**pyNetLogo: Linking NetLogo with Python**

**Jan H. Kwakkel, Marc Jaxa-Rozen**

*Faculty of Technology, Policy and Management, Delft University of Technology*

**Abstract:** …

***Keywords****:* agent-based modelling; social-ecological systems; NetLogo; Python

1. **INTRODUCTION**

This work introduces the pyNetLogo library, which can be used to link the NetLogo agent-based modelling platform with a Python environment.

Section 2 describes the different software platforms used in this work; Section 3 describes the pyNetLogo connector, which provides real-time interaction mechanisms between NetLogo and Python, then illustrates these mechanisms for a simple predator-prey model.

1. **SOFTWARE DESCRIPTION**
   1. **NetLogo**

NetLogo (Wilensky, 1999) is an open-source environment for the design and testing of agent-based models, which has become a leading platform for this purpose due to its user-friendliness and active user community. This tool is primarily implemented in Java and Scala, and includes a range of functions and methods to support the rapid development of spatially-explicit agent-based models. Different extension modules are also available, for instance to allow an interface with GIS datasets, or to link NetLogo with the R package (Thiele, 2015; Thiele, Kurth, & Grimm, 2012a).

* 1. **Python**

Python is a general-purpose, object-oriented programming language which is increasingly popular for scientific and engineering applications. An extensive set of libraries is available for general data manipulation and analysis (e.g. Numpy/Pandas), as well as interfaces with specific software packages and other environments. As such, the JPype library can be used to access Java classes from Python and provides the ability to interactively communicate with the NetLogo API. The pyNetLogo connector is also included in the EMA Workbench Python package (Kwakkel, 2017), which offers support for experiment design and exploratory modeling and analysis.

1. **SOFTWARE IMPLEMENTATION**

This section first describes basic interactions between the Python environment and a NetLogo model, using the pyNetLogo connector. These interactions are demonstrated using the wolf-sheep predation example which is available in NetLogo’s model library.

The Python modules used in this work will be made available at <https://github.com/quaquel/pyNetLogo>. These modules have been tested with a standard distribution for scientific Python (Continuum Anaconda 3.6); using this distribution, the modules require the additional installation of the JPype Python package, which is available with the standard *pip* package manager.

* 1. **Controlling NetLogo through Python with pyNetLogo**

The pyNetLogo connector is composed of a Python module and a Java class (respectively pyNetLogo and NetLogoLink, in Figure 1 below), which are linked with the JPype package through a Java Native Interface (JNI). The NetLogoLink Java class in turn communicates with the NetLogo API. This allows for bidirectional data exchanges between a Python environment (which can for instance be an interactive IPython Notebook) and a NetLogo model at runtime, with appropriate data type conversions between the two environments. The connector currently supports NetLogo 5.x and NetLogo 6.0.

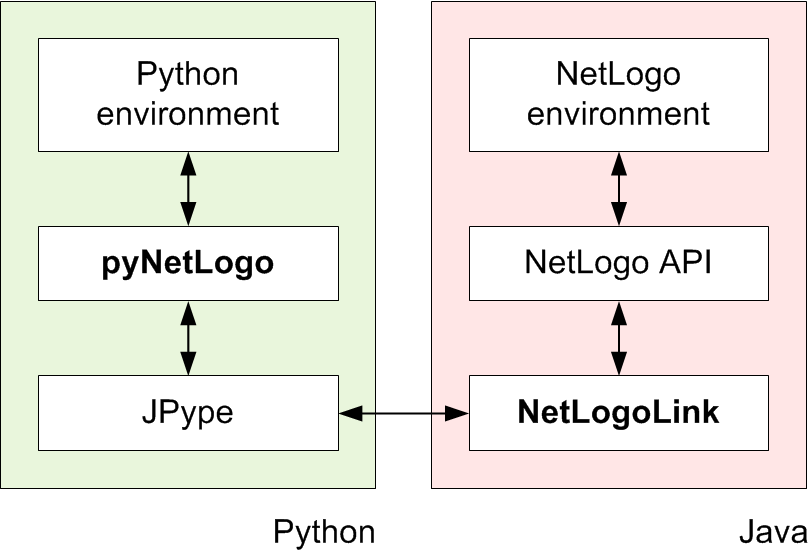


Figure : Interactions between Python and NetLogo

The table below summarizes the basic functions available through the pyNetLogo connector. These functions are intended to provide “building blocks” for the interactive analysis of NetLogo models with Python, and largely replicate the basic functionality of the RNetLogo connector for the R environment (Thiele, Kurth, & Grimm, 2012b).

