Mesa Documentation

Release .1

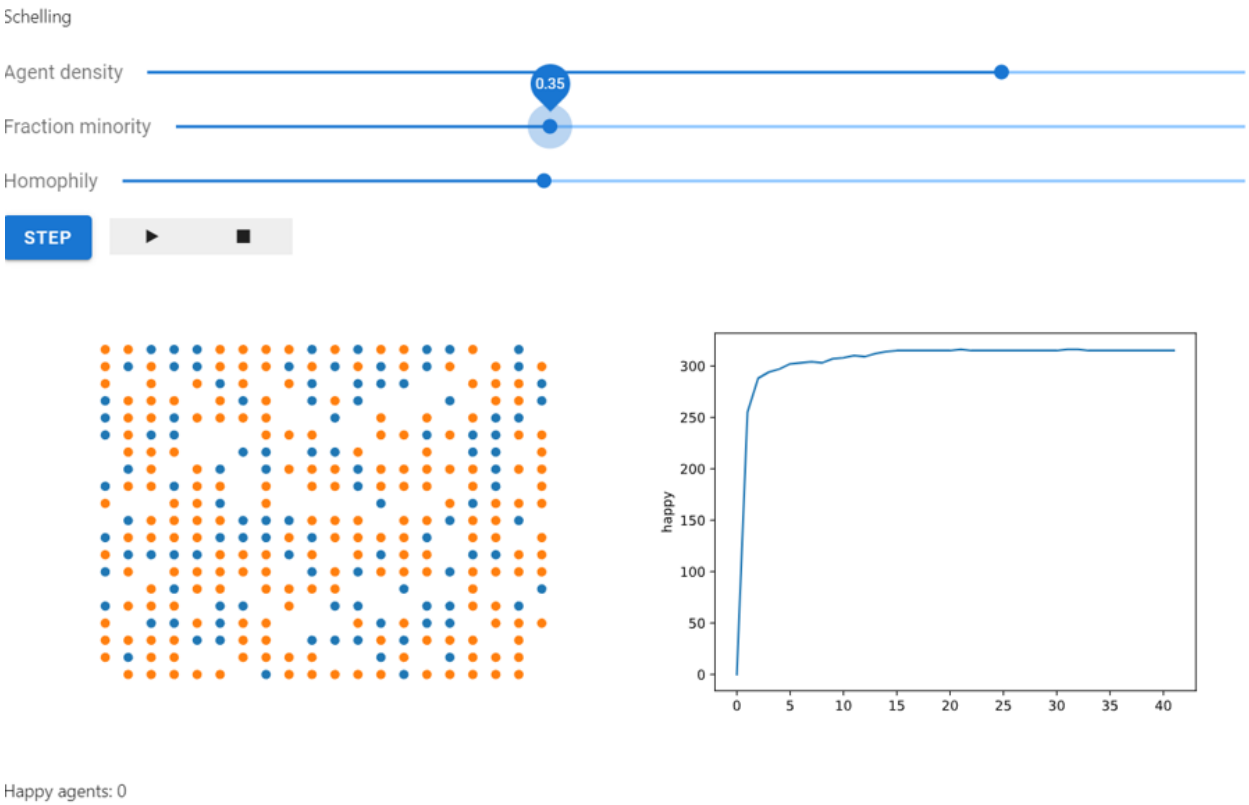
Project Mesa Team

May 07, 2024

CONTENTS

Mesa is an Apache2 licensed agent-based modeling (or ABM) framework in Python.

Mesa allows users to quickly create agent-based models using built-in core components (such as spatial grids and agent schedulers) or customized implementations; visualize them using a browser-based interface; and analyze their results using Python's data analysis tools. Its goal is to be the Python-based counterpart to NetLogo, Repast, or MASON.



Above: A Mesa implementation of the Schelling segregation model, being visualized in a browser window and analyzed in a Jupyter notebook.

CHAPTER

ONE

FEATURES

- Modular components
- Browser-based visualization
- Built-in tools for analysis

CHAPTER

TWO

USING MESA

Getting started quickly:

```
pip install mesa
```

To launch an example model, clone the repository folder and invoke `mesa runserver` for one of the `examples/` subdirectories:

```
mesa runserver examples/wolf_sheep
```

For more help on using Mesa, check out the following resources:

- *Mesa Introductory Tutorial*
- *Mesa Visualization Tutorial*
- GitHub Issue Tracker
- Email list
- PyPI

CHAPTER

THREE

CONTRIBUTING BACK TO MESA

If you run into an issue, please file a ticket for us to discuss. If possible, follow up with a pull request.

If you would like to add a feature, please reach out via ticket or the email list for discussion. A feature is most likely to be added if you build it!

- Contributors guide
- Github

CHAPTER

FOUR

MESA PACKAGES

ABM features users have shared that you may want to use in your model

- See the Packages
- Mesa-Packages

4.1 Mesa Overview

Mesa is a modular framework for building, analyzing and visualizing agent-based models.

Agent-based models are computer simulations involving multiple entities (the agents) acting and interacting with one another based on their programmed behavior. Agents can be used to represent living cells, animals, individual humans, even entire organizations or abstract entities. Sometimes, we may have an understanding of how the individual components of a system behave, and want to see what system-level behaviors and effects emerge from their interaction. Other times, we may have a good idea of how the system overall behaves, and want to figure out what individual behaviors explain it. Or we may want to see how to get agents to cooperate or compete most effectively. Or we may just want to build a cool toy with colorful little dots moving around.

4.1.1 Mesa Modules

Mesa is modular, meaning that its modeling, analysis and visualization components are kept separate but intended to work together. The modules are grouped into three categories:

1. **Modeling:** Modules used to build the models themselves: a model and agent classes, a scheduler to determine the sequence in which the agents act, and space for them to move around on.
2. **Analysis:** Tools to collect data generated from your model, or to run it multiple times with different parameter values.
3. **Visualization:** Classes to create and launch an interactive model visualization, using a server with a JavaScript interface.

4.1.1.1 Modeling modules

Most models consist of one class to represent the model itself; one class (or more) for agents; a scheduler to handle time (what order the agents act in), and possibly a space for the agents to inhabit and move through. These are implemented in Mesa's modeling modules:

- `mesa.Model`, `mesa.Agent`
- *mesa.time*
- *mesa.space*

The skeleton of a model might look like this:

```
import mesa

class MyAgent(mesa.Agent):
    def __init__(self, name, model):
        super().__init__(name, model)
        self.name = name

    def step(self):
        print("{} activated".format(self.name))
        # Whatever else the agent does when activated

class MyModel(mesa.Model):
    def __init__(self, n_agents):
        super().__init__()
        self.schedule = mesa.time.RandomActivation(self)
        self.grid = mesa.space.MultiGrid(10, 10, torus=True)
        for i in range(n_agents):
            a = MyAgent(i, self)
            self.schedule.add(a)
            coords = (self.random.randrange(0, 10), self.random.randrange(0, 10))
            self.grid.place_agent(a, coords)

    def step(self):
        self.schedule.step()
```

If you instantiate a model and run it for one step, like so:

```
model = MyModel(5)
model.step()
```

You should see agents 0-4, activated in random order. See the *tutorial* or API documentation for more detail on how to add model functionality.

To bootstrap a new model install mesa and run `mesa startproject`

4.1.1.2 Analysis modules

If you're using modeling for research, you'll want a way to collect the data each model run generates. You'll probably also want to run the model multiple times, to see how some output changes with different parameters. Data collection and batch running are implemented in the appropriately-named analysis modules:

- *mesa.datacollection*
- *mesa.batchrunner*

You'd add a data collector to the model like this:

```
import mesa

# ...

class MyModel(mesa.Model):
    def __init__(self, n_agents):
        # ...
        self.dc = mesa.DataCollector(model_reporters={"agent_count":
                                     lambda m: m.schedule.get_agent_count()},
                                    agent_reporters={"name": lambda a: a.name})

    def step(self):
        self.schedule.step()
        self.dc.collect(self)
```

The data collector will collect the specified model- and agent-level data at each step of the model. After you're done running it, you can extract the data as a pandas DataFrame:

```
model = MyModel(5)
for t in range(10):
    model.step()
model_df = model.dc.get_model_vars_dataframe()
agent_df = model.dc.get_agent_vars_dataframe()
```

To batch-run the model while varying, for example, the n_agents parameter, you'd use the `batch_run` function:

```
import mesa

parameters = {"n_agents": range(1, 20)}
mesa.batch_run(
    MyModel,
    parameters,
    max_steps=10,
)
```

As with the data collector, once the runs are all over, you can extract the data as a data frame.

```
batch_df = batch_run.get_model_vars_dataframe()
```

4.1.1.3 Visualization modules

Finally, you may want to directly observe your model as it runs. Mesa's main visualization tool uses a small local web server to render the model in a browser, using JavaScript. There are different components for drawing different types of data: for example, grids for drawing agents moving around on a grid, or charts for showing how some data changes as the model runs. A few core modules are:

- mesa.visualization.ModularVisualization
- mesa.visualization.modules

To quickly spin up a model visualization, you might do something like:

```
import mesa

def agent_portrayal(agent):
    portrayal = {"Shape": "circle",
                 "Filled": "true",
                 "Layer": 0,
                 "Color": "red",
                 "r": 0.5}
    return portrayal

grid = mesa.visualization.CanvasGrid(agent_portrayal, 10, 10, 500, 500)
server = mesa.visualization.ModularServer(MyModel,
                       [grid],
                       "My Model",
                       {'n_agents': 10})
server.launch()
```

This will launch the browser-based visualization, on the default port 8521.

4.2 Introductory Tutorial

4.2.1 The Boltzmann Wealth Model

Important:

- If you are just exploring Mesa and want the fastest way to execute the code we recommend executing this tutorial online in a Colab notebook.
- If you have installed mesa and are running locally, please ensure that your Mesa version is up-to-date in order to run this tutorial.

4.2.2 Tutorial Description

Mesa is a Python framework for agent-based modeling. This tutorial will assist you in getting started. Working through the tutorial will help you discover the core features of Mesa. Through the tutorial, you are walked through creating a starter-level model. Functionality is added progressively as the process unfolds. Should anyone find any errors, bugs, have a suggestion, or just are looking for clarification, let us know!

The premise of this tutorial is to create a starter-level model representing agents exchanging money. This exchange of money affects wealth.

Next, *space* is added to allow agents to move based on the change in wealth as time progresses.

Two of Mesa's analytic tools: the *data collector* and *batch runner* are then used to examine the dynamics of this simple model.

4.2.2.1 More Tutorials:

Visualization: There is a separate visualization tutorial that will take users through building a visualization for this model (aka Boltzmann Wealth Model).

Advanced Visualization (legacy): There is also an advanced visualization tutorial that will show users how to use the JavaScript based visualization option, which also uses this model as its base.

4.2.3 Model Description

This is a starter-level simulated agent-based economy. In an agent-based economy, the behavior of an individual economic agent, such as a consumer or producer, is studied in a market environment. This model is drawn from the field econophysics, specifically a paper prepared by Drăgulescu et al. for additional information on the modeling assumptions used in this model. [Drăgulescu, 2002].

The assumption that govern this model are:

1. There are some number of agents.
2. All agents begin with 1 unit of money.
3. At every step of the model, an agent gives 1 unit of money (if they have it) to some other agent.

Even as a starter-level model the yielded results are both interesting and unexpected to individuals unfamiliar with it the specific topic. As such, this model is a good starting point to examine Mesa's core features.

4.2.3.1 Tutorial Setup

Create and activate a virtual environment. *Python version 3.9 or higher is required.*

Install Mesa:

```
pip install --upgrade mesa
```

Install Jupyter Notebook (optional):

```
pip install jupyter
```

Install Seaborn (which is used for data visualization):

```
pip install seaborn
```

If running in Google Colab run the below cell to install Mesa. (This will also work in a locally installed version of Jupyter.)

```
# SKIP THIS CELL unless running in colab

!pip install --quiet mesa
# The exclamation points tell jupyter to do the command via the command line
```

```
Note: you may need to restart the kernel to use updated packages.
```

4.2.4 Building the Sample Model

After Mesa is installed a model can be built. A jupyter notebook is recommended for this tutorial, this allows for small segments of codes to be examined one at a time. As an option this can be created using python script files.

Good Practice: Place a model in its own folder/directory. This is not specifically required for the starter_model, but as other models become more complicated and expand multiple python scripts, documentation, discussions and notebooks may be added.

4.2.4.1 Create New Folder/Directory

- Using operating system commands create a new folder/directory named 'starter_model'.
- Change into the new folder/directory.

4.2.4.2 Creating Model With Jupyter Notebook

Write the model interactively in Jupyter Notebook cells.

Start Jupyter Notebook:

```
jupyter notebook
```

Create a new Notebook named `money_model.ipynb` (or whatever you want to call it).

4.2.4.3 Creating Model With Script File (IDE, Text Editor, Colab, etc.)

Create a new file called `money_model.py` (or whatever you want to call it)

Code will be added as the tutorial progresses.

4.2.4.4 Import Dependencies

This includes importing of dependencies needed for the tutorial.

```
import mesa

# Data visualization tools.
import seaborn as sns

# Has multi-dimensional arrays and matrices. Has a large collection of
# mathematical functions to operate on these arrays.
import numpy as np

# Data manipulation and analysis.
import pandas as pd
```

4.2.4.5 Create Agent

First create the agent. As the tutorial progresses, more functionality will be added to the agent.

Background: Agents are the individual entities that act in the model. It is a good modeling practice to make certain each Agent can be uniquely identified.

Model-specific information: Agents are the individuals that exchange money, in this case the amount of money an individual agent has is represented as wealth. Additionally, agents each have a unique identifier.

Code implementation: This is done by creating a new class (or object) that extends `mesa.Agent` creating a subclass of the `Agent` class from mesa. The new class is named `MoneyAgent`. The technical details about the agent object can be found in the mesa repo.

The `MoneyAgent` class is created with the following code:

```
class MoneyAgent(mesa.Agent):
    """An agent with fixed initial wealth."""

    def __init__(self, unique_id, model):
        # Pass the parameters to the parent class.
        super().__init__(unique_id, model)

        # Create the agent's variable and set the initial values.
        self.wealth = 1
```

4.2.4.6 Create Model

Next, create the model. Again, as the tutorial progresses, more functionality will be added to the model.

Background: The model can be visualized as a grid containing all the agents. The model creates, holds and manages all the agents on the grid. The model evolves in discrete time steps.

Model-specific information: When a model is created the number of agents within the model is specified. The model then creates the agents and places them on the grid. The model also contains a scheduler which controls the order in which agents are activated. The scheduler is also responsible for advancing the model by one step. The model also contains a data collector which collects data from the model. These topics will be covered in more detail later in the tutorial.

Code implementation: This is done by creating a new class (or object) that extends `mesa.Model` and calls `super().__init__()`, creating a subclass of the `Model` class from mesa. The new class is named `MoneyModel`. The technical details about the agent object can be found in the mesa repo.

The `MoneyModel` class is created with the following code:

```
class MoneyModel(mesa.Model):
    """A model with some number of agents."""

    def __init__(self, N):
        super().__init__()
        self.num_agents = N
        # Create agents
        for i in range(self.num_agents):
            a = MoneyAgent(i, self)
```

4.2.4.7 Adding the Scheduler

Now the model will be modified to add a scheduler.

Background: The scheduler controls the order in which agents are activated, causing the agent to take their defined action. The scheduler is also responsible for advancing the model by one step. A step is the smallest unit of time in the model, and is often referred to as a tick. The scheduler can be configured to activate agents in different orders. This can be important as the order in which agents are activated can impact the results of the model [Comer2014]. At each step of the model, one or more of the agents – usually all of them – are activated and take their own step, changing internally and/or interacting with one another or the environment.

Model-specific information: A new class is named `RandomActivationByAgent` is created which extends `mesa.time.RandomActivation` creating a subclass of the `RandomActivation` class from Mesa. This class activates all the agents once per step, in random order. Every agent is expected to have a `step` method. The step method is the action the agent takes when it is activated by the model schedule. We add an agent to the schedule using the `add` method; when we call the schedule's `step` method, the model shuffles the order of the agents, then activates and executes each agent's `step` method. The scheduler is then added to the model.

Code implementation: The technical details about the timer object can be found in the mesa repo. Mesa offers a few different built-in scheduler classes, with a common interface. That makes it easy to change the activation regime a given model uses, and see whether it changes the model behavior. The details pertaining to the scheduler interface can be located in the same mesa repo.

With that in mind, the `MoneyAgent` code is modified below to visually show when a new agent is created. The MoneyModel code is modified by adding the RandomActivation method to the model. with the scheduler added looks like this:

```
class MoneyAgent(mesa.Agent):
    """An agent with fixed initial wealth."""

    def __init__(self, unique_id, model):
        # Pass the parameters to the parent class.
        super().__init__(unique_id, model)

        # Create the agent's attribute and set the initial values.
        self.wealth = 1

    def step(self):
        # The agent's step will go here.
        # For demonstration purposes we will print the agent's unique_id
        print(f"Hi, I am an agent, you can call me {str(self.unique_id)}.")


class MoneyModel(mesa.Model):
    """A model with some number of agents."""

    def __init__(self, N):
        super().__init__()
        self.num_agents = N
        # Create scheduler and assign it to the model
        self.schedule = mesa.time.RandomActivation(self)

        # Create agents
        for i in range(self.num_agents):
            a = MoneyAgent(i, self)
```

(continues on next page)

(continued from previous page)

```
            # Add the agent to the scheduler
            self.schedule.add(a)

    def step(self):
        """Advance the model by one step."""

        # The model's step will go here for now this will call the step method of each␣
↪agent and print the agent's unique_id
        self.schedule.step()
```

4.2.4.8 Running the Model

A basic model has now been created. The model can be run by creating a model object and calling the step method. The model will run for one step and print the unique_id of each agent. You may run the model for multiple steps by calling the step method multiple times.

