**pyNetLogo: Linking NetLogo with Python**

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

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

**Abstract:** Methods for testing and analyzing agent-based models have drawn increasing attention in the literature, in the context of efforts to establish standard frameworks for the development and documentation of models. This process can benefit from the use of established software environments for data analysis and visualization. For instance, the popular NetLogo agent-based modelling software can be interfaced with Mathematica and R, letting modellers use the advanced analysis capabilities available in these programming languages. To extend these capabilities to an additional user base, this paper presents the pyNetLogo connector, which allows NetLogo to be controlled from the Python general-purpose programming language. Given Python’s increasing popularity for scientific computing, this provides additional flexibility for modellers and analysts. PyNetLogo’s features are demonstrated by controlling one of NetLogo’s example models from an interactive Python environment, then performing a global sensitivity analysis with parallel processing.

***Keywords****:* agent-based modelling; NetLogo; Python

**INTRODUCTION**

Agent-based models (ABMs) are a well-established method for the study of complex adaptive systems, in which the interactions of heterogeneous entities yield emergent large-scale behaviors. As such, this approach has been applied across a wide variety of fields such as economics, ecology, or socio-technical systems (e.g. Tesfatsion & Judd, 2006; Grimm & Railsback, 2012; Nikolic, Dam, & Kasmire, 2013).

However, the computational nature of ABMs can make them more difficult to understand and communicate than analytical models (Grimm et al., 2006). Without the use of standard frameworks to structure their analysis and documentation, ABMs may yield *ad hoc*, poorly reproducible results (Thiele, 2015). Different initiatives are attempting to address this gap, such as the ODD and TRACE protocols for documentation (Grimm et al., 2006; Schmolke, Thorbek, DeAngelis, & Grimm, 2010).

In practice, these documentation protocols are easier to apply when supported by suitable computational tools -- for instance to generate experimental designs for uncertain inputs, visualize output data, or apply standard statistical methods. While many agent-based modelling platforms include basic analysis tools, these are typically not sufficient to meet the requirements of a comprehensive analysis and documentation process. Conversely, using standalone analysis software to process input and output data files can quickly become unwieldy for complex models – making the analysis workflow more difficult to reproduce.

The literature therefore presents different connectors to directly interface agent-based modelling software with analysis environments. In particular, the popular open-source NetLogo modelling software can be linked at runtime with Mathematica (Bakshy & Wilensky, 2007) and R (Thiele, Kurth, & Grimm, 2012), which allows modellers to use the comprehensive analysis and visualization functionalities available in these programming languages.

As a complement to these connectors, this work introduces the pyNetLogo library, which can be used to control NetLogo through the Python programming language. Python is a general-purpose language which is consistently ranked as one the five most popular languages on the TIOBE Programming Community index (TIOBE, 2017); it is increasingly used for scientific computing, and offers a variety of libraries which can support ABM development and testing. It should be emphasized that pyNetLogo is not intended as a replacement for the existing R and Mathematica connectors, or as a comment on the suitability of these various environments for ABM analysis. However, given the popularity of the Python language, pyNetLogo extends the benefits of a specialized analysis environment to a broader audience.

The following section of this paper describes the different software platforms used in this work. A software implementation section then introduces pyNetLogo and its key features, and illustrates these mechanisms for a simple predator-prey model. As an example of the analysis workflow which is enabled by pyNetLogo, this model is controlled interactively from a Python environment, then tested using a global sensitivity analysis.

**SOFTWARE DESCRIPTION**

**NetLogo**

NetLogo (Wilensky, 1999) is an open-source environment for the design and testing of agent-based models. While NetLogo was initially intended as an educational tool, its ease of use, robust performance and active user community have made it a pragmatic choice for a wide range of research applications (see e.g. Kravari & Bassiliades, 2015; Railsback, Lytinen, & Jackson, 2006 for comparisons of common ABM platforms). It has therefore established itself as a leading platform for agent-based modelling (Thiele, 2015).

NetLogo 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. In addition to connectors for Mathematica and R, different extension modules are available, for instance to allow an interface with GIS datasets.

**Python**

Python is a widely used high-level, general-purpose open source programming language that supports various programming paradigms. Python places a strong emphasis on code readability and code expressiveness. A large collection of libraries for many typical programming tasks is readily available. Python is increasingly popular for scientific computing purposes due to the rapidly expanding scientific computing ecosystem available for Python.

This ecosystem includes NumPy (Walt, Colbert, & Varoquaux, 2011) and pandas (McKinney, 2010) for data manipulation, SciPy (Jones, Oliphant, Peterson, & others, 2001) for general numerical tasks, Matplotlib (Hunter, 2007) for plotting and visualization, and IPython (Pérez & Granger, 2007) for interactive analysis. These libraries are pre-packaged in several scientific distributions for Python, such as Continuum Anaconda. Additional libraries can be installed through standard package managers such as *pip* and *conda*.

Python is often used as a “glue” language, meaning that it connects pieces of software written in different languages together into a bigger application. For instance, the JPype library (Menard & Nell, 2014) can be used to access Java class libraries through interfacing the Python interpreter and the Java Virtual Machine. PyNetlogo therefore relies on JPype for interacting with NetLogo.

The pyNetLogo modules used in this work are available at <https://github.com/quaquel/pyNetLogo>, along with interactive IPython/Jupyter notebooks which replicate the analysis presented in the next section. These modules have been tested with Continuum Anaconda 2.7 and 3.6. Using this distribution, the modules require the additional installation of JPype through the *conda* package manager. 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.