Table : Basic PyNetLogo functions

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Description** | **Arguments** | **Returns** |
| load\_model() | Load a NetLogo model file | Model path (string) | - |
| kill\_workspace() | Close the NetLogo instance and shut down the Java virtual machine | - | - |
| command() | Execute a given command in the NetLogo environment | Valid NetLogo command (string) | - |
| report() | Return the value of a NetLogo reporter | Valid NetLogo reporter (string) | Reported value, converted to appropriate Python data type |
| patch\_report() | Return values for an attribute of the NetLogo patches | Valid NetLogo patch attribute (string) | Pandas dataframe of patch attribute values, with column labels and row indices following NetLogo patch coordinates |
| patch\_set() | Set NetLogo patch attributes from a Pandas dataframe | - Valid NetLogo patch attribute (string)  - Pandas dataframe with same dimensions as the NetLogo world, containing attribute values to be set | - |
| repeat\_command() | Execute a given command a number of times in the NetLogo environment | - Valid NetLogo command (string)  - Number of repetitions (integer) | - |
| repeat\_report() | Return the values of one or multiple NetLogo reporters over a given number of ticks | - Valid NetLogo reporter (string)  - Number of repetitions (integer) | Pandas dataframe of reported values with columns for each reporter, indexed by NetLogo ticks |
| write\_NetLogo\_attriblist() | Update a set of NetLogo agents of the same type with multiple attributes | - Pandas dataframe containing attribute values to be set, indexed by agent  - Valid NetLogo agent type (breed) | - |

To illustrate this functionality, a simple example follows below, using the Wolf Sheep Predation model which is included in the NetLogo example library. The IPython Notebook attached to this paper demonstrates the key functions of the PyNetLogo connector in more detail using this model.

A link is first instantiated to NetLogo, using the *load\_model* function, followed by basic commands to set up the model and run it for 100 ticks. The *report* function is then used to return Numpy arrays to the Python workspace, containing the NetLogo coordinates of the "sheep" agents, and the energy attribute of the “sheep” and “wolf” agents. These arrays can then for instance be used with conventional Python functions to plot the coordinates of the agents, or the distribution of energy across agents.

netlogo = pyNetLogo.NetLogoLink(gui=True)

netlogo.load\_model(r'Wolf Sheep Predation\_v6.nlogo')

netlogo.command('setup')

netlogo.repeat\_command('go', 100)

x = netlogo.report('map [[?1] -> [xcor] of ?1] sort sheep')

y = netlogo.report('map [[?1] -> [ycor] of ?1] sort sheep')

energy\_sheep = netlogo.report('map [[?1] -> [energy] of ?1] sort sheep')

energy\_wolves = netlogo.report('map [[?1] -> [energy] of ?1] sort wolves')

