



# Helipad: A Framework for Agent-Based Modeling in Python

SOFTWARE  
METAPAPER

CAMERON HARWICK 

 ubiquity press

## ABSTRACT

Agent-based modeling tools commonly trade off usability against power and vice versa. On the one hand, full development environments like NetLogo feature a shallow learning curve, but have a relatively limited proprietary language. Others written in Python or Matlab, for example, have the advantage of a full-featured language with a robust community of third-party libraries, but are typically more skeletal and require more setup and boilerplate in order to write a model. Helipad is introduced to fill this gap. Helipad is a new agent-based modeling framework for Python with the goal of a shallow learning curve, extensive flexibility, minimal boilerplate, and powerful yet easy to set up visualization, in a full Python environment. We summarize Helipad's general architecture and capabilities, and briefly preview a variety of models from a variety of disciplines, including multilevel models, matching models, network models, spatial models, and others.

## CORRESPONDING AUTHOR:

**Cameron Harwick**

SUNY Brockport, US

[cameron@cameronharwick.com](mailto:cameron@cameronharwick.com)

---

## KEYWORDS:

Agent-based modeling; Spatial modeling; Research software; Simulation software

## TO CITE THIS ARTICLE:

Harwick C 2025 Helipad: A Framework for Agent-Based Modeling in Python. *Journal of Open Research Software*, 13: 25. DOI: <https://doi.org/10.5334/jors.547>

## INTRODUCTION

Agent-based modeling is an alternative to traditional analytical modeling that simulates interactions among agents algorithmically [5, 8]. It is particularly valuable for modeling dynamic systems that are difficult to describe with a closed-form analytical solution. In an agent-based model, discrete agents are programmed to interact under conditions that simulate the environment in question, and carry their state with them. Agent behavior can be directly specified in an open-ended way, allowing models to be interpreted much more easily than highly stylized analytical models where agent state can be specified only at an aggregate level. In addition, equilibrium can emerge from such a model – or not – without building the equilibrium into the assumptions of the model [2: ch. 1].

Agent-based modeling necessarily involves programming, and there are numerous frameworks available in a variety of languages. NetLogo is one popular integrated development environment (IDE) that includes a code editor, a proprietary language, and extensive visualization tools, especially for spatial models [4, 27, 28]. Its popularity is due to its shallow learning curve and its integrated environment: very little setup is necessary, models can be easily packaged, and visualizations are simple to set up.

Nevertheless, NetLogo is limited in important ways. First, its language is only object-oriented in a very restricted sense, limiting some of the advantages of agent-based models that involve the states of individual agents.<sup>1</sup> And second, while its self-containedness is an advantage in some respects, it also limits the ability to interact with outside libraries and to use code written for more traditional object-oriented languages.

Agent-based modeling frameworks in other languages on the other hand – for example, in Python [21], Java [22], MatLab, or Mathematica – have the potential to be far more powerful with the ability to draw on general language features, outside libraries, and wider communities of users. However, they are not integrated IDEs: they tend to be skeletal, to provide a basic structure with some visualization capabilities, but generally require a great deal more setup and boilerplate – especially for visualization – than an IDE like NetLogo.

This paper introduces a new agent-based modeling framework for Python, Helipad, to fill this gap. Helipad is a framework rather than an IDE (although the distinction is blurrier when used in a Jupyter notebook), but it has the goal of reducing boilerplate to a minimum and allowing models to be built, tested, and visualized in incremental steps, an important trait for rapid debugging [10]. In the following section we introduce Helipad's general architecture and its array of modeling capabilities.<sup>2</sup> This is followed by an overview of various sample models that have been written

to demonstrate Helipad's capabilities. The paper concludes with suggestions for future applications.

## IMPLEMENTATION AND ARCHITECTURE

### PREREQUISITES

Helipad runs cross-platform on Python 3.9 or higher. It has minimal dependencies, requiring only Matplotlib [18] and NetworkX [11] for visualizations, and Pandas [25] for data collection, both of which in turn rely on Numpy [12]. Shapely is optional at install time, but required for geospatial models.

Helipad can be run “headlessly” or with a GUI, which consists of a control panel and/or a visualization window. The GUI can be run in two different environments. First, a Helipad model can be run directly as a .py file, using Tkinter to provide a cross-platform windowed application interface (Figure 1).

Helipad can also be run in a Jupyter notebook (Figure 3), a format that allows code and exposition to be mixed together and run in-browser [23] in an environment very nearly approaching an IDE. Model code and features are identical in both frontends. Doing so requires, in addition to Jupyter Lab, the Ipywidgets and Ipyimpl libraries.

### HOOKS

There are two distinct strategies that can govern the relationship between user code and the code of an agent-based modeling framework (Figure 2). The two are not entirely mutually exclusive, but in practice, frameworks will hew toward one or the other.

1. *An imperative strategy.* Many frameworks are simply a collection of functions and classes that must be called or subclassed explicitly from a user-specified loop. The advantage of this strategy is that it provides explicit and precise control over every aspect of a model's runtime. The disadvantage is that a great deal of boilerplate must be written in each model.
2. *A hook strategy.* Helipad, by contrast, incorporates the boilerplate and takes care of the looping, allowing user code to be inserted in specific places in the model's runtime through hooks. The advantage of this strategy is that it allows a logical organization of code by topic and minimal boilerplate code. The disadvantage is that the framework makes certain assumptions about model structure, though there are ways to mitigate this disadvantage.