**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. This functionality is then extended to illustrate a typical model analysis workflow, using the SALib Python library (Herman & Usher, 2017) to perform a global sensitivity analysis.

**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 the Java Native Interface (JNI). An instance of 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 Jupyter notebook) and a NetLogo model at runtime, with appropriate data type conversions between the two environments. The connector currently supports both NetLogo 5.x and NetLogo 6.0.

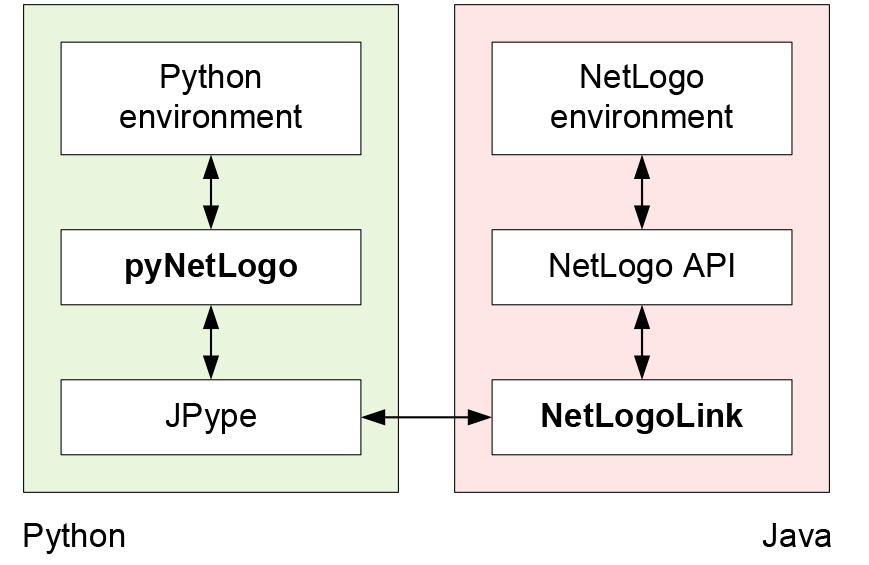


Figure 1: Interactions between Python and NetLogo

The table below summarizes the basic methods available in pyNetLogo. These 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 et al., 2012).

Table 1: Basic PyNetLogo methods

|  |  |  |  |
| --- | --- | --- | --- |
| **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 the functionality of pyNetLogo, a simple example follows below, using the Wolf Sheep Predation model which is included in the NetLogo example library. The Jupyter notebook attached to this paper demonstrates the key functions of the PyNetLogo connector in more detail using this model.

First, a link to NetLogo is instantiated. This involves starting a Java VM, followed by starting NetLogo. All interactions with NetLogo are handled by an instance of the NetLogoLink class. Next, we can load a model using the *load\_model* method, followed by basic commands to set up the model and run it for 100 ticks. The *report* method is then used to return Numpy arrays to Python, 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')

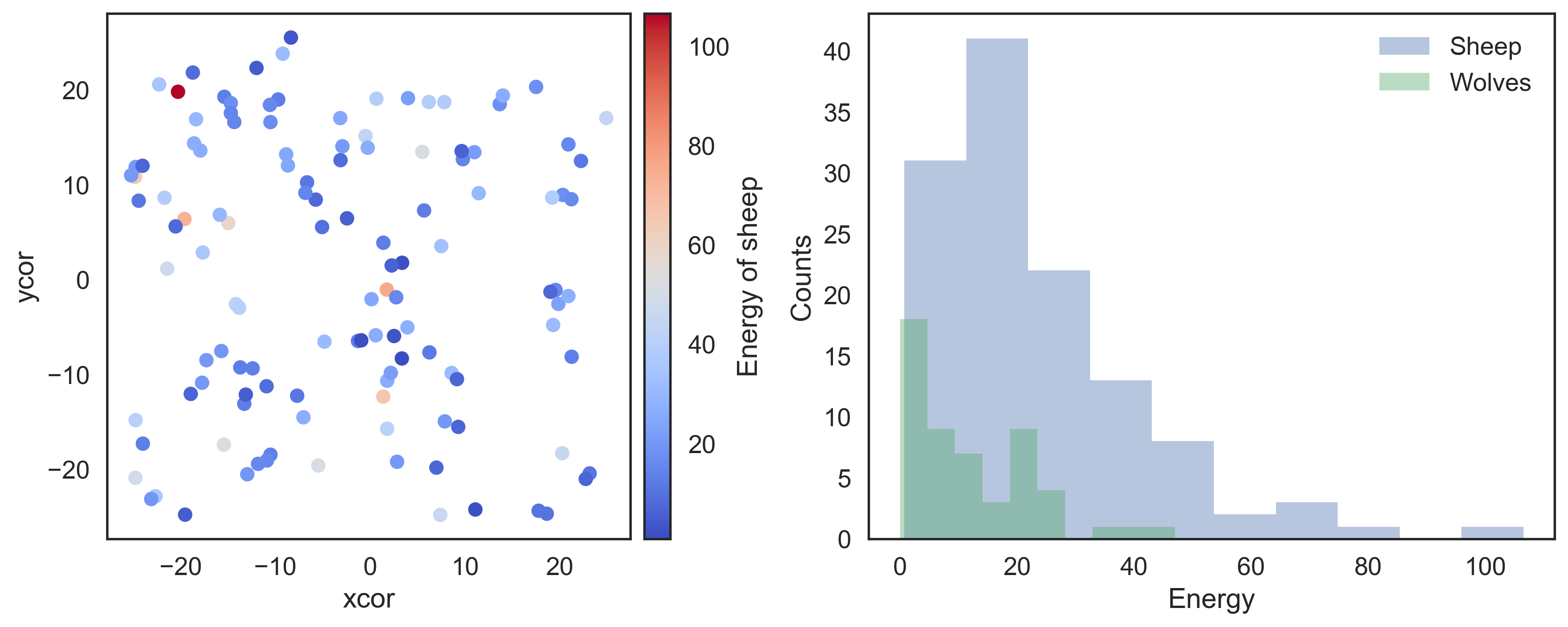


Figure 2: Basic plots generated in Python: agent coordinates (left); distribution of energy attribute across agents (right)