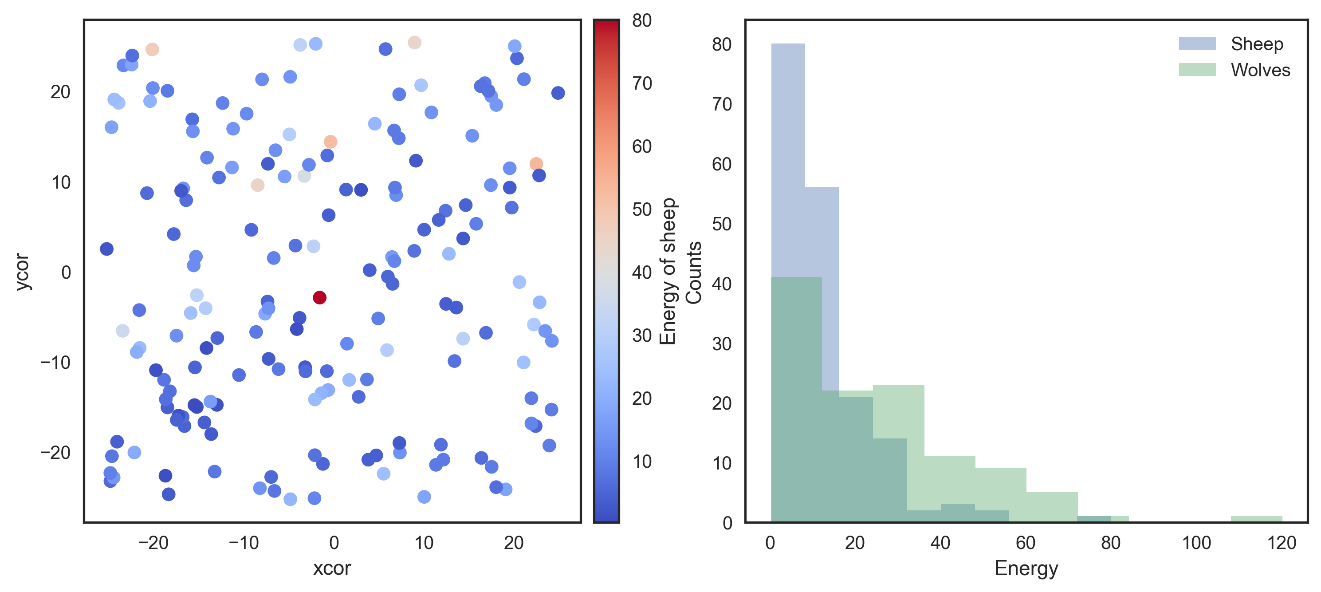


Figure 2: Basic plots generated in Python: agent coordinates (left); distribution of energy attribute across agents (right)

Building on this functionality, the *repeat\_report* function returns a Pandas dataframe containing reported values over a given number of ticks, for one or multiple NetLogo reporters. In this case, we can first track the number of “wolf” and “sheep” agents over 200 ticks; the *repeat\_report* function can also be used with reporters that return an array – in this case, for the energy of both agent types for 5 ticks:

counts = netlogo.repeat\_report(['count wolves','count sheep'], 200)

energy\_df = netlogo.repeat\_report(['[energy] of wolves', '[energy] of sheep'], 5)

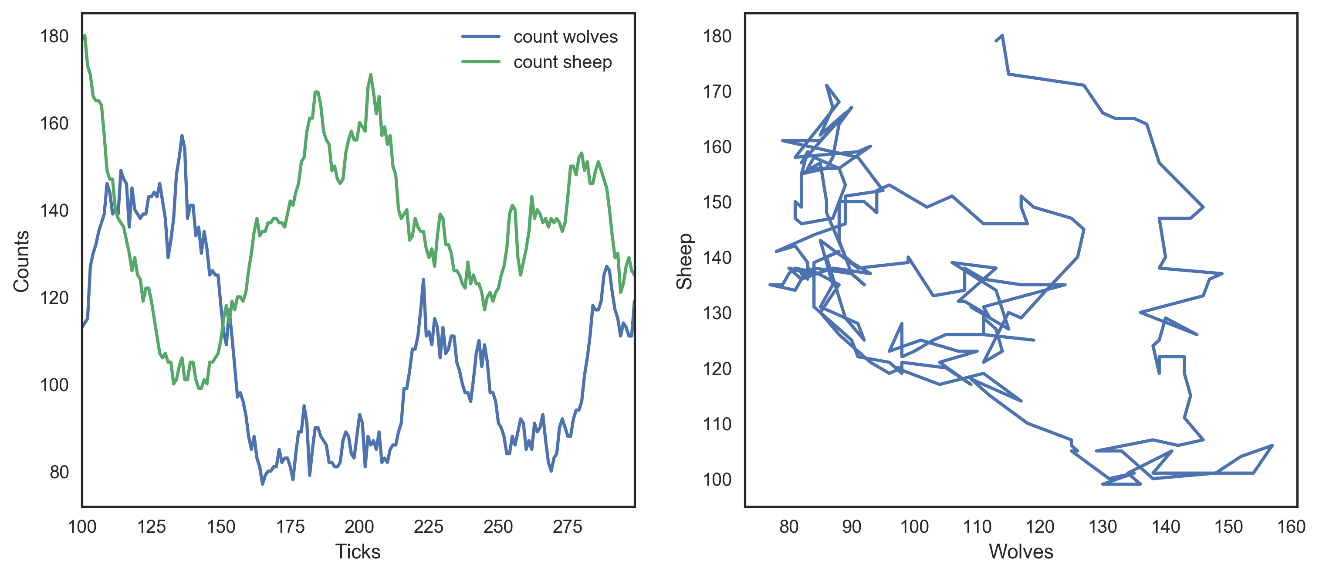


Figure 3: Python plots using *repeat\_report* function: number of agents as a function of time (left); number of sheep agents as a function of wolf agents (right)

