Utilizing Python for Agent-Based Modeling: The Mesa Framework

Jackie Kazil[1,2(✉)], David Masad[1], and Andrew Crooks[1]

[1] George Mason University, Fairfax, VA 22020, USA
{jkazil,dmasad,acrooks2}@gmu.edu
[2] Rebellion Defense, Washington DC 20001, USA

Abstract. Mesa is an agent-based modeling framework written in Python. Originally started in 2013, it was created to be the go-to tool in for researchers wishing to build agent-based models with Python. Within this paper we present Mesa's design goals, along with its underlying architecture. This includes its core components: 1) the model (Model, Agent, Schedule, and Space), 2) analysis (Data Collector and Batch Runner) and the visualization (Visualization Server and Visualization Browser Page). We then discuss how agent-based models can be created in Mesa. This is followed by a discussion of applications and extensions by other researchers to demonstrate how Mesa design is decoupled and extensible and thus creating the opportunity for a larger decentralized ecosystem of packages that people can share and reuse for their own needs. Finally, the paper concludes with a summary and discussion of future development areas for Mesa.

Keywords: Agent-based modeling · Python · Framework · Complex systems

1 Introduction

Agent-based modeling (ABM) is a way to simulate the behaviors and interactions of many autonomous entities, or agents, over time. Such a methodology has many advantages over other mathematical approaches to studying complex systems including the ability to capture the temporal paths, the spatial paths, and their end states as well as the ability to study the dynamics of a system and the impact of individual actions and reactions (Crooks et al. 2019). One of the most novel aspects of ABM is its ability to explore "transient, non-equilibrium, non-stationary behavior" of a system and along with that ability, to computationally trace it (Epstein and Axtell 1996). ABMs have seen tremendous growth over the last 20 years (Crooks et al. 2019), leading a growth of ABM frameworks (which we further discuss below). However, there also was a void. There was no framework for easily building a model in Python, as well as no ability to serve a model over Hypertext Transfer Protocol (HTTP) which takes advantage of modern browser-based technologies. In response to this, we created Mesa[1], an open source

[1] We chose the name Mesa for three weak reasons: (1) It sounded like Mason, (2) It evoked the mesas around Santa Fe, the location of the Santa Fe Institute and home to much complexity research, and (3) It was a short and memorable name that was available on the Python Package Index (PyPI).

© Springer Nature Switzerland AG 2020
R. Thomson et al. (Eds.): SBP-BRiMS 2020, LNCS 12268, pp. 308–317, 2020.
https://doi.org/10.1007/978-3-030-61255-9_30

framework for creating agent-based models in Python. Mesa was released under Apache 2 ("Apache License, Version 2.0" 2004) in order to be flexible both for academia and private sector use. Mesa can be installed directly by "pip install mesa" or by downloading it from Github: https://github.com/projectmesa/mesa.

To create and study complex systems from the bottom up, a number of open source frameworks have been developed (see Crooks et al. 2019 for a review). The most widely used one being NetLogo (Wilensky 1999) which made agent-based models accessible for non-professional programmers. In an analysis of comses.net (formally known as openabm.org) which allows researchers to share their agent-based models McCabe (2016) found that approximately 60% of the models created were NetLogo. However, NetLogo does not scale (i.e., in terms of numbers of agents) (North et al. 2013), nor does it have the same execution speed of Mason (Luke et al. 2005) as noted by Railsback et al. (2006). Despite the plethora of frameworks to build agent-based models until the creation of Mesa there was not one utilizing Python. In the remainder of this paper we discuss Mesa's design goals (Sect. 1.1) before introducing its architecture and usage in Sect. 2. In, Sect. 3 we provide insight into applications and extensions to Mesa from its user community, while Sect. 4 provides a summary of the paper and outlines future development directions.

1.1 Mesa's Design Goals

Mesa's design goals go beyond building agent-based models rapidly in Python, headless or displaying them in the browser. Mesa has a permissive license (i.e., Apache 2) and was built to be accessible to a wide range of users, similar to NetLogo, but extensible, like Mason. Similar to other frameworks, Mesa is focused on the core functionality that is needed when building agent-based models (e.g., reusable objects, scheduling, and graphical user interfaces), thus allowing modelers to focus on the development of models rather than parts of the simulation that are not content specific (e.g., the display of the model). Mesa is intended to be extensible, which allows users to easily develop and share their own components through open source ecosystems such as Gitlab and Github. However, it should be noted that Mesa is not intended to be an all-encompassing toolbox; we believe specialized components should be offered as separate packages, similar to how GeoMason (Sullivan et al. 2010) is an extension of Mason. Lastly, Mesa was built to take advantage of the Python and JavaScript ecosystems. Our rationale for using Python was that it is a rapidly growing programing language used throughout academia and industry (Robinson 2017). Furthermore, it seamlessly integrates with popular data science tools such as Jupyter notebooks and Pandas for ease of analysis of data. For example, the continuous space module in Mesa uses the NumPy arrays in the background to speed up neighborhood lookups.