Note: If you are using `.py` (script) files instead of `.ipynb` (Jupyter), the common convention is to have a `run.py` in the same directory as your model code. You then (1) import the `MoneyModel` class, (2) create a model object and (3) run it for a few steps. As shown below:

```
from money_model import MoneyModel

starter_model = MoneyModel(10)
starter_model.step()
```

Create the model object, and run it for one step:

```
starter_model = MoneyModel(10)
starter_model.step()
```

```
Hi, I am an agent, you can call me 5.
Hi, I am an agent, you can call me 0.
Hi, I am an agent, you can call me 7.
Hi, I am an agent, you can call me 6.
Hi, I am an agent, you can call me 4.
Hi, I am an agent, you can call me 1.
Hi, I am an agent, you can call me 3.
Hi, I am an agent, you can call me 2.
Hi, I am an agent, you can call me 9.
Hi, I am an agent, you can call me 8.
```

```
# Run this step overnight and see what happens! Notice the order of the agents changes␣
↪each time.
starter_model.step()
```

```
Hi, I am an agent, you can call me 0.
Hi, I am an agent, you can call me 2.
Hi, I am an agent, you can call me 3.
Hi, I am an agent, you can call me 4.
Hi, I am an agent, you can call me 9.
Hi, I am an agent, you can call me 1.
```

(continues on next page)

(continued from previous page)

```
Hi, I am an agent, you can call me 8.
Hi, I am an agent, you can call me 7.
Hi, I am an agent, you can call me 5.
Hi, I am an agent, you can call me 6.
```

4.2.4.8.1 Exercise

Modifying the code below to have every agent print out its `wealth` when it is activated.

```
class MoneyAgent(mesa.Agent):
    """An agent with fixed initial wealth."""

    def __init__(self, unique_id, model):
        # Pass the parameters to the parent class.
        super().__init__(unique_id, model)

        # Create the agent's variable and set the initial values.
        self.wealth = 1

    def step(self):
        # The agent's step will go here.
        # FIXME: Need to print the agent's wealth
        print(f"Hi, I am an agent and I am broke!")
```

Create a model for 12 Agents, and run it for a few steps to see the output.

```
# Fixme: Create the model object, and run it
```

4.2.4.9 Agent Step

Returning back to the MoneyAgent the actual step process is now going to be created.

Background: This is where the agent's behavior as it relates to each step or tick of the model is defined.

Model-specific information: In this case, the agent will check its wealth, and if it has money, give one unit of it away to another random agent.

Code implementation: The agent's step method is called by the scheduler during each step of the model. To allow the agent to choose another agent at random, we use the `model.random` random-number generator. This works just like Python's `random` module, but with a fixed seed set when the model is instantiated, that can be used to replicate a specific model run later.

To pick an agent at random, we need a list of all agents. Notice that there isn't such a list explicitly in the model. The scheduler, however, does have an internal list of all the agents it is scheduled to activate.

With that in mind, we rewrite the agent `step` method as shown below:

```
import copy


class MoneyAgent(mesa.Agent):
    """An agent with fixed initial wealth."""
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, unique_id, model):
        # Pass the parameters to the parent class.
        super().__init__(unique_id, model)

        # Create the agent's variable and set the initial values.
        self.wealth = 1

    def step(self):
        # Verify agent has some wealth
        if self.wealth > 0:
            other_agent = self.random.choice(self.model.schedule.agents)
            if other_agent is not None:
                other_agent.wealth += 1
                self.wealth -= 1
```

4.2.4.10 Running your first model

With that last piece in hand, it's time for the first rudimentary run of the model.

If you've written the code in its own script file (`money_model.py` or a different name) you can now modify your `run.py` or even launch a Jupyter Notebook. You then just follow the same three steps of (1) import your model class `MoneyModel`, (2) create the model object and (3) run it for a few steps. If you wrote the code in one Notebook then step 1, importing, is not necessary.

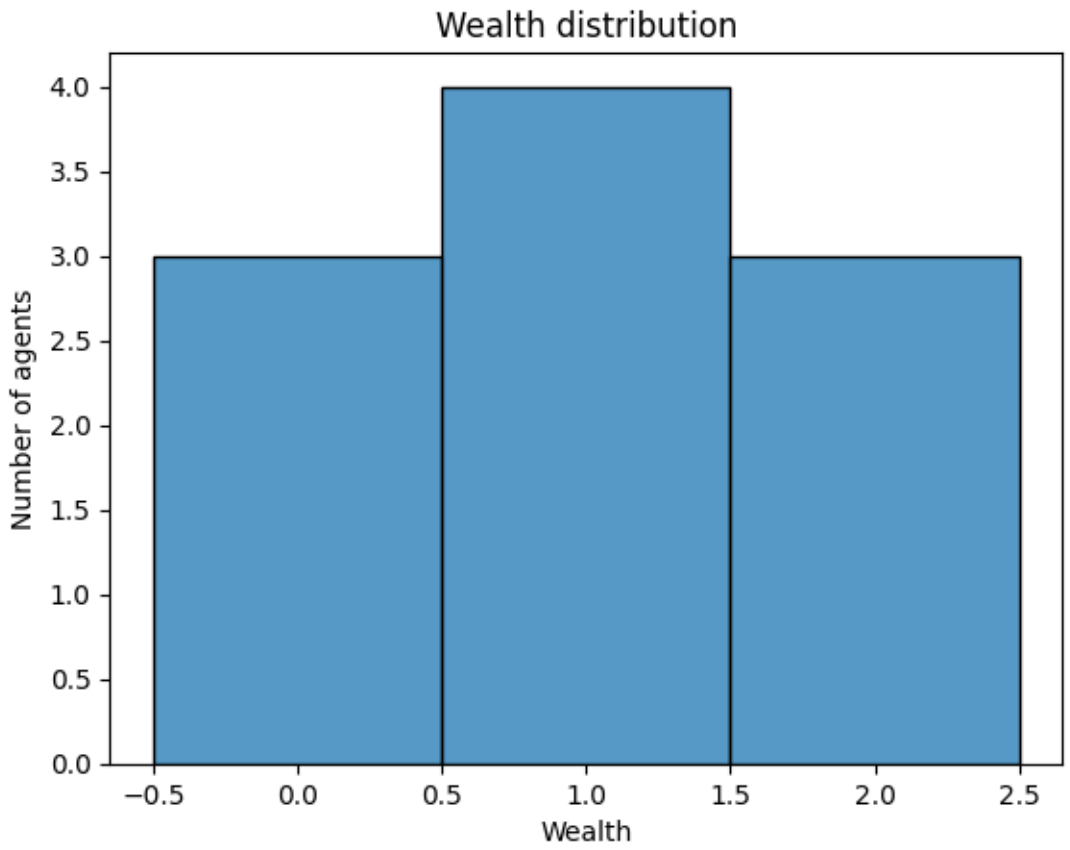
```
from money_model import MoneyModel
```

Now let's create a model with 10 agents, and run it for 10 steps.

```
model = MoneyModel(10)
for i in range(10):
    model.step()
```

Next, we need to get some data out of the model. Specifically, we want to see the distribution of the agent's wealth. We can get the wealth values with list comprehension, and then use seaborn (or another graphics library) to visualize the data in a histogram.

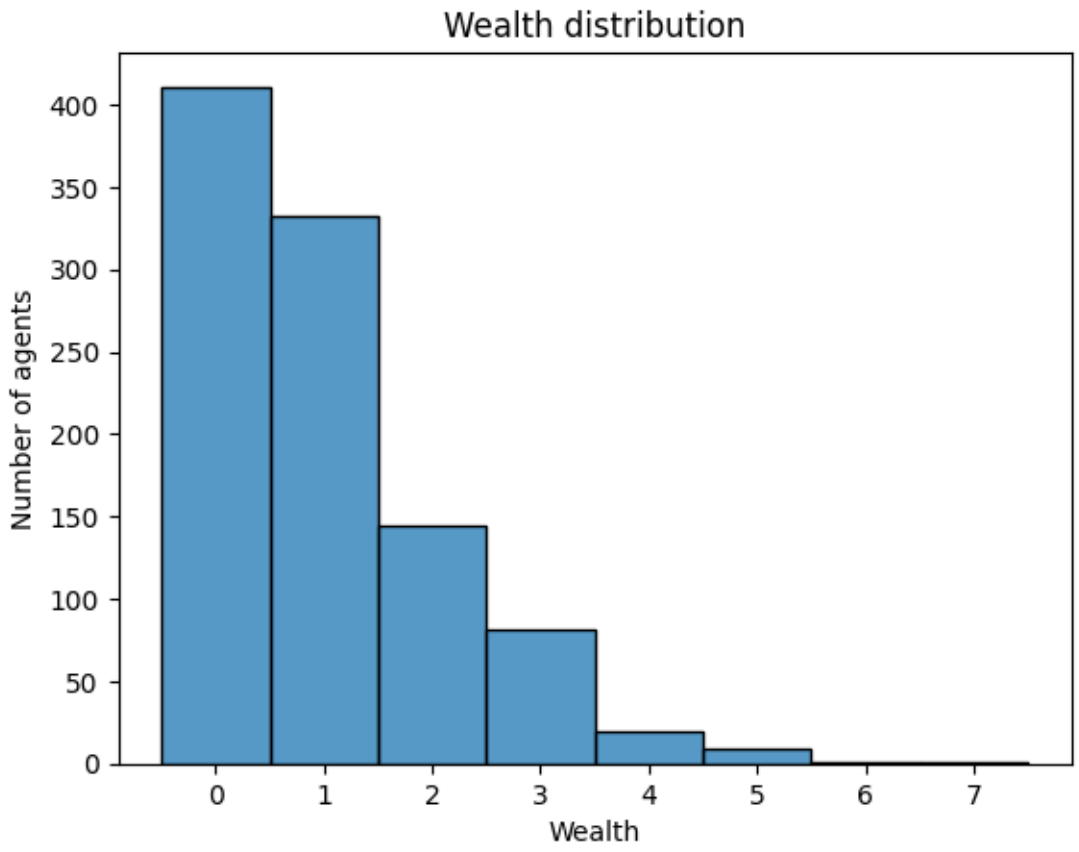
```
agent_wealth = [a.wealth for a in model.schedule.agents]
# Create a histogram with seaborn
g = sns.histplot(agent_wealth, discrete=True)
g.set(
    title="Wealth distribution", xlabel="Wealth", ylabel="Number of agents"
);  # The semicolon is just to avoid printing the object representation
```



You'll should see something like the distribution above. Yours will almost certainly look at least slightly different, since each run of the model is random, after all.

To get a better idea of how a model behaves, we can create multiple model runs and see the distribution that emerges from all of them. We can do this with a nested for loop:

```
all_wealth = []
# This runs the model 100 times, each model executing 10 steps.
for j in range(100):
    # Run the model
    model = MoneyModel(10)
    for i in range(10):
        model.step()

    # Store the results
    for agent in model.schedule.agents:
        all_wealth.append(agent.wealth)

# Use seaborn
g = sns.histplot(all_wealth, discrete=True)
g.set(title="Wealth distribution", xlabel="Wealth", ylabel="Number of agents");
```



This runs 100 instantiations of the model, and runs each for 10 steps. (Notice that we set the histogram bins to be integers, since agents can only have whole numbers of wealth). This distribution looks a lot smoother. By running the model 100 times, we smooth out some of the 'noise' of randomness, and get to the model's overall expected behavior.

This outcome might be surprising. Despite the fact that all agents, on average, give and receive one unit of money every step, the model converges to a state where most agents have a small amount of money and a small number have a lot of money.

4.2.4.11 Adding space

Many ABMs have a spatial element, with agents moving around and interacting with nearby neighbors. Mesa currently supports two overall kinds of spaces: grid, and continuous. Grids are divided into cells, and agents can only be on a particular cell, like pieces on a chess board. Continuous space, in contrast, allows agents to have any arbitrary position. Both grids and continuous spaces are frequently toroidal, meaning that the edges wrap around, with cells on the right edge connected to those on the left edge, and the top to the bottom. This prevents some cells having fewer neighbors than others, or agents being able to go off the edge of the environment.

Let's add a simple spatial element to our model by putting our agents on a grid and make them walk around at random. Instead of giving their unit of money to any random agent, they'll give it to an agent on the same cell.

Mesa has two main types of grids: `SingleGrid` and `MultiGrid`*. `SingleGrid` enforces at most one agent per cell; `MultiGrid` allows multiple agents to be in the same cell. Since we want agents to be able to share a cell, we use `MultiGrid`.

*However there are more types of space to include `HexGrid`, `NetworkGrid`, and the previously mentioned `ContinuousSpace`. Similar to `mesa.time` context is retained with `mesa.space.[enter class]`. You can inspect the different classes at mesa.space.

We instantiate a grid with width and height parameters, and a boolean as to whether the grid is toroidal. Let's make width and height model parameters, in addition to the number of agents, and have the grid always be toroidal. We can place agents on a grid with the grid's `place_agent` method, which takes an agent and an (x, y) tuple of the coordinates to place the agent.

```
class MoneyModel(mesa.Model):
    """A model with some number of agents."""

    def __init__(self, N, width, height):
        super().__init__()
        self.num_agents = N
        self.grid = mesa.space.MultiGrid(width, height, True)
        self.schedule = mesa.time.RandomActivation(self)

        # Create agents
        for i in range(self.num_agents):
            a = MoneyAgent(i, self)
            self.schedule.add(a)

            # Add the agent to a random grid cell
            x = self.random.randrange(self.grid.width)
            y = self.random.randrange(self.grid.height)
            self.grid.place_agent(a, (x, y))
```

Under the hood, each agent's position is stored in two ways: the agent is contained in the grid in the cell it is currently in, and the agent has a `pos` variable with an (x, y) coordinate tuple. The `place_agent` method adds the coordinate to the agent automatically.

Now we need to add to the agents' behaviors, letting them move around and only give money to other agents in the same cell.

First let's handle movement, and have the agents move to a neighboring cell. The grid object provides a `move_agent` method, which like you'd imagine, moves an agent to a given cell. That still leaves us to get the possible neighboring cells to move to. There are a couple ways to do this. One is to use the current coordinates, and loop over all coordinates +/- 1 away from it. For example:

```
neighbors = []
x, y = self.pos
for dx in [-1, 0, 1]:
    for dy in [-1, 0, 1]:
        neighbors.append((x+dx, y+dy))
```

But there's an even simpler way, using the grid's built-in `get_neighborhood` method, which returns all the neighbors of a given cell. This method can get two types of cell neighborhoods: Moore (includes all 8 surrounding squares), and Von Neumann(only up/down/left/right). It also needs an argument as to whether to include the center cell itself as one of the neighbors.

With that in mind, the agent's `move` method looks like this:

```
class MoneyAgent(mesa.Agent):
   #...
    def move(self):
        possible_steps = self.model.grid.get_neighborhood(
            self.pos,
            moore=True,
```

(continues on next page)

(continued from previous page)

```
            include_center=False)
        new_position = self.random.choice(possible_steps)
        self.model.grid.move_agent(self, new_position)
```

Next, we need to get all the other agents present in a cell, and give one of them some money. We can get the contents of one or more cells using the grid's `get_cell_list_contents` method, or by accessing a cell directly. The method accepts a list of cell coordinate tuples, or a single tuple if we only care about one cell.

```
class MoneyAgent(mesa.Agent):
    #...
    def give_money(self):
        cellmates = self.model.grid.get_cell_list_contents([self.pos])
        if len(cellmates) > 1:
            other = self.random.choice(cellmates)
            other.wealth += 1
            self.wealth -= 1
```

And with those two methods, the agent's `step` method becomes:

```
class MoneyAgent(mesa.Agent):
    # ...
    def step(self):
        self.move()
        if self.wealth > 0:
            self.give_money()
```

Now, putting that all together should look like this:

```
class MoneyAgent(mesa.Agent):
    """An agent with fixed initial wealth."""

    def __init__(self, unique_id, model):
        super().__init__(unique_id, model)
        self.wealth = 1

    def move(self):
        possible_steps = self.model.grid.get_neighborhood(
            self.pos, moore=True, include_center=False
        )
        new_position = self.random.choice(possible_steps)
        self.model.grid.move_agent(self, new_position)

    def give_money(self):
        cellmates = self.model.grid.get_cell_list_contents([self.pos])
        if len(cellmates) > 1:
            other_agent = self.random.choice(cellmates)
            other_agent.wealth += 1
            self.wealth -= 1

    def step(self):
        self.move()
        if self.wealth > 0:
            self.give_money()
```

(continues on next page)

(continued from previous page)

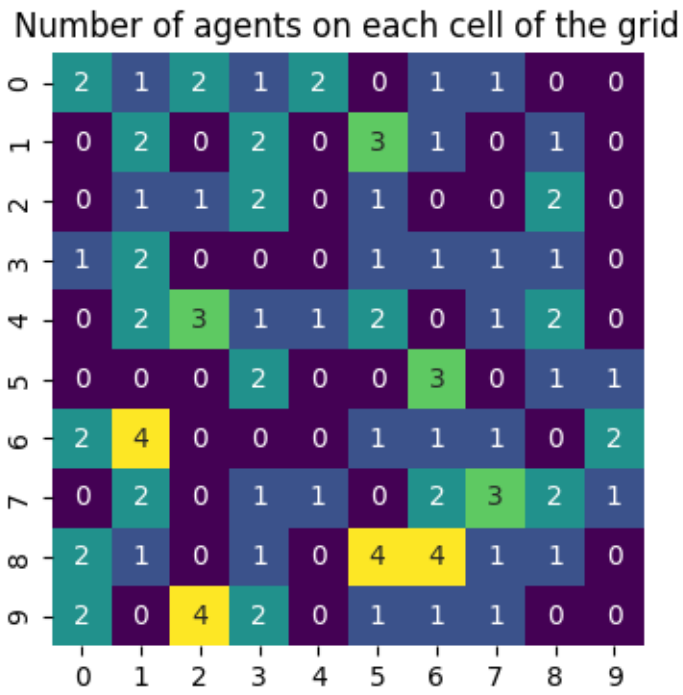
```
class MoneyModel(mesa.Model):
    """A model with some number of agents."""

    def __init__(self, N, width, height):
        super().__init__()
        self.num_agents = N
        self.grid = mesa.space.MultiGrid(width, height, True)
        self.schedule = mesa.time.RandomActivation(self)
        # Create agents
        for i in range(self.num_agents):
            a = MoneyAgent(i, self)
            self.schedule.add(a)
            # Add the agent to a random grid cell
            x = self.random.randrange(self.grid.width)
            y = self.random.randrange(self.grid.height)
            self.grid.place_agent(a, (x, y))

    def step(self):
        self.schedule.step()
```

Let's create a model with 100 agents on a 10x10 grid, and run it for 20 steps.

```
model = MoneyModel(100, 10, 10)
for i in range(20):
    model.step()
```

Now let's use seaborn and numpy to visualize the number of agents residing in each cell. To do that, we create a numpy array of the same size as the grid, filled with zeros. Then we use the grid object's `coord_iter()` feature, which lets us loop over every cell in the grid, giving us each cell's positions and contents in turn.

```
agent_counts = np.zeros((model.grid.width, model.grid.height))
for cell_content, (x, y) in model.grid.coord_iter():
    agent_count = len(cell_content)
    agent_counts[x][y] = agent_count
# Plot using seaborn, with a size of 5x5
g = sns.heatmap(agent_counts, cmap="viridis", annot=True, cbar=False, square=True)
g.figure.set_size_inches(4, 4)
g.set(title="Number of agents on each cell of the grid");
```



4.2.4.12 Collecting Data

So far, at the end of every model run, we've had to go and write our own code to get the data out of the model. This has two problems: it isn't very efficient, and it only gives us end results. If we wanted to know the wealth of each agent at each step, we'd have to add that to the loop of executing steps, and figure out some way to store the data.

Since one of the main goals of agent-based modeling is generating data for analysis, Mesa provides a class which can handle data collection and storage for us and make it easier to analyze.