In addition to these reporting functions, the *patch\_report* function can also be used to return a dataframe which (for this example) contains the *countdown* attribute of each NetLogo patch:

patch\_df = netlogo.patch\_report('countdown')

This dataframe essentially replicates the NetLogo environment, with column labels corresponding to the p*xcor* patch coordinates, and row indices following the *pycor* coordinates. The dataframes can be manipulated with any of the existing Pandas functions, for instance by exporting to an Excel file. The *patch\_set* function provides the inverse functionality to *patch\_report*, and updates the NetLogo environment from a dataframe.

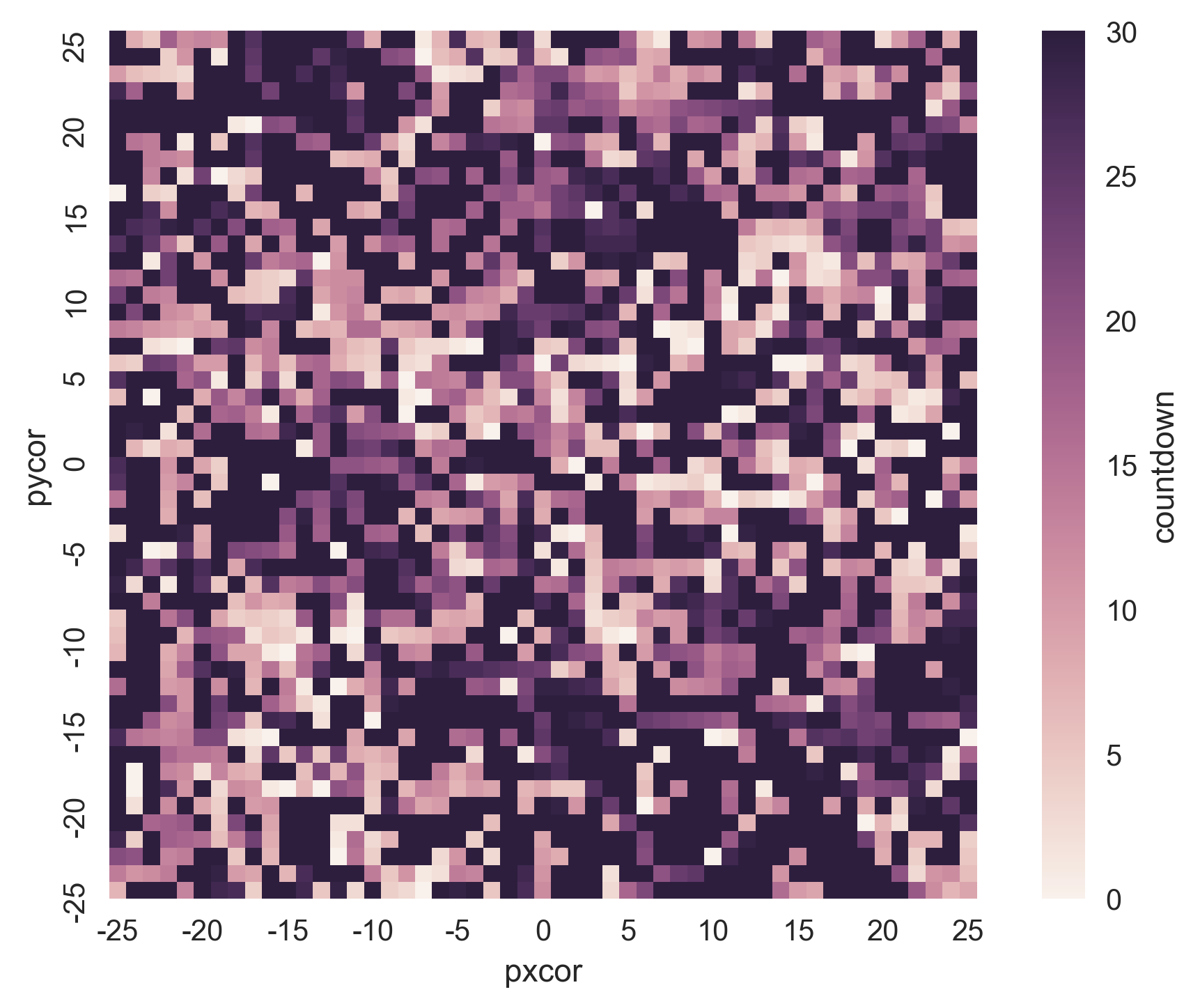


Figure 4: Python plot using *patch\_report* function: distribution of the *countdown* patch attribute across the NetLogo environment