Generally, the approach for Mesa has followed the well-known programming philosophy: "Make it work. Make it right. Make it fast." Mesa was not intended to be a high-performance tool when it was first designed, although over time, some contributions have been made to improve performance, such as the addition of a multi-processing batch runner, which allows for multiple cores to be used to run multiple simulations at the same time. Moreover, we have prioritized accessibility over performance in the building of Mesa. This decision was a part of exercising core Python principles such as simplicity

and reliability. In addition to this, Mesa was written to run on a single core. Multicore processing is was not an initial priority, because of its complexity, but we haven't ruled this out as possible future development. In comparison to other well adopted ABM tools, Mesa has two major advantages. The first is that it is written in Python, which is accessible and integrates well with a many of open data science tools (e.g., Jupyter notebooks and Pandas). The second is the architecture of Mesa, which allows users to easily replace components, which is what we turn to next.

2 Architecture and Usage

Mesa is written in Python and has a front end that takes advantage of front-end browser-based technologies. The underlying structure for how Mesa is laid and designed is influenced by Django (2013), a web framework written in Python. Django's design decouples the models, views, and controller architecture. In a similar fashion, we decoupled the components of Mesa to be easily replaceable and in the case of the model, could be used independent of the other components. There are three major components which make up Mesa from a user perspective. These are the model (Model, Agent, Schedule, and Space), analysis (Data Collector and Batch Runner) and the visualization (Visualization Server and Visualization Browser Page) and the relationship of these components can be seen in Fig. 1.

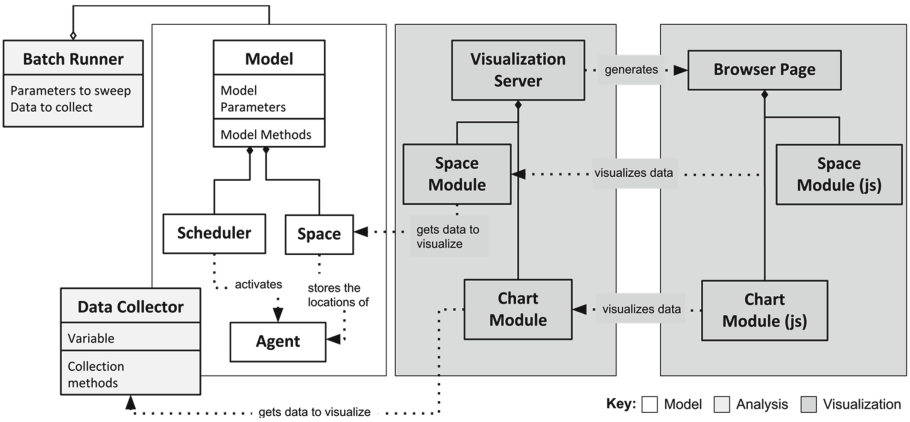


Fig. 1. Mesa model components: model, analysis and visualization.

2.1 The Model

The model is the core of Mesa and the other two components, analysis and visualization would not exist without it. However, the model can stand alone without the other two components. The model as it is referred to here contains the major components of what it means to be an ABM: agents, the space they exist within, and the time and activation

controllers. Separately from what is called "the model", there is a Model class, which is the core class for creating a model. In this class, the user defines the initial state of the model, what happens when the model runs, what occurs each step for the environment, and also the space the agents inhabit. In addition to the model class, there is also the Agent class, which is the object that is subclassed when defining the agents. The default Mesa package comes with predefined spaces (which we discuss further below), which are located in the space module. This module provides the following space types: continuous, single grid, multiple grids, hexagonal grid, or a network grid. The default Mesa package also comes with a time module which contains schedules, which handles agent activation. Due to the complexity and nuances of the space and schedule (i.e., time) modules, we will go into depth into those next.

