: +49-(0)551-3933414

Suggested running headline:

(1) nlrx: next-gen NetLogo model analysis framework

This article has been accepted for publication and undergone full peer review but has not been through the copyediting, typesetting, pagination and proofreading process, which may lead to differences between this version and the Version of Record. Please cite this article as doi: 10.1111/2041-210X.13286

Abstract

1. Agent-based models find wide application in all fields of science where large-scale patterns emerge from properties of individuals. Due to increasing capacities of computing resources it was possible to improve the level of detail and structural realism of next-generation models in recent years. However, this is at the expense of increased model complexity, which requires more efficient tools for model exploration, analysis and documentation that enable reproducibility, repeatability and parallelisation. NetLogo is a widely used environment for agent-based model development, but it does not provide sufficient built-in tools for extensive model exploration, such as sensitivity analyses. One tool for controlling NetLogo externally is the R-package RNetLogo. However, this package is not suited for efficient, reproducible research as it has stability and resource allocation issues, is not straightforward to be setup and used on high performance computing clusters and does not provide utilities, such as storing and exchanging metadata, in an easy way.
2. We present the R-package nlrx, which overcomes stability and resource allocation issues by running NetLogo simulations via dynamically created XML experiment files. Class objects make setting up experiments more convenient and helper functions provide many parameter exploration approaches, such as Latin Hypercube designs, Sobol sensitivity analyses or optimization approaches. Output is automatically collected in user-friendly formats and can be post-processed with provided utility functions. nlrx enables reproducibility by storing all relevant information and simulation output of experiments in one R object which can conveniently be archived and shared.
3. We provide a detailed description of the nlrx package functions and the overall workflow. We also present a use case scenario using a NetLogo model, for which we performed a sensitivity analysis and a genetic algorithm optimization.
4. The nlrx package is the first framework for documentation and application of reproducible NetLogo simulation model analysis.

Keywords: agent-based modelling, individual-based modelling, reproducible workflow, R package, NetLogo

Introduction

Agent-based models are an increasingly popular method for understanding complex systems (DeAngelis & Grimm, 2014). They are developed and applied across many different research disciplines from natural sciences over archeology to socio-economic sciences (e.g., Dislich *et al.*, 2018; Vincenot, 2018). Agent-based models incorporate the heterogeneity of entities at the individual level in order to observe patterns emerging on broader scales (Grimm & Railsback, 2005). Due to the release from computing power constraints in recent years, next-generation agent-based models have evolved that are structurally realistic, powerful and detailed enough to address real-world problems (Cabral *et al.*, 2017; Grimm & Berger, 2016). However, next-generation agent-based models require reproducability, repeatability and parallelisation of model analyses. Access to scripts, runs and results of statistical analyses is a key criterion for reproducible research (Sandve *et al.*, 2013; Peng, 2011). Yet, incentives for sharing code of agent-based models and model analyses alongside publications are still lacking (Janssen, 2017).

A widely used programming language to develop agent-based models in ecological and socio-economic sciences is NetLogo (Vincenot, 2018; Abar *et al.*, 2017; Wilensky, 1999). NetLogo is a Java-based modelling environment that features a very comprehensible syntax, which allows for fast prototyping of agent-based models but also offers capabilities to formulate and implement complex agent-based models efficiently (Railsback *et al.*, 2017). With rising complexity of NetLogo models, increasing efforts need to be spent on model analyses and documentation of such analyses. Model analysis is a crucial part of the modelling cycle, not only for understanding model processes but also during model refinement and development (Grimm & Railsback, 2005). Often sensitivity analyses are the central part of such refinements, because they give access to detailed information on the relative importance of model parameters for model outputs (Saltelli *et al.*, 2008). Sensitivity analyses requires the simulation thousands of different parameter value combinations to gain a better understanding of the modelled systems. Unfortunately, the capabilities of the built-in tool of NetLogo to run such analyses, the NetLogo BehaviorSpace, are limited. If more than one parameter is changed within one experiment, BehaviorSpace automatically creates a full-factorial parameter matrix including all possible combinations of parameter values. This might be generally suitable for simple models, but with rising model complexity more efficient ways of scanning the parameter space and a better control of the design of parameter matrices are required. Additionally, post-processing of NetLogo model output is mostly done within statistical software, such as R (the most prominent programming language for ecologists; Lai *et al.*,

2019). Transfer of the output data from NetLogo into R may not be straightforward, especially when dealing with spatially-explicit model output. This is because spatial NetLogo output is mostly reported in nested lists which need to be parsed and split to convert the data into a usable format.

The R package *RNetLogo* currently is the only R package available to set up NetLogo model simulations directly from R (Thiele *et al.*, 2012, 2014). However, the package establishes an interactive connection between the R session and the Java virtual machine that runs the NetLogo model application, which has two main drawbacks: (1) *RNetLogo* establishes the interactive connection via the *rJava* package, which may be cumbersome to set up, especially on remote systems, and requires access to system-wide environmental variables. (2) Resource allocation of *rJava* and *RNetLogo* may be problematic when running large jobs with many simulations. Memory may not be cleared efficiently between runs and Java virtual machines may freeze due to memory constraints. Furthermore, *RNetLogo* does not provide a convenient workflow to set up experiments with minimal coding, utility functions to generate simulations, or features to enable reproducible research (see a full comparison in Tab. 1).

We have, therefore, developed the R package *nlrx* to provide a flexible framework for self-contained and reproducible analysis of NetLogo models from R, while also delivering performance gains (see Fig. 1 and Supplementary Material 1). The *nlrx* framework serves four main needs for next-generation NetLogo modellers: (1) it provides an interface between R and NetLogo. Simulations are completely defined and stored within R objects. This allows the application of a huge toolbox of statistical methods to create simulation experiments and corresponding parameter input matrices. The *nlrx* package utilizes the controlling API of NetLogo's BehaviorSpace to set up and run experiments. Thus, in contrast to *RNetLogo*, the connection between R and NetLogo is not interactive and does not rely on the *rJava* package. (2) it enables reproducible research (Sandve *et al.*, 2013) by storing a simulation experiment in one single R object (including parameter input matrix, applied simulation method and simulation results) and allows easy sharing of code and results among colleagues, with students and for publication. (3) *nlrx* enables utilization of high performance computing clusters (HPCs) in a very convenient way (see Supplementary Material 2). (4) *nlrx* provides post-processing analysis functions of NetLogo model output for several applications, such as sensitivity analyses and calculation of descriptive statistics. *nlrx* also enables gathering spatial output from models, such as coordinates and properties of individuals and patches and provides functions to convert the output into spatial R objects (raster and vector data).