* 1. **Using Python for global sensitivity analysis on a NetLogo model**

The Python environment enables access to a wide variety of packages to support the development and analysis of NetLogo models. As an example, this subsection uses the SALib Python library (Herman & Usher, 2017) for a Sobol global sensitivity analysis on the Wolf Sheep Predation model. The SALib library can be applied to extend the functionality of NetLogo’s native BehaviorSpace tool, with sampling and analysis modules for techniques including Sobol indices, Morris elementary effects, and Fourier amplitude sensitivity testing.

SALib relies on a problem definition dictionary which contains the number of input parameters to sample, their names (which should here correspond to a NetLogo global variable), and the sampling bounds:

problem = {

'num\_vars': 6,

'names': ['random-seed','grass-regrowth-time','sheep-gain-from-food',

'wolf-gain-from-food','sheep-reproduce','wolf-reproduce'],

'bounds': [[1, 100000], [20., 40.], [2., 8.],

[16., 32.], [2., 8.], [2., 8.]]

}

The SALib sampler will generate an appropriate experimental design based on the analysis technique to be used. To calculate first-order, second-order and total Sobol sensitivity indices, this gives a sample size of *n\*(2p+2)*, where *p* is the number of input parameters, and *n* is a baseline sample size which should be large enough to stabilize the estimation of the indices. For this example, we use *n* = 200 to reduce runtime, for a total of 2800 experiments. For more complex cases, the EMA Workbench library (Kwakkel, 2017) can be used to parallelize the simulation.

from SALib.sample import saltelli

from SALib.analyze import sobol

n = 200

#Generates an input array of shape (n\*(2p+2), p) with rows for each experiment and columns for each input

param\_values = saltelli.sample(problem, n, calc\_second\_order=True)

Assuming we are interested in the mean number of sheep and wolf agents over a timeframe of 100 ticks, we first create an empty dataframe to store the results. We then simulate the model over the 2800 experiments, reading input parameters from the *param\_values* array generated by SALib.

For simplicity, the *repeat\_report* command is used to track the outcomes of interest over time. Performance can be improved by using NetLogo's text output commands to store time series outcomes; this method is also implemented in the EMA Workbench.

results = pd.DataFrame(columns=['Avg. sheep', 'Avg. wolves'])

for run in range(param\_values.shape[0]):

#Set the input parameters

for i, name in enumerate(problem['names']):

if name == 'random-seed':

#The NetLogo random seed requires a different syntax

netlogo.command('random-seed {}'.format(param\_values[run,i]))

else:

#Otherwise, assume the input parameters are global variables

netlogo.command('set {0} {1}'.format(name, param\_values[run,i]))

netlogo.command('setup')

#Run for 100 ticks and return the number of sheep and wolf agents at each time step

counts = netlogo.repeat\_report(['count sheep','count wolves'], 100)

#For each run, save the mean value of the agent counts over time

results.loc[run, 'Avg. sheep'] = counts['count sheep'].values.mean()

results.loc[run, 'Avg. wolves'] = counts['count wolves'].values.mean()

We can then proceed with the analysis, first using a histogram to visualize output distributions for each outcome.