The Scheduler. Special attention was placed into the functionality of Mesa's scheduler. Generally, speaking, most agent-based models are discrete-event simulations and rarely are they continuous (Masad and Kazil 2015). As a result, the activation order of agents can significantly impact the behavior and results of a simulation (Comer 2014). Mesa takes this into account and offers a variety of methods to implement the activation of agents. In comparison to other ABM frameworks, this is a unique feature to Mesa. Each model step, or "a tick", results in the activation of one or more agents. There are many different approaches to scheduling agent activation including synchronous or simultaneous activation, uniform activation, random activation, random interval activation, and more complex activation regimes (Masad and Kazil 2015). It was Comer's (2014) research and impact of different approaches to scheduling that led to the design of the scheduler in Mesa. The simplest scheduler is in BaseScheduler class which is a uniform activation and was created with the intent to replicate the scheduler in Mason. In addition to the BaseScheduler, there is also the RandomActivation class, which behaviors similar to the scheduler in NetLogo, the SimultaneousActivation class, and the StagedActivation class. More details on these activation schedulers are found in Table 1. To understand the impact of various scheduling routines, see Fig. 2, where we show three different activation schemes (i.e., sequential, random, and simultaneous) for the same step even when the initial model configuration was the same.

Table 1. Activation schedules within Mesa

Activation	Agent activation details
BaseScheduler	Agents are activated one at a time, in the order they were added to the scheduler (i.e., sequential activation)
RandomActivation	Agents are activated one at time, once per step, in random order. Reshuffled every time tick
SimultaneousActivation	Each agent's actions are queued based on the state of the model at the end of the previous step. Then all agents advance at the same time
StagedActivation	Allows agent activation to be divided into several stages instead of a single step. All agents execute one stage before moving on to the next. This scheduler tracks steps and time separately

Fig. 2. An illustration of how different activation schemes impact a model, in this case the Prisoner's Dilemma. Defecting agents are in red and cooperating agents are in blue. Each image is from the same step, but different activation schemes are used. (Color figure online)

Space. While the concept of space is not required for all agent-based models (e.g., the Zero Intelligence Traders model (Gode and Sunder 1993)), an ABM framework would be incomplete without a spatial component. Mesa has three general space definitions: continuous, grid, and networks (which are also common in MASON and Repast). Both the continuous and grid spaces have a method, which allows users to designate whether the space is toroidal or not. All the classes have a similar set of methods which allow the user to get information on the agent's position and location, move the agent, and get information on the agent's neighbors.

Neighborhood identification is handled slightly differently in the general space categories. In the continuous space, neighbors are determined by a defined radius. In the grid space, neighborhoods can be defined as Moore neighborhoods (includes diagonal neighbors) or von Neumann neighborhoods (excludes diagonals), except in the case of a hexagonal grid, which provides access to neighbors on each of the six sides of the hexagon. Lastly, neighbors in networks are provided by adjacent nodes.

Each class of spaces works slightly differently. The ContinuousSpace class, agents have an (x, y) position, while all of the grid classes are discrete spaces. The most basic grid class, which all other grid classes are based off of, represents cells as rectangular spaces. The SingleGrid class limits cells to only have one object per cell, while the MultiGrid class allows for a set of objects per cell. The last extension of the Grid class is the HexGrid class. At the time of this paper, agents placed on the HexGrid grid have an (x, y) position, but it is possible for this class to be extended to offer a (x, y, z) position and offer a 3D-like modeling grid (Patel 2019). Finally, the NetworkGrid has nodes that hold zero or more agents. The NetworkGrid requires a graph object as an argument that is created with the Python library NetworkX (Hagberg et al. 2008). By using NetworkX, Mesa is able to take advantage of all the graph metrics and operations that the NetworkX library provides. It is also possible to create multilayer networks by instantiating multiple graphs. Space would be incomplete without mentioning geospatial models. Similar to the early development of other frameworks (e.g., NetLogo and MASON), the current version of Mesa does not have specific support for geospatial data. However, it allows for importation of text files to create artificial landscapes so in a way it does allow for raster data to be added to models like in Sugarscape. Our rationale for not including GIS support into core Mesa is its dependencies on many third-party packages for importing and exporting data etc. However, this is one area we plan on exploring in the future. If

readers of this paper want to use geospatial data with Mesa, a core contributor known as Corvince, has created a package which offers this functionality entitled Mesa-Geo (https://github.com/Corvince/mesa-geo).