Helipad uses a hook strategy, and minimizes the disadvantages by providing direct access to the model class in most hooks, allowing as much fine-grained control over the model's runtime as would be possible with an imperative framework.

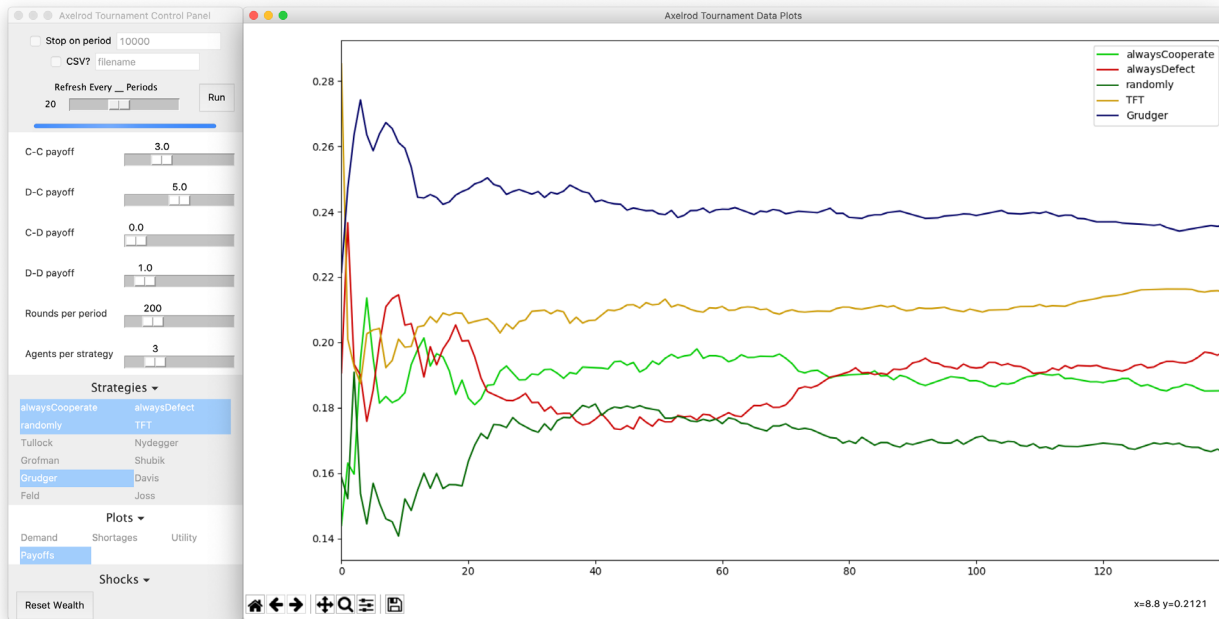


Figure 1 An Axelrod tournament model [2] running in Helipad’s Tkinter frontend.

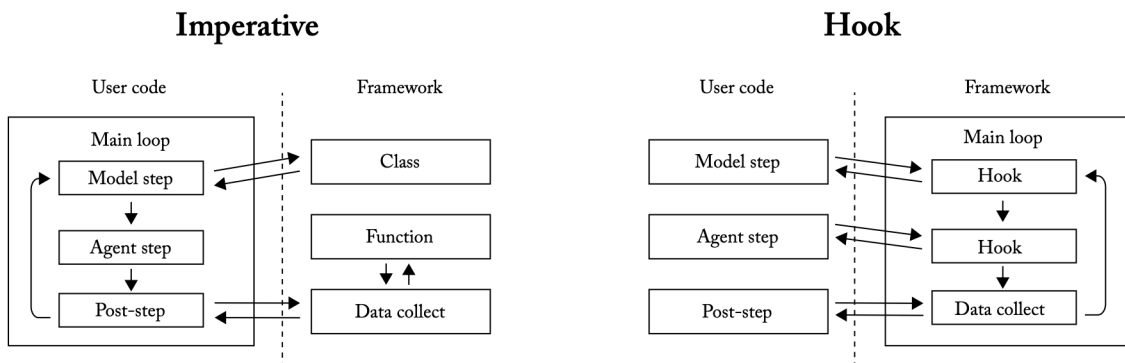


Figure 2 A representation of the relationship between user code and hook code under imperative and hook strategies.

Helipad provides a loop structure for a model, into the elements of which user code can be inserted via hooks. Some of these will be critical to a model’s functioning (e.g. the agents’ step function); others offer low-level control over various aspects of the model (e.g. the `cpanel` hooks).

The `@heli.hook` decorator inserts a user function into these specified points in the model’s runtime. For example, the following agent step function instructs agents to work in the first stage of each period, and consume their product in the second stage.

```
from helipad import *
heli = Helipad()

@heli.hook
def agentStep(agent, model, stage):
    if stage==1: agent.wealth +=
produce(agent.id, agent.laborSupply)
    elif stage==2:
        agent.utility += sqrt(agent.wealth)
```

```
agent.wealth = 0
agent.laborSupply = random.
normal(100, 10)
```

To tell Helipad to run this code for every agent every stage of every period, the `@heli.hook` decorator is added to the top of the function, which is named so that Helipad runs it during the `agentStep` hook. The decorator can also take the hook name as an argument (e.g. `@heli.hook('agentStep')`) and be placed above a function with any name.

There are a few things to notice about this example.