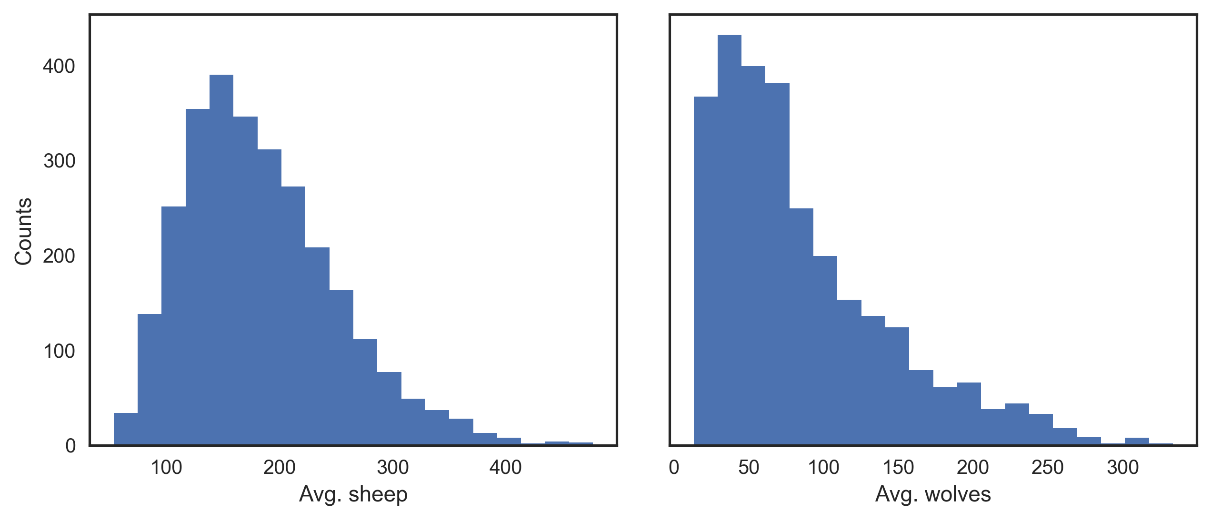


Figure 5: Output distributions for the average number of sheep agents (left) and wolf agents (right) over 100 ticks

Bivariate scatter plots can be useful to visualize relationships between each input parameter and the outputs. Taking the outcome for the average sheep count as an example, we obtain the following, using the scipy Python library to calculate the Pearson correlation coefficient (r) for each parameter:

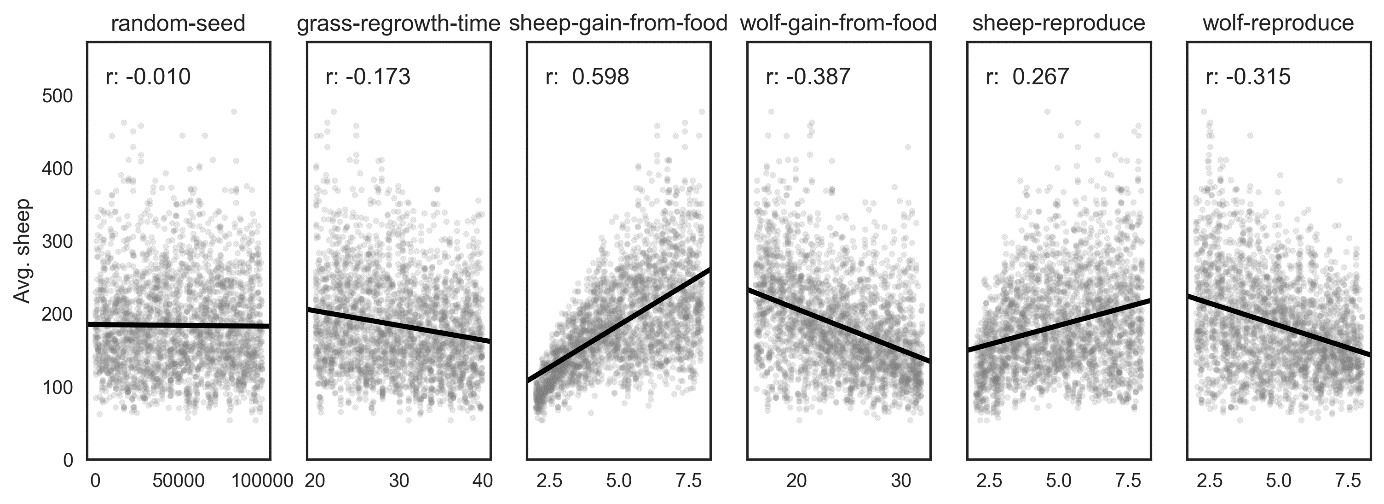


Figure 6: Scatter plots for the average number of sheep agents as a function of each input parameter

This indicates a positive relationship between the *sheep-gain-from-food* parameter and the mean sheep count, and negative relationships for the *wolf-gain-from-food* and *wolf-reproduce* parameters.