2.2 Model Analysis

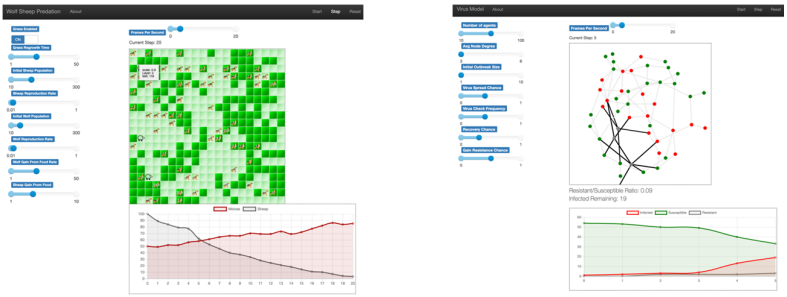
Data Collection. While agent-based models can be interesting to run, it is difficult to gain insights into the model without gathering data and conducting an analysis. To address this issue, Mesa provides the DataCollector class which records, stores, and exports data from the model and agents as well as data that isn't covered in model or agent data abstractions. The DataCollector is initiated with model and agent variables and their respective collection functions. The collector will return the computed value of the model and agent collector at their current state. Data not covered by model or agents can be stored by passing a dictionary object for a table row. One use case might be to log model events or state as the model progresses. These types of data points do not occur at regular intervals. The DataCollector makes data exports easy as well, by using dictionaries and lists to store the data, it makes it easy to export to common data formats such as Pandas DataFrames, JSON, or CSV. By doing this we can take the data out of Mesa and into a popular browser-based workbook-like tool called Jupyter Notebooks, which is used by data scientists for analysis and storytelling with data.

Batch Runner. While it is possible to collect data for individual model runs, this is not the most efficient use of time. Researchers carry out parameter sweeps in order to get a more representative picture of the potential outcomes of a model. To do this, Mesa provides the BatchRunner class. The BatchRunner is instantiated with model and agent-level reporters which are dictionaries with a variable name and function mapping. This class works by generating runs for all possible combinations of values that the user passed to the runner. The iterations argument in the BatchRunner allows user to define how many times they want to run a particular combination of settings in order to account for the stochasticity of their model. Each run terminates after a set number of steps or until the model terminates. At that time, the Batchrunner will collect the reporters. By default, the BatchRunner only collects at the end of a run, but it can be set to collect the whole run by storing the whole DataCollector object.

2.3 Model Visualization

While models can be run headless in Mesa (i.e., no visualization), Mesa also provides a front-end browser-based visualization. We choose a browser-based visualization system over a desktop-based graphical user interface (GUI) for two reasons. First, desktop GUIs lack flexibility in sharing models. By making a model browser based, users can run a model locally on their personal computer or make it accessible to the others to run via the Internet and web browser. Secondly, browser-based front-end technologies develop more rapidly with the changing nature of web application design. Since the creation of Mesa, the front end has been rewritten completely once and changes more rapidly with improvements than the more stable back end. At the time of this publication, the

front-end technologies used are HTML5, Bootstrap, D3, JQuery, Sigma.js for displaying networks, and various charting libraries. A screenshot of the front end of two models created with Mesa can be found in Fig. 3.



A B

Fig. 3. Model visualization of two Mesa applications within a web browser: (A) Wolf-sheep predation Model. (B) Virus on a network (Source: https://github.com/projectmesa).

Models built in Mesa are served to the browser using Tornado, a Python based web server. Mesa uses Tornado's coroutines to ensure that the model does not block the front end from being served. In the browser, the user can control the model run with the expected tooling such "start," "stop," and "step" as well as any controls that they have defined. The commands in the browser trigger the back end. The ModularServer class is what handles the passing of the model and visualizations to the front end. At the end of each step, data in JSON form is sent to front end via a WebSocket connection. This data is then displayed to show the current form of the model and to update any charts or counts that the user also defined to be displayed. When launched locally, the front end can be accessed in any browser window at http://127.0.0.1:8521/.

The user defines which data from the DataCollector is passed to the browser and in what form. Mesa will then render the page for them, so the user does not have to think about styling of the page. To do this, Mesa offers a few preset visualization models and controls. Each visualization module in Mesa has a component on the client-side, in JavaScript, and server-side, in Python. The two of these are developed in tandem, because one does not work with other. When a model is being written, the user passes the visualization objects and the model to the ModularServer to pass them to the front end.

Visualizations provided correlate with the offerings in the space module on the back end, which include a CanvasGridVisualization, used to visual the grid objects from the back end, a HexGridVisualization for hexagonal grids, and a NetworkVisualization to display networks. There are also charting modules to render line charts, bar charts, and pie charts. Lastly, there is a TextVisualization, which renders text, such as count values on the front end. All of these modules update when the model is running. By providing these modules, users only have to consider what values they want displayed.