1. Helipad passes three arguments to any function used in the `agentStep` hook: `agent`, `model`, and `stage`. The documentation specifies the exact function signature to be used for each hook.
2. The `agent` and `model` objects are passed as arguments, allowing access to their properties and methods during each agent step.

- Multiple functions can be added to a single hook, in which case they will be executed in the order they were registered (though the `prioritize` argument can be used to move a hook to the front of the queue).

### AGENTS, BREEDS, GOODS, AND PRIMITIVES

Helipad provides for agent heterogeneity through agent *breeds*. A breed is registered during model setup, and assigned to an agent when it is initialized, whether at the beginning of the model run or through reproduction. An agent's breed can be accessed with the `agent.breed` property.

Goods are items that agents can hold stocks of, which are kept track of in the `agent.stocks` property. Goods are also registered during model setup, along with user-defined per-agent properties (for example, agents might have a reservation price for each good), and can be exchanged using `agent.trade()`. One good may optionally be used as a medium of exchange, which allows the `agent.buy()` and `agent.pay()` functions to be used. Agents can optionally take a variety of utility functions over these goods, including Cobb Douglas, Leontief, and CES.

Agent primitives are a way to specify deeper heterogeneity than breeds. Primitives are registered using a separate agent class, subclassed from `baseAgent`, and their behavior is specified using separate hooks. For example, a model might use primitives to distinguish between permanently distinct `'buyer'` and `'seller'` agents that share no common code, while using breeds within each agent to distinguish separate buying and selling strategies. An agent cannot switch primitives once instantiated. Agents of a given primitive can be accessed using `model.agents[primitive]`, and by breed within that primitive with `model.agents[primitive][breed]`. The default primitive is `'agent'`, which is registered automatically at the initialization of a model. Agents of all primitives can be hooked with the `baseAgent` set of hooks.

One included primitive is the `MultiLevel` class, allowing for multi-level agent-based models where the agents at one level are themselves full models [24]. `MultiLevel` therefore inherits from both the `baseAgent` and the `Helipad` classes.

Helipad's `agent` class is well-suited to evolutionary models and genetic algorithms [15, 16]. Agents can reproduce both haploid and polyploid through a powerful `agent.reproduce()` method that allows child traits to be inherited in a variety of ways from one or multiple parents, along with mutations to those traits. Ancestry is tracked with a directed network (see below on networks). Agents keep track of their age in periods in the `agent.age` property, and can be killed with `agent.die()`.

### PARAMETERS

Helipad constructs its control panel GUI primarily with user parameters that allow model variables to be adjusted before and, optionally, during model runtime.

Parameters can be registered with `model.params.add()`. Helipad supports the following parameter types, depending on the format of the variable in question:

- Sliders*, for numerical variables over a range. Sliders can also be set to slide over a discrete set of values, for example on a logarithmic scale.
- Checkboxes*, for boolean variables.
- Menus*, for categorical choices.
- Checkentries*, for a variable equal to the value of a text box if a checkbox is checked, or `False` otherwise. The text box can be numeric or a string.
- Checkgrids*, for a series of related booleans.
- Hidden* parameters can be retrieved and set in user code, but do not display in the control panel.

Figure 1 above shows six slider parameters and two checkgrids, along with two checkentries and a slider in the top configuration section.

Helipad also allows parameters to be specified on a per-breed and per-good basis, with the parameter taking separate values for each registered breed or good. Current parameter values can be accessed at any point in model code using `model.param()`. Parameters can also be set in model code, in which case they are also reflected in the control panel GUI.

Helipad provides a shocks API for numeric parameters. A shock consists of a parameter, a timer function that takes the current model time and outputs a boolean, and a value function that takes the current parameter value and outputs a new value. The value function will then update the parameter value whenever the timer function returns `True`. This can be used for one-time, regular, or stochastic shocks, possibly generating data for impulse response functions as in [12]. Registered shocks can be toggled on and off before and during the model's runtime in the control panel.

### SPATIAL MODELS

Spatial models are the bread and butter of many agent-based models [4, 27, 28], especially in epidemiology where they are commonly used to model infection spread [1, 17]. Helipad can optionally instantiate a spatial map on which agents can move using `model.spatial()`. Spatial models are created by initializing a `Patch` primitive and creating an undirected grid network connecting neighboring patches, with optional diagonal connections.

Spatial models can be initialized with a number of geometries:

- Rectangular, with  $x \times y$  dimensions. Wrapping can be toggled in either or both dimensions (i.e. moving past the edge will wrap to the other edge), for cylindrical or toroidal geometries.
- Polar, with  $\theta \times r$  dimensions, which are useful in certain ontogenetic models in biology [e.g. 20: 556].

3. Geospatial, where arbitrary polygonal patches can be imported from GIS files with libraries like GeoPandas. These are especially useful in applied work, for example in urban studies, where the specific topology of local regions is important [19, 26].

Agents in spatial models acquire `position` and `orientation` properties, along with methods for motion appropriate to the coordinate system (i.e. `agent.up()`, `.down()`, `.left()`, and `.right()` in rectangular geometries, and `agent.clockwise()`, `.counterclockwise()`, `.inward()`, and `.outward()` in polar geometries). Agents can also move absolutely and relatively, as well as `.forward()` in their oriented direction. Orientations can be set and accessed in either degrees or radians by setting the `baseAgent.angleUnit` static property.