The data collector stores three categories of data: model-level variables, agent-level variables, and tables (which are a catch-all for everything else). Model- and agent-level variables are added to the data collector along with a function for collecting them. Model-level collection functions take a model object as an input, while agent-level collection functions take an agent object as an input. Both then return a value computed from the model or each agent at their current state. When the data collector's `collect` method is called, with a model object as its argument, it applies each model-level collection function to the model, and stores the results in a dictionary, associating the current value with the current step of the model. Similarly, the method applies each agent-level collection function to each agent currently in the schedule, associating the resulting value with the step of the model, and the agent's `unique_id`.

Let's add a DataCollector to the model with `mesa.DataCollector`, and collect two variables. At the agent level, we want to collect every agent's wealth at every step. At the model level, let's measure the model's Gini Coefficient, a measure of wealth inequality.

```
def compute_gini(model):
    agent_wealths = [agent.wealth for agent in model.schedule.agents]
    x = sorted(agent_wealths)
    N = model.num_agents
    B = sum(xi * (N - i) for i, xi in enumerate(x)) / (N * sum(x))
    return 1 + (1 / N) - 2 * B
```

(continues on next page)

(continued from previous page)

```
class MoneyAgent(mesa.Agent):
    """An agent with fixed initial wealth."""

    def __init__(self, unique_id, model):
        super().__init__(unique_id, model)
        self.wealth = 1

    def move(self):
        possible_steps = self.model.grid.get_neighborhood(
            self.pos, moore=True, include_center=False
        )
        new_position = self.random.choice(possible_steps)
        self.model.grid.move_agent(self, new_position)

    def give_money(self):
        cellmates = self.model.grid.get_cell_list_contents([self.pos])
        cellmates.pop(
            cellmates.index(self)
        )  # Ensure agent is not giving money to itself
        if len(cellmates) > 1:
            other = self.random.choice(cellmates)
            other.wealth += 1
            self.wealth -= 1
            if other == self:
                print("I JUST GAVE MONEY TO MYSELF HEHEHE!")

    def step(self):
        self.move()
        if self.wealth > 0:
            self.give_money()


class MoneyModel(mesa.Model):
    """A model with some number of agents."""

    def __init__(self, N, width, height):
        super().__init__()
        self.num_agents = N
        self.grid = mesa.space.MultiGrid(width, height, True)
        self.schedule = mesa.time.RandomActivation(self)

        # Create agents
        for i in range(self.num_agents):
            a = MoneyAgent(i, self)
            self.schedule.add(a)
            # Add the agent to a random grid cell
            x = self.random.randrange(self.grid.width)
            y = self.random.randrange(self.grid.height)
            self.grid.place_agent(a, (x, y))

        self.datacollector = mesa.DataCollector(
            model_reporters={"Gini": compute_gini}, agent_reporters={"Wealth": "wealth"}
```

(continues on next page)

(continued from previous page)

```
        )

    def step(self):
        self.datacollector.collect(self)
        self.schedule.step()
```

At every step of the model, the datacollector will collect and store the model-level current Gini coefficient, as well as each agent's wealth, associating each with the current step.

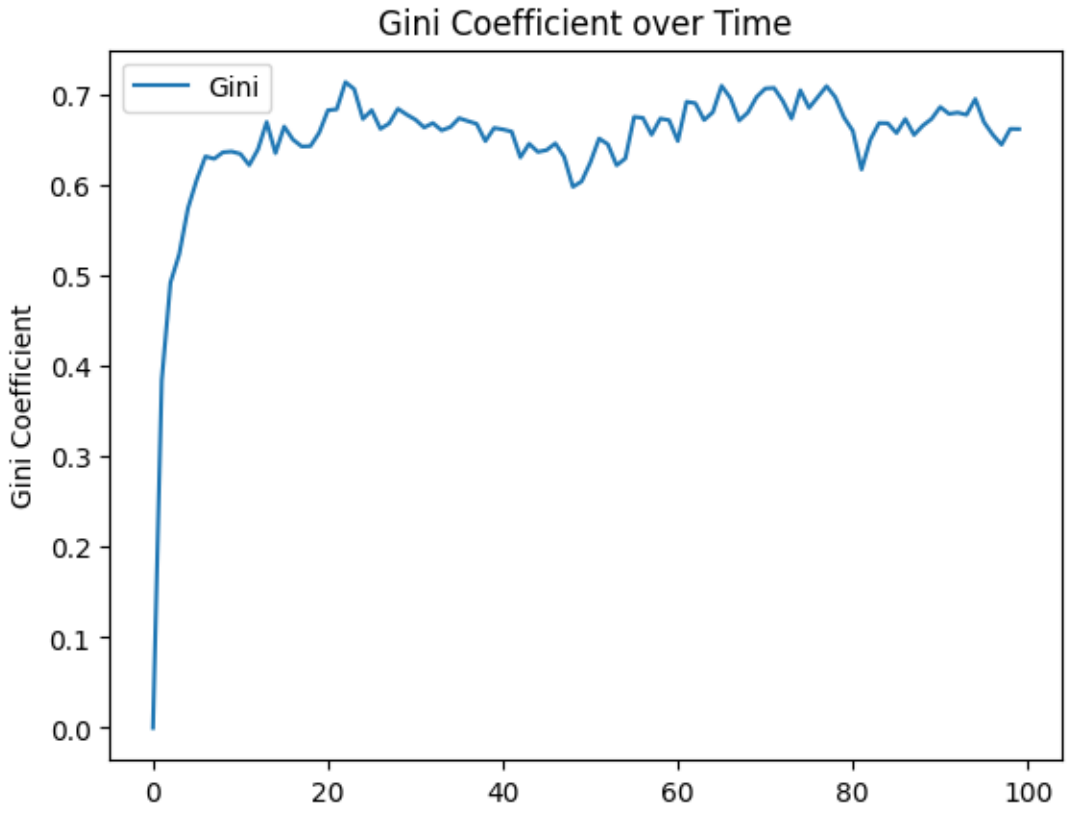
We run the model just as we did above. Now is when an interactive session, especially via a Notebook, comes in handy: the DataCollector can export the data its collected as a pandas* DataFrame, for easy interactive analysis.

*If you are new to Python, please be aware that pandas is already installed as a dependency of Mesa and that pandas is a "fast, powerful, flexible and easy to use open source data analysis and manipulation tool". pandas is great resource to help analyze the data collected in your models.

```
model = MoneyModel(100, 10, 10)
for i in range(100):
    model.step()
```

To get the series of Gini coefficients as a pandas DataFrame:

```
gini = model.datacollector.get_model_vars_dataframe()
# Plot the Gini coefficient over time
g = sns.lineplot(data=gini)
g.set(title="Gini Coefficient over Time", ylabel="Gini Coefficient");
```



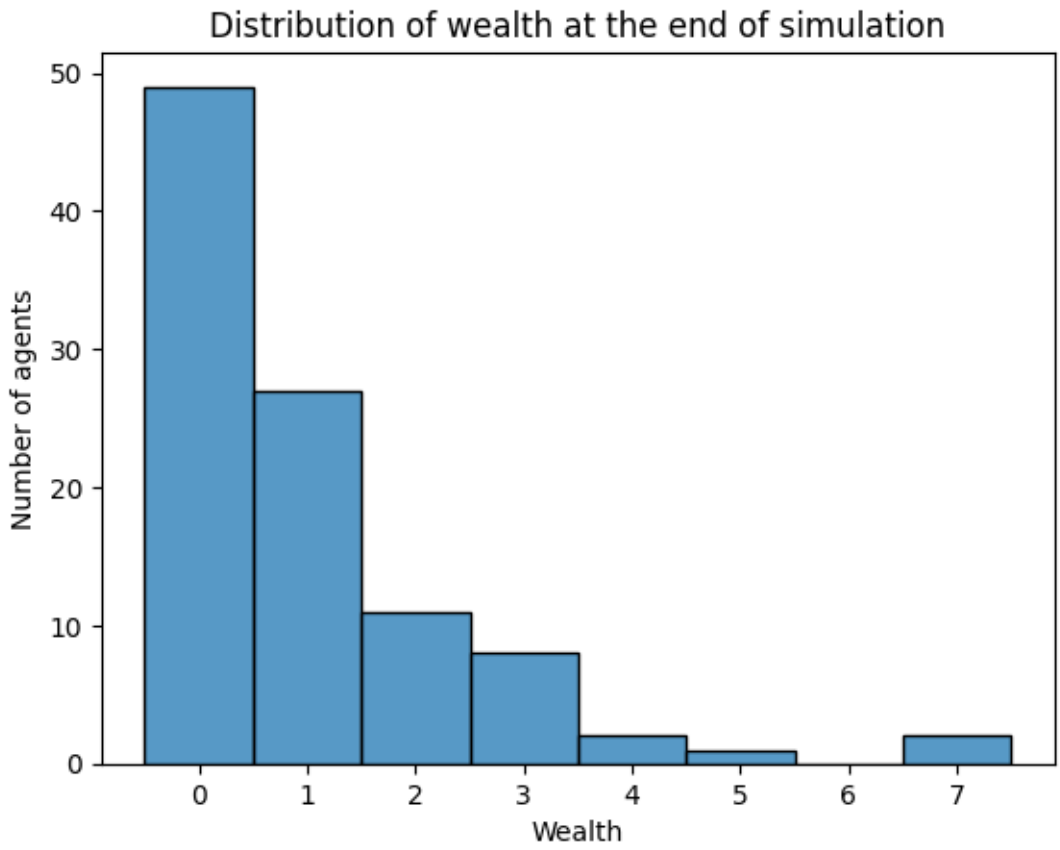
Similarly, we can get the agent-wealth data:

```
agent_wealth = model.datacollector.get_agent_vars_dataframe()
agent_wealth.head()
```

```
              Wealth
Step AgentID
0    0             1
     1             1
     2             1
     3             1
     4             1
```

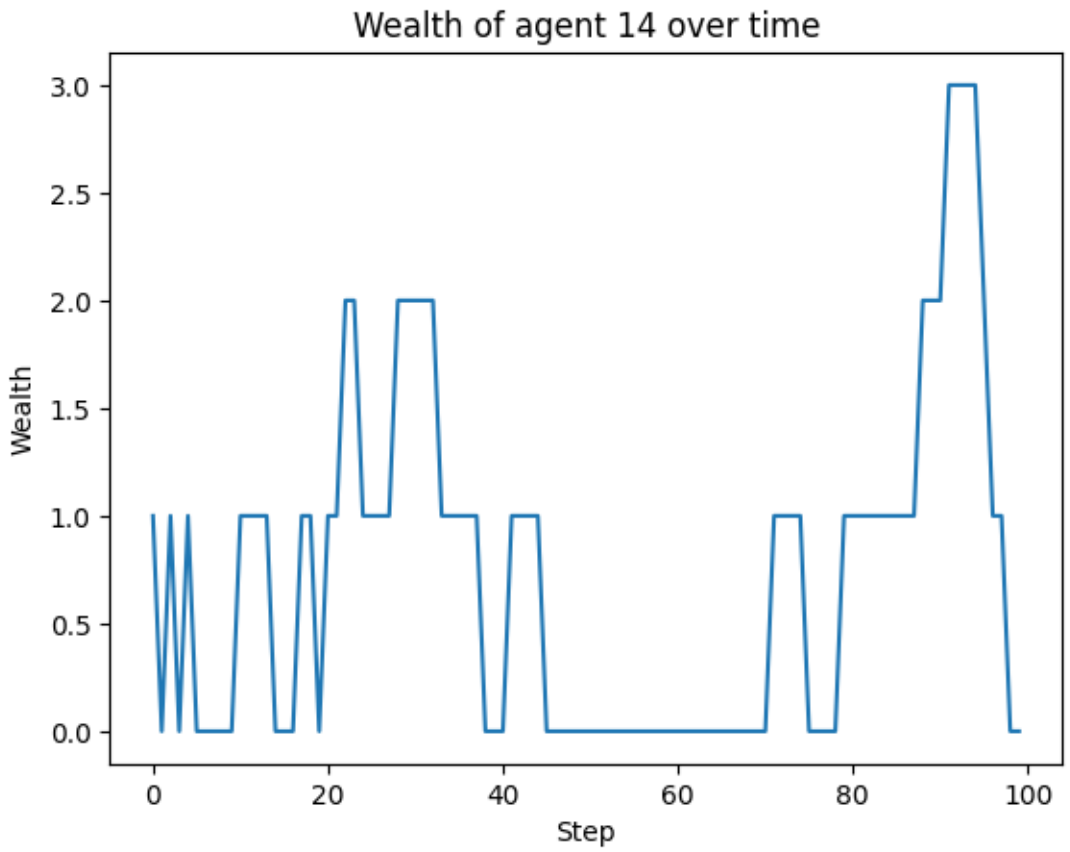
You'll see that the DataFrame's index is pairings of model step and agent ID. This is because the data collector stores the data in a dictionary, with the step number as the key, and a dictionary of agent ID and variable value pairs as the value. The data collector then converts this dictionary into a DataFrame, which is why the index is a pair of (model step, agent ID). You can analyze it the way you would any other DataFrame. For example, to get a histogram of agent wealth at the model's end:

```
last_step = agent_wealth.index.get_level_values("Step").max()
end_wealth = agent_wealth.xs(last_step, level="Step")["Wealth"]
# Create a histogram of wealth at the last step
g = sns.histplot(end_wealth, discrete=True)
g.set(
    title="Distribution of wealth at the end of simulation",
    xlabel="Wealth",
    ylabel="Number of agents",
);
```



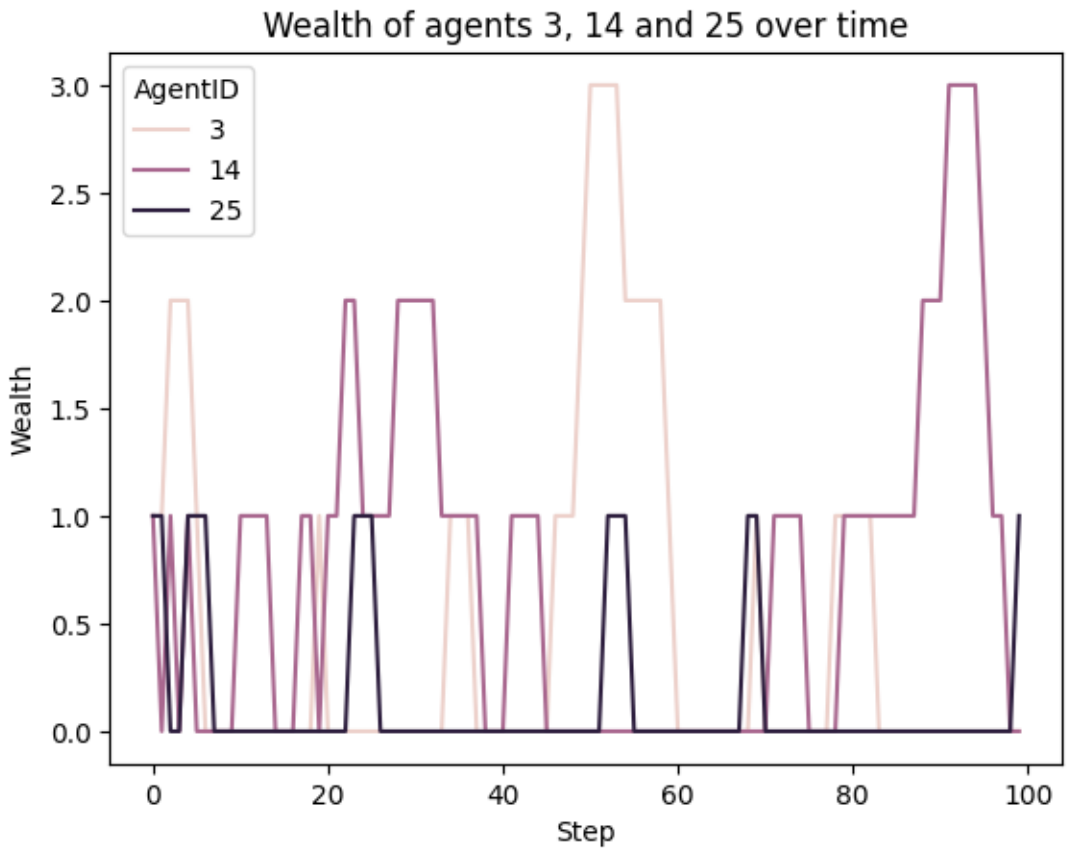
Or to plot the wealth of a given agent (in this example, agent 14):

```
# Get the wealth of agent 14 over time
one_agent_wealth = agent_wealth.xs(14, level="AgentID")

# Plot the wealth of agent 14 over time
g = sns.lineplot(data=one_agent_wealth, x="Step", y="Wealth")
g.set(title="Wealth of agent 14 over time");
```



You can also plot a reporter of multiple agents over time.

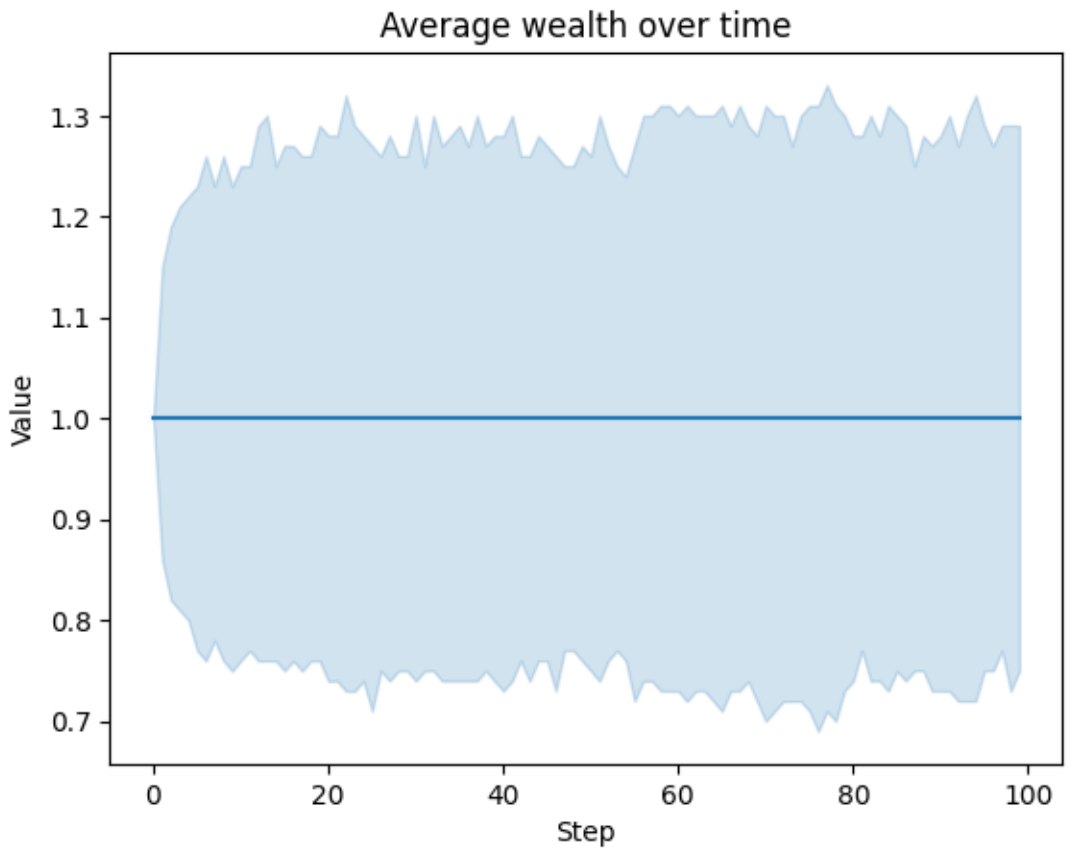
```
agent_list = [3, 14, 25]

# Get the wealth of multiple agents over time
multiple_agents_wealth = agent_wealth[
    agent_wealth.index.get_level_values("AgentID").isin(agent_list)
]
# Plot the wealth of multiple agents over time
g = sns.lineplot(data=multiple_agents_wealth, x="Step", y="Wealth", hue="AgentID")
g.set(title="Wealth of agents 3, 14 and 25 over time");
```



We can also plot the average of all agents, with a 95% confidence interval for that average.

```
# Transform the data to a long format
agent_wealth_long = agent_wealth.T.unstack().reset_index()
agent_wealth_long.columns = ["Step", "AgentID", "Variable", "Value"]
agent_wealth_long.head(3)

# Plot the average wealth over time
g = sns.lineplot(data=agent_wealth_long, x="Step", y="Value", errorbar=("ci", 95))
g.set(title="Average wealth over time")
```

```
[Text(0.5, 1.0, 'Average wealth over time')]
```



Which is exactly 1, as expected in this model, since each agent starts with one wealth unit, and each agent gives one wealth unit to another agent at each step.