2.4 Creating a Model

While the act of writing code brings a lot of freedom, it can create inconsistencies from one model to another. One example is file organization and layouts. In a framework like NetLogo, there is only one place to write code (i.e., the code tab), but in Mesa, without guidance you can place your code in one file or in twenty files. As a result, we and members of the community converged on standards. To explain the standards, we will use the Wolf-Sheep Predation Model found in the examples folder (http://bit.ly/projectmesa-examples) in Mesa code repository (see http://bit.ly/WolfSheepMesa). At the top level, a model should be laid out with a Readme.md, a requirements.txt, a run.py, and a folder named after the model using Python naming conventions PEP 8. For the Wolf-Sheep model, the folder is called 'wolf_sheep'. The Readme.md describes the model and is similar to the "Info" tab in NetLogo. The requirements.txt is a Python standard that holds information on the dependencies for a project. Every model will have 'mesa' as one of its dependencies. When building models, it is important to be explicit about which version of Mesa, i.e., 'mesa == 0.8.6', so when core Mesa updates, the model that depends on a certain version of Mesa still continues to function properly. Lastly, the run.py is what launches the server if you are using the front end.

Inside a model folder there should be at least three files: agents.py, model.py, and server.py. In some cases, there may be a schedule.py or other files that are used in the model (e.g., the Wolf-Sheep example). Agents.py house agents, model.py houses the model, and the visualization server details such as charts and grids are in the server.py. When a user launches a model, it looks for the details that define how the server should behave. We tried to make adhering to the standards as easy as possible, so we built a command line tool into Mesa. To start a new project, which lays out all the base files, objects, and text, a user only has to type 'mesa startproject' on the command line in an environment where Mesa is installed. This will prompt a few questions, which after answering will generates the files. In addition to 'mesa startproject' the command line tool will also run the server from inside a model by running 'mesa runserver'.

In addition to the tools and prescriptions we provide, we also encourage users of Mesa to share their models openly so others can learn from them, as well as to isolate development environments by using Virtualenv or something similar in order to make sure dependencies don't come into conflict when building a lot of models, and finally use tools like Flake8 and Black to keep the model code well formatted and clean of extra variables. The purpose of the standards in this section is to make models easy to understand.

3 Applications and Extensions of Mesa

While above, we have introduced Mesa, since its initial release numerous social scientists and researchers have utilized it in a wide range of applications. Over 50 published papers have cited Mesa (Google Scholar 2020) and more than 250 code repositories on GitHub have Mesa as a requirement, which cross many domains such as economics, biology, infrastructure, workplace dynamics etc. (e.g., Pires et al. 2017; Neves et al. 2019). For further applications areas where Mesa has been used see: bit.ly/mesa-publications.

Turning to extensions, as discussed in Sect. 1.1, one of the goals of Mesa is to be extensible and to have interchangeable parts, which allows people to easily integrate specific functionality that might not be a part of the core Mesa package (such as Mesa-Geo). That is not the only geo-spatial modification we found. For example, Heinz (2017) created a modified version which took a simulation server and embedded it into a Django-Channels application and to create a front end with leaflet maps. Another is an open source package called Simulation Occupancy based Agents (SOBA, https://github.com/gsi-upm/soba) which simulates the occupancy of agents in buildings (Delgado 2017). The SOBA project has led to the creation of the simulation tool of building evacuations (Escobar 2017). Lastly, Pike (2018), has created two extensions, Bilateral Shapley and Multi-level Mesa (https://github.com/tpike3/).

4 Conclusion

Motivated by the lack of a Python framework for agent-based modeling, this paper has introduced Mesa. Specially its design goals, the model architecture (along with its key components), how to use it, and some examples of usage and extensions. The success of this framework was highlighted in Sect. 3 with respect to how the Python and agent-based modeling community are utilizing and extending it to meet their modeling needs. However, Mesa is a community effort, and we belief Mesa will continue to evolve to meet the needs of the researchers with the help of the community. For example, as noted in Sect. 1, Mesa only allows for single core processing, however, as multiple cores in machines are becoming the norm, efforts need to be made to explore multithreading and distributed processing. Additional areas of opportunity in core Mesa include the scaling of the data collector by using check pointing, increasing the front-end modules and controls, along with exploring the addition of 3D grids. Beyond Mesa, we look forward to a community of Mesa packages, which extend functionality not offered in core like Mesa-Geo and domain specific extensions that extend the model and agent objects like SOBA.