Patches are a *fixed* primitive, meaning they cannot reproduce or move. They can die, however, but unlike other killed agents, can be revived later. Agents on a patch that dies will either die themselves or return `None` for their `patch` property, depending on whether agents have been allowed `offmap` in the initializing `spatial()` function.

## DATA AND VISUALIZATION

Ease of visualization is the key advantage of Helipad over other Python-based agent-based modeling frameworks. Unlike some others which have the user create plots directly in (e.g.) Matplotlib, or require the launch of a webserver, Helipad includes an extensible visualization API to manage a full-featured and interactive visualization window. It can also be easily extended with custom visualizations written in Python, without a frontend/backend division; thus Helipad models – even with custom visualizations – can generally be self-contained in one .py file without becoming unwieldy. Visualizations of various types can be registered in only a few lines of code. For example, in the Price Discovery model (described in the following section) where the `LastPrice` property is set in the `agentStep` hook, the following five lines are all that is necessary to collect trade price data and register a live-updating plot of the geometric mean with percentile bars as the model runs.

```
from helipad.visualize import TimeSeries
viz = heli.useVisual(TimeSeries)

heli.data.addReporter('ssprice', heli.
data.agentReporter('lastPrice', 'agent',
stat='gmean', percentiles=[0,100]))
pricePlot = viz.addPlot('price', 'Price',
logscale=True, selected=True)
pricePlot.addSeries('ssprice', 'Soma/Shmoo
Price', '#119900')
```

Model data is collected each period into *reporters*, corresponding to data columns, that return a value each

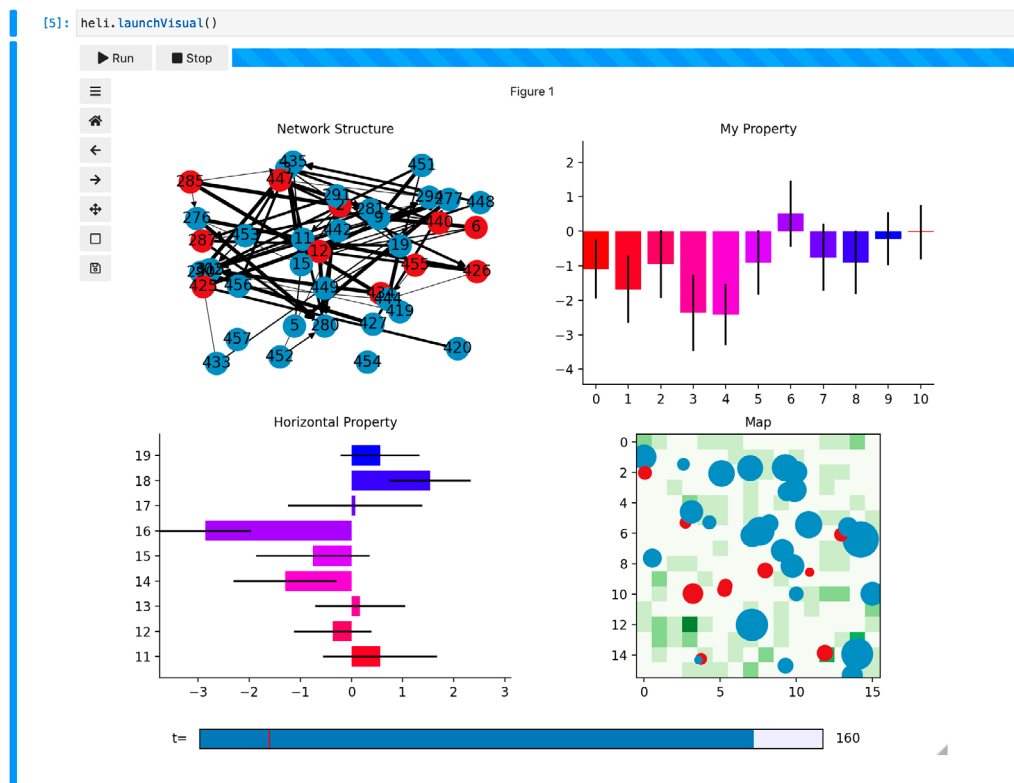
period and record column data. The data as a whole can be accessed during model runtime, exported to a Pandas dataframe, analyzed after the model's run using the `terminate` hook, or exported to a CSV through the control panel.

Parameter values are automatically registered as reporters. Helipad also includes functions to generate reporter functions for summary statistics (including arithmetic and geometric means, sum, maximum, minimum, standard deviation, and percentiles) of agent properties, as in the code block above, but reporter functions can be entirely user-defined as well. For reporters generated as summary statistics of agent properties, percentile marks and  $\pm$  multiples of the standard deviation can be automatically plotted above and/or below the mean, as additional dotted lines above and below a time series line, or as error bars on a bar chart. Reporters can also be registered as a decaying average of any strength with the `smooth` parameter.

There are two included overarching visualizers: `TimeSeries` and `charts`, for diachronic (over time) and synchronic (a particular point in time) data, respectively. Custom visualizations can also be written using the extensible `BaseVisualization` or `MPLVisualization` classes, the latter of which provides low-level access to the Matplotlib API while also maintaining important two-way links between the visualizer and the underlying model, such as automatic updating and interactivity through event handling. Both `TimeSeries` and `Charts` divide the visualization window into *Plots*, areas in the graph view onto which data series can be plotted as the model runs.