Building on this functionality, the *repeat\_report* method 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)

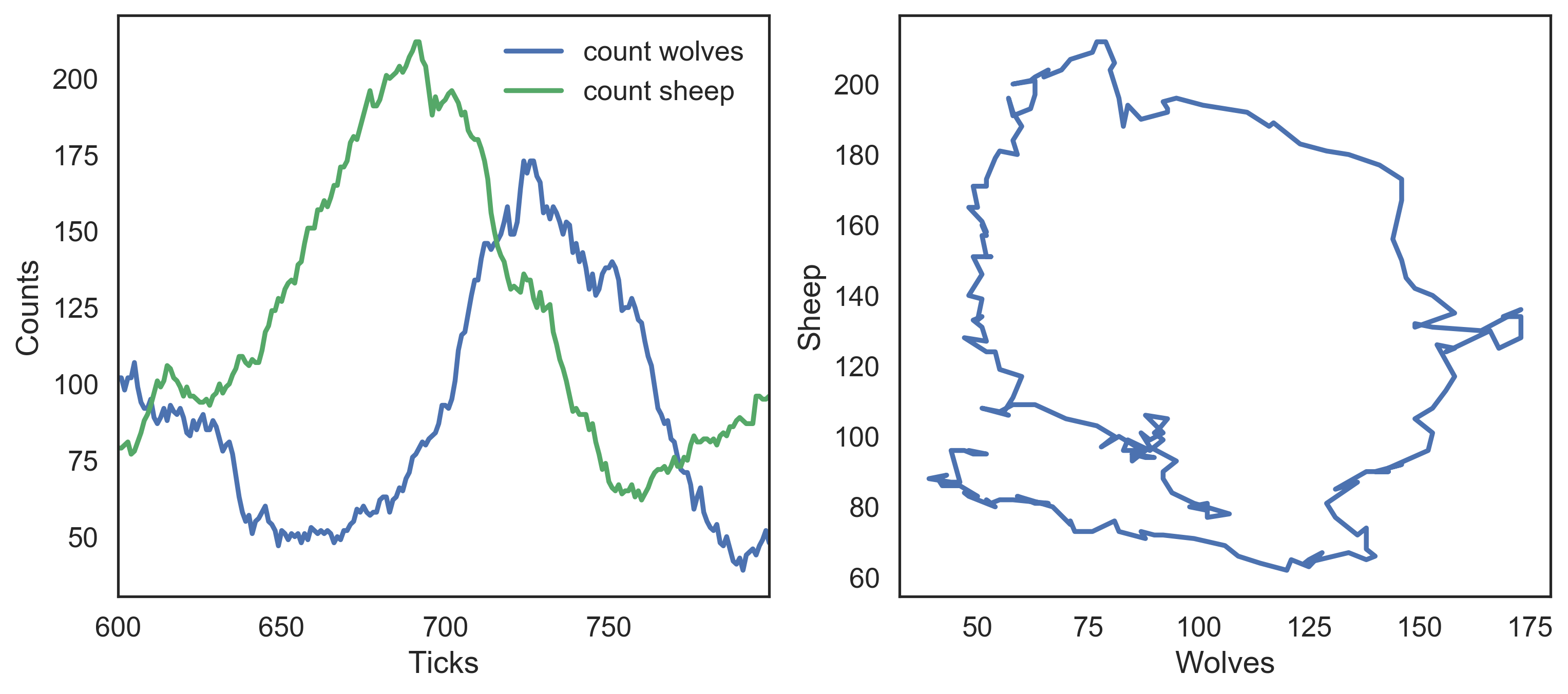


Figure 3: Python plots using *repeat\_report* function: number of agents as a function of time (left); number of sheep agents as a function of wolf agents (right)

In addition to these reporting methods, the *patch\_report* method 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.