Acknowledgements. While originally developed by Jackie Kazil and David Masad, Mesa has had over 70 contributors. A special thank you to Corvince, rht, Taylor Mulch, and Tom Pike for their contributions or continuing support to Mesa.

References

Apache License, Version 2.0 (2004). https://www.apache.org/licenses/LICENSE-2.0. Accessed 28 Feb 2020

Comer, K.W.: Who goes first? An examination of the impact of activation on outcome behavior in agent-based models. Ph.D. dissertation, George Mason University, Fairfax, VA (2014)

Crooks, A.T., Malleson, N., Manley, E., Heppenstall, A.J.: Agent-Based Modelling and Geographical Information Systems: A Practical Primer. Sage, London, UK (2019)

Delgado, P.A.: design and development of an agent-based social simulation visualization tool for indoor crowd analytics based on the library Three.Js. Ph.D. dissertation, Universidad Politecnica De Madrid, Madrid, Spain (2017)

Django: Django (version 1.5) (2013). https://www.djangoproject.com/. Accessed 28 Feb 2020
Epstein, J.M., Axtell, R.: Growing Artificial Societies: Social Science from the Bottom Up. MIT Press, Cambridge (1996)
Escobar, G.F.: Design and implementation of an agent-based crowd simulation model for evacuation of university buildings using Python. Ph.D. dissertation, Universidad Politecnica De Madrid, Madrid, Spain (2017)
Gode, D.K., Sunder, S.: Allocative efficiency of markets with zero-intelligence traders: market as a partial substitute for individual rationality. J. Polit. Econ. **101**, 119–137 (1993)
Google Scholar. Papers Citing Mesa (2020). bit.ly/GScholarMesa. Accessed 28 Feb 2020
Hagberg, A., Swart, P., Chult, D.S.: Exploring Network Structure, Dynamics, and Function using NetworkX, Los Alamos National Lab (No. LA-UR-08-05495; LA-UR-08-5495), Los Alamos, NM (2008)
Heinz, T.: Location-based game design pattern exploration through agent-based simulation. In: AGILE 2017 Workshop on Geogames and Geoplay, Wageningen, Netherlands (2017)
Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., Balan, G.: MASON: a multi-agent simulation environment. Simulation **81**(7), 517–527 (2005)
Masad, D., Kazil, J.: Mesa: an agent-based modeling framework. In: Huff, K., Bergstra, J. (eds.) Proceedings of the 14th Python in Science Conference, Austin, TX, pp. 53–60 (2015)
McCabe, S.: Communicating sequential agents: an analysis of concurrent agent scheduling. MA thesis, George Mason University, Fairfax, VA (2016)
Neves, F., Campos, P., Silva, S.: Innovation and employment: an agent-based approach. J. Artif. Soc. Soc. Simul. **22**(1), 8 (2019)
North, M.J., et al.: Complex adaptive systems modeling with repast simphony. Complex Adapt. Syst. Model. **1**(1) (2013). https://doi.org/10.1186/2194-3206-1-3
Patel, A.: Red Blob Games: Hexagonal Grids (2019). https://www.redblobgames.com/grids/hexagons/. Accessed 28 Feb 2020
Pike, T.: Integrating computational tools into foreign policy: introducing mesa packages with a coalition algorithm. J. Policy Complex Syst. **4**(2) (2018). https://doi.org/10.18278/jpcs.4.2.5
Pires, B., Goldstein, J., Molfino, E., Ziemer, K.S.: Knowledge sharing in a dynamic, multi-level organization: exploring cascade and threshold models of diffusion. In: Proceedings of the 2017 International Conference of the Computational Social Science Society of the Americas Santa Fe, NM (2017)
Railsback, S.F., Lytinen, S.L., Jackson, S.K.: Agent-based simulation platforms: review and development recommendations. Simulation **82**(9), 609–623 (2006)
Robinson, D.: Why Is Python Growing So Quickly? (2017). https://stackoverflow.blog/2017/09/14/python-growing-quickly/. Accessed 28 Feb 2020
Sullivan, K., Coletti, M., Luke, S.: GeoMason: GeoSpatial Support for MASON, Department of Computer Science, George Mason University, Technical Report Series, Fairfax, VA (2010)
Wilensky, U.: NetLogo. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL (1999). http://ccl.northwestern.edu/netlogo