`TimeSeries` stacks its plots vertically with a separate vertical axis for each plot, and model time as the shared horizontal axis. The visibility of plots can be toggled in the control panel prior to the model's run. Plots can be displayed on a logarithmic or linear scale. Once a plot area is registered, reporters can be drawn to it by registering a *series* that connects a reporter function to a plot area. [Figure 1](#), for example, shows only one active plot with multiple series displayed on it. When this is done, the plot area will update live with reporter data as the model runs. Series can be drawn as independent lines, or stacked on top of one another within a plot with the `stack` parameter, for example if the sum of several series is important.

The `Charts` visualizer is divided into a grid of plots, where slice-in-time data of any form can be plotted and updated live as the model runs. The `charts` visualizer also features a slider to scrub through time and display the state on each plot at any point in the model history. Bundled plot types include bar charts, network graph diagrams (with a variety of layouts including networks laid on top of a spatial coordinate grid), spatial maps, scatterplots of agent properties, and bar charts, all of



**Figure 3** The Charts visualizer running in Jupyter Lab, displaying Network, Bar Chart, and Spatial plots.

which can be displayed alongside one another in the visualization window.

Custom plot types can be registered and displayed within the Charts visualizer by extending the `ChartPlot` class, and a tutorial notebook for building a custom 3D bar chart visualizer is included. Finally, the appearance of spatial and network plots can be extensively customized, with color, size, and text all set to convey meaningful per-agent information such as breed, wealth, age, ID, and so on.

## OTHER CAPABILITIES

In addition to linear activation and random activation style models, Helipad also supports matching models through a `match` hook that activates agents  $n$  at a time (with  $n = 2$  for pairwise matching). Pairwise matching is an important feature of models in both monetary theory and game theory. Periods can be divided into multiple stages, with activation style (linear, random, or matching) customizable on a per-stage basis, for example with random activation in the first stage, order preserved in the second stage, and matching in the third stage.

Agents can be linked together with multiple named networks, directed or undirected, and weighted or unweighted. Agents can be explicitly connected with an `agent.addEdge()` method, or a network of a given density can be generated automatically. This network structure can be exported for further analysis to dedicated network packages like NetworkX [11].

Ancestry relationships, as mentioned earlier, are kept track of using a directed ‘lineage’ network.

Helipad also includes tools for batch processing model runs, most importantly, a parameter sweep function that runs a model up to a given termination period through every permutation of one or more parameters.<sup>3</sup> The resulting data from each run can be exported to CSV files or passed to further user code for analysis.

Finally, Helipad supports *events*, which trigger when the model satisfies a user-defined criterion, registered by placing an `@event` decorator over a criterion function. For example, an event might be “population stabilizes” or “death rate exceeds 5% over 10 periods”. An event may repeat or not. On the firing of an event, Helipad records the current model data into the `Event` object and notifies the user in the visualization window: `TimeSeries` draws a line at the event mark, and `Charts` flashes the window. Events can be used to stop a model after it reaches a certain state, or to automatically move the model into a different phase.

## PERFORMANCE

Helipad provides a great deal of flexibility and assumes very little out of the box about the structure of the model. This generality does entail some performance cost compared to a pre-compiled model, especially in single-threaded Python, although for moderate numbers of agents the majority of the time cost will be spent in Matplotlib for visualization. A model attribute `heli.timer` can be set to `True` to print live performance

AGENTS	10	100	1,000	10,000
Visualization	184	150	55	7
No Visuals	1,653	553	79	7.4

**Table 1** Performance of the Helicopters model (described below) in periods-per-second on an M2 MacBook Pro, visualized with four time series charts.

data to the console during model run, split between model time and visualization. Although some parallelization is planned for a future release, users seeking performant models with tens of thousands of agents or very long model times should consider frameworks in than Python. Illustrative performance data are shown in [Table 1](#).

## QUALITY CONTROL

Every feature of Helipad is used by least one model described in the following section, plus several additional specialized models for systematically testing control panel functions and visualization subclasses (a 3D bar chart is implemented this way). Testing occurs before each release on the most recent version of Python with MacOS, Windows, Jupyter Lab, and the Spyder IDE. Helipad is designed to validate most function arguments so that errors can be caught on initialization rather than on run.

## EXAMPLE MODELS

This section describes the various sample models that have been written in Helipad, and the features they exemplify, to give a sense of the variety of models possible. All of the following models can be downloaded from Helipad's Github page.<sup>4</sup> This list, of course, is by no means exhaustive.

### HELICOPTER DROPS

Helipad's origin and namesake is a model of relative price responses to monetary shocks in the presence or absence of a banking system, i.e. depending on whether money is injected through helicopter drops or open-market operations [13]. The model features two agent breeds – Hobbits and Dwarves – who consume one of two goods, jam and axes, respectively, and who have differing demand for money set with a per-breed parameter. The relative demand for money balances and goods is determined by a CES utility function. The model also features a store and (optionally) a bank, both registered as separate primitives. The control panel uses the `callback` argument of `model.params.add()` to enforce relationships between certain parameters, and post-model analysis using `statsmodels` is run using the `'terminate'` hook.

## MATCHING MODELS

A price discovery model with random matching is described in [7: ch. 9]. In this model – significantly, written in under 50 lines of code – agents are randomly endowed with two goods, and repeatedly randomly paired to trade along the contract curve of a standard Edgeworth Box setup with Cobb-Douglas utility. The model runs until the per-period trade volume falls below a certain threshold, leading to convergence on a uniform equilibrium price.