Table 1: Main differences between NetLogo's BehaviorSpace, RNetLogo and nlrx.

Feature & Description	Behavior Space	*RNetLogo*	*nlrx*
High-performance computing (HPC) Straightforward setup of HPC machines to submit and run large NetLogo simulation jobs.	-	-	✓
Controllable Java environment Possibility to open a new Java virtual machine for each model simulation.	-	-	✓
Interactive Java environment Possibility to control model parameters during model run.	-	✓	-
Reproducible research tools Possibility to perform model analyses in a non-interactive, reproducible way. Means to control for random number generator seeds and exchange of model files.	-	-	✓
Spatial integration of agent metrics Coerce NetLogo agent metrics to geospatial data objects (vector and raster data).	-	-	✓
Tidy data format Data format in which each observation is a unique cell and each variable a unique column (Wickham, 2014).	-	-	✓
Utility functionality Functionality to perform pre- and post-processing of simulation data.	-	-	✓

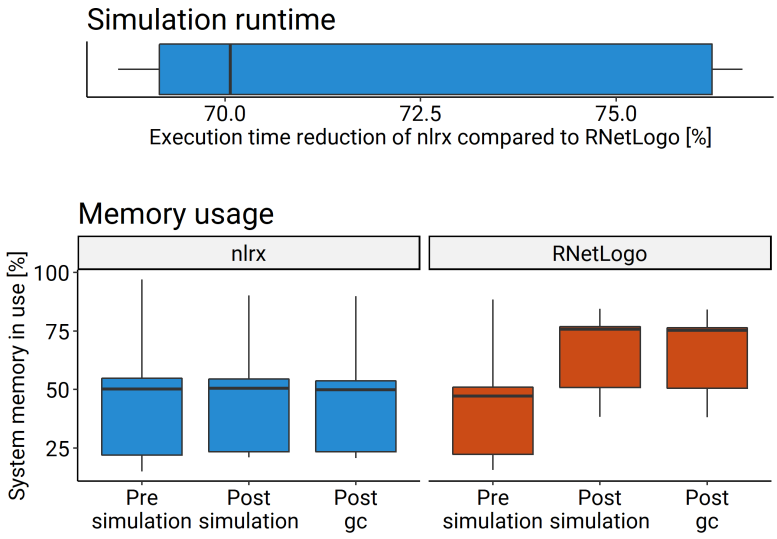


Figure 1: Comparison of execution time (top panel) and memory usage (bottom panel) of *nlrx* (blue) and *RNetLogo* (red). Benchmarks were calculated on different machines with various processor types, available memory and operating systems (details see Supplementary material 1, boxplots show variation between systems). Both packages were used to execute NetLogo model simulations with the Wolf Sheep Model from the NetLogo Models Library. With each package we simulated 8 replications of a Latin-Hypercube Sampling design with 100 samples and a run time of 500 ticks of each simulation. Available memory was measured before starting simulations (Pre simulation) and after simulations have finished (Post simulation). A third measurement of available memory was taken after manually executing the R garbage collection function *gc* (Post gc).

The R package *nlrx*

The *nlrx* package utilizes the NetLogo BehaviorSpace interface by calling NetLogo from the command line via a bash script (*.sh, linux) or a batch file (*.bat, windows). By default, NetLogo BehaviorSpace experiments are stored in XML format within the *.nlogo model file. However, when NetLogo is started in headless mode from command line, it is also possible to commit XML BehaviorSpace experiment files to setup and run experiments. The *nlrx* package creates such XML BehaviorSpace experiment files dynamically in order to run simulation experiments from R.

nlrx workflow

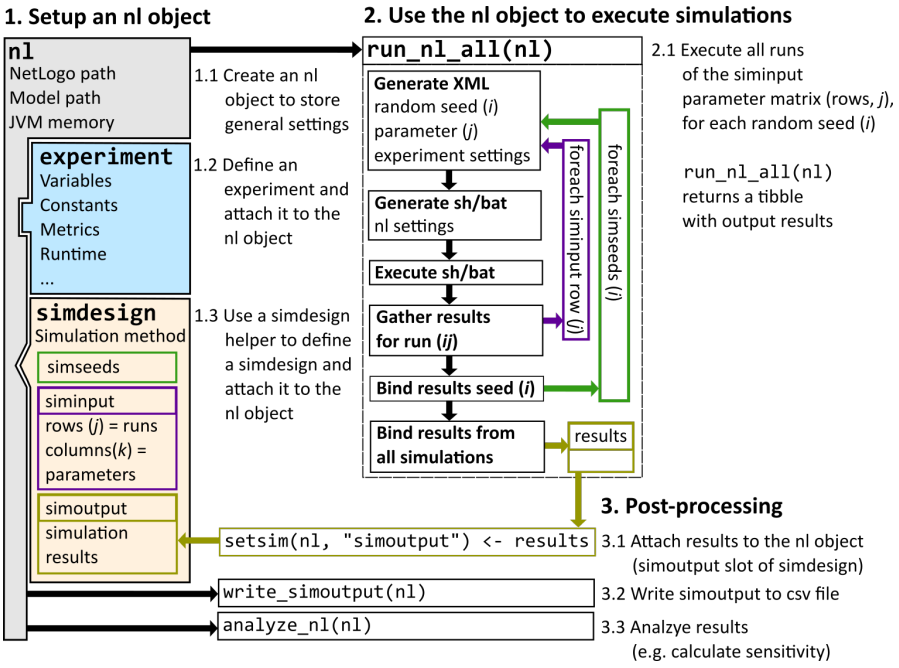


Figure 2: Workflow of the *nlrx* package. The `nl` object stores all the information needed to run NetLogo models. The `experiment` object, which contains all model-specific information, such as names of setup and go procedures, runtime, variable range and metrics and the `simdesign` object, which contains all simulation-related information, such as input parameter matrix, simulation method and simulation output, are nested within this `nl` object. Abbreviations: JVM=java virtual machine; XML=Extensible Markup Language; sh=bash script; bat=batch file

The typical workflow of the *nlrx* package starts with creation of an `nl` object, which

Table 2: List of slots of an `nl` S4 class object.