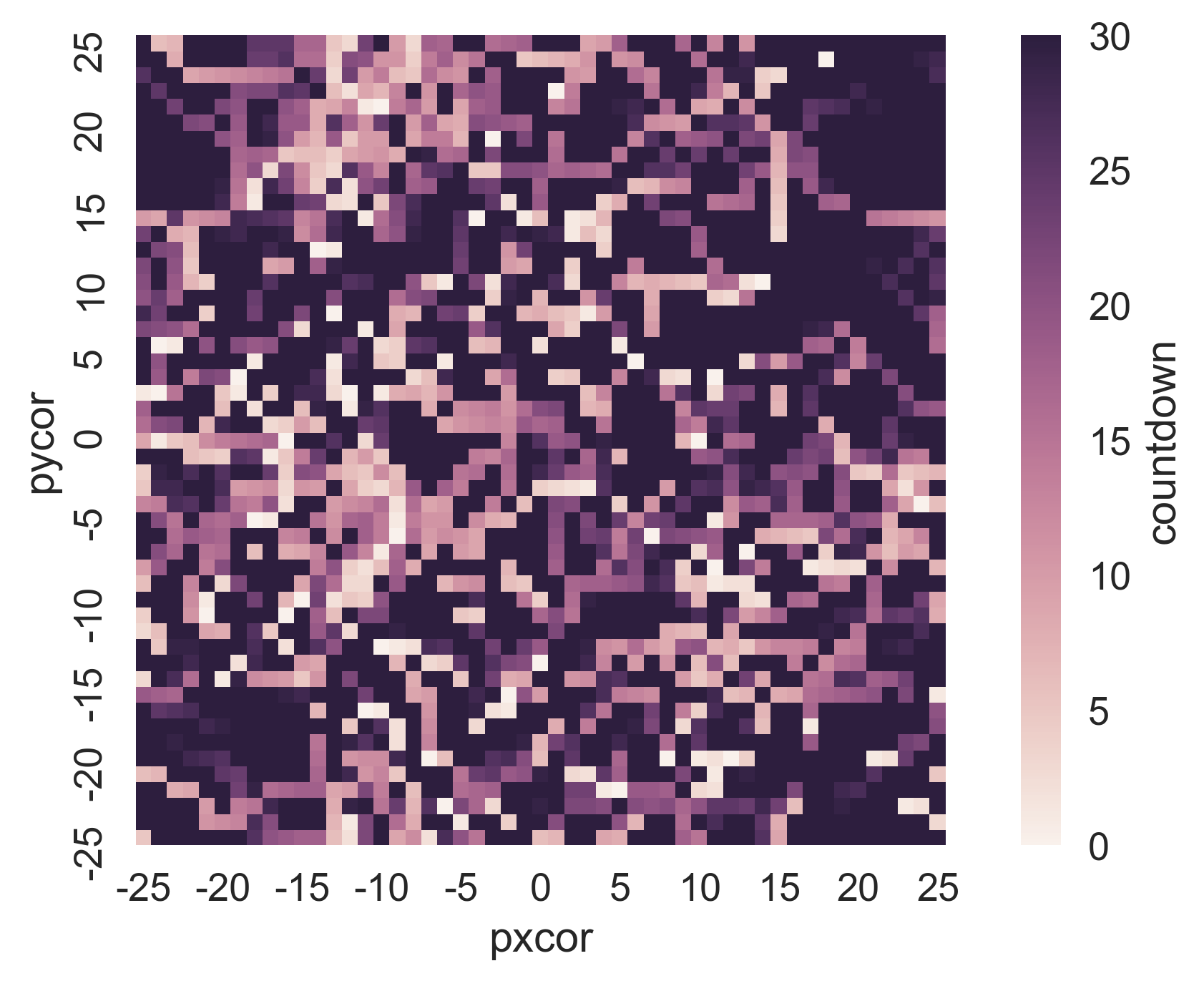


Figure 4: Python plot using *patch\_report* function: distribution of the *countdown* patch attribute across the NetLogo environment

**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 for a global sensitivity analysis (GSA) on the Wolf Sheep Predation model.

By contrast to “one-at-a-time” sensitivity analysis, which evaluates the response of a model to changes in individual parameters, GSA aims to capture the behavior of the model across the full domain of uncertain inputs (see e.g. Saltelli et al., 2008 for a comprehensive overview). This is especially useful for models in which interactions between parameters can be expected to be significant; a simple example of GSA would be generating a Monte Carlo sample of all uncertain inputs, then applying a multiple linear regression to the model output. For more complex, non-linear models, variance-based approaches such as Sobol indices (Sobol, 1993) can accurately capture each parameter’s contribution to the variance of model output. However, this requires specific techniques for input sampling and output analysis.

In this context, the SALib library provides sampling and analysis modules for methods including Sobol indices, Morris elementary effects (Morris, 1991), and derivative-based global sensitivity measures (Sobol’ & Kucherenko, 2009). Integrating these methods within a NetLogo workflow significantly extends the functionality of NetLogo’s BehaviorSpace tool, which has limited sampling options. This example will demonstrate the use of SALib for a Sobol analysis, which provides a reliable (albeit computationally expensive) estimation of input importances.

SALib relies on a problem definition dictionary (i.e., a key-value map), 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 then 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. The first-order and total indices respectively estimate the fractional contribution of each input to output variance on its own, and inclusive of interactions with other inputs; second-order indices estimate pairwise interactions between the inputs.

For this example, we use *n* = 200, for a total of 2800 experiments. The next subsection will demonstrate the use of IPyparallel to parallelize the simulations and reduce runtime.

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 (Kwakkel, 2017).

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 as shown in Figure 5.