Another matching model is the Axelrod tournament [3] displayed in [Figure 1](#). In the Axelrod tournament, strategies are assigned to breeds, and paired randomly against other strategies in 200-round repeated prisoner's dilemmas. As it turns out, the much-celebrated dominance of tit-for-tat is not robust to the collection of strategies it plays against; indeed, in [Figure 1](#), Grudger comes out ahead by a substantial margin.

## EVOLUTIONARY MODELS

An evolutionary model is described in [14], where an agent's reproductive fitness depends positively on a partially-heritable human capital parameter, but negatively on local population density. The fact that population density increases the economic returns to human capital leads to cyclical human capital dynamics. Not only the mean, but also the variance in human capital over time can be seen with the plotted error bands.

Similarly, a model of the evolution of altruism through deme selection [6: ch. 7.1] is included as a multi-level model, with the agents in the top-level model representing competing demes, and the agents of each deme representing individuals. Cooperation is selected against within demes, but demes with a higher proportion of cooperative members are more likely to prevail against and colonize demes with fewer cooperative members. Altruism can survive as a stable strategy provided the benefits to cooperation are high enough, and the likelihood of inter-deme conflict is sufficiently strong. Relative proportions of selfish and altruistic agents are plotted on a stacked plot adding up to 100%.

## SPATIAL MODELS

Conway's Game of Life [9], a cellular automaton where a grid evolves according to simple rules but whose results cannot be predicted from the initial state without stepping through algorithmically, can be implemented as a spatial model in Helipad in just 27 lines of code, including 5 for interactivity in the visualizer, i.e. the ability to toggle cells on and off.

A standard spatial ecological population model of predator-prey relationships (in this case, sheep and grass) is included. The productivity of grass places a hard limit on the sheep population, especially when the latter reproduce sexually. The model can be easily extended to other coordinate systems, for example polar or geospatial models.

## CONCLUSION AND REUSE POTENTIAL

Helipad is a powerful and extensible agent-based modeling framework for Python that ensures a shallow learning curve with a hook-based architecture. It has specialized tools for economic, biological, game-theoretic, and network models, but has ready applications in ecological, epidemiological, organizational, and urban systems – and indeed any context where interacting heterogeneous agents generate emergent structure. A “bootstrap” model template is included, with a minimal skeleton for a well-specified model. The sample models are also written so as to be importable and extensible, with the initialized Helipad object returned using a `setup()` function. Bug reports, feature requests, and support tickets can be filed on Github, and there are numerous user-contributed usage examples of particular features and functions in the documentation (<https://helipad.dev>).

Since its initial public release, Helipad has added a number of significant modeling and architectural features, and is now API stable, ensuring backward-compatibility for future versions. The source code is open, and as agent-based simulations gain traction in social-scientific work, Helipad has the potential to aid the packagability and legibility of model code – especially in notebook format – as well as to lower the barriers to creating new agent-based simulations in a variety of fields.

## APPENDIX 1. GLOSSARY

**Agent.** The basic unit of an agent-based model. In Helipad, an agent is a self-contained object with code and functions for interacting with other agents and with the model environment.

**Breed.** An *agent* type that allows heterogeneity within a *primitive*. Agents of different breeds share code, but conditional statements can make them behave differently. For example, the model might be split between agents playing a ‘hawk’ strategy and those playing a ‘dove’ strategy.

**Control panel.** The window (in the Tkinter frontend) or cell (in a Jupyter notebook) where model settings can be adjusted prior to and during a model’s run.

**Good.** Something which *agents* can hold stocks of and receive utility from. Helipad registers goods and provides functions allowing them to be traded, bought, and so on.

**Edge.** A connection between two *agents*. An edge may have a weight, and may have a direction or no direction. The structure of edges in a model constitutes a *network*.

**Event.** A user-defined condition used to mark phases of a model. An event is triggered when its criterion function first returns `True`. It then records the model

time and marks it in the visualization area, and records all the data from that period.

**Hook.** A predetermined place in the model’s runtime where user code can be inserted.

**Matching Model.** A model where a *period* consists in *agents* being matched in groups, rather than being stepped individually.

**Money.** A *good* serving as a medium of exchange, allowing the use of monetary functions such as `agent.buy()`, `agent.pay()`, and so on.

**Multi-level Model.** A model in which *agents* are themselves full models with sub-agents. The `MultiLevel` *primitive* in this case might represent firms, demes, or other kinds of groups.

**Network.** A structure of connections (*edges*) among agents. There are dedicated Python tools for analyzing networks, such as NetworkX, which can interface with Helipad. Networks may be visualized in a variety of layouts, including on top of a *spatial model*.

**Parameter.** A variable whose value can be adjusted from the *control panel*. Parameters can be global, or split out on a per-breed or per-good basis (e.g. if the productivity of each good were to be set separately).

**Patch.** A fixed agent *primitive* used in *spatial models*, representing the coordinate grid upon which other agent primitives are placed. Patches are placed in a grid *network* with connections to their immediate neighbors, and – optionally – their diagonal neighbors as well. Patch shape will depend on the spatial geometry: a rectangle (in a rectilinear coordinate system), an annular sector (in a polar coordinate system), or an arbitrary polygon (in a geospatial model).