We can then use SALib to calculate first-order (S1), second-order (S2) and total (ST) Sobol indices, to estimate each input's contribution to the variance of the average sheep count. By default, 95% confidence intervals are also estimated for each index. The analysis function returns a Python dictionary.

Si = sobol.analyze(problem, results['Avg. sheep'].values, calc\_second\_order=True, print\_to\_console=False)

As a simple example, we first visualize the first-order and total indices and their confidence bounds using the default Pandas plotting functions, after converting the dictionary returned by SALib to a dataframe:

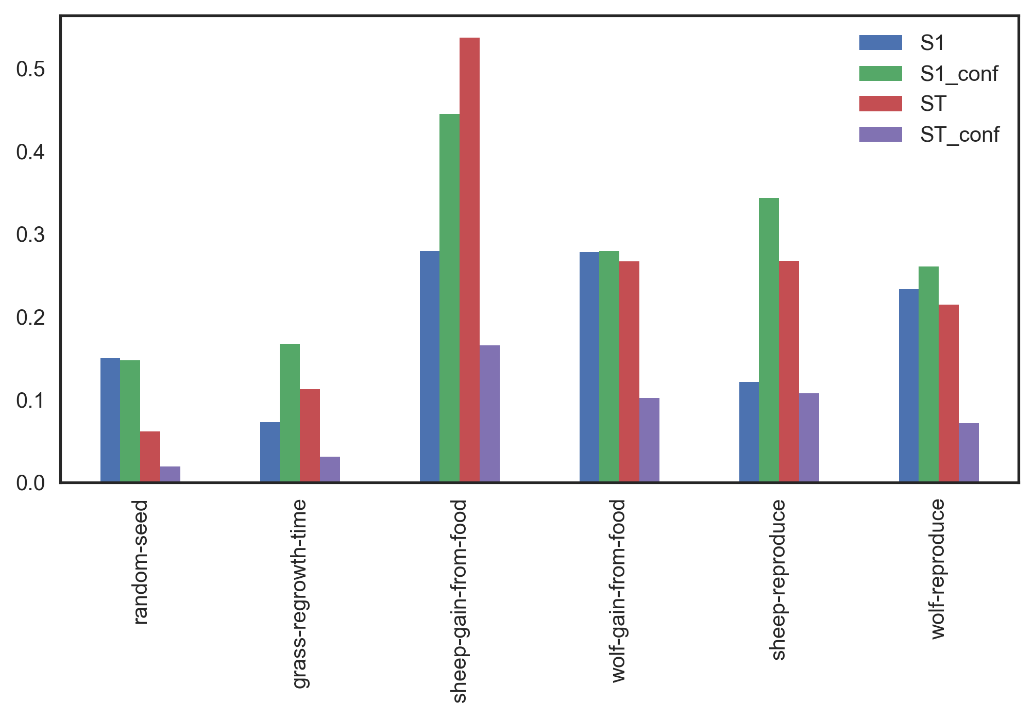


Figure 7: First-order and total Sobol indices with confidence bounds, for the average number of sheep agents

The *sheep-gain-from-food* parameter has the highest ST index, indicating that it contributes over 50% of output variance when accounting for interactions with other parameters. However, the confidence bounds are overly broad due to the small *n* value used for sampling, so that a larger sample would be required for reliable results. For instance, the S1 index is estimated to be larger than ST for the *random-seed* parameter, which is an artifact of the small sample size.

We can use a more sophisticated visualization to include the second-order interactions between inputs:

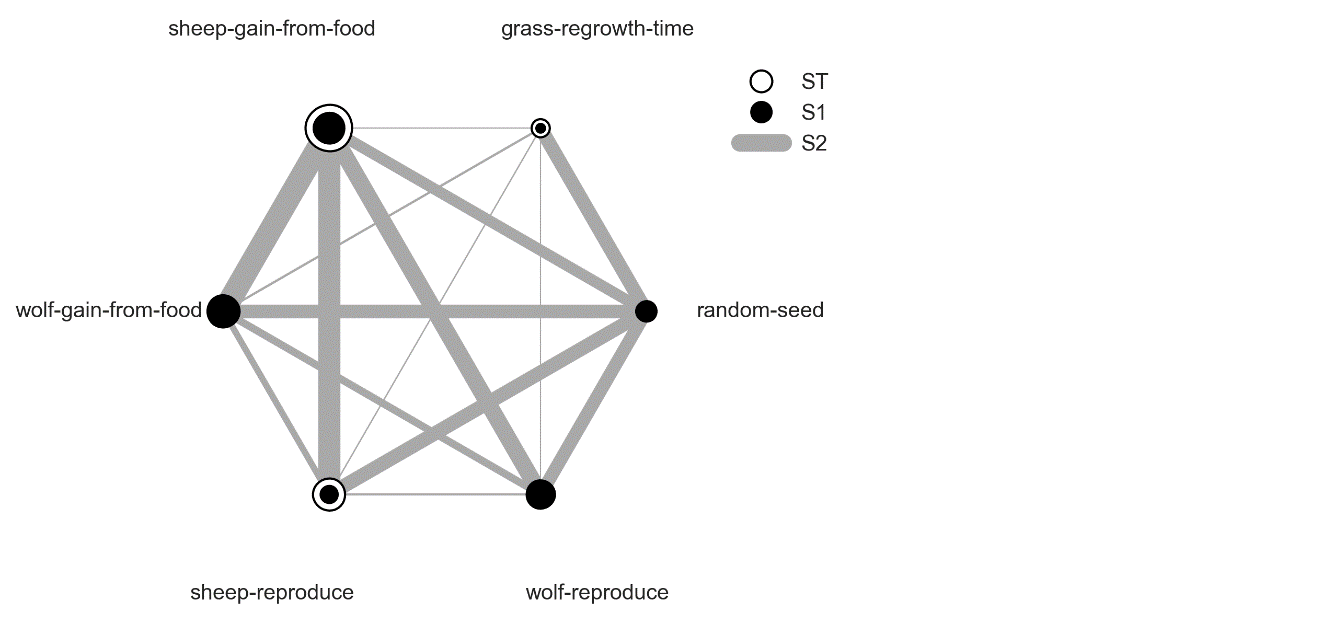


Figure 8: First-order, second-order and total Sobol indices for the average number of sheep agents

In this case, the *sheep-gain-from-food* variable has strong interactions with the *wolf-gain-from-food* and *sheep-reproduce* inputs in particular, as indicated by their thicker connecting lines. The size of the ST and S1 circles correspond to the normalized total and first-order indices.

1. **CONCLUSIONS**

This paper first introduced the pyNetLogo connector, which can be used to interface NetLogo agent-based models with a Python environment. This connector provides basic functionalities similar to the RNetLogo package in R (Thiele et al., 2012b), which were illustrated by controlling one of NetLogo’s sample models from an IPython Notebook environment.

The use of the Python language also addresses issues with the coupled analysis of the models, by enabling the straightforward integration of the simulation architecture with different packages available in Python.

**ACKNOWLEDGMENTS**

This research was supported by the Netherlands Organization for Scientific Research (NWO) under the project Aquifer Thermal Energy Storage Smart Grids (ATES-SG), grant number 408-13-030.

**REFERENCES**

Herman, J., & Usher, W. (2017). SALib: An open-source Python library for Sensitivity Analysis. *The Journal of Open Source Software*, *2*(9). https://doi.org/10.21105/joss.00097

Kwakkel, J. H. (2017). The Exploratory Modeling Workbench: An open source toolkit for exploratory modeling, scenario discovery, and (multi-objective) robust decision making. *Environmental Modelling & Software*, *96*, 239–250.

Thiele, J. C. (2015). *Towards Rigorous Agent-Based Modelling* (Doctoral dissertation). Georg-August-Universität Göttingen, Göttingen. Retrieved from https://ediss.uni-goettingen.de/handle/11858/00-1735-0000-0023-997B-8

Thiele, J. C., Kurth, W., & Grimm, V. (2012a). Agent-Based Modelling: Tools for Linking NetLogo and r. *Journal of Artificial Societies and Social Simulation*, *15*(3), 8. https://doi.org/10.18564/jasss.2018

Thiele, J. C., Kurth, W., & Grimm, V. (2012b). RNETLOGO: an R package for running and exploring individual-based models implemented in NETLOGO. *Methods in Ecology and Evolution*, *3*(3), 480–483. https://doi.org/10.1111/j.2041-210X.2011.00180.x

Wilensky, U. (1999). NetLogo (Version 5.1.0). Center for Connected Learning and Computer-Based Modeling, Northwestern University.