You can also use pandas to export the data to a CSV (comma separated value), which can be opened by any common spreadsheet application or opened by pandas.

If you do not specify a file path, the file will be saved in the local directory. After you run the code below you will see two files appear (*model_data.csv* and *agent_data.csv*)

```
# save the model data (stored in the pandas gini object) to CSV
gini.to_csv("model_data.csv")

# save the agent data (stored in the pandas agent_wealth object) to CSV
agent_wealth.to_csv("agent_data.csv")
```

4.2.4.13 Batch Run

Like we mentioned above, you usually won't run a model only once, but multiple times, with fixed parameters to find the overall distributions the model generates, and with varying parameters to analyze how they drive the model's outputs and behaviors. Instead of needing to write nested for-loops for each model, Mesa provides a `batch_run` function which automates it for you.

The batch runner also requires an additional variable `self.running` for the MoneyModel class. This variable enables conditional shut off of the model once a condition is met. In this example it will be set as True indefinitely.

4.2.4.13.1 Additional agent reporter

To make the results a little bit more interesting, we will also calculate the number of consecutive time steps an agent hasn't given any wealth as an agent reporter.

This way we can see how data is handled when multiple reporters are used.

```
def compute_gini(model):
    agent_wealths = [agent.wealth for agent in model.schedule.agents]
    x = sorted(agent_wealths)
    N = model.num_agents
    B = sum(xi * (N - i) for i, xi in enumerate(x)) / (N * sum(x))
    return 1 + (1 / N) - 2 * B


class MoneyModel(mesa.Model):
    """A model with some number of agents."""

    def __init__(self, N, width, height):
        super().__init__()
        self.num_agents = N
        self.grid = mesa.space.MultiGrid(width, height, True)
        self.schedule = mesa.time.RandomActivation(self)
        self.running = True

        # Create agents
        for i in range(self.num_agents):
            a = MoneyAgent(i, self)
            self.schedule.add(a)
            # Add the agent to a random grid cell
            x = self.random.randrange(self.grid.width)
            y = self.random.randrange(self.grid.height)
            self.grid.place_agent(a, (x, y))

        self.datacollector = mesa.DataCollector(
            model_reporters={"Gini": compute_gini},
            agent_reporters={"Wealth": "wealth", "Steps_not_given": "steps_not_given"},
        )

    def step(self):
        self.datacollector.collect(self)
        self.schedule.step()


class MoneyAgent(mesa.Agent):
    """An agent with fixed initial wealth."""

    def __init__(self, unique_id, model):
        super().__init__(unique_id, model)
        self.wealth = 1
        self.steps_not_given = 0

    def move(self):
        possible_steps = self.model.grid.get_neighborhood(
```

(continues on next page)

(continued from previous page)

```
            self.pos, moore=True, include_center=False
        )
        new_position = self.random.choice(possible_steps)
        self.model.grid.move_agent(self, new_position)

    def give_money(self):
        cellmates = self.model.grid.get_cell_list_contents([self.pos])
        if len(cellmates) > 1:
            other = self.random.choice(cellmates)
            other.wealth += 1
            self.wealth -= 1
            self.steps_not_given = 0
        else:
            self.steps_not_given += 1

    def step(self):
        self.move()
        if self.wealth > 0:
            self.give_money()
        else:
            self.steps_not_given += 1
```

4.2.4.13.2 Batch run

We call `batch_run` with the following arguments:

- `model_cls` The model class that is used for the batch run.
- `parameters` A dictionary containing all the parameters of the model class and desired values to use for the batch run as key-value pairs. Each value can either be fixed (e.g. `{"height": 10, "width": 10}`) or an iterable (e.g. `{"N": range(10, 500, 10)}`). `batch_run` will then generate all possible parameter combinations based on this dictionary and run the model `iterations` times for each combination.
- `number_processes` If not specified, defaults to 1. Set it to `None` to use all the available processors. Note: Multiprocessing does make debugging challenging. If your parameter sweeps are resulting in unexpected errors set `number_processes=1`.
- `iterations` The number of iterations to run each parameter combination for. Optional. If not specified, defaults to 1.
- `data_collection_period` The length of the period (number of steps) after which the model and agent reporters collect data. Optional. If not specified, defaults to -1, i.e. only at the end of each episode.
- `max_steps` The maximum number of time steps after which the model halts. An episode does either end when `self.running` of the model class is set to `False` or when `model.schedule.steps == max_steps` is reached. Optional. If not specified, defaults to 1000.
- `display_progress` Display the batch run progress. Optional. If not specified, defaults to `True`.

In the following example, we hold the height and width fixed, and vary the number of agents. We tell the batch runner to run 5 instantiations of the model with each number of agents, and to run each for 100 steps.

We want to keep track of

1. the Gini coefficient value at each time step, and
2. the individual agent's wealth development and steps without giving money.

Important: Since for the latter changes at each time step might be interesting, we set `data_collection_period=1`. By default, it only collects data at the end of each episode.

Note: The total number of runs is 245 (= 49 different populations * 5 iterations per population). However, the resulting list of dictionaries will be of length 6186250 (= 250 average agents per population * 49 different populations * 5 iterations per population * 101 steps per iteration).

Note for Windows OS users: If you are running this tutorial in Jupyter, make sure that you set `number_processes = 1` (single process). If `number_processes` is greater than 1, it is less straightforward to set up. You can read Mesa's how-to guide, in 'Using multi-process `batch_run` on Windows' section for how to do it.

```
params = {"width": 10, "height": 10, "N": range(5, 100, 5)}

results = mesa.batch_run(
    MoneyModel,
    parameters=params,
    iterations=7,
    max_steps=100,
    number_processes=1,
    data_collection_period=1,
    display_progress=True,
)
```

```
  0%|          | 0/133 [00:00<?, ?it/s]
```

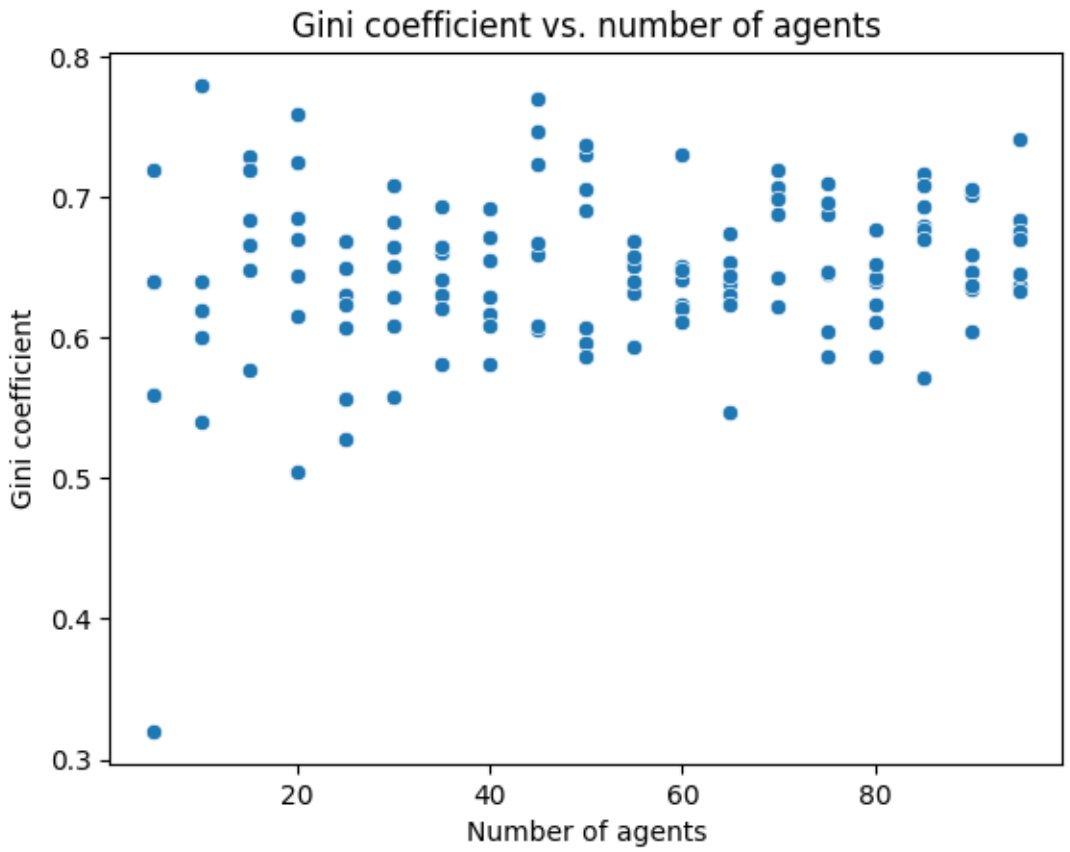
To further analyze the return of the `batch_run` function, we convert the list of dictionaries to a Pandas DataFrame and print its keys.

```
results_df = pd.DataFrame(results)
print(results_df.keys())
```

```
Index(['RunId', 'iteration', 'Step', 'width', 'height', 'N', 'Gini', 'AgentID',
       'Wealth', 'Steps_not_given'],
      dtype='object')
```

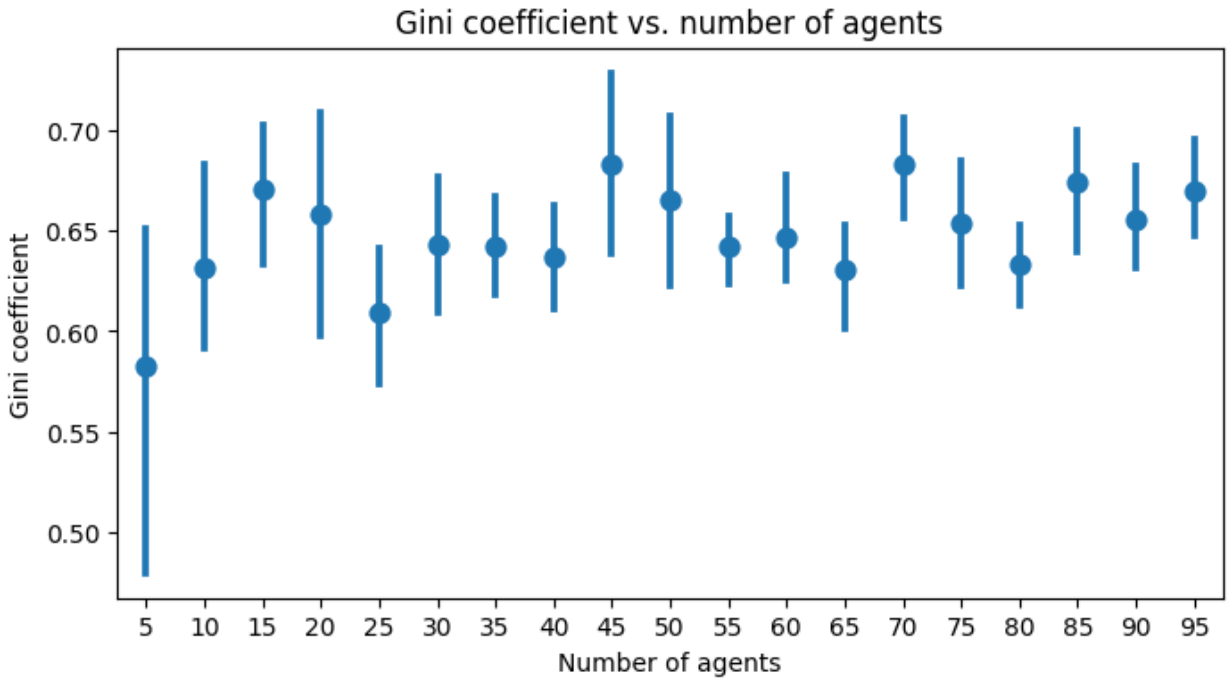
First, we want to take a closer look at how the Gini coefficient at the end of each episode changes as we increase the size of the population. For this, we filter our results to only contain the data of one agent (the Gini coefficient will be the same for the entire population at any time) at the 100th step of each episode and then scatter-plot the values for the Gini coefficient over the the number of agents. Notice there are five values for each population size since we set `iterations=5` when calling the batch run.

```
# Filter the results to only contain the data of one agent (the Gini coefficient will be␣
↪the same for the entire population at any time) at the 100th step of each episode
results_filtered = results_df[(results_df.AgentID == 0) & (results_df.Step == 100)]
results_filtered[["iteration", "N", "Gini"]].reset_index(
    drop=True
).head()  # Create a scatter plot
g = sns.scatterplot(data=results_filtered, x="N", y="Gini")
g.set(
    xlabel="Number of agents",
    ylabel="Gini coefficient",
    title="Gini coefficient vs. number of agents",
);
```



We can create different kinds of plot from this filtered DataFrame. For example, a point plot with error bars.

```
# Create a point plot with error bars
g = sns.pointplot(data=results_filtered, x="N", y="Gini", linestyle='none')
g.figure.set_size_inches(8, 4)
g.set(
    xlabel="Number of agents",
    ylabel="Gini coefficient",
    title="Gini coefficient vs. number of agents",
);
```



Second, we want to display the agent's wealth at each time step of one specific episode. To do this, we again filter our large data frame, this time with a fixed number of agents and only for a specific iteration of that population. To print the results, we convert the filtered data frame to a string specifying the desired columns to print.

Pandas has built-in functions to convert to a lot of different data formats. For example, to display as a table in a Jupyter Notebook, we can use the `to_html()` function which takes the same arguments as `to_string()` (see commented lines).

```
# First, we filter the results
one_episode_wealth = results_df[(results_df.N == 10) & (results_df.iteration == 2)]
# Then, print the columns of interest of the filtered data frame
print(
    one_episode_wealth.to_string(
        index=False, columns=["Step", "AgentID", "Wealth"], max_rows=10
    )
)
# For a prettier display we can also convert the data frame to html, uncomment to test␣
↪in a Jupyter Notebook
# from IPython.display import display, HTML
# display(HTML(one_episode_wealth.to_html(index=False, columns=['Step', 'AgentID', 'Wealth'],
↪ max_rows=25)))
```

```
 Step  AgentID  Wealth
    0        0       1
    0        1       1
    0        2       1
    0        3       1
    0        4       1
  ...      ...     ...
  100        0       3
  100        2       0
```

(continues on next page)

(continued from previous page)

```
  100        4        2
  100        8        1
  100        6        0
```

Lastly, we want to take a look at the development of the Gini coefficient over the course of one iteration. Filtering and printing looks almost the same as above, only this time we choose a different episode.

```
results_one_episode = results_df[
    (results_df.N == 10) & (results_df.iteration == 1) & (results_df.AgentID == 0)
]
print(results_one_episode.to_string(index=False, columns=["Step", "Gini"], max_rows=10))
```

```
 Step  Gini
    0  0.00
    1  0.00
    2  0.00
    3  0.00
    4  0.00
  ...   ...
   96  0.62
   97  0.62
   98  0.62
   99  0.62
  100  0.62
```

4.2.4.14 Analyzing model reporters: Comparing 5 scenarios

Other insights might be gathered when we compare the Gini coefficient of different scenarios. For example, we can compare the Gini coefficient of a population with 25 agents to the Gini coefficient of a population with 400 agents. While doing this, we increase the number of iterations to 25 to get a better estimate of the Gini coefficient for each population size and get usable error estimations.