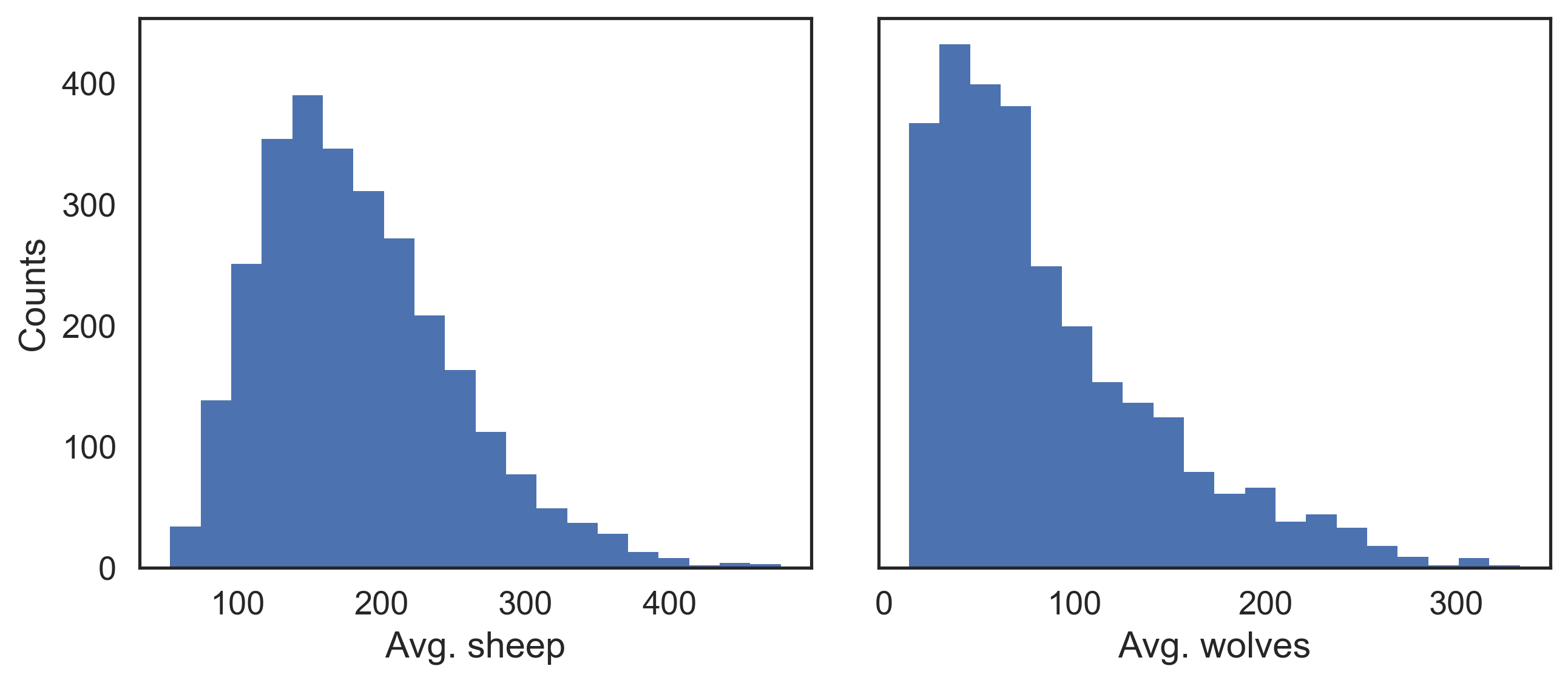


Figure 5: Output distributions for the average number of sheep agents (left) and wolf agents (right) over 100 ticks

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 Figure 6, using Scipy to calculate the Pearson correlation coefficient (r) for each parameter. This indicates a positive correlation between the *sheep-gain-from-food* parameter and the mean sheep count, and negative correlations for the *wolf-gain-from-food* and *wolf-reproduce* parameters.