Slot	Example	Description
nlversion	`"6.0.4"`	Version number of the NetLogo version
nlpath	`"/home/NetLogo~6.0.4"`	Path to a NetLogo installation
modelpath	`"/home/NetLogo~6.0.4/app/models/Sample Models/Biology/Wolf~Sheep~Predation.nlogo"`	Path to a *.nlogo model file
jvmmem	`1024`	Java virtual machine memory in megabytes
experiment		Stores an `experiment` S4 class object
simdesign		Stores a `simdesign` S4 class object

is an S4 class object that stores basic information on NetLogo (see Fig. 2, and Tab. 2). The `nl` object contains information on the NetLogo version (`nlversion`), the path to the NetLogo directory (`nlpath`), the path to the NetLogo model file (`modelpath`) and the amount of reserved memory for each Java virtual machine (`jvmmem`).

Next, an `experiment` needs to be created and attached to the `nl` object (see Fig. 2, and Tab. 3). The `experiment` object stores all information that would be entered into a typical BehaviorSpace experiment (see Tab. 3). The `experiment` object contains the name of the experiment (`expname`), the directory where output is written to (`outpath`), the number of repeated runs within BehaviorSpace (`repetition`), a logical variable whether output should be measured on each tick or on the final tick only (`tickmetrics`), names of setup and go procedures (`idsetup, idgo`), the maximum number of ticks that should be simulated (`runtime`, 0=infinite), a vector of ticks for which output is reported (`evalticks`), a stop condition (`stopcond`), output metrics (`metrics`, `metrics.turtles`, `metrics.patches`, `metrics.links`), variable parameters and corresponding value ranges (`variables`), and constant parameters (`constants`).

Finally, a `simdesign` needs to be attached to the `nl` object (see Fig. 2, and Tab. 4). The *nlrx* package provides many helper functions to create pre-defined `simdesigns` for simple parameter screenings (simple, distinct, full-factorial), sensitivity analyses (Morris (Morris, 1991), Sobol (Sobol, 1990), eFast (Saltelli *et al.*, 1999)) or parameter optimizations (simulated annealing (Kirkpatrick *et al.*, 1983), genetic algorithm (Holland, 1992)). These helper functions take information from the experiment (`variables`, `constants`) to create a `simdesign` object that contains the name of the simulation design method (`simmethod`), a vector of random seeds (`simseeds`), the generated parameter matrix (`siminput`) and an empty tibble slot to attach output results after simulations have been finished (`simoutput`). The siminput and simoutput slots store tibble data, a modern representation of rectangular data in R (Müller & Wickham, 2018). The sensitivity

Table 3: List of slots of an `experiment` S4 class object.

Slot	Example	Description
expname	`"nlrx-example"`	Name of the BehaviorSpace experiment
outpath	`"/home/nlrxout"`	Path to an output directory
repetition	`1`	Number of repetitions
tickmetrics	`"true"`	If true, metrics are collected on each model tick, otherwise only on the last tick
idsetup	`"setup"`	Name or vector of names containing NetLogo procedures that are called to setup the model
idgo	`"go"`	Name or vector of names containing NetLogo procedures that are called to run the model
idfinal	`"final"`	Name or vector of names containing NetLogo procedures that are called at the end of one run
idrunnum	`"id-input"`	Name of a NetLogo numeric widget to transfer the current nlrx run number to NetLogo
runtime	`100`	Maximum runtime of model simulations (number of ticks)
evalticks	`seq(90, 100)`	Number or vector of ticks indicating when metrics should be collected
stopcond	`"not any? turtles"`	A NetLogo reporter that reports TRUE/-FALSE. If it reports TRUE the current simulation run is stopped
metrics	`c("count turtles")`	Valid NetLogo reporters that are used to collect output data
metrics.turtles	`list("turtles"=c("who", "pxcor","pycor","color"))`	A list with named vectors of strings defining valid turtles-own variables that are taken as output measurements
metrics.patches	`c("pxcor","pycor", "pcolor")`	A vector of strings defining valid patches-own variables that are taken as output measurements
metrics.links	`list("links"=c("end1", "end2"))`	A list with named vectors of strings defining valid links-own variables that are taken as output measurements
variables	`list("variable.parameter. 1"=list(min=50, max= 150, step=10, qfun= "qunif"))`	A list with NetLogo globals that should be varied in a `simdesign`
constants	`list("constant.parameter. 1"=4, "constant.parameter. 2"=0.1)`	A list with NetLogo globals that should stay constant in a `simdesign`

Table 4: List of slots of a `simdesign` S4 class object.

Slot	Description
simmethod	Name of the `simdesign` method (e.g. "ff"=full factorial, "lhs"=latin hypercube sampling)
siminput	tibble providing input parameterisations for the NetLogo model (cols=parameter, rows=runs)
simobject	List containing a `simdesign` object with further details on the simmethod that may be needed for output analysis
simseeds	Vector containing model seeds
simoutput	Slot to attach simulation results to the `nl` object

analyses and optimization simdesign helper functions also create a simulation object (`simobject`) that contains further information needed for post-processing, such as calculation of sensitivity indices. The optimization algorithms do not generate a parameter matrix in advance, because the parameterization of later runs depends on the results of previous runs. It is also possible to write user-defined helper functions to create `simdesign` objects.

After initialization of the `nl` object, simulations can be run by calling one of the `run_nl_*()` functions that come with *nlrx*. For `simdesigns` that generate a parameter matrix, `run_nl_one()` can be used to execute one specific row of the `simdesign` parameter matrix `siminput` by specifying the row ID and a model seed. This can be useful for testing specific parameterizations. The function `run_nl_all()` executes all simulations from the parameter matrix `siminput` across all seeds. `run_nl_all()` uses the `future_map_dfr()` function (*furrr* package) to loop over random seeds and rows of the `siminput` parameter matrix (see Fig. 2). This allows users to run the function sequentially or in parallel, on local machines and remote HPC machines. Parallelisation options can be easily adjusted by using different *future* plans before calling the function (for more details see documentation of R package *future*). For optimization `simdesigns` that do not come with a pre-generated parameter input matrix, `run_nl_dyn()` can be used to execute model simulations with dynamically generated parameter settings.