**Period.** An agent-based model repeatedly runs the *step* functions of all the agents. Each time this happens constitutes one period, and model time is kept by the number of periods elapsed since initialization.

**Plot.** A discrete area in the visualization window. The `Charts` visualizer, for example, can take a variety of plot types including network, spatial, and bar plots, and `TimeSeries` takes a time series plot. Some plots, like the bar chart and time series plot, can take individual *series*.

**Primitive.** An agent type more basic than a *breed*. For example, *agents* could be divided by breed to employ different strategies, but a different type of agent entirely – for example, firms – could be registered as a separate primitive, with its own set of breeds. By default all agents are registered with the ‘`agent`’ primitive. Agents cannot change primitives once initialized.

**Reporter.** A function that gathers model data and outputs a numerical value. Registered reporters are run each period to collect data. Helipad provides functions to generate reporter functions for certain common data types, but custom reporter functions can also be written.

**Series.** A line on a time series *plot* or a bar on a bar chart plot, drawing model data from a *reporter*.

**Shock.** An exogenous shift in a *parameter* value. Shocks may be timed automatically with a timer function, or initiated by the user via buttons in the *control panel*.

**Spatial model.** A model where *agents* have a spatial location on a grid of *patches*. Instantiating a spatial model provides agents with methods and properties for orientation and motion, depending on the geometry.

**Stage.** The division of a *period* into multiple parts, for example if *agents* must all run some code before any of them run other code.

**Step.** A function that runs each *period*. Each *agent* has a step function, as well as the model as a whole.

## APPENDIX II. AVAILABILITY

### OPERATING SYSTEM

MacOS, Windows, and Linux with Python 3.10 or higher, or as a web app using Jupyter Lab. Spyder also supported.

### PROGRAMMING LANGUAGE

Python 3.10 or higher.

### ADDITIONAL SYSTEM REQUIREMENTS

None.

### DEPENDENCIES

**Required:** Matplotlib, NetworkX, Pandas

**Optional:** Jupyter Lab, Ipympl, Ipywidgets, Shapely

### LIST OF CONTRIBUTORS

Cameron Harwick, James Caton

### SOFTWARE LOCATION

**Name:** Helipad

**Persistent identifier:** <https://github.com/charwick/helipad>

**Documentation:** <https://helipad.dev>

**License:** MIT

**Version published:** 1.6.3

**Date published:** 10/7/2025

### PERMANENT ARCHIVE

**URL:** [zenodo.org/records/17291353](https://zenodo.org/records/17291353)

**DOI:** [10.5281/zenodo.17291353](https://doi.org/10.5281/zenodo.17291353)

### LANGUAGE

English, but localization supported.

## NOTES

1 Specifically, while agents are themselves objects, there is no way to create user-defined objects.

2 A complete and up-to-date API reference can be found at <https://helipad.dev>.


3 The checkentry is the only parameter type with an open-ended value range, and thus cannot be swept. All other parameter types have finite value ranges.

4 <https://github.com/charwick/helipad/tree/master/sample-models>. Some are also available as Jupyter notebooks: <https://github.com/charwick/helipad/tree/master/sample-notebooks>.

## COMPETING INTERESTS

The author has no competing interests to declare.

## AUTHOR AFFILIATIONS

**Cameron Harwick**  [orcid.org/0000-0003-2703-1627](https://orcid.org/0000-0003-2703-1627)  
SUNY Brockport, US

## REFERENCES

1. **Arifin SM, Madey G, Collins F.** *Spatial Agent-Based Simulation Modeling in Public Health: Design, Implementation, and Applications for Malaria Epidemiology*. John Wiley & Sons; 2016. DOI: <https://doi.org/10.1002/9781118964385>
2. **Arthur B.** *Complexity and the Economy*. New York: Oxford University Press; 2015.
3. **Axelrod R.** Effective Choice in the Prisoner's Dilemma. *Journal of Conflict Resolution*. 1980;24(1):3–25. DOI: <https://doi.org/10.1177/002200278002400101>
4. **Banos A, Lang C, Marilleau N.** *Agent-Based Spatial Simulation with NetLogo*. London: ISTE Press; 2015.
5. **Bonabeau E.** Agent-based modeling: Methods and techniques for simulating human systems. *Proceedings of the National Academy of Sciences*. 2002;99(3):7280–7287. DOI: <https://doi.org/10.1073/pnas.082080899>
6. **Bowles S, Gintis H.** *A Cooperative Species: Human Reciprocity and its Evolution*. Princeton, NJ: Princeton University Press; 2011. DOI: <https://doi.org/10.23943/princeton/9780691151250.001.0001>
7. **Caton J.** *Learn Python for Economic Computation: A Crash Course*. 2020. Available at <https://github.com/jlcatonjr/Learn-Python-for-Stats-and-Econ/tree/master/Textbook>
8. **Epstein J, Axtell R.** *Growing Artificial Societies: Social Science from the Bottom Up*. Cambridge, MA: MIT University Press; 1996. DOI: <https://doi.org/10.7551/mitpress/3374.001.0001>
9. **Gardner M.** Mathematical Games: The Fantastic Combinations of John Conway's New Solitaire Game 'Life'. *Scientific American*. 1970;223:120–123. DOI: <https://doi.org/10.1038/scientificamerican1070-120>
10. **Gilbert N, Troitzsch KG.** *Simulation for the Social Scientist, Second Edition*. New York: Open University Press; 2005.