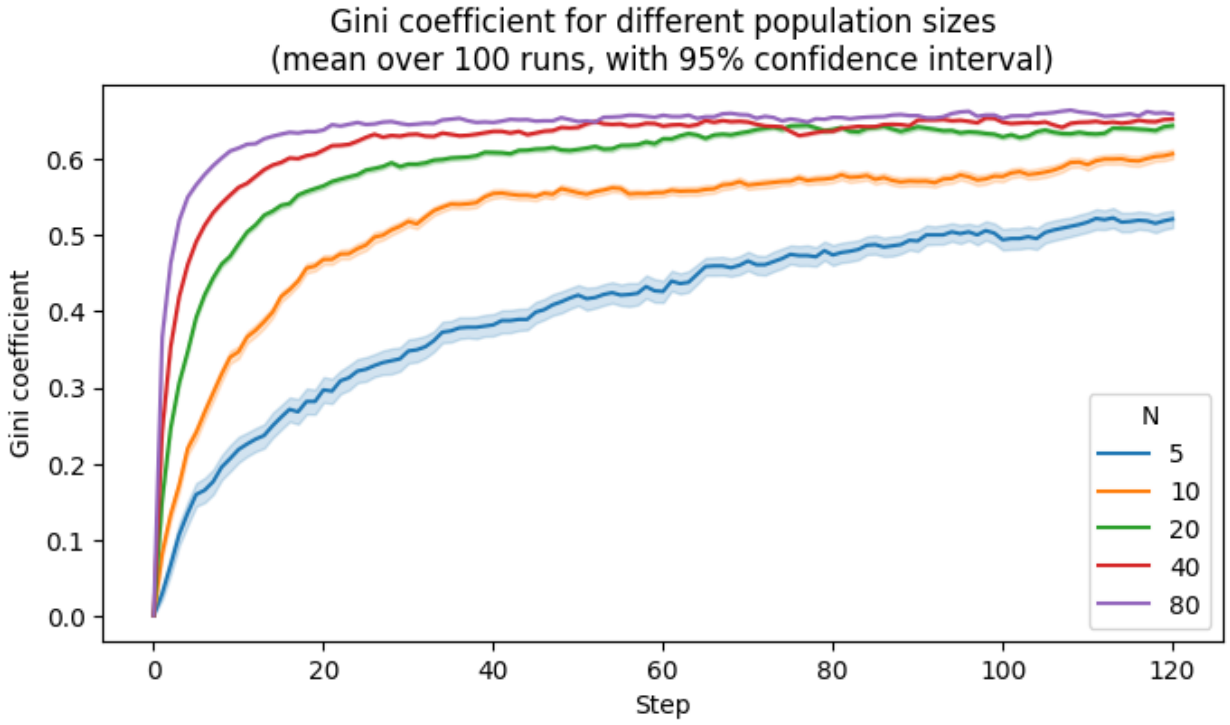
```
params = {"width": 10, "height": 10, "N": [5, 10, 20, 40, 80]}

results_5s = mesa.batch_run(
    MoneyModel,
    parameters=params,
    iterations=100,
    max_steps=120,
    number_processes=1,
    data_collection_period=1,  # Important, otherwise the datacollector will only␣
↪collect data of the last time step
    display_progress=True,
)

results_5s_df = pd.DataFrame(results_5s)
```

```
  0%|          | 0/500 [00:00<?, ?it/s]
```

```
# Again filter the results to only contain the data of one agent (the Gini coefficient␣
↪will be the same for the entire population at any time)
results_5s_df_filtered = results_5s_df[(results_5s_df.AgentID == 0)]
results_5s_df_filtered.head(3)
```

```
    RunId  iteration  Step  width  height  N  Gini  AgentID  Wealth  \
0       0          0     0     10      10  5   0.0        0       1
6       0          0     1     10      10  5   0.0        0       1
11      0          0     2     10      10  5   0.0        0       1

    Steps_not_given
0                 0
6                 1
11                2
```

```
# Create a lineplot with error bars
g = sns.lineplot(
    data=results_5s_df,
    x="Step",
    y="Gini",
    hue="N",
    errorbar=("ci", 95),
    palette="tab10",
)
g.figure.set_size_inches(8, 4)
plot_title = "Gini coefficient for different population sizes\n(mean over 100 runs, with␣
↪95% confidence interval)"
g.set(title=plot_title, ylabel="Gini coefficient");
```



In this case it looks like the Gini coefficient increases slower for smaller populations. This can be because of different things, either because the Gini coefficient is a measure of inequality and the smaller the population, the more likely it is that the agents are all in the same wealth class, or because there are less interactions between agents in smaller populations, which means that the wealth of an agent is less likely to change.

4.2.4.15 Analyzing agent reporters

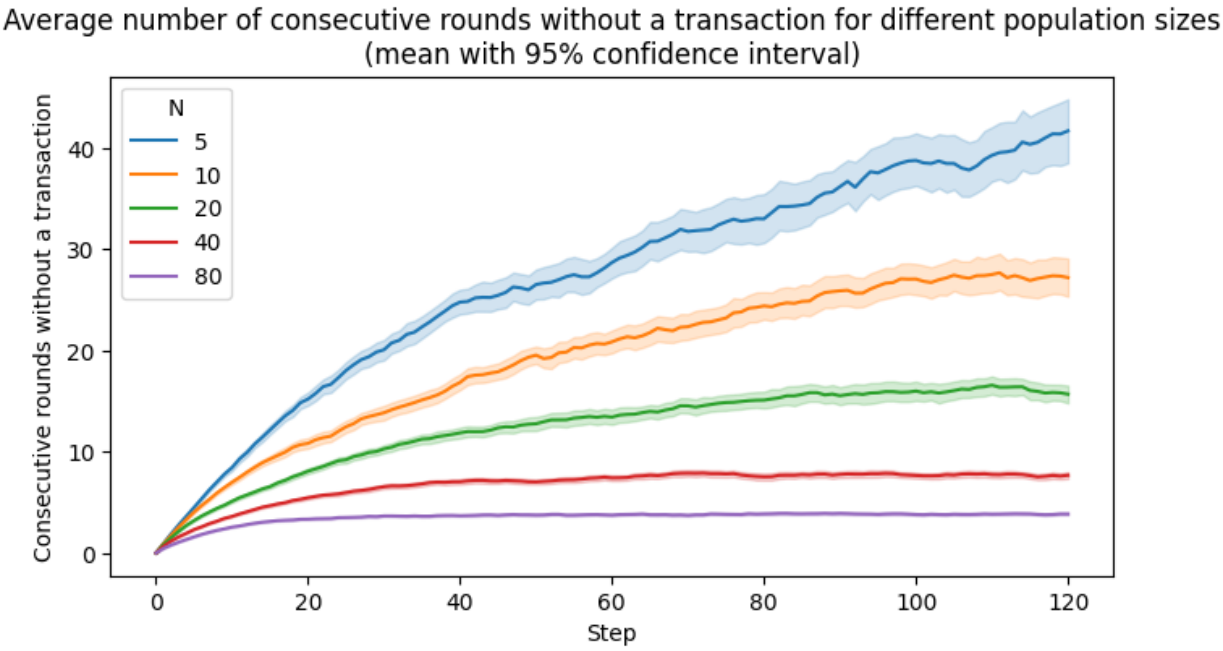
From the agents we collected the wealth and the number of consecutive rounds without a transaction. We can compare the 5 different population sizes by plotting the average number of consecutive rounds without a transaction for each population size.

Note that we're aggregating multiple times here: First we take the average of all agents for each single replication. Then we plot the averages for all replications, with the error band showing the 95% confidence interval of that first average (over all agents). So this error band is representing the uncertainty of the mean value of the number of consecutive rounds without a transaction for each population size.

```
# Calculate the mean of the wealth and the number of consecutive rounds for all agents␣
↪in each episode
agg_results_df = (
    results_5s_df.groupby(["iteration", "N", "Step"])
    .agg({"Wealth": "mean", "Steps_not_given": "mean"})
    .reset_index()
)
agg_results_df.head(3)
```

	iteration	N	Step	Wealth	Steps_not_given
0	0	5	0	1.0	0.0
1	0	5	1	1.0	1.0
2	0	5	2	1.0	2.0

```
# Create a line plot with error bars
g = sns.lineplot(
    data=agg_results_df, x="Step", y="Steps_not_given", hue="N", palette="tab10"
)
g.figure.set_size_inches(8, 4)
g.set(
    title="Average number of consecutive rounds without a transaction for different␣
↪population sizes\n(mean with 95% confidence interval)",
    ylabel="Consecutive rounds without a transaction",
);
```



It can be clearly seen that the lower the number of agents, the higher the number of consecutive rounds without a transaction. This is because the agents have fewer interactions with each other and therefore the wealth of an agent is less likely to change.

4.2.4.15.1 General steps for analyzing results

Many other analysis are possible based on the policies, scenarios and uncertainties that you might be interested in. In general, you can follow these steps to do your own analysis:

1. Determine which metrics you want to analyse. Add these as model and agent reporters to the datacollector of your model.
2. Determine the input parameters you want to vary. Add these as parameters to the batch_run function, using ranges or lists to test different values.
3. Determine the hyperparameters of the batch_run function. Define the number of iterations, the number of processes, the number of steps, the data collection period, etc.
4. Run the batch_run function and save the results.
5. Transform, filter and aggregate the results to get the data you want to analyze. Make sure it's in long format, so that each row represents a single value.
6. Choose a plot type, what to plot on the x and y axis, which columns to use for the hue. Seaborn also has an amazing Example Gallery.
7. Plot the data and analyze the results.

4.2.4.16 Happy Modeling!

This document is a work in progress. If you see any errors, exclusions or have any problems please contact us.

[Comer2014] Comer, Kenneth W. "Who Goes First? An Examination of the Impact of Activation on Outcome Behavior in AgentBased Models." George Mason University, 2014. http://mars.gmu.edu/bitstream/handle/1920/9070/Comer_gmu_0883E_10539.pdf

[Dragulescu2002] Drăgulescu, Adrian A., and Victor M. Yakovenko. "Statistical Mechanics of Money, Income, and Wealth: A Short Survey." arXiv Preprint Cond-mat/0211175, 2002. http://arxiv.org/abs/cond-mat/0211175.

4.3 Visualization Tutorial

Important:

- If you are just exploring Mesa and want the fastest way to execute the code we recommend executing this tutorial online in a Colab notebook.
- If you have installed mesa and are running locally, please ensure that your Mesa version is up-to-date in order to run this tutorial.

4.3.1 Adding visualization

So far, we've built a model, run it, and analyzed some output afterwards. However, one of the advantages of agent-based models is that we can often watch them run step by step, potentially spotting unexpected patterns, behaviors or bugs, or developing new intuitions, hypotheses, or insights. Other times, watching a model run can explain it to an unfamiliar audience better than static explanations. Like many ABM frameworks, Mesa allows you to create an interactive visualization of the model. In this section we'll walk through creating a visualization using built-in components, and (for advanced users) how to create a new visualization element.

First, a quick explanation of how Mesa's interactive visualization works. The visualization is done in a browser window, using the Solara framework, a pure Python, React-style web framework. Running `solara run app.py` will launch a web server, which runs the model, and displays model detail at each step via the Matplotlib plotting library. Alternatively, you can execute everything inside a Jupyter environment.

4.3.1.1 Grid Visualization

To start with, let's have a visualization where we can watch the agents moving around the grid. Let us use the same `MoneyModel` created in the Introductory Tutorial.

```
!pip install --quiet mesa
import mesa

# You can either define the BoltzmannWealthModel (aka MoneyModel) or install mesa-models:
!pip install --quiet -U git+https://github.com/projectmesa/mesa-examples#egg=mesa-models

from mesa_models.boltzmann_wealth_model.model import BoltzmannWealthModel
```

```
Note: you may need to restart the kernel to use updated packages.
```

```
Note: you may need to restart the kernel to use updated packages.
```

Mesa's grid visualizer works by looping over every cell in a grid, and generating a portrayal for every agent it finds. A portrayal is a dictionary (which can easily be turned into a JSON object) which tells Matplotlib the color and size of the scatterplot markers (each signifying an agent). The only thing we need to provide is a function which takes an agent, and returns a portrayal dictionary. Here's the simplest one: it'll draw each agent as a blue, filled circle, with a radius size of 50.

```
def agent_portrayal(agent):
    return {
        "color": "tab:blue",
        "size": 50,
    }
```

In addition to the portrayal method, we instantiate the model parameters, some of which are modifiable by user inputs. In this case, the number of agents, N, is specified as a slider of integers.

```
model_params = {
    "N": {
        "type": "SliderInt",
        "value": 50,
        "label": "Number of agents:",
        "min": 10,
        "max": 100,
        "step": 1,
    },
    "width": 10,
    "height": 10,
}
```

Next, we instantiate the visualization object which (by default) displays the grid containing the agents, and timeseries of of values computed by the model's data collector. In this example, we specify the Gini coefficient.

There are 3 buttons:

- the step button, which advances the model by 1 step
- the play button, which advances the model indefinitely until it is paused, or until `model.running` is False (you may specify the stopping condition)
- the pause button, which pauses the model

To reset the model, simply change the model parameter from the user input (e.g. the "Number of agents" slider).

```
from mesa.experimental import JupyterViz

page = JupyterViz(
    BoltzmannWealthModel,
    model_params,
    measures=["Gini"],
    name="Money Model",
    agent_portrayal=agent_portrayal,
)
# This is required to render the visualization in the Jupyter notebook
page
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```

```
Cannot show ipywidgets in text
```

4.3.1.2 Changing the agents

In the visualization above, all we could see is the agents moving around – but not how much money they had, or anything else of interest. Let's change it so that agents who are broke (wealth 0) are drawn in red, smaller. (TODO: currently, we can't predict the drawing order of the circles, so a broke agent may be overshadowed by a wealthy agent. We should fix this by doing a hollow circle instead)

To do this, we go back to our `agent_portrayal` code and add some code to change the portrayal based on the agent properties and launch the server again.

```
def agent_portrayal(agent):
    size = 10
    color = "tab:red"
    if agent.wealth > 0:
        size = 50
        color = "tab:blue"
    return {"size": size, "color": color}
```

```
page = JupyterViz(
    BoltzmannWealthModel,
    model_params,
    measures=["Gini"],
    name="Money Model",
    agent_portrayal=agent_portrayal,
)
# This is required to render the visualization in the Jupyter notebook
page
```

```
Cannot show ipywidgets in text
```

4.3.2 Building your own visualization component

Note: This section is for users who have a basic familiarity with Python's Matplotlib plotting library.

If the visualization elements provided by Mesa aren't enough for you, you can build your own and plug them into the model server.

For this example, let's build a simple histogram visualization, which can count the number of agents with each value of wealth.

```
import solara
from matplotlib.figure import Figure


def make_histogram(model):
    # Note: you must initialize a figure using this method instead of
    # plt.figure(), for thread safety purpose
```

(continues on next page)

(continued from previous page)

```
    fig = Figure()
    ax = fig.subplots()
    wealth_vals = [agent.wealth for agent in model.schedule.agents]
    # Note: you have to use Matplotlib's OOP API instead of plt.hist
    # because plt.hist is not thread-safe.
    ax.hist(wealth_vals, bins=10)
    solara.FigureMatplotlib(fig)
```

Next, we reinitialize the visualization object, but this time with the histogram (see the measures argument).

```
page = JupyterViz(
    BoltzmannWealthModel,
    model_params,
    measures=["Gini", make_histogram],
    name="Money Model",
    agent_portrayal=agent_portrayal,
)
# This is required to render the visualization in the Jupyter notebook
page
```

```
Cannot show ipywidgets in text
```

4.3.3 Happy Modeling!

This document is a work in progress. If you see any errors, exclusions or have any problems please contact us.

4.4 Best Practices

Here are some general principles that have proven helpful for developing models.

4.4.1 Model Layout

A model should be contained in a folder named with lower-case letters and underscores, such as `thunder_cats`. Within that directory:

- `README.md` describes the model, how to use it, and any other details. Github will automatically show this file to anyone visiting the directory.
- `model.py` should contain the model class. If the file gets large, it may make sense to move the complex bits into other files, but this is the first place readers will look to figure out how the model works.
- `server.py` should contain the visualization support, including the server class.
- `run.py` is a Python script that will run the model when invoked via `mesa runserver`.

After the number of files grows beyond a half-dozen, try to use sub-folders to organize them. For example, if the visualization uses image files, put those in an `images` directory.

The Schelling model is a good example of a small well-packaged model.

It's easy to create a cookiecutter mesa model by running `mesa startproject`

4.4.2 Randomization

If your model involves some random choice, you can use the built-in `random` property that Mesa `Model` and `Agent` objects have. This works exactly like the built-in `random` library.

```
class AwesomeModel(Model):
  # ...

  def cool_method(self):
    interesting_number = self.random.random()
    print(interesting_number)

class AwesomeAgent(Agent):
  # ...
  def __init__(self, unique_id, model, ...):
    super().__init__(unique_id, model)
    # ...

  def my_method(self):
    random_number = self.random.randint(0, 100)
```

(The agent's random property is just a reference to its parent model's `random` property).

When a model object is created, its random property is automatically seeded with the current time. The seed determines the sequence of random numbers; if you instantiate a model with the same seed, you will get the same results. To allow you to set the seed, make sure your model has a `seed` argument in its constructor.

```
class AwesomeModel(Model):

  def __init__(self, seed=None):
    pass

  def cool_method(self):
    interesting_number = self.random.random()
    print(interesting_number)

>>> model0 = AwesomeModel(seed=0)
>>> model0._seed
0
>>> model0.cool_method()
0.8444218515250481
>>> model1 = AwesomeModel(seed=0)
>>> model1.cool_method()
0.8444218515250481
```

4.5 How-to Guide

This guide provides concise instructions and examples to help you start with common tasks in Mesa.

4.5.1 Models with Discrete Time

For models involving agents of multiple types, including those with a time attribute, you can construct a discrete-time model. This setup allows each agent to perform actions in steps that correspond to the model's discrete time.

Example:

```
if self.model.schedule.time in self.discrete_time:
    self.model.space.move_agent(self, new_pos)
```

4.5.2 Implementing Model-Level Functions with Staged Activation

In staged activation scenarios, to designate functions that should only operate at the model level (and not at the agent level), prepend the function name with "model." in the function list definition. This approach is useful for model-wide operations, like adjusting a wage rate based on demand.

Example:

```
stage_list = [Send_Labour_Supply, Send_Labour_Demand, model.Adjust_Wage_Rate]
self.schedule = StagedActivation(self, stage_list, shuffle=True)
```

4.5.3 Using `numpy.random`

To incorporate `numpy`'s random functions, such as for generating a Poisson distribution, initialize a random number generator in your model's constructor.

Example:

```
import mesa
import numpy as np

class MyModel(mesa.Model):
    def __init__(self, seed=None):
        super().__init__()
        self.random = np.random.default_rng(seed)
```

Usage example:

```
lambda_value = 5
sample = self.random.poisson(lambda_value)
```

4.5.4 Multi-process `batch_run` on Windows

When using `batch_run` with `number_processes = None` on Windows, you might encounter progress display issues or `AttributeError: Can't get attribute 'MoneyModel' on <module '__main__' (built-in)>`. To resolve this, run your code outside of Jupyter notebooks and use the following pattern, incorporating `freeze_support()` for multiprocessing support.

Example:

```
from mesa.batchrunner import batch_run
from multiprocessing import freeze_support

params = {"width": 10, "height": 10, "N": range(10, 500, 10)}

if __name__ == '__main__':
    freeze_support()
    results = batch_run(
        MoneyModel,
        parameters=params,
        iterations=5,
        max_steps=100,
        number_processes=None,
        data_collection_period=1,
        display_progress=True,
    )
```

If you would still like to run your code in Jupyter, adjust your code as described and consider integrating external libraries like nbmultitask or refer to Stack Overflow for multiprocessing tips.

4.6 APIs

4.6.1 Base Classes

class Agent(*unique_id: int*, *model: Model*)

Base class for a model agent in Mesa.

Attributes:

unique_id (int): A unique identifier for this agent. model (Model): A reference to the model instance. self.pos: Position | None = None

Create a new agent.

Args:

unique_id (int): A unique identifier for this agent. model (Model): The model instance in which the agent exists.

remove() → None

Remove and delete the agent from the model.

step() → None

A single step of the agent.

class Model(**args: Any*, ***kwargs: Any*)

Base class for models in the Mesa ABM library.

This class serves as a foundational structure for creating agent-based models. It includes the basic attributes and methods necessary for initializing and running a simulation model.

Attributes:

running: A boolean indicating if the model should continue running. schedule: An object to manage the order and execution of agent steps. current_id: A counter for assigning unique IDs to agents. **agents_**: A defaultdict mapping each agent type to a dict of its instances.

This private attribute is used internally to manage agents.

Properties:

agents: An AgentSet containing all agents in the model, generated from the _agents attribute. agent_types: A list of different agent types present in the model.

Methods:

get_agents_of_type: Returns an AgentSet of agents of the specified type. run_model: Runs the model's simulation until a defined end condition is reached. step: Executes a single step of the model's simulation process. next_id: Generates and returns the next unique identifier for an agent. reset_randomizer: Resets the model's random number generator with a new or existing seed. initialize_data_collector: Sets up the data collector for the model, requiring an initialized scheduler and agents.

Create a new model. Overload this method with the actual code to start the model. Always start with super().__init__() to initialize the model object properly.

property agents: AgentSet

Provides an AgentSet of all agents in the model, combining agents from all types.

property agent_types: list[type]

Return a list of different agent types.

get_agents_of_type(*agenttype: type[Agent]*) → AgentSet

Retrieves an AgentSet containing all agents of the specified type.

run_model() → None

Run the model until the end condition is reached. Overload as needed.

step() → None

A single step. Fill in here.

next_id() → int

Return the next unique ID for agents, increment current_id

reset_randomizer(*seed: int | None = None*) → None

Reset the model random number generator.

Args:

seed: A new seed for the RNG; if None, reset using the current seed

4.6.2 Mesa Time Module

Objects for handling the time component of a model. In particular, this module contains Schedulers, which handle agent activation. A Scheduler is an object which controls when agents are called upon to act, and when.

The activation order can have a serious impact on model behavior, so it's important to specify it explicitly. Example simple activation regimes include activating all agents in the same order every step, shuffling the activation order every time, activating each agent *on average* once per step, and more.

Key concepts:

Step: Many models advance in 'steps'. A step may involve the activation of all agents, or a random (or selected) subset of them. Each agent in turn may have their own step() method.

Time: Some models may simulate a continuous 'clock' instead of discrete steps. However, by default, the Time is equal to the number of steps the model has taken.

`class BaseScheduler`(*model: Model*, *agents: Iterable[Agent] | None = None*)

A simple scheduler that activates agents one at a time, in the order they were added.

This scheduler is designed to replicate the behavior of the scheduler in MASON, a multi-agent simulation toolkit. It assumes that each agent added has a *step* method which takes no arguments and executes the agent's actions.

Attributes:

- model (Model): The model instance associated with the scheduler.
- steps (int): The number of steps the scheduler has taken.
- time (TimeT): The current time in the simulation. Can be an integer or a float.

Methods:

- add: Adds an agent to the scheduler.
- remove: Removes an agent from the scheduler.
- step: Executes a step, which involves activating each agent once.
- get_agent_count: Returns the number of agents in the scheduler.
- agents (property): Returns a list of all agent instances.

Create a new BaseScheduler.

Args:

model (Model): The model to which the schedule belongs agents (Iterable[Agent], None, optional): An iterable of agents who are controlled by the schedule

`add`(*agent: Agent*) → None

Add an Agent object to the schedule.

Args:

agent: An Agent to be added to the schedule. NOTE: The agent must have a step() method.

`remove`(*agent: Agent*) → None

Remove all instances of a given agent from the schedule.

Note:

It is only necessary to explicitly remove agents from the schedule if the agent is not removed from the model.

Args:

agent: An agent object.

`step`() → None

Execute the step of all the agents, one at a time.

`get_agent_count`() → int

Returns the current number of agents in the queue.

class RandomActivation(*model: Model*, *agents: Iterable[Agent] | None = None*)

A scheduler that activates each agent once per step, in a random order, with the order reshuffled each step.

This scheduler is equivalent to the NetLogo 'ask agents...' behavior and is a common default for ABMs. It assumes that all agents have a *step* method.

The random activation ensures that no single agent or sequence of agents consistently influences the model due to ordering effects, which is crucial for certain types of simulations.

Inherits all attributes and methods from BaseScheduler.

Methods:

- step: Executes a step, activating each agent in a random order.

Create a new BaseScheduler.

Args:
model (Model): The model to which the schedule belongs agents (Iterable[Agent], None, optional): An iterable of agents who are controlled by the schedule

step() → None

Executes the step of all agents, one at a time, in random order.

add(*agent: Agent*) → None

Add an Agent object to the schedule.

Args:
agent: An Agent to be added to the schedule. NOTE: The agent must have a step() method.

get_agent_count() → int

Returns the current number of agents in the queue.

remove(*agent: Agent*) → None

Remove all instances of a given agent from the schedule.

Note:
It is only necessary to explicitly remove agents from the schedule if the agent is not removed from the model.

Args:
agent: An agent object.

class SimultaneousActivation(*model: Model*, *agents: Iterable[Agent] | None = None*)

A scheduler that simulates the simultaneous activation of all agents.

This scheduler is unique in that it requires agents to have both *step* and *advance* methods. - The *step* method is for activating the agent and staging any changes without applying them immediately. - The *advance* method then applies these changes, simulating simultaneous action.

This scheduler is useful in scenarios where the interactions between agents are sensitive to the order of execution, and a quasi-simultaneous execution is more realistic.

Inherits all attributes and methods from BaseScheduler.

Methods:

- step: Executes a step for all agents, first calling *step* then *advance* on each.

Create a new BaseScheduler.

Args:
model (Model): The model to which the schedule belongs agents (Iterable[Agent], None, optional): An iterable of agents who are controlled by the schedule

step() → None

Step all agents, then advance them.

add(*agent: Agent*) → None

Add an Agent object to the schedule.

Args:
agent: An Agent to be added to the schedule. NOTE: The agent must have a step() method.

get_agent_count() → int

Returns the current number of agents in the queue.

remove(*agent: Agent*) → None

Remove all instances of a given agent from the schedule.

Note:
It is only necessary to explicitly remove agents from the schedule if the agent is not removed from the model.

Args:
agent: An agent object.

class StagedActivation(*model: Model*, *agents: Iterable[Agent] | None = None*, *stage_list: list[str] | None = None*, *shuffle: bool = False*, *shuffle_between_stages: bool = False*)

A scheduler allowing agent activation to be divided into several stages, with all agents executing one stage before moving on to the next. This class is a generalization of SimultaneousActivation.

This scheduler is useful for complex models where actions need to be broken down into distinct phases for each agent in each time step. Agents must implement methods for each defined stage.

The scheduler also tracks steps and time separately, allowing fractional time increments based on the number of stages. Time advances in fractional increments of 1 / (# of stages), meaning that 1 step = 1 unit of time.

Inherits all attributes and methods from BaseScheduler.

Attributes:

- stage_list (list[str]): A list of stage names that define the order of execution.
- shuffle (bool): Determines whether to shuffle the order of agents each step.
- shuffle_between_stages (bool): Determines whether to shuffle agents between each stage.

Methods:

- step: Executes all the stages for all agents in the defined order.

Create an empty Staged Activation schedule.

Args:
model (Model): The model to which the schedule belongs agents (Iterable[Agent], None, optional): An iterable of agents who are controlled by the schedule stage_list (`list` of `str`): List of strings of names of stages to run, in the

order to run them in.

shuffle (bool, optional): If True, shuffle the order of agents each step. shuffle_between_stages (bool, optional): If True, shuffle the agents after each

stage; otherwise, only shuffle at the start of each step.

step() → None

Executes all the stages for all agents.

add(*agent: Agent*) → None

Add an Agent object to the schedule.

Args:
agent: An Agent to be added to the schedule. NOTE: The agent must have a step() method.

get_agent_count() → int

Returns the current number of agents in the queue.

remove(*agent: Agent*) → None

Remove all instances of a given agent from the schedule.

Note:
It is only necessary to explicitly remove agents from the schedule if the agent is not removed from the model.

Args:
agent: An agent object.

class RandomActivationByType(*model: Model*, *agents: Iterable[Agent] | None = None*)

A scheduler that activates each type of agent once per step, in random order, with the order reshuffled every step.

This scheduler is useful for models with multiple types of agents, ensuring that each type is treated equitably in terms of activation order. The randomness in activation order helps in reducing biases due to ordering effects.

Inherits all attributes and methods from BaseScheduler.

If you want to do some computations / data collections specific to an agent type, you can either: - loop through all agents, and filter by their type - access via *your_model.scheduler.agents_by_type[your_type_class]*

Attributes:

- agents_by_type (defaultdict): A dictionary mapping agent types to dictionaries of agents.

Methods:

- step: Executes the step of each agent type in a random order.
- step_type: Activates all agents of a given type.
- get_type_count: Returns the count of agents of a specific type.

Create a new BaseScheduler.

Args:
model (Model): The model to which the schedule belongs agents (Iterable[Agent], None, optional): An iterable of agents who are controlled by the schedule

add(*agent: Agent*) → None

Add an Agent object to the schedule

Args:
agent: An Agent to be added to the schedule.

remove(*agent: Agent*) → None

Remove all instances of a given agent from the schedule.

step(*shuffle_types: bool = True*, *shuffle_agents: bool = True*) → None

Executes the step of each agent type, one at a time, in random order.

Args:

shuffle_types: If True, the order of execution of each types is
shuffled.

shuffle_agents: If True, the order of execution of each agents in a
type group is shuffled.

`step_type`(*agenttype: type[Agent]*, *shuffle_agents: bool = True*) → None

Shuffle order and run all agents of a given type. This method is equivalent to the NetLogo 'ask [breed]...'.

Args:
agenttype: Class object of the type to run.

`get_type_count`(*agenttype: type[Agent]*) → int

Returns the current number of agents of certain type in the queue.

`get_agent_count`() → int

Returns the current number of agents in the queue.

`class DiscreteEventScheduler`(*model: Model*, *time_step: float | int = 1*)

This class has been deprecated and replaced by the functionality provided by experimental.devs

Args:
model (Model): The model to which the schedule belongs time_step (TimeT): The fixed time step between steps

`add`(*agent: Agent*) → None

Add an Agent object to the schedule.

Args:
agent: An Agent to be added to the schedule. NOTE: The agent must have a step() method.

`get_agent_count`() → int

Returns the current number of agents in the queue.

`remove`(*agent: Agent*) → None

Remove all instances of a given agent from the schedule.

Note:
It is only necessary to explicitly remove agents from the schedule if the agent is not removed from the model.

Args:
agent: An agent object.

`step`() → None

Execute the step of all the agents, one at a time.

4.6.3 Mesa Space Module

Objects used to add a spatial component to a model.

Grid: base grid, which creates a rectangular grid. SingleGrid: extension to Grid which strictly enforces one agent per cell. MultiGrid: extension to Grid where each cell can contain a set of agents. HexGrid: extension to Grid to handle hexagonal neighbors. ContinuousSpace: a two-dimensional space where each agent has an arbitrary

position of *float*'s.

NetworkGrid: a network where each node contains zero or more agents.

`accept_tuple_argument`(*wrapped_function: F*) → F

Decorator to allow grid methods that take a list of (x, y) coord tuples to also handle a single position, by automatically wrapping tuple in single-item list rather than forcing user to do it.

`is_single_argument_function`(*function*)

Check if a function is a single argument function.

`class PropertyLayer`(*name: str*, *width: int*, *height: int*, *default_value*, *dtype=<class 'numpy.float64'>*)

A class representing a layer of properties in a two-dimensional grid. Each cell in the grid can store a value of a specified data type.

Attributes:

name (str): The name of the property layer. width (int): The width of the grid (number of columns). height (int): The height of the grid (number of rows). data (numpy.ndarray): A NumPy array representing the grid data.

Methods:

set_cell(position, value): Sets the value of a single cell. set_cells(value, condition=None): Sets the values of multiple cells, optionally based on a condition. modify_cell(position, operation, value): Modifies the value of a single cell using an operation. modify_cells(operation, value, condition_function): Modifies the values of multiple cells using an operation. select_cells(condition, return_list): Selects cells that meet a specified condition. aggregate_property(operation): Performs an aggregate operation over all cells.

Initializes a new PropertyLayer instance.

Args:

name (str): The name of the property layer. width (int): The width of the grid (number of columns). Must be a positive integer. height (int): The height of the grid (number of rows). Must be a positive integer. default_value: The default value to initialize each cell in the grid. Should ideally

be of the same type as specified by the dtype parameter.

dtype (data-type, optional): The desired data-type for the grid's elements. Default is np.float64.

Raises:

ValueError: If width or height is not a positive integer.

Notes:

A UserWarning is raised if the default_value is not of a type compatible with dtype. The dtype parameter can accept both Python data types (like bool, int or float) and NumPy data types (like np.int64 or np.float64). Using NumPy data types is recommended (except for bool) for better control over the precision and efficiency of data storage and computations, especially in cases of large data volumes or specialized numerical operations.

`set_cell`(*position: tuple[int, int]*, *value*)

Update a single cell's value in-place.

`set_cells`(*value*, *condition=None*)

Perform a batch update either on the entire grid or conditionally, in-place.

Args:

value: The value to be used for the update. condition: (Optional) A callable (like a lambda function or a NumPy ufunc)

that returns a boolean array when applied to the data.

`modify_cell`(*position: tuple[int, int]*, *operation*, *value=None*)

Modify a single cell using an operation, which can be a lambda function or a NumPy ufunc. If a NumPy ufunc is used, an additional value should be provided.

Args:

position: The grid coordinates of the cell to modify. operation: A function to apply. Can be a lambda function or a NumPy ufunc. value: The value to be used if the operation is a NumPy ufunc. Ignored for lambda functions.

`modify_cells`(*operation*, *value=None*, *condition_function=None*)

Modify cells using an operation, which can be a lambda function or a NumPy ufunc. If a NumPy ufunc is used, an additional value should be provided.

Args:

operation: A function to apply. Can be a lambda function or a NumPy ufunc. value: The value to be used if the operation is a NumPy ufunc. Ignored for lambda functions. condition_function: (Optional) A callable that returns a boolean array when applied to the data.

`select_cells`(*condition*, *return_list=True*)

Find cells that meet a specified condition using NumPy's boolean indexing, in-place.

Args:

condition: A callable that returns a boolean array when applied to the data. return_list: (Optional) If True, return a list of (x, y) tuples. Otherwise, return a boolean array.

Returns:

A list of (x, y) tuples or a boolean array.

`aggregate_property`(*operation*)

Perform an aggregate operation (e.g., sum, mean) on a property across all cells.

Args:

operation: A function to apply. Can be a lambda function or a NumPy ufunc.

`class SingleGrid`(*width: int*, *height: int*, *torus: bool*, *property_layers: None | PropertyLayer | list[PropertyLayer] = None*)

Rectangular grid where each cell contains exactly at most one agent.

Grid cells are indexed by [x, y], where [0, 0] is assumed to be the bottom-left and [width-1, height-1] is the top-right. If a grid is toroidal, the top and bottom, and left and right, edges wrap to each other.

This class provides a property *empties* that returns a set of coordinates for all empty cells in the grid. It is automatically updated whenever agents are added or removed from the grid. The *empties* property should be used for efficient access to current empty cells rather than manually iterating over the grid to check for emptiness.

Properties:

width, height: The grid's width and height. torus: Boolean which determines whether to treat the grid as a torus. empties: Returns a set of (x, y) tuples for all empty cells. This set is

maintained internally and provides a performant way to query the grid for empty spaces.

Initializes a new _PropertyGrid instance with specified dimensions and optional property layers.

Args:

width (int): The width of the grid (number of columns). height (int): The height of the grid (number of rows). torus (bool): A boolean indicating if the grid should behave like a torus. property_layers (None | PropertyLayer | list[PropertyLayer], optional): A single PropertyLayer instance,

a list of PropertyLayer instances, or None to initialize without any property layers.

Raises:

ValueError: If a property layer's dimensions do not match the grid dimensions.

`remove_agent`(*agent: Agent*) → None

Remove the agent from the grid and set its pos attribute to None.

`add_property_layer`(*property_layer: PropertyLayer*)

Adds a new property layer to the grid.

Args:

property_layer (PropertyLayer): The PropertyLayer instance to be added to the grid.

Raises:
ValueError: If a property layer with the same name already exists in the grid. ValueError: If the dimensions of the property layer do not match the grid's dimensions.

`coord_iter()` → Iterator[tuple[Agent | None, tuple[int, int]]]

An iterator that returns positions as well as cell contents.

`static default_val()` → None

Default value for new cell elements.

`property empty_mask: ndarray`

Returns a boolean mask indicating empty cells on the grid.

`exists_empty_cells()` → bool

Return True if any cells empty else False.

`get_neighborhood`(*pos: tuple[int, int]*, *moore: bool*, *include_center: bool = False*, *radius: int = 1*) → Sequence[tuple[int, int]]

Return a list of cells that are in the neighborhood of a certain point.

Args:
pos: Coordinate tuple for the neighborhood to get. moore: If True, return Moore neighborhood

(including diagonals) If False, return Von Neumann neighborhood (exclude diagonals)

include_center: If True, return the (x, y) cell as well.