The connection between R and NetLogo is realized within these three `run_nl_*()` functions. First, a BehaviorSpace experiment XML file is created from the information that is stored within the provided `nl` object. This XML file contains the random seed, parameter settings and experiment settings. Next, depending on the operating system, a bash script or batch file is generated that is used to execute NetLogo via the command line in headless mode. The file contains information that is stored within the `nl` object (such as NetLogo path, model path, Java virtual machine memory and the path to the

previously generated XML file). Afterwards, the script is executed to run the NetLogo simulations. When the simulation is finished, the NetLogo model output data is read into R. In the case of multiple simulations, results are bound together and reported as one large tibble.

After simulations are finished, the resulting output table can be attached to the `nl` object by using the provided setter function `setsim`. This stores a complete experiment setup, including output results, in one R object. After outputs have been attached to the `nl` object, the output results can also be written to a *.csv file by calling `write_simoutput()`. The output can also be further analyzed by calling `analyze_nl()`. For example, for an `nl` object with a Morris `simdesign` and attached simulation output results running `analyze_nl()`, lists Morris sensitivity indices for each parameter and each measured model output. For reproducible research, it is important to attach the output to the `nl` object even if one does not want to conduct further analyses. By attaching the output to the `nl` object, all information of the simulation experiment, including parameter inputs and simulation output, is stored within an `nl` object, which can be easily stored as *.rds file for documentation purposes.

Further functionality

nlrx offers a function `unnest_simoutput()` to turn agent-specific metrics that were collected during the simulation into a wide-format table that splits every type of agent, patch and link in a unique column and nests all measured variables in this column. To be able to derive this information, the `nl` object offers slots to specify agent, patch and link metrics. The unnested spatial data can easily be visualized to illustrate model behavior (Fig. 3). Furthermore, the functions `nl_to_points()`, `nl_to_raster()` and `nl_to_graph()` coerce the `nl` object directly into a spatial data type (e.g. spatial points and raster data for agents and patches, as well as undirected network graphs for links). To be able to use *nlrx* as a fully reproducible framework, the functions `export_nl()` and `import_nl()` store R and corresponding NetLogo model scripts in a zip file, which uses relative paths and thus enables easy collaboration. Furthermore, *nlrx* provides functions to generate documentation files as *.html, *.pdf and *.docx files from NetLogo model files containing specific documentation comment sections. *nlrx* also provides a function to generate a network of NetLogo model procedures (for details see Supplementary material 3). *nlrx* will thus enable ecologists to make use of agent-based NetLogo models in their research while permitting a workflow that follows modern scientific standards.

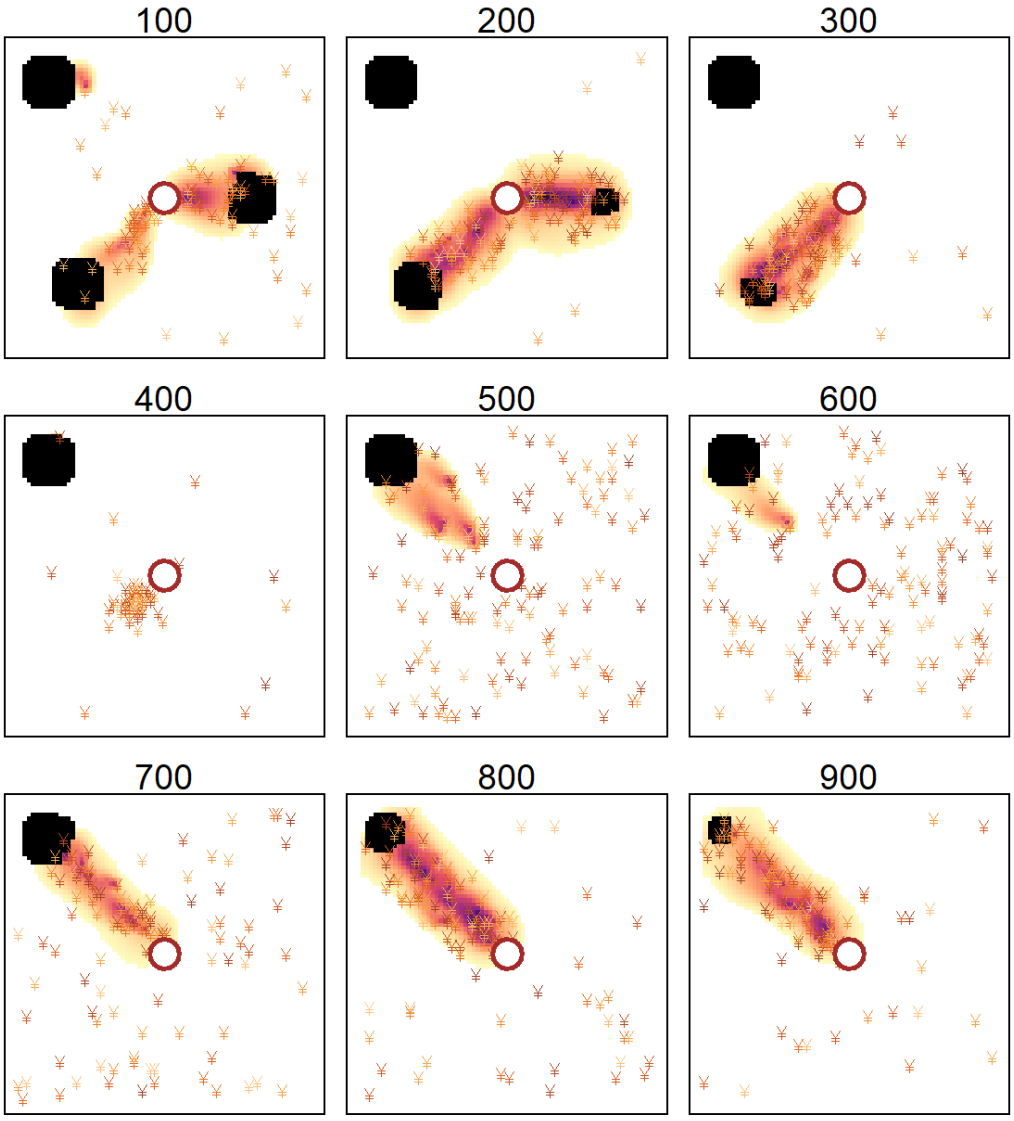


Figure 3: Visualization of spatial output gathered from the Ants model (see section 'Use case: Ants model' for a description of the model) via the nlrx package. Ants positions (brown yen signs), position of food cells (black), position of the nest cell (brown circle), and amount of chemicals on all cells (heat shading, with darker colors indicating higher amount of chemicals) were measured using the metrics.turtles and metrics.patches slots of the nlrx experiment. Panels show output for every 100th tick of a model simulation.