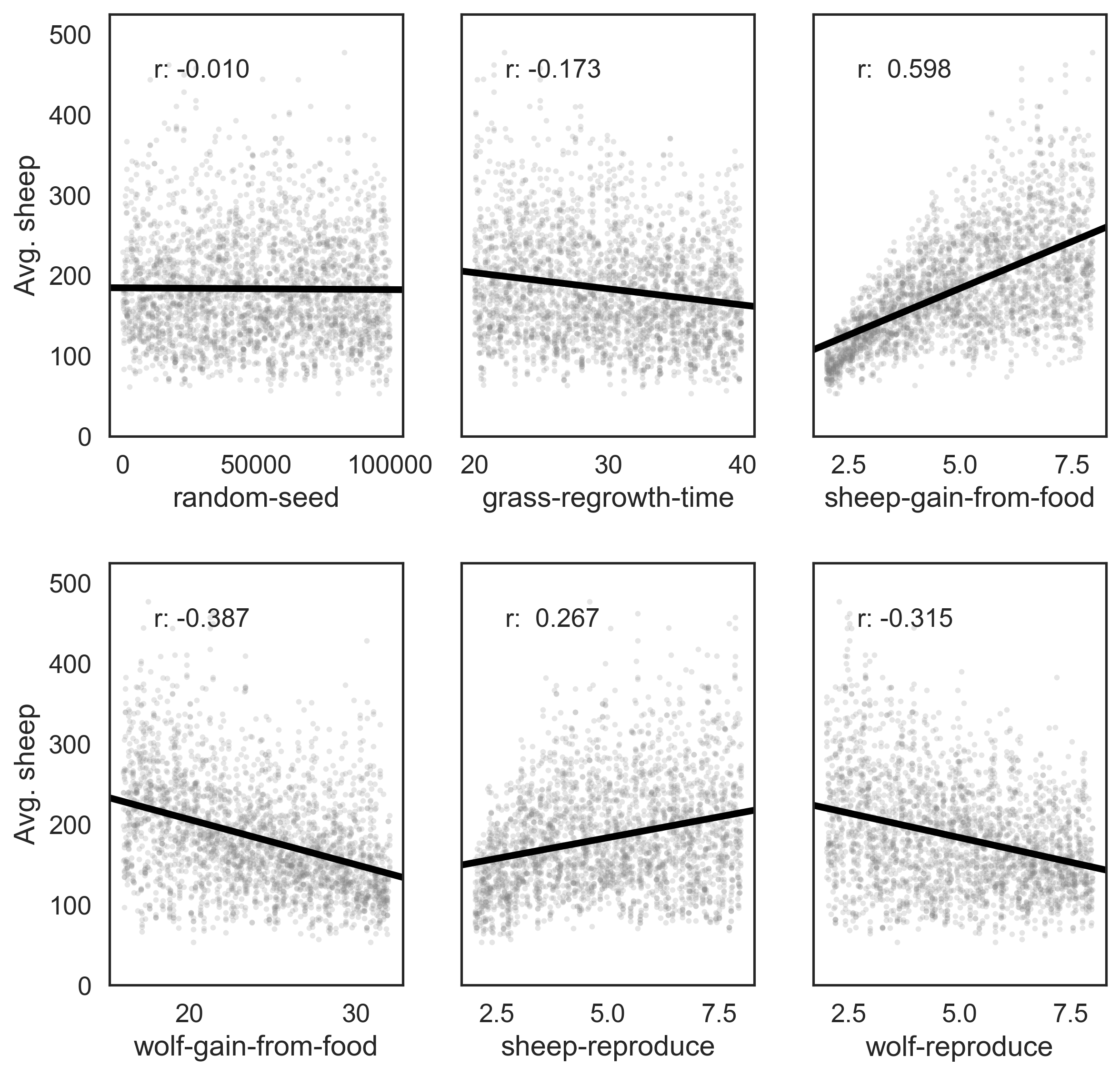


Figure 6: Scatter plots with linear trendlines for the average number of sheep agents as a function of each input parameter

We can 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)

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:

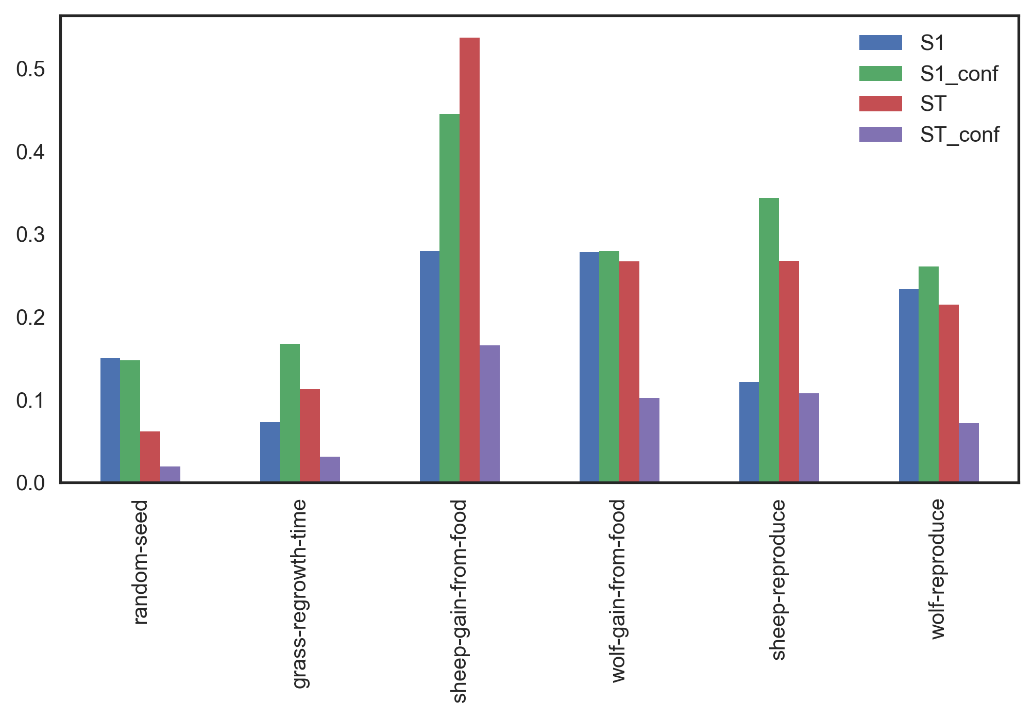


Figure 7: First-order and total Sobol indices with confidence bounds, for the average number of sheep agents

The *sheep-gain-from-food* parameter has the highest S1 and ST indices, indicating that it contributes roughly 30% of output variance on its own, and over 50% when accounting for interactions with other parameters. However, the confidence bounds are overly broad due to the small *n* value used for sampling (i.e. 200), indicating 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 pairwise interactions between inputs:

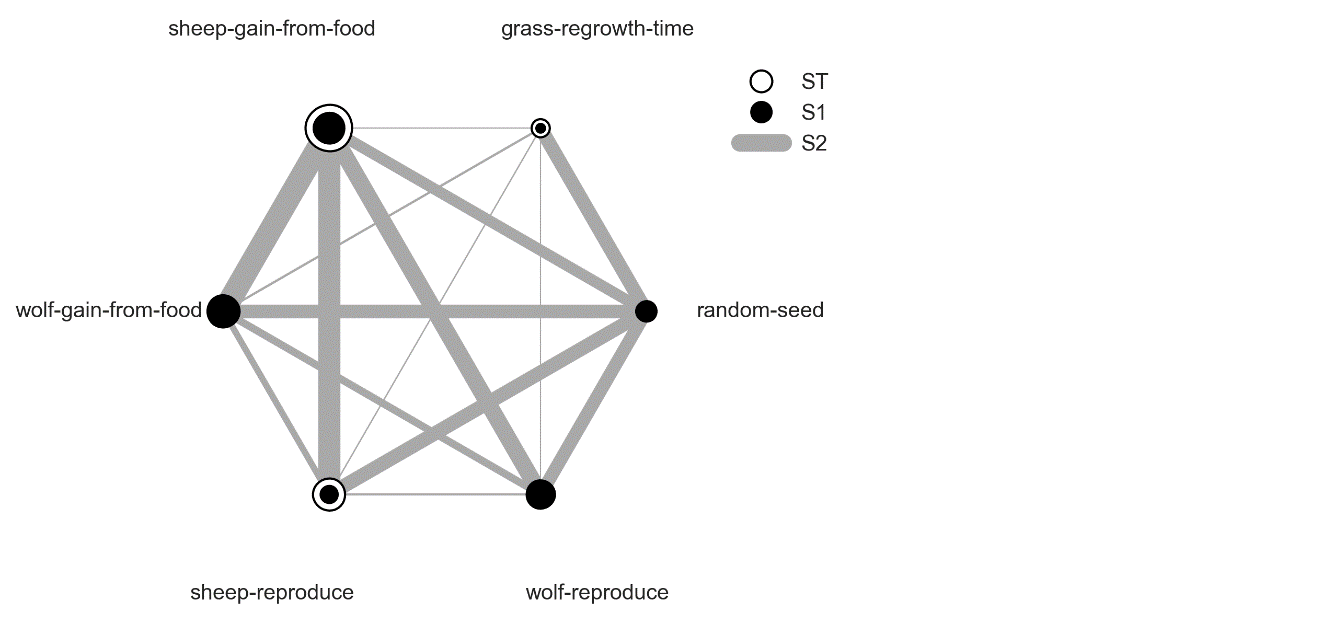


Figure 8: First-order, second-order and total Sobol indices for the average number of sheep agents

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.

**Using Ipyparallel for parallel simulation**

Ipyparallel is a standalone package (available through the *conda* package manager) which can be used to interactively run parallel tasks from IPython on a single PC, but also on multiple computers. On machines with multiple cores, this can significantly improve performance: for instance, the multiple simulations required for a sensitivity analysis are easy to run in parallel. This subsection will repeat the SALib global sensitivity analysis presented in the previous subsection, this time using IPyparallel to distribute the simulations across multiple cores on a single computer.

Ipyparallel first requires starting a controller and multiple engines, which can be done from a terminal or command prompt with the following:

ipcluster start -n 4

The optional *-n* argument specifies the number of processes to start (4 in this case). By default, the number of logical processor cores will be used.

Next, we can connect the interactive notebook to the cluster by instantiating a client (within a notebook), and checking that *client.ids* returns a list of 4 available engines.

import ipyparallel

client = ipyparallel.Client()

print(client.ids)

After defining the SALib problem dictionary and input sample as in the previous subsection, we can then set up the engines so that they can run the simulations, using a "direct view" that accesses all engines. We first need to change the working directories to import pyNetLogo on the engines (assuming the pyNetLogo module is located in the same directory as the interactive notebook, rather than being on the Python path). We can then also ensure the engines have access to the SALib problem definition dictionary.

direct\_view = client[:]

import os

# Push the current working directory of the notebook to a

# "cwd" variable on the engines that can be accessed later

direct\_view.push(dict(cwd=os.getcwd()))

# Push the "problem" variable from the notebook to a

# corresponding variable on the engines

direct\_view.push(dict(problem=problem))

The *%%px* command can be added to a notebook cell to run it in parallel on each of the engines. Here the code first involves some imports and a change of the working directory. We then start a link to NetLogo, and load the example model (assumed to be in the working directory) on each of the engines.

%%px

import os

os.chdir(cwd)

import jpype

import pyNetLogo

import pandas as pd

import numpy as np

netlogo = pyNetLogo.NetLogoLink(gui=False)

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

We can then use the IPyparallel map functionality to run the sampled experiments, now using a "load balanced" view to automatically handle the scheduling and distribution of the simulations across the engines. This is for instance useful when simulations may take different amounts of time.

We slightly modify the simulation code used previously, setting up a simulation function that takes a single experiment (i.e. a vector of input parameters) as an argument, and returns the outcomes of interest in a pandas Series.

def simulation(experiment):

#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(experiment[i]))

else:

#Otherwise, assume the input parameters are global variables

netlogo.command('set {0} {1}'.format(name, experiment[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)

results = pd.Series([counts['count sheep'].values.mean(),

counts['count wolves'].values.mean()],

index=['Avg. sheep', 'Avg. wolves'])

return results

We then create a load balanced view and run the simulation with the *map\_sync()* method, which takes a function and a Python sequence as arguments, applies the function to each element of the sequence, and returns results once all computations are finished. In this case, we pass the simulation function and the array of experiments (*param\_values*), so that the function will be executed for each row of the array.

The DataFrame constructor is used to immediately build a DataFrame from the results (which are returned as a list of Series).

lview = client.load\_balanced\_view()

results = pd.DataFrame(lview.map\_sync(simulation, param\_values))

We can then proceed with the analysis as in the previous subsection. Figure 9 compares the runtimes obtained with IPyparallel and a sequential simulation (using an Intel i7-6700HQ CPU) for 2800 experiments. The elapsed parallel runtime is approximately one-third of the sequential runtime; given that we were using 4 engines, this is slightly more than could be expected from a perfectly parallel computation, due to the overhead involved in data exchanges, etc.