Otherwise, return surrounding cells only.

radius: radius, in cells, of neighborhood to get.

Returns:
A list of coordinate tuples representing the neighborhood; With radius 1, at most 9 if Moore, 5 if Von Neumann (8 and 4 if not including the center).

`get_neighborhood_mask`(*pos: tuple[int, int]*, *moore: bool*, *include_center: bool*, *radius: int*) → ndarray

Generate a boolean mask representing the neighborhood. Helper method for select_cells_multi_properties() and move_agent_to_random_cell()

Args:
pos (Coordinate): Center of the neighborhood. moore (bool): True for Moore neighborhood, False for Von Neumann. include_center (bool): Include the central cell in the neighborhood. radius (int): The radius of the neighborhood.

Returns:
np.ndarray: A boolean mask representing the neighborhood.

`get_neighbors`(*pos: tuple[int, int]*, *moore: bool*, *include_center: bool = False*, *radius: int = 1*) → list[Agent]

Return a list of neighbors to a certain point.

Args:
pos: Coordinate tuple for the neighborhood to get. moore: If True, return Moore neighborhood

(including diagonals)

If False, return Von Neumann neighborhood
(exclude diagonals)

include_center: If True, return the (x, y) cell as well.
Otherwise, return surrounding cells only.

radius: radius, in cells, of neighborhood to get.

Returns:
A list of non-None objects in the given neighborhood; at most 9 if Moore, 5 if Von-Neumann (8 and 4 if not including the center).

`is_cell_empty`(*pos: tuple[int, int]*) → bool

Returns a bool of the contents of a cell.

`iter_neighborhood`(*pos: tuple[int, int]*, *moore: bool*, *include_center: bool = False*, *radius: int = 1*) → Iterator[tuple[int, int]]

Return an iterator over cell coordinates that are in the neighborhood of a certain point.

Args:
pos: Coordinate tuple for the neighborhood to get. moore: If True, return Moore neighborhood

(including diagonals)

If False, return Von Neumann neighborhood
(exclude diagonals)

include_center: If True, return the (x, y) cell as well.
Otherwise, return surrounding cells only.

radius: radius, in cells, of neighborhood to get.

Returns:
An iterator of coordinate tuples representing the neighborhood. For example with radius 1, it will return list with number of elements equals at most 9 (8) if Moore, 5 (4) if Von Neumann (if not including the center).

`iter_neighbors`(*pos: tuple[int, int]*, *moore: bool*, *include_center: bool = False*, *radius: int = 1*) → Iterator[Agent]

Return an iterator over neighbors to a certain point.

Args:
pos: Coordinates for the neighborhood to get. moore: If True, return Moore neighborhood

(including diagonals)

If False, return Von Neumann neighborhood
(exclude diagonals)

include_center: If True, return the (x, y) cell as well.
Otherwise, return surrounding cells only.

radius: radius, in cells, of neighborhood to get.

Returns:
An iterator of non-None objects in the given neighborhood; at most 9 if Moore, 5 if Von-Neumann (8 and 4 if not including the center).

`move_agent`(*agent: Agent*, *pos: tuple[int, int]*) → None

Move an agent from its current position to a new position.

Args:

agent: Agent object to move. Assumed to have its current location
stored in a 'pos' tuple.

pos: Tuple of new position to move the agent to.

`move_agent_to_one_of`(*agent: Agent*, *pos: list[tuple[int, int]]*, *selection: str = 'random'*, *handle_empty: str | None = None*) → None

Move an agent to one of the given positions.

Args:
agent: Agent object to move. Assumed to have its current location stored in a 'pos' tuple. pos: List of possible positions. selection: String, either "random" (default) or "closest". If "closest" is selected and multiple

cells are the same distance, one is chosen randomly.

handle_empty: String, either "warning", "error" or None (default). If "warning" or "error" is selected
and no positions are given (an empty list), a warning or error is raised respectively.

`move_to_empty`(*agent: Agent*) → None

Moves agent to a random empty cell, vacating agent's old cell.

`out_of_bounds`(*pos: tuple[int, int]*) → bool

Determines whether position is off the grid, returns the out of bounds coordinate.

`remove_property_layer`(*property_name: str*)

Removes a property layer from the grid by its name.

Args:
property_name (str): The name of the property layer to be removed.

Raises:
ValueError: If a property layer with the given name does not exist in the grid.

`select_cells`(*conditions: dict | None = None*, *extreme_values: dict | None = None*, *masks: ndarray | list[ndarray] = None*, *only_empty: bool = False*, *return_list: bool = True*) → list[tuple[int, int]] | ndarray

Select cells based on property conditions, extreme values, and/or masks, with an option to only select empty cells.

Args:
conditions (dict): A dictionary where keys are property names and values are callables that return a boolean when applied. extreme_values (dict): A dictionary where keys are property names and values are either 'highest' or 'lowest'. masks (np.ndarray | list[np.ndarray], optional): A mask or list of masks to restrict the selection. only_empty (bool, optional): If True, only select cells that are empty. Default is False. return_list (bool, optional): If True, return a list of coordinates, otherwise return a mask.

Returns:
Union[list[Coordinate], np.ndarray]: Coordinates where conditions are satisfied or the combined mask.

`swap_pos`(*agent_a: Agent*, *agent_b: Agent*) → None

Swap agents positions

`torus_adj`(*pos: tuple[int, int]*) → tuple[int, int]

Convert coordinate, handling torus looping.

class MultiGrid(*width: int*, *height: int*, *torus: bool*, *property_layers: None | PropertyLayer | list[PropertyLayer] = None*)

Rectangular grid where each cell can contain more than one agent.

Grid cells are indexed by [x, y], where [0, 0] is assumed to be at bottom-left and [width-1, height-1] is the top-right. If a grid is toroidal, the top and bottom, and left and right, edges wrap to each other.

This class maintains an *empties* property, which is a set of coordinates for all cells that currently contain no agents. This property is updated automatically as agents are added to or removed from the grid.

Properties:
width, height: The grid's width and height. torus: Boolean which determines whether to treat the grid as a torus. empties: Returns a set of (x, y) tuples for all empty cells.

Initializes a new _PropertyGrid instance with specified dimensions and optional property layers.

Args:
width (int): The width of the grid (number of columns). height (int): The height of the grid (number of rows). torus (bool): A boolean indicating if the grid should behave like a torus. property_layers (None | PropertyLayer | list[PropertyLayer], optional): A single PropertyLayer instance,

a list of PropertyLayer instances, or None to initialize without any property layers.

Raises:
ValueError: If a property layer's dimensions do not match the grid dimensions.

static default_val() → list[Agent]

Default value for new cell elements.

remove_agent(*agent: Agent*) → None

Remove the agent from the given location and set its pos attribute to None.

iter_neighbors(*pos: tuple[int, int]*, *moore: bool*, *include_center: bool = False*, *radius: int = 1*) → Iterator[Agent]

Return an iterator over neighbors to a certain point.

Args:
pos: Coordinates for the neighborhood to get. moore: If True, return Moore neighborhood

(including diagonals)

If False, return Von Neumann neighborhood
(exclude diagonals)

include_center: If True, return the (x, y) cell as well.
Otherwise, return surrounding cells only.

radius: radius, in cells, of neighborhood to get.

Returns:
An iterator of non-None objects in the given neighborhood; at most 9 if Moore, 5 if Von-Neumann (8 and 4 if not including the center).

add_property_layer(*property_layer: PropertyLayer*)

Adds a new property layer to the grid.

Args:
property_layer (PropertyLayer): The PropertyLayer instance to be added to the grid.

Raises:
ValueError: If a property layer with the same name already exists in the grid. ValueError: If the dimensions of the property layer do not match the grid's dimensions.

`coord_iter()` → Iterator[tuple[Agent | None, tuple[int, int]]]

An iterator that returns positions as well as cell contents.

`property empty_mask: ndarray`

Returns a boolean mask indicating empty cells on the grid.

`exists_empty_cells()` → bool

Return True if any cells empty else False.

`get_neighborhood`(*pos: tuple[int, int]*, *moore: bool*, *include_center: bool = False*, *radius: int = 1*) → Sequence[tuple[int, int]]

Return a list of cells that are in the neighborhood of a certain point.

Args:
pos: Coordinate tuple for the neighborhood to get. moore: If True, return Moore neighborhood

(including diagonals) If False, return Von Neumann neighborhood (exclude diagonals)

include_center: If True, return the (x, y) cell as well.
Otherwise, return surrounding cells only.

radius: radius, in cells, of neighborhood to get.

Returns:
A list of coordinate tuples representing the neighborhood; With radius 1, at most 9 if Moore, 5 if Von Neumann (8 and 4 if not including the center).

`get_neighborhood_mask`(*pos: tuple[int, int]*, *moore: bool*, *include_center: bool*, *radius: int*) → ndarray

Generate a boolean mask representing the neighborhood. Helper method for select_cells_multi_properties() and move_agent_to_random_cell()

Args:
pos (Coordinate): Center of the neighborhood. moore (bool): True for Moore neighborhood, False for Von Neumann. include_center (bool): Include the central cell in the neighborhood. radius (int): The radius of the neighborhood.

Returns:
np.ndarray: A boolean mask representing the neighborhood.

`get_neighbors`(*pos: tuple[int, int]*, *moore: bool*, *include_center: bool = False*, *radius: int = 1*) → list[Agent]

Return a list of neighbors to a certain point.

Args:
pos: Coordinate tuple for the neighborhood to get. moore: If True, return Moore neighborhood

(including diagonals)

If False, return Von Neumann neighborhood
(exclude diagonals)

include_center: If True, return the (x, y) cell as well.
Otherwise, return surrounding cells only.

radius: radius, in cells, of neighborhood to get.

Returns:
A list of non-None objects in the given neighborhood; at most 9 if Moore, 5 if Von-Neumann (8 and 4 if not including the center).

`is_cell_empty`(*pos: tuple[int, int]*) → bool

Returns a bool of the contents of a cell.

`iter_neighborhood`(*pos: tuple[int, int]*, *moore: bool*, *include_center: bool = False*, *radius: int = 1*) → Iterator[tuple[int, int]]

Return an iterator over cell coordinates that are in the neighborhood of a certain point.

Args:
pos: Coordinate tuple for the neighborhood to get. moore: If True, return Moore neighborhood

(including diagonals)

If False, return Von Neumann neighborhood
(exclude diagonals)

include_center: If True, return the (x, y) cell as well.
Otherwise, return surrounding cells only.

radius: radius, in cells, of neighborhood to get.

Returns:
An iterator of coordinate tuples representing the neighborhood. For example with radius 1, it will return list with number of elements equals at most 9 (8) if Moore, 5 (4) if Von Neumann (if not including the center).

`move_agent`(*agent: Agent*, *pos: tuple[int, int]*) → None

Move an agent from its current position to a new position.

Args:

agent: Agent object to move. Assumed to have its current location
stored in a 'pos' tuple.

pos: Tuple of new position to move the agent to.

`move_agent_to_one_of`(*agent: Agent*, *pos: list[tuple[int, int]]*, *selection: str = 'random'*, *handle_empty: str | None = None*) → None

Move an agent to one of the given positions.

Args:
agent: Agent object to move. Assumed to have its current location stored in a 'pos' tuple. pos: List of possible positions. selection: String, either "random" (default) or "closest". If "closest" is selected and multiple

cells are the same distance, one is chosen randomly.

handle_empty: String, either "warning", "error" or None (default). If "warning" or "error" is selected
and no positions are given (an empty list), a warning or error is raised respectively.

`move_to_empty`(*agent: Agent*) → None

Moves agent to a random empty cell, vacating agent's old cell.

`out_of_bounds`(*pos: tuple[int, int]*) → bool

Determines whether position is off the grid, returns the out of bounds coordinate.

remove_property_layer(*property_name: str*)

Removes a property layer from the grid by its name.

Args:
property_name (str): The name of the property layer to be removed.

Raises:
ValueError: If a property layer with the given name does not exist in the grid.

select_cells(*conditions: dict | None = None*, *extreme_values: dict | None = None*, *masks: ndarray | list[ndarray] = None*, *only_empty: bool = False*, *return_list: bool = True*) → list[tuple[int, int]] | ndarray

Select cells based on property conditions, extreme values, and/or masks, with an option to only select empty cells.

Args:
conditions (dict): A dictionary where keys are property names and values are callables that return a boolean when applied. extreme_values (dict): A dictionary where keys are property names and values are either 'highest' or 'lowest'. masks (np.ndarray | list[np.ndarray], optional): A mask or list of masks to restrict the selection. only_empty (bool, optional): If True, only select cells that are empty. Default is False. return_list (bool, optional): If True, return a list of coordinates, otherwise return a mask.

Returns:
Union[list[Coordinate], np.ndarray]: Coordinates where conditions are satisfied or the combined mask.

swap_pos(*agent_a: Agent*, *agent_b: Agent*) → None

Swap agents positions

torus_adj(*pos: tuple[int, int]*) → tuple[int, int]

Convert coordinate, handling torus looping.

class HexSingleGrid(*width: int*, *height: int*, *torus: bool*, *property_layers: None | PropertyLayer | list[PropertyLayer] = None*)

Hexagonal SingleGrid: a SingleGrid where neighbors are computed according to a hexagonal tiling of the grid.

Functions according to odd-q rules. See http://www.redblobgames.com/grids/hexagons/#coordinates for more.

This class also maintains an *empties* property, similar to SingleGrid, which provides a set of coordinates for all empty hexagonal cells.

Properties:
width, height: The grid's width and height. torus: Boolean which determines whether to treat the grid as a torus. empties: Returns a set of hexagonal coordinates for all empty cells.

Initializes a new _PropertyGrid instance with specified dimensions and optional property layers.

Args:
width (int): The width of the grid (number of columns). height (int): The height of the grid (number of rows). torus (bool): A boolean indicating if the grid should behave like a torus. property_layers (None | PropertyLayer | list[PropertyLayer], optional): A single PropertyLayer instance,

a list of PropertyLayer instances, or None to initialize without any property layers.

Raises:
ValueError: If a property layer's dimensions do not match the grid dimensions.

add_property_layer(*property_layer: PropertyLayer*)

Adds a new property layer to the grid.

Args:
property_layer (PropertyLayer): The PropertyLayer instance to be added to the grid.

Raises:
ValueError: If a property layer with the same name already exists in the grid. ValueError: If the dimensions of the property layer do not match the grid's dimensions.

`coord_iter()` → Iterator[tuple[Agent | None, tuple[int, int]]]

An iterator that returns positions as well as cell contents.

`static default_val()` → None

Default value for new cell elements.

`property empty_mask: ndarray`

Returns a boolean mask indicating empty cells on the grid.

`exists_empty_cells()` → bool

Return True if any cells empty else False.

`get_neighborhood`(*pos: tuple[int, int]*, *include_center: bool = False*, *radius: int = 1*) → list[tuple[int, int]]

Return a list of coordinates that are in the neighborhood of a certain point. To calculate the neighborhood for a HexGrid the parity of the x coordinate of the point is important, the neighborhood can be sketched as:

Always: (0,-), (0,+) When x is even: (-,+), (-,0), (+,+), (+,0) When x is odd: (-,0), (-,-), (+,0), (+,-)

Args:
pos: Coordinate tuple for the neighborhood to get. include_center: If True, return the (x, y) cell as well.

Otherwise, return surrounding cells only.

radius: radius, in cells, of neighborhood to get.

Returns:
A list of coordinate tuples representing the neighborhood. For example with radius 1, it will return list with number of elements equals at most 9 (8) if Moore, 5 (4) if Von Neumann (if not including the center).

`get_neighborhood_mask`(*pos: tuple[int, int]*, *moore: bool*, *include_center: bool*, *radius: int*) → ndarray

Generate a boolean mask representing the neighborhood. Helper method for select_cells_multi_properties() and move_agent_to_random_cell()

Args:
pos (Coordinate): Center of the neighborhood. moore (bool): True for Moore neighborhood, False for Von Neumann. include_center (bool): Include the central cell in the neighborhood. radius (int): The radius of the neighborhood.

Returns:
np.ndarray: A boolean mask representing the neighborhood.

`get_neighbors`(*pos: tuple[int, int]*, *include_center: bool = False*, *radius: int = 1*) → list[Agent]

Return a list of neighbors to a certain point.

Args:
pos: Coordinate tuple for the neighborhood to get. include_center: If True, return the (x, y) cell as well.

Otherwise, return surrounding cells only.

radius: radius, in cells, of neighborhood to get.

Returns:
A list of non-None objects in the given neighborhood

is_cell_empty(*pos: tuple[int, int]*) → bool

Returns a bool of the contents of a cell.

iter_neighborhood(*pos: tuple[int, int]*, *include_center: bool = False*, *radius: int = 1*) → Iterator[tuple[int, int]]

Return an iterator over cell coordinates that are in the neighborhood of a certain point.

Args:

pos: Coordinate tuple for the neighborhood to get. include_center: If True, return the (x, y) cell as well.

Otherwise, return surrounding cells only.

radius: radius, in cells, of neighborhood to get.

Returns:

An iterator of coordinate tuples representing the neighborhood.

iter_neighbors(*pos: tuple[int, int]*, *include_center: bool = False*, *radius: int = 1*) → Iterator[Agent]

Return an iterator over neighbors to a certain point.

Args:

pos: Coordinates for the neighborhood to get. include_center: If True, return the (x, y) cell as well.

Otherwise, return surrounding cells only.

radius: radius, in cells, of neighborhood to get.

Returns:

An iterator of non-None objects in the given neighborhood

move_agent(*agent: Agent*, *pos: tuple[int, int]*) → None

Move an agent from its current position to a new position.

Args:

agent: Agent object to move. Assumed to have its current location
stored in a 'pos' tuple.

pos: Tuple of new position to move the agent to.

move_agent_to_one_of(*agent: Agent*, *pos: list[tuple[int, int]]*, *selection: str = 'random'*, *handle_empty: str | None = None*) → None

Move an agent to one of the given positions.

Args:

agent: Agent object to move. Assumed to have its current location stored in a 'pos' tuple. pos: List of possible positions. selection: String, either "random" (default) or "closest". If "closest" is selected and multiple

cells are the same distance, one is chosen randomly.

handle_empty: String, either "warning", "error" or None (default). If "warning" or "error" is selected
and no positions are given (an empty list), a warning or error is raised respectively.

move_to_empty(*agent: Agent*) → None

Moves agent to a random empty cell, vacating agent's old cell.

out_of_bounds(*pos: tuple[int, int]*) → bool

Determines whether position is off the grid, returns the out of bounds coordinate.

remove_agent(*agent: Agent*) → None

Remove the agent from the grid and set its pos attribute to None.

remove_property_layer(*property_name: str*)

Removes a property layer from the grid by its name.

Args:
property_name (str): The name of the property layer to be removed.

Raises:
ValueError: If a property layer with the given name does not exist in the grid.

select_cells(*conditions: dict | None = None*, *extreme_values: dict | None = None*, *masks: ndarray | list[ndarray] = None*, *only_empty: bool = False*, *return_list: bool = True*) → list[tuple[int, int]] | ndarray

Select cells based on property conditions, extreme values, and/or masks, with an option to only select empty cells.

Args:
conditions (dict): A dictionary where keys are property names and values are callables that return a boolean when applied. extreme_values (dict): A dictionary where keys are property names and values are either 'highest' or 'lowest'. masks (np.ndarray | list[np.ndarray], optional): A mask or list of masks to restrict the selection. only_empty (bool, optional): If True, only select cells that are empty. Default is False. return_list (bool, optional): If True, return a list of coordinates, otherwise return a mask.

Returns:
Union[list[Coordinate], np.ndarray]: Coordinates where conditions are satisfied or the combined mask.

swap_pos(*agent_a: Agent*, *agent_b: Agent*) → None

Swap agents positions

torus_adj(*pos: tuple[int, int]*) → tuple[int, int]

Convert coordinate, handling torus looping.

class HexMultiGrid(*width: int*, *height: int*, *torus: bool*, *property_layers: None | PropertyLayer | list[PropertyLayer] = None*)

Hexagonal MultiGrid: a MultiGrid where neighbors are computed according to a hexagonal tiling of the grid.

Functions according to odd-q rules. See http://www.redblobgames.com/grids/hexagons/#coordinates for more.