Use case: Ants model

This section describes two example use cases of *nlrx* using the Ants simulation model from the NetLogo Models Library: (i) a Sobol sensitivity analysis based on R's advanced statistical packages, (ii) a genetic algorithm optimization approach to optimize foraging speed.

The Ants model is a spatially-explicit model of an ant colony foraging for food. The landscape is set up as a patch lattice. The center cell of the landscape is the nest of the ant colony and all ants (number depending on parameter *population*) are created within this nest cell on initialization. Additionally, there are three clusters of food source cells created within the landscape. Food source clusters are created at different distances from the ant nest, with food source 1 being closest to the nest and food source 3 being furthest away from the nest. During a model run, ants move randomly in order to find food. If any ant finds a food source cell, a certain amount of food is transferred from that cell to the ant, which carries the food back to the nest. Each food cell only supplies a certain amount of food and is reset to a non-food cell once all food from this cell has been gathered. Any ant carrying food releases chemicals at its current position. Chemicals also diffuse to neighboring cells, depending on the parameter *difussion-rate* and evaporate depending on the parameter *evaporation-rate*. If ants sense chemicals on their current or neighboring patch, they change their movement pattern from random walk to directed walk following the trail of chemicals by moving into the direction which contains the most chemicals. The main output of the model simulation is the time needed to gather all food from all food sources. We also measure the time that is needed to completely deplete specific food source clusters. Furthermore, we are interested in the spatial distribution of ants under different parameterizations of the model (for details see Supplementary material 4).

Sensitivity Analysis with *nlrx*

In order to demonstrate the usability and workflow of the *nlrx* package, we performed a global sensitivity analysis of the Ants model (Sobol, 1990). We varied the three model parameters *population*, *diffusion-rate* and *evaporation-rate* by applying a Sobol parameter variation approach. As main output we measured the simulation time (ticks) until all food sources were completely depleted by the ants.

To set up the sensitivity analysis with *nlrx*, we first defined an `nl` object (List. 1).

```
library(nlrx)
nl <- nl(nlversion = "6.0.4",
         nlpath = "/home/usr/NetLogo_6.0.4/",
         modelpath = "/home/usr/NetLogo_6.0.4/app/models/Biology/Ants.nlogo",
         jvmmem = 1024)
```

Listing 1: Defining an `nl` object.

In the next step, we attached an `experiment` to the previously generated `nl` object (List. 2). We defined that output should be measured only on the final tick (`tickmetrics` = "false"). We set the runtime to infinite (`runtime`, 0=infinite) and defined a stop condition to stop model execution once no food cell is left in the model landscape (`stopcond`). As output metrics we measured the number of ticks simulated when the model run finishes (`metrics`).

```
nl@experiment <- experiment(expname = "nlrx_ants_sobol",
                            outpath = "/home/usr/output",
                            repetition = 1,
                            tickmetrics = "false",
                            idsetup = "setup",
                            idgo = "go",
                            runtime = 0,
                            stopcond = "not any? patches with [food > 0]",
                            metrics = c("ticks"),
                            variables = list(
                                "population" = list(min=10, max=200, qfun="qunif"),
                                "diffusion-rate" = list(min=1, max=99, qfun="qunif"),
                                "evaporation-rate" = list(min=1, max=99, qfun="qunif")
                                )
                            )
```

Listing 2: Attaching an `experiment` to an `nl` object.

To construct an input parameter matrix from the defined experiment and variable ranges, a `simdesign` needs to be attached to the `nl` object (List. 3). For our Sobol sensitivity analysis, we used the `simdesign` helper function `simdesign_sobol()`. We defined the number of samples (`samples`), the order of interaction effects (`sobolorder`), the number of bootstrapping replicates (`sobolnboot`), the confidence interval (`sobolconf`), the number of Sobol analysis repetitions (`nseeds`) and the precision level of values within the parameter matrix (number of decimal digits, `precision`).

```
nl@simdesign <- simdesign_sobol(nl = nl,
                                samples = 1000,
                                sobolorder = 2,
                                sobolnboot = 100,
                                sobolconf = 0.95,
                                nseeds = 6,
                                precision = 3)
```

Listing 3: Attaching a Sobol `simdesign` to an `nl` object.

This `nl` object stores all information needed to run the sensitivity analysis. We can execute all simulations by calling `run_nl_all()` (List. 4). Because in our experiment we set `tickmetrics` to `"false"`, output will only be reported for the final simulation step.

```
results <- run_nl_all(nl)
```

Listing 4: Executing all simulations of an `nl` object.

In order to execute the `analyze_nl()` function, the results tibble first needs to be attached to the `nl` object (List. 5).

```
setsim(nl, "simoutput") <- results
sensitivityIndices <- analyze_nl(nl)
```

Listing 5: Attaching simulation results to an `nl` object and calculating sensitivity indices.

The Sobol sensitivity analysis revealed a very large main effect of the *population* parameter on the effectiveness of food collection (measured as number of ticks until all food sources are depleted) by the ant colony (see Fig. 4). The more ants were present, the faster the colony depleted all food sources. The two chemical-related parameters *evaporation-rate* and *diffusion-rate* showed only small main effects (see Fig. 4). However, we found some interaction effects between *population* and *evaporation-rate*, albeit at a very low level. There were no interactions between *population* and *diffusion-rate* and between *diffusion-rate* and *evaporation-rate*.

```
write_simoutput(nl)
export_nl(nl, nl@modelpath, <path-to-disk>)
```

Listing 6: Export simulation output and model files.