11. **Hagberg A, Schult D, Swart P.** Exploring network structure, dynamics, and function using NetworkX. In Varoquaux G, Vaught T, Millman J, editors. *Proceedings of the 7th Python in Science Conference*. Pasadena, CA; 2008. pp. 11–15. DOI: <https://doi.org/10.25080/TCWV9851>
12. **Harris CR, Millman KJ, van der Walt SJ,** et al. Array programming with NumPy. *Nature*. 2020;585:357–362. DOI: <https://doi.org/10.1038/s41586-020-2649-2>
13. **Harwick C.** *Helicopters and the Neutrality of Money*. Working paper; 2018. DOI: <https://doi.org/10.2139/ssrn.2545488>
14. **Harwick C.** *Cities in the Rise and Decline of Civilizations*. Working paper; 2021.
15. **Holland J.** Genetic Algorithms: computer programs that ‘evolve’ in was that resemble natural selection can solve complex problems even their creators do not fully understand. *Scientific American*. 1992;267:66–72. DOI: <https://doi.org/10.1038/scientificamerican0792-66>
16. **Holland J.** *Emergence: From Chaos to Order*. New York: Basic Books; 1998. DOI: <https://doi.org/10.1093/oso/9780198504092.001.0001>
17. **Hunter E, Namee BM, Kelleher J.** A Taxonomy for Agent-Based Models in Human Infectious Disease Epidemiology. *Journal of Artificial Societies and Social Simulation*. 2017;20(3). DOI: <https://doi.org/10.18564/jasss.3414>
18. **Hunter JD.** Matplotlib: a 2D graphics environment. *Computing in Science & Engineering*. 2007;9(3):90–95. DOI: <https://doi.org/10.1109/MCSE.2007.55>
19. **Jiang Na, Crooks A, Wang W, Xie Y.** Simulating Urban Shrinkage in Detroit via Agent-Based Modeling. *Sustainability*. 2021;13(4). DOI: <https://doi.org/10.3390/su13042283>
20. **Kauffman SA.** *The Origins of Order: Self-Organization and Selection in Evolution*. Oxford: Oxford University Press; 1993. DOI: <https://doi.org/10.1093/oso/9780195079517.001.0001>
21. **Kazil J, Masad D, Crooks A.** Utilizing Python for Agent-Based Modeling: The Mesa Framework. In: Thompson R, Bisgin H, Dancy C, Hyder A, Hussain M, editors. *Social, Cultural, and Behavioral Modeling: 13th International Conference Proceedings*. Basel: Springer Nature; 2020. DOI: [https://doi.org/10.1007/978-3-030-61255-9\\_30](https://doi.org/10.1007/978-3-030-61255-9_30)
22. **Kluyver T, Ragan-Kelley B, Pérez F,** et al. Jupyter Notebooks—a publishing format for reproducible computational workflows. In: Loizides F, Schmidt B, editors. *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. Amsterdam: IOS Press; 2016. DOI: <https://doi.org/10.3233/978-1-61499-649-1-87>
23. **Luke S, Simon R, Crooks A, Wang H, Wei E, Freelan D, Spagnuolo C, Scarano V, Cordasco G, Cioffi-Revilla C.** The MASON Simulation Toolkit: Past, Present, and Future. *Multi-Agent-Based Simulation*. 2018;19:75–86. DOI: [https://doi.org/10.1007/978-3-030-22270-3\\_6](https://doi.org/10.1007/978-3-030-22270-3_6)
24. **Mathieu P, Morvan G, Picault S.** Multi-level agent-based simulations: Four design patterns. *Simulation Modelling Practice and Theory*. 2018;83:51–64. DOI: <https://doi.org/10.1016/j.simpat.2017.12.015>
25. **McKinney W.** Data structures for statistical computing in Python. In: Van der Walt S, Millman KJ, editors. *Proceedings of the 9th Python in Science Conference*; 2010. pp. 56–61. DOI: <https://doi.org/10.25080/Majora-92bf1922-00a>
26. **Meng X, Xie Y, Crooks A, Wu J, Welsh H, Zeng S.** Examining spatial expansion and stemming strategies of urban shrinkage: evidence from Detroit, USA. *NPJ Urban Sustainability*. 2025;52. DOI: <https://doi.org/10.1038/s42949-025-00245-5>
27. **Railsback SF, Grimm V.** *Agent-Based and Individual-Based Modeling: A Practical Introduction*. Princeton, NJ: Princeton University Press; 2012.
28. **Wilensky U, Rand W.** *An Introduction to Agent-Based Modeling: Modeling Natural, Social, and Engineered Complex Systems with NetLogo*. Cambridge, MA: MIT University Press; 2015.

---

#### TO CITE THIS ARTICLE:

Harwick C 2025 Helipad: A Framework for Agent-Based Modeling in Python. *Journal of Open Research Software*, 13: 25. DOI: <https://doi.org/10.5334/jors.547>

**Submitted:** 18 December 2024    **Accepted:** 27 October 2025    **Published:** 05 November 2025

#### COPYRIGHT:

© 2025 The Author(s). This is an open-access article distributed under the terms of the Creative Commons Attribution 4.0 International License (CC-BY 4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited. See <http://creativecommons.org/licenses/by/4.0/>.

*Journal of Open Research Software* is a peer-reviewed open access journal published by Ubiquity Press.