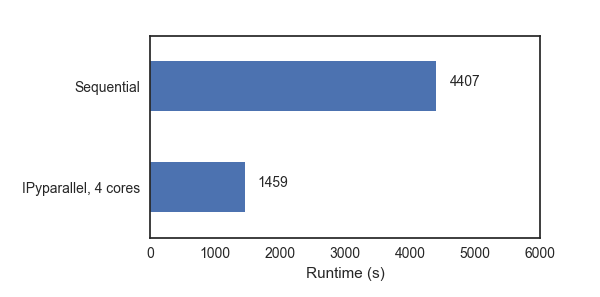


Figure 9: Comparison of runtimes for sensitivity analysis (2800 total experiments), using sequential and parallel simulations

**CONCLUSIONS**

The analysis and communication of agent-based models can benefit from the comprehensive analysis features which are available in specialized software environments. To this end, this paper first introduced the pyNetLogo connector, which interfaces the NetLogo agent-based modelling software with a Python environment. This connector provides basic functionalities similar to the RNetLogo package in R (Thiele et al., 2012), which were illustrated by controlling one of NetLogo’s sample models from an interactive Jupyter notebook. As an example of the more complex analyses which are enabled by a Python interface, the SALib Python library was then used for a Sobol variance-based global sensitivity analysis of the model. This analysis was performed using sequential simulations, then parallelized for improved performance using the IPyparallel library.

The current implementation of pyNetLogo relies on a Java Native Interface (JNI) through the JPype library, which allows Java classes (and thus NetLogo) to be called from Python. However, this does not support a bidirectional linkage through which a NetLogo model could also directly execute Python code. This limitation could potentially be solved by using the jpy library[[1]](#footnote-1) as a replacement for JPype.

**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**

Bakshy, E., & Wilensky, U. (2007). *NetLogo-Mathematica Link*. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL. Retrieved from http://ccl.northwestern.edu/netlogo/mathematica.html

Grimm, V., Berger, U., Bastiansen, F., Eliassen, S., Ginot, V., Giske, J., … DeAngelis, D. L. (2006). A standard protocol for describing individual-based and agent-based models. *Ecological Modelling*, *198*(1), 115–126. https://doi.org/10.1016/j.ecolmodel.2006.04.023

Grimm, V., & Railsback, S. F. (2012). Pattern-oriented modelling: a “multi-scope” for predictive systems ecology. *Philosophical Transactions of the Royal Society of London B: Biological Sciences*, *367*(1586), 298–310. https://doi.org/10.1098/rstb.2011.0180

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

Hunter, J. D. (2007). Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering*, *9*(3), 90–95. https://doi.org/10.1109/MCSE.2007.55

Jones, E., Oliphant, T., Peterson, P., & others. (2001). *SciPy: Open source scientific tools for Python*. Retrieved from http://www.scipy.org/

Kravari, K., & Bassiliades, N. (2015). A Survey of Agent Platforms. *Journal of Artificial Societies and Social Simulation*, *18*(1), 11.

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.

McKinney, W. (2010). Data Structures for Statistical Computing in Python. In S. van der Walt & J. Millman (Eds.), *Proceedings of the 9th Python in Science Conference* (pp. 51–56).

Menard, S., & Nell, L. (2014). *Jpype*. Retrieved from https://pypi.python.org/pypi/JPype1

Morris, M. D. (1991). Factorial Sampling Plans for Preliminary Computational Experiments. *Technometrics*, *33*(2), 161–174. https://doi.org/10.1080/00401706.1991.10484804

Nikolic, I., Dam, K. H. van, & Kasmire, J. (2013). Practice. In K. H. van Dam, I. Nikolic, & Z. Lukszo (Eds.), *Agent-Based Modelling of Socio-Technical Systems* (pp. 73–137). Springer Netherlands. Retrieved from http://link.springer.com/chapter/10.1007/978-94-007-4933-7\_3

Pérez, F., & Granger, B. E. (2007). IPython: A System for Interactive Scientific Computing. *Computing in Science & Engineering*, *9*(3), 21–29. https://doi.org/10.1109/MCSE.2007.53

Railsback, S. F., Lytinen, S. L., & Jackson, S. K. (2006). Agent-based Simulation Platforms: Review and Development Recommendations. *SIMULATION*, *82*(9), 609–623. https://doi.org/10.1177/0037549706073695

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

Schmolke, A., Thorbek, P., DeAngelis, D. L., & Grimm, V. (2010). Ecological models supporting environmental decision making: a strategy for the future. *Trends in Ecology & Evolution*, *25*(8), 479–486. https://doi.org/10.1016/j.tree.2010.05.001

Sobol, I. M. (1993). Sensitivity estimates for nonlinear mathematical models. *Mathematical Modelling and Computational Experiments*, *1*(4), 407–414.

Sobol’, I. M., & Kucherenko, S. (2009). Derivative based global sensitivity measures and their link with global sensitivity indices. *Mathematics and Computers in Simulation*, *79*(10), 3009–3017. https://doi.org/10.1016/j.matcom.2009.01.023

Tesfatsion, L., & Judd, K. L. (2006). Preface. In L. T. and K. L. Judd (Ed.), *Handbook of Computational Economics* (Vol. 2, pp. xi–xv). Elsevier. Retrieved from http://www.sciencedirect.com/science/article/pii/S1574002105020393

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. (2012). 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

TIOBE. (2017). TIOBE Programming Community index. Retrieved from https://www.tiobe.com/tiobe-index/

Walt, S. van der, Colbert, S. C., & Varoquaux, G. (2011). The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science Engineering*, *13*(2), 22–30. https://doi.org/10.1109/MCSE.2011.37

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

1. https://github.com/bcdev/jpy [↑](#footnote-ref-1)