After running all simulations with *nlrx*, we store the simulation output as a *.csv file using the `write_simoutput()` function (see List. 6). Additionally, we store the

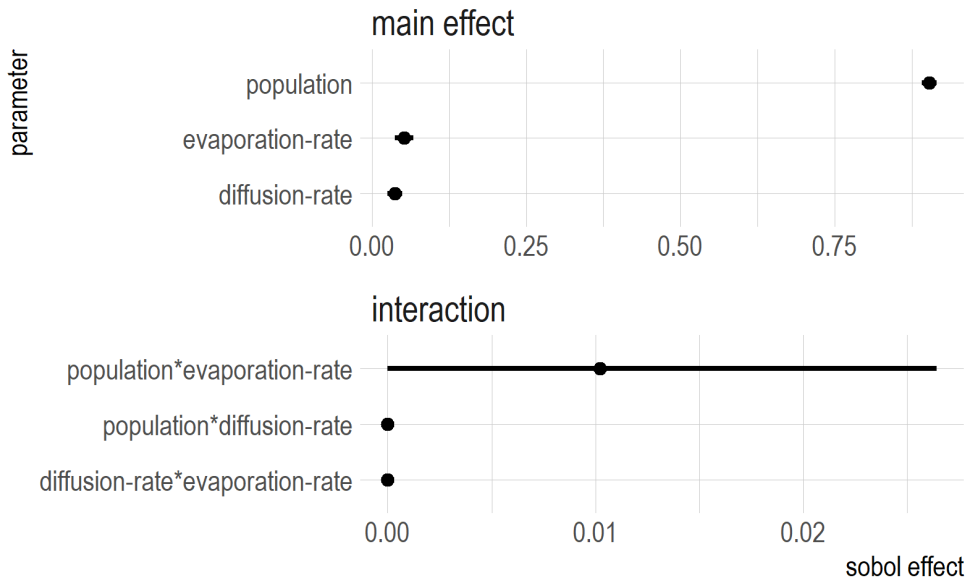


Figure 4: Main effects (upper panel) and interaction effects (lower panel) of each parameter of the Ants model on the total number of simulation steps that were simulated until all food sources were depleted by the ant individuals. Parameter effects were derived by applying a Sobol sensitivity analysis approach. Points indicate parameter mean sensitivity (Sobol effect) of 6 model simulation replicates, linebars indicate the corresponding standard deviation.

nl object, together with underlying model files as a zip folder using the `export_nl()` function.

Genetic Algorithm with *nlrx*

In order to demonstrate the optimization functions of *nlrx*, we applied a genetic algorithm to the Ants model. The genetic algorithm varies parameters of the model within defined ranges, in order to minimize a defined fitness criterion. In our case, we want to minimize the number of time steps needed by the ant colony to deplete all food sources completely.

For this analysis, we do not need to make any changes to the `nl` object and experiment that we used in the Sobol sensitivity analysis example, thus we can reuse the `nl` object and just add a different `simdesign`. In order to set up a genetic algorithm optimization `simdesign`, we used the `simdesign` helper function `simdesign_GenAlg()` (List. 7). We

defined the number of initial parameterizations (`popSize`), the number of iterations (`iters`), the index of a defined output metric (see `metrics` slot in `experiment`) that is used as evaluation criterion (`evalcrit`), the elitism rate (`elitism`), the mutation probability (`mutationChance`) and the number of random seeds for replications of the algorithm (`nseeds`).

```
nl@simdesign <- simdesign_GenAlg(nl = nl,
                                 popSize = 100,
                                 iters = 10,
                                 evalcrit = 1,
                                 elitism = NA,
                                 mutationChance = NA,
                                 nseeds = 1)
```

Listing 7: Attaching a genetic algorithm `simdesign` to an `nl` object.

In contrast to sensitivity analysis `simdesigns`, for optimization `simdesigns` there is no parameter input matrix set up in advance. Instead, random starting values are chosen within the defined parameter ranges. The parameterization of the next simulation run is dynamically created, depending on the output of the previous simulation. Thus, for optimization `simdesigns`, the *nlrx* function `run_nl_dyn()` needs to be used to execute the simulations (instead of `run_nl_all()` which iterates over a pre-generated parameter matrix)(List. 8).

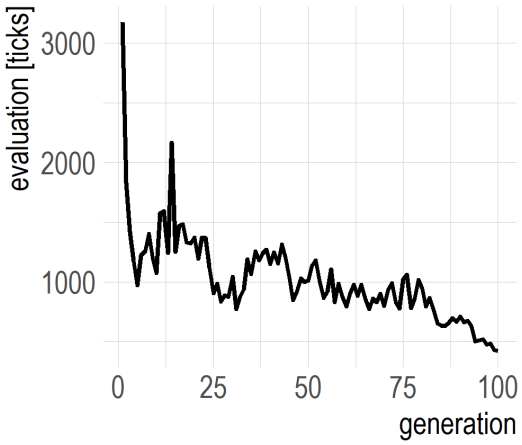
```
results <- run_nl_dyn(nl = nl,
                      seed = getsim(nl, "simseeds")[1])

setsim(nl, "simoutput") <- results
```

Listing 8: Executing dynamic simulations of an `nl` object with an optimization approach.

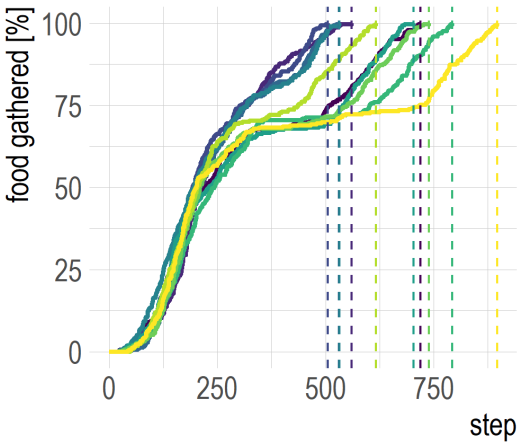
The results from the genetic algorithm `simdesign` are reported as a list which contains information of parameterizations and the evaluation criterion of each population of the genetic algorithm. We can also identify the best found parameterization, i.e. the parameterization that resulted in the smallest fitness value.

The genetic algorithm was able to decrease the time to deplete all food sources from more than 3000 ticks to below 500 ticks over 100 iterations (see Fig. 5, upper left). The parameterization of the best found solution had a moderately large *population* (166), a medium *diffusion-rate* (18.28), and a low *evaporation-rate* (8.41) (see Fig. 5, upper right).

To evaluate the stability of the optimization result under different random seeds, we simulated an additional 10 replicates (each with a different random seed) of the best



parameter	value
population	166.00
diffusion-rate	18.28
evaporation-rate	8.41



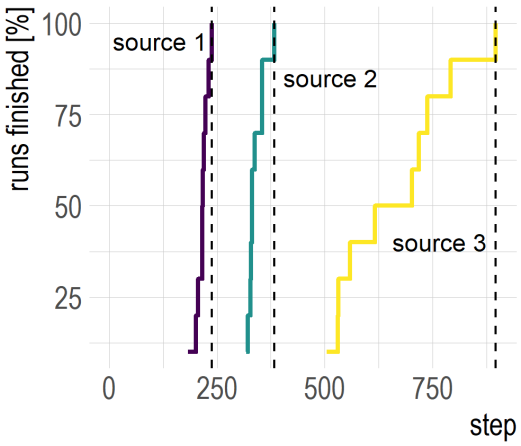


Figure 5: A genetic algorithm was applied to minimize simulation time until all food sources were depleted. The figure illustrates the minimization process over iterations of the algorithm (upper left) and the best found parameterization in terms of minimum simulation time (upper right). This best found parameterization was used to run the Ants model with 10 replicates and measuring the percentage of food gathered at each tick (lower left, colors indicate individual simulations). We also aggregated these 10 runs to measure at which time steps the three individual food source clusters were completely depleted (lower right). Dashed lines illustrate final time step of each curve.

parameterization. In order to run a specific parameterization, the *nlrx* package provides the `simdesign` helper function `simdesign_simple()`, which creates a parameter table for one single simulation that uses only defined values of constant parameters (`constants` slot of `experiment`). We removed all variable parameters from the experiment and defined those parameters as `constants` instead (List. 9). We also added counts of food

sources as output metrics to the `metrics` slot of the experiment. Finally, we used the `simdesign_simple()` function to attach a `simdesign` containing a `siminput` matrix with only one parameterization and a `simseeds` vector with 10 random seeds (`nseeds = 10`).

Again, we used the `run_nl_all()` function to execute the simulations and calculated the percentage of food gathered at each tick and identified the tick at which specific food source clusters were depleted completely. We observed some variation in total simulation time between these replicates, but all runs lay between 500 and 800 ticks (see Fig. 5, lower left). While the first two food sources were depleted very fast in all replicates, depletion times were more variable for the third food source, which had the largest distance from the ant nest (see Fig. 5, lower right).

```
nl@experiment@expname <- "ants_genalg_valid"
nl@experiment@tickmetrics <- "true"
nl@experiment@variables <- list()
nl@experiment@constants <- list("population" = 166,
                                "diffusion-rate" = 18.28,
                                "evaporation-rate" = 8.41)
nl@experiment@metrics <- c("ticks",
                           "sum [food] of patches with [pcolor = cyan]",
                           "sum [food] of patches with [pcolor = sky]",
                           "sum [food] of patches with [pcolor = blue]")

nl@simdesign <- simdesign_simple(nl, nseeds = 10)
results.best <- run_nl_all(nl)
```

Listing 9: Setting specific slots of the `experiment` within an `nl` object and attaching a simple `simdesign`.

Conclusion and outlook

Next-generation agent-based models must be reproducible and repeatable. The new *nlrx* R-package fulfills the need for an efficient tool for running reproducible and repeatable analyses for the popular programming language NetLogo. *nlrx* has several advantages in comparison to previous approaches that connect R and NetLogo: (1) the package does not rely on the *rJava* package which is known to be unstable and causes many problems especially when working on machines with different configurations; (2) setup of model experiments is very similar to NetLogo BehaviorSpace. Thus, NetLogo users who do not have much experience with R will feel familiar with the workflow of *nlrx*. (3) multiprocessing and execution of large model simulations on remote HPC machines can be easily done with *nlrx* by adjusting the future plan of the R session. (4) all information

on the model version, simulation experiment definitions and simulation design definitions are stored within one single R object. By using the *nlrx* export and import functions, these objects can be archived and shared, together with all required NetLogo model files. This functionality enables a reproducible workflow and easy collaboration.

In contrast to *RNetLogo*, NetLogo instances of *nlrx* are not interactive. There is no direct connection from R to NetLogo and thus, it is not possible to manipulate the NetLogo model from within R while the model is running. However, for most use cases it is sufficient to set up and run model simulations in a convenient way because manual exploration of NetLogo models can also be done directly in NetLogo.

The presented use cases of the Ants model highlight the flexibility and convenient workflow of the *nlrx* framework. With the nested class layout the same `nl` object and `experiment` can be reused for several simulation design approaches (as demonstrated in the genetic algorithm use case). Thus, research questions can be tackled from very different angles with minimal code adjustments. Because reported output from model simulations is reported in a tidy data format (see section 3 Data Accessibility for additional R code examples), post-processing, data analysis and result visualization can benefit from the well-established tool set provided by the *tidyverse* framework (e.g. *dplyr*, *ggplot2* packages). Collecting agent metrics and coercing them to spatial objects is now possible with *nlrx* in two lines of code, thus enabling ecologists to use advanced statistical methods on the individual level of their simulation models (see Supplementary Material 4).

Increasing model complexity across many disciplines increases the need of standardized tools and open standards for model documentation, application and analysis. For example, the ODD (Overview, Design concepts, and Details) protocol and the TRACE (TRAnsparent and Comprehensive Ecological modelling documentation) framework are such standards for documentation of agent-based models that are widely accepted and applied within the field of agent-based modelling (Grimm *et al.*, 2014, 2006, 2010; Müller *et al.*, 2013). However, given the rising importance of reproducible research as a standard in academia, standards for documentation and reproducibility of applied model analysis and corresponding code are still lacking. *nlrx* is a first step to provide a framework for documentation and application of reproducible NetLogo simulation model analysis. While *nlrx* mainly provides tools that enable reproducible research, future work might also give recommendations on a *best practice* approach that also incorporates data management, collaboration and version control.

nlrx has been officially released on CRAN (current version 0.2.0) and peer-reviewed by rOpenSci (https://github.com/ropensci/onboarding/issues/188).

1 Acknowledgments

JS and KW were funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - project number 192626868 - SFB 990 in the framework of the collaborative German - Indonesian research project CRC 990. MS and KW were supported by the DFG through grant number WI 1816/18-1 (FOR2432) and DFG-GRK 1644/3. We thank rOpenSci for their extensive code review and three reviewers for their helpful and constructive comments on an earlier version of this manuscript.

2 Author Contribution

JS and MS developed the R package. The case study was designed by JS. JS and MS drafted the manuscript and all authors contributed critically to the manuscript and gave final approval for publication.

3 Data Accessibility

The *nlrx* package is hosted at DOI:10.5281/zenodo.3367095. All the code for the figures and use cases in this manuscript can be found at DOI:10.5281/zenodo.2650853.

4 References

References

Abar, S., Theodoropoulos, G.K., Lemarinier, P. & O'Hare, G.M. (2017) Agent Based Modelling and Simulation tools: A review of the state-of-art software. *Computer Science Review*, **24**, 13–33.

Cabral, J.S., Valente, L. & Hartig, F. (2017) Mechanistic simulation models in macroecology and biogeography: state-of-art and prospects. *Ecography*, **40**, 267–280.

DeAngelis, D.L. & Grimm, V. (2014) Individual-based models in ecology after four decades. *F1000Prime Reports*, **6**.

Dislich, C., Hettig, E., Salecker, J., Heinonen, J., Lay, J., Meyer, K.M., Wiegand, K. & Tarigan, S. (2018) Land-use change in oil palm dominated tropical landscapes—An agent-based model to explore ecological and socio-economic trade-offs. *PLOS ONE*, **13**, e0190506.

Grimm, V., Berger, U., Bastiansen, F., Eliassen, S., Ginot, V., Giske, J., Goss-Custard, J., Grand, T., Heinz, S.K., Huse, G., Huth, A., Jepsen, J.U., Jorgensen, C., Mooij, W.M., Müller, B., Pe'er, G., Piou, C., Railsback, S.F., Robbins, A.M., Robbins, M.M., Rossmanith, E., Rüger, N., Strand, E., Souissi, S., Stillman, R.A., Vabo, R., Visser, U. & DeAngelis, D. (2006) A standard protocol for describing individual-based and agent-based models. *Ecological Modelling*, **198**, 115–126.

Grimm, V. & Railsback, S.F. (2005) *Individual-Based Modeling and Ecology*. Princeton University Press.

Grimm, V., Augusiak, J., Focks, A., Frank, B.M., Gabsi, F., Johnston, A.S., Liu, C., Martin, B.T., Meli, M., Radchuk, V., Thorbek, P. & Railsback, S.F. (2014) Towards better modelling and decision support: Documenting model development, testing, and analysis using trace. *Ecological Modelling*, **280**, 129 – 139.

Grimm, V. & Berger, U. (2016) Structural realism, emergence, and predictions in next-generation ecological modelling: Synthesis from a special issue. *Ecological Modelling*, **326**, 177–187.

Grimm, V., Berger, U., DeAngelis, D.L., Polhill, J.G., Giske, J. & Railsback, S.F. (2010) The ODD protocol: A review and first update. *Ecological Modelling*, **221**, 2760–2768.

Holland, J.H. (1992) *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA.

Janssen, M.A. (2017) The practice of archiving model code of agent-based models. *Journal of Artificial Societies and Social Simulation*, **20**, 2.

Kirkpatrick, S., Gelatt, C.D. & Vecchi, M.P. (1983) Optimization by simulated annealing. *Science*, **220**, 671–680.

Lai, J., Lortie, C.J., Muenchen, R.A., Yang, J. & Ma, K. (2019) Evaluating the popularity of R in ecology. *Ecosphere*, **10**, e02567.

Morris, M.D. (1991) Factorial sampling plans for preliminary computational experiments. *Technometrics*, **33**, 161–174.

Müller, B., Bohn, F., Dreßler, G., Groeneveld, J., Klassert, C., Martin, R., Schlüter, M., Schulze, J., Weise, H. & Schwarz, N. (2013) Describing human decisions in agent-based

models – ODD + D, an extension of the ODD protocol. *Environmental Modelling & Software*, **48**, 37–48.

Müller, K. & Wickham, H. (2018) *tibble: Simple Data Frames*. R package version 1.4.2.

Peng, R.D. (2011) Reproducible research in computational science. *Science*, **334**, 1226–1227.

Railsback, S., Ayllón, D., Berger, U., Grimm, V., Lytinen, S., Sheppard, C. & Thiele, J. (2017) Improving execution speed of models implemented in Netlogo. *Jasss*, **20**, 3.

Saltelli, A., Tarantola, S. & Chan, K.S. (1999) A quantitative model-independent method for global sensitivity analysis of model output. *Technometrics*, **41**, 39–56.

Saltelli, A., Ratto, M., Andres, T., Campolongo, F., Cariboni, J., Gatelli, D., Saisana, M. & Tarantola, S. (2008) *Global sensitivity analysis: the primer*. John Wiley & Sons.

Sandve, G.K., Nekrutenko, A., Taylor, J. & Hovig, E. (2013) Ten simple rules for reproducible computational research. *PLOS Computational Biology*, **9**, 1–4.

Sobol, I.M. (1990) Sensitivity analysis for nonlinear mathematical models. *Matematicheskoe Modelirovanie*, **2**, 112–118.

Thiele, J.C., Kurth, W. & Grimm, V. (2012) RNetLogo: An R package for running and exploring individual-based models implemented in NetLogo. *Methods in Ecology and Evolution*, **3**, 480–483.

Thiele, J.C., Kurth, W. & Grimm, V. (2014) Facilitating Parameter Estimation and Sensitivity Analysis of Agent-Based Models : A Cookbook Using NetLogo and R. *Journal of Artificial Societies and Social Simulation*, **17**, 11.

Vincenot, C.E. (2018) How new concepts become universal scientific approaches: insights from citation network analysis of agent-based complex systems science. *Proceedings Biological sciences*, **285**, 20172360.

Wickham, H. (2014) Tidy data. *Journal of Statistical Software, Articles*, **59**, 1–23.

Wilensky, U. (1999) *NetLogo*. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.