Similar to the standard MultiGrid, this class maintains an *empties* property, which is a set of coordinates for all hexagonal cells that currently contain no agents. This property is updated automatically as agents are added to or removed from the grid.

Properties:
width, height: The grid's width and height. torus: Boolean which determines whether to treat the grid as a torus. empties: Returns a set of hexagonal coordinates for all empty cells.

Initializes a new _PropertyGrid instance with specified dimensions and optional property layers.

Args:
width (int): The width of the grid (number of columns). height (int): The height of the grid (number of rows). torus (bool): A boolean indicating if the grid should behave like a torus. property_layers (None | PropertyLayer | list[PropertyLayer], optional): A single PropertyLayer instance,

a list of PropertyLayer instances, or None to initialize without any property layers.

Raises:
ValueError: If a property layer's dimensions do not match the grid dimensions.

add_property_layer(*property_layer: PropertyLayer*)

Adds a new property layer to the grid.

Args:
property_layer (PropertyLayer): The PropertyLayer instance to be added to the grid.

Raises:
ValueError: If a property layer with the same name already exists in the grid. ValueError: If the dimensions of the property layer do not match the grid's dimensions.

coord_iter() → Iterator[tuple[Agent | None, tuple[int, int]]]

An iterator that returns positions as well as cell contents.

static default_val() → list[Agent]

Default value for new cell elements.

property empty_mask: ndarray

Returns a boolean mask indicating empty cells on the grid.

exists_empty_cells() → bool

Return True if any cells empty else False.

get_neighborhood(*pos: tuple[int, int]*, *include_center: bool = False*, *radius: int = 1*) → list[tuple[int, int]]

Return a list of coordinates that are in the neighborhood of a certain point. To calculate the neighborhood for a HexGrid the parity of the x coordinate of the point is important, the neighborhood can be sketched as:

Always: (0,-), (0,+) When x is even: (-,+), (-,0), (+,+), (+,0) When x is odd: (-,0), (-,-), (+,0), (+,-)

Args:
pos: Coordinate tuple for the neighborhood to get. include_center: If True, return the (x, y) cell as well.

Otherwise, return surrounding cells only.

radius: radius, in cells, of neighborhood to get.

Returns:
A list of coordinate tuples representing the neighborhood. For example with radius 1, it will return list with number of elements equals at most 9 (8) if Moore, 5 (4) if Von Neumann (if not including the center).

get_neighborhood_mask(*pos: tuple[int, int]*, *moore: bool*, *include_center: bool*, *radius: int*) → ndarray

Generate a boolean mask representing the neighborhood. Helper method for select_cells_multi_properties() and move_agent_to_random_cell()

Args:
pos (Coordinate): Center of the neighborhood. moore (bool): True for Moore neighborhood, False for Von Neumann. include_center (bool): Include the central cell in the neighborhood. radius (int): The radius of the neighborhood.

Returns:
np.ndarray: A boolean mask representing the neighborhood.

get_neighbors(*pos: tuple[int, int]*, *include_center: bool = False*, *radius: int = 1*) → list[Agent]

Return a list of neighbors to a certain point.

Args:
pos: Coordinate tuple for the neighborhood to get. include_center: If True, return the (x, y) cell as well.

Otherwise, return surrounding cells only.

radius: radius, in cells, of neighborhood to get.

Returns:
A list of non-None objects in the given neighborhood

`is_cell_empty`(*pos: tuple[int, int]*) → bool

Returns a bool of the contents of a cell.

`iter_neighborhood`(*pos: tuple[int, int]*, *include_center: bool = False*, *radius: int = 1*) → Iterator[tuple[int, int]]

Return an iterator over cell coordinates that are in the neighborhood of a certain point.

Args:
pos: Coordinate tuple for the neighborhood to get. include_center: If True, return the (x, y) cell as well.

Otherwise, return surrounding cells only.

radius: radius, in cells, of neighborhood to get.

Returns:
An iterator of coordinate tuples representing the neighborhood.

`iter_neighbors`(*pos: tuple[int, int]*, *include_center: bool = False*, *radius: int = 1*) → Iterator[Agent]

Return an iterator over neighbors to a certain point.

Args:
pos: Coordinates for the neighborhood to get. include_center: If True, return the (x, y) cell as well.

Otherwise, return surrounding cells only.

radius: radius, in cells, of neighborhood to get.

Returns:
An iterator of non-None objects in the given neighborhood

`move_agent`(*agent: Agent*, *pos: tuple[int, int]*) → None

Move an agent from its current position to a new position.

Args:

agent: Agent object to move. Assumed to have its current location
stored in a 'pos' tuple.

pos: Tuple of new position to move the agent to.

`move_agent_to_one_of`(*agent: Agent*, *pos: list[tuple[int, int]]*, *selection: str = 'random'*, *handle_empty: str | None = None*) → None

Move an agent to one of the given positions.

Args:
agent: Agent object to move. Assumed to have its current location stored in a 'pos' tuple. pos: List of possible positions. selection: String, either "random" (default) or "closest". If "closest" is selected and multiple

cells are the same distance, one is chosen randomly.

handle_empty: String, either "warning", "error" or None (default). If "warning" or "error" is selected
and no positions are given (an empty list), a warning or error is raised respectively.

move_to_empty(*agent: Agent*) → None

Moves agent to a random empty cell, vacating agent's old cell.

out_of_bounds(*pos: tuple[int, int]*) → bool

Determines whether position is off the grid, returns the out of bounds coordinate.

remove_agent(*agent: Agent*) → None

Remove the agent from the given location and set its pos attribute to None.

remove_property_layer(*property_name: str*)

Removes a property layer from the grid by its name.

Args:
property_name (str): The name of the property layer to be removed.

Raises:
ValueError: If a property layer with the given name does not exist in the grid.

select_cells(*conditions: dict | None = None*, *extreme_values: dict | None = None*, *masks: ndarray | list[ndarray] = None*, *only_empty: bool = False*, *return_list: bool = True*) → list[tuple[int, int]] | ndarray

Select cells based on property conditions, extreme values, and/or masks, with an option to only select empty cells.

Args:
conditions (dict): A dictionary where keys are property names and values are callables that return a boolean when applied. extreme_values (dict): A dictionary where keys are property names and values are either 'highest' or 'lowest'. masks (np.ndarray | list[np.ndarray], optional): A mask or list of masks to restrict the selection. only_empty (bool, optional): If True, only select cells that are empty. Default is False. return_list (bool, optional): If True, return a list of coordinates, otherwise return a mask.

Returns:
Union[list[Coordinate], np.ndarray]: Coordinates where conditions are satisfied or the combined mask.

swap_pos(*agent_a: Agent*, *agent_b: Agent*) → None

Swap agents positions

torus_adj(*pos: tuple[int, int]*) → tuple[int, int]

Convert coordinate, handling torus looping.

class HexGrid(*width: int*, *height: int*, *torus: bool*)

Hexagonal Grid: a Grid where neighbors are computed according to a hexagonal tiling of the grid.

Functions according to odd-q rules. See http://www.redblobgames.com/grids/hexagons/#coordinates for more.

Properties:
width, height: The grid's width and height. torus: Boolean which determines whether to treat the grid as a torus.

Initializes a new _PropertyGrid instance with specified dimensions and optional property layers.

Args:
width (int): The width of the grid (number of columns). height (int): The height of the grid (number of rows). torus (bool): A boolean indicating if the grid should behave like a torus. property_layers (None | PropertyLayer | list[PropertyLayer], optional): A single PropertyLayer instance,

a list of PropertyLayer instances, or None to initialize without any property layers.

Raises:
ValueError: If a property layer's dimensions do not match the grid dimensions.

add_property_layer(*property_layer: PropertyLayer*)

Adds a new property layer to the grid.

Args:
property_layer (PropertyLayer): The PropertyLayer instance to be added to the grid.

Raises:
ValueError: If a property layer with the same name already exists in the grid. ValueError: If the dimensions of the property layer do not match the grid's dimensions.

coord_iter() → Iterator[tuple[Agent | None, tuple[int, int]]]

An iterator that returns positions as well as cell contents.

static default_val() → None

Default value for new cell elements.

property empty_mask: ndarray

Returns a boolean mask indicating empty cells on the grid.

exists_empty_cells() → bool

Return True if any cells empty else False.

get_neighborhood(*pos: tuple[int, int]*, *include_center: bool = False*, *radius: int = 1*) → list[tuple[int, int]]

Return a list of coordinates that are in the neighborhood of a certain point. To calculate the neighborhood for a HexGrid the parity of the x coordinate of the point is important, the neighborhood can be sketched as:

Always: (0,-), (0,+) When x is even: (-,+), (-,0), (+,+), (+,0) When x is odd: (-,0), (-,-), (+,0), (+,-)

Args:
pos: Coordinate tuple for the neighborhood to get. include_center: If True, return the (x, y) cell as well.

Otherwise, return surrounding cells only.

radius: radius, in cells, of neighborhood to get.

Returns:
A list of coordinate tuples representing the neighborhood. For example with radius 1, it will return list with number of elements equals at most 9 (8) if Moore, 5 (4) if Von Neumann (if not including the center).

get_neighborhood_mask(*pos: tuple[int, int]*, *moore: bool*, *include_center: bool*, *radius: int*) → ndarray

Generate a boolean mask representing the neighborhood. Helper method for select_cells_multi_properties() and move_agent_to_random_cell()

Args:
pos (Coordinate): Center of the neighborhood. moore (bool): True for Moore neighborhood, False for Von Neumann. include_center (bool): Include the central cell in the neighborhood. radius (int): The radius of the neighborhood.

Returns:
np.ndarray: A boolean mask representing the neighborhood.

get_neighbors(*pos: tuple[int, int]*, *include_center: bool = False*, *radius: int = 1*) → list[Agent]

Return a list of neighbors to a certain point.

Args:
pos: Coordinate tuple for the neighborhood to get. include_center: If True, return the (x, y) cell as well.

Otherwise, return surrounding cells only.

radius: radius, in cells, of neighborhood to get.

Returns:
A list of non-None objects in the given neighborhood

`is_cell_empty`(*pos: tuple[int, int]*) → bool

Returns a bool of the contents of a cell.

`iter_neighborhood`(*pos: tuple[int, int]*, *include_center: bool = False*, *radius: int = 1*) → Iterator[tuple[int, int]]

Return an iterator over cell coordinates that are in the neighborhood of a certain point.

Args:
pos: Coordinate tuple for the neighborhood to get. include_center: If True, return the (x, y) cell as well.

Otherwise, return surrounding cells only.

radius: radius, in cells, of neighborhood to get.

Returns:
An iterator of coordinate tuples representing the neighborhood.

`iter_neighbors`(*pos: tuple[int, int]*, *include_center: bool = False*, *radius: int = 1*) → Iterator[Agent]

Return an iterator over neighbors to a certain point.

Args:
pos: Coordinates for the neighborhood to get. include_center: If True, return the (x, y) cell as well.

Otherwise, return surrounding cells only.

radius: radius, in cells, of neighborhood to get.

Returns:
An iterator of non-None objects in the given neighborhood

`move_agent`(*agent: Agent*, *pos: tuple[int, int]*) → None

Move an agent from its current position to a new position.

Args:

agent: Agent object to move. Assumed to have its current location
stored in a 'pos' tuple.

pos: Tuple of new position to move the agent to.

`move_agent_to_one_of`(*agent: Agent*, *pos: list[tuple[int, int]]*, *selection: str = 'random'*, *handle_empty: str | None = None*) → None

Move an agent to one of the given positions.

Args:
agent: Agent object to move. Assumed to have its current location stored in a 'pos' tuple. pos: List of possible positions. selection: String, either "random" (default) or "closest". If "closest" is selected and multiple

cells are the same distance, one is chosen randomly.

handle_empty: String, either "warning", "error" or None (default). If "warning" or "error" is selected
and no positions are given (an empty list), a warning or error is raised respectively.

move_to_empty(*agent: Agent*) → None

Moves agent to a random empty cell, vacating agent's old cell.

out_of_bounds(*pos: tuple[int, int]*) → bool

Determines whether position is off the grid, returns the out of bounds coordinate.

remove_agent(*agent: Agent*) → None

Remove the agent from the grid and set its pos attribute to None.

remove_property_layer(*property_name: str*)

Removes a property layer from the grid by its name.

Args:
property_name (str): The name of the property layer to be removed.

Raises:
ValueError: If a property layer with the given name does not exist in the grid.

select_cells(*conditions: dict | None = None*, *extreme_values: dict | None = None*, *masks: ndarray | list[ndarray] = None*, *only_empty: bool = False*, *return_list: bool = True*) → list[tuple[int, int]] | ndarray

Select cells based on property conditions, extreme values, and/or masks, with an option to only select empty cells.

Args:
conditions (dict): A dictionary where keys are property names and values are callables that return a boolean when applied. extreme_values (dict): A dictionary where keys are property names and values are either 'highest' or 'lowest'. masks (np.ndarray | list[np.ndarray], optional): A mask or list of masks to restrict the selection. only_empty (bool, optional): If True, only select cells that are empty. Default is False. return_list (bool, optional): If True, return a list of coordinates, otherwise return a mask.

Returns:
Union[list[Coordinate], np.ndarray]: Coordinates where conditions are satisfied or the combined mask.

swap_pos(*agent_a: Agent*, *agent_b: Agent*) → None

Swap agents positions

torus_adj(*pos: tuple[int, int]*) → tuple[int, int]

Convert coordinate, handling torus looping.

class ContinuousSpace(*x_max: float*, *y_max: float*, *torus: bool*, *x_min: float = 0*, *y_min: float = 0*)

Continuous space where each agent can have an arbitrary position.

Assumes that all agents have a pos property storing their position as an (x, y) tuple.

This class uses a numpy array internally to store agents in order to speed up neighborhood lookups. This array is calculated on the first neighborhood lookup, and is updated if agents are added or removed.

The concept of 'empty cells' is not directly applicable in continuous space, as positions are not discretized.

Create a new continuous space.

Args:
x_max, y_max: Maximum x and y coordinates for the space. torus: Boolean for whether the edges loop around. x_min, y_min: (default 0) If provided, set the minimum x and y

coordinates for the space. Below them, values loop to the other edge (if torus=True) or raise an exception.

move_agent(*agent: Agent*, *pos: tuple[float, float] | ndarray[Any, dtype[float]]*) → None

Move an agent from its current position to a new position.

Args:
agent: The agent object to move. pos: Coordinate tuple to move the agent to.

remove_agent(*agent: Agent*) → None

Remove an agent from the space.

Args:
agent: The agent object to remove

get_neighbors(*pos: tuple[float, float] | ndarray[Any, dtype[float]]*, *radius: float*, *include_center: bool = True*) → list[Agent]

Get all agents within a certain radius.

Args:
pos: (x,y) coordinate tuple to center the search at. radius: Get all the objects within this distance of the center. include_center: If True, include an object at the *exact* provided

coordinates. i.e. if you are searching for the neighbors of a given agent, True will include that agent in the results.

get_heading(*pos_1: tuple[float, float] | ndarray[Any, dtype[float]]*, *pos_2: tuple[float, float] | ndarray[Any, dtype[float]]*) → tuple[float, float] | ndarray[Any, dtype[float]]

Get the heading vector between two points, accounting for toroidal space. It is possible to calculate the heading angle by applying the atan2 function to the result.

Args:
pos_1, pos_2: Coordinate tuples for both points.

get_distance(*pos_1: tuple[float, float] | ndarray[Any, dtype[float]]*, *pos_2: tuple[float, float] | ndarray[Any, dtype[float]]*) → float

Get the distance between two point, accounting for toroidal space.

Args:
pos_1, pos_2: Coordinate tuples for both points.

torus_adj(*pos: tuple[float, float] | ndarray[Any, dtype[float]]*) → tuple[float, float] | ndarray[Any, dtype[float]]

Adjust coordinates to handle torus looping.

If the coordinate is out-of-bounds and the space is toroidal, return the corresponding point within the space. If the space is not toroidal, raise an exception.

Args:
pos: Coordinate tuple to convert.

out_of_bounds(*pos: tuple[float, float] | ndarray[Any, dtype[float]]*) → bool

Check if a point is out of bounds.

class NetworkGrid(*g: Any*)

Network Grid where each node contains zero or more agents.

Create a new network.

Args:
G: a NetworkX graph instance.

static default_val() → list

Default value for a new node.

get_neighborhood(*node_id: int*, *include_center: bool = False*, *radius: int = 1*) → list[int]

Get all adjacent nodes within a certain radius

get_neighbors(*node_id: int*, *include_center: bool = False*, *radius: int = 1*) → list[Agent]

Get all agents in adjacent nodes (within a certain radius).

move_agent(*agent: Agent*, *node_id: int*) → None

Move an agent from its current node to a new node.

remove_agent(*agent: Agent*) → None

Remove the agent from the network and set its pos attribute to None.

is_cell_empty(*node_id: int*) → bool

Returns a bool of the contents of a cell.

get_cell_list_contents(*cell_list: list[int]*) → list[Agent]

Returns a list of the agents contained in the nodes identified in *cell_list*; nodes with empty content are excluded.

get_all_cell_contents() → list[Agent]

Returns a list of all the agents in the network.

iter_cell_list_contents(*cell_list: list[int]*) → Iterator[Agent]

Returns an iterator of the agents contained in the nodes identified in *cell_list*; nodes with empty content are excluded.

4.6.4 Mesa Data Collection Module

DataCollector is meant to provide a simple, standard way to collect data generated by a Mesa model. It collects three types of data: model-level data, agent-level data, and tables.

A DataCollector is instantiated with two dictionaries of reporter names and associated variable names or functions for each, one for model-level data and one for agent-level data; a third dictionary provides table names and columns. Variable names are converted into functions which retrieve attributes of that name.

When the collect() method is called, each model-level function is called, with the model as the argument, and the results associated with the relevant variable. Then the agent-level functions are called on each agent.

Additionally, other objects can write directly to tables by passing in an appropriate dictionary object for a table row.

The DataCollector then stores the data it collects in dictionaries:

- model_vars maps each reporter to a list of its values
- tables maps each table to a dictionary, with each column as a key with a list as its value.
- _agent_records maps each model step to a list of each agents id and its values.

Finally, DataCollector can create a pandas DataFrame from each collection.

The default DataCollector here makes several assumptions:

- The model has an agent list called agents
- For collecting agent-level variables, agents must have a unique_id

class DataCollector(*model_reporters=None*, *agent_reporters=None*, *tables=None*)

Class for collecting data generated by a Mesa model.

A DataCollector is instantiated with dictionaries of names of model- and agent-level variables to collect, associated with attribute names or functions which actually collect them. When the collect(...) method is called, it collects these attributes and executes these functions one by one and stores the results.

Instantiate a DataCollector with lists of model and agent reporters. Both model_reporters and agent_reporters accept a dictionary mapping a variable name to either an attribute name, a function, a method of a class/instance, or a function with parameters placed in a list.

Model reporters can take four types of arguments: 1. Lambda function:

{"agent_count": lambda m: len(m.agents)}

2. Method of a class/instance: {"agent_count": self.get_agent_count} # self here is a class instance {"agent_count": Model.get_agent_count} # Model here is a class
3. Class attributes of a model: {"model_attribute": "model_attribute"}
4. Functions with parameters that have been placed in a list: {"Model_Function": [function, [param_1, param_2]]}

Agent reporters can similarly take: 1. Attribute name (string) referring to agent's attribute:

{"energy": "energy"}

2. Lambda function: {"energy": lambda a: a.energy}
3. Method of an agent class/instance: {"agent_action": self.do_action} # self here is an agent class instance {"agent_action": Agent.do_action} # Agent here is a class
4. Functions with parameters placed in a list: {"Agent_Function": [function, [param_1, param_2]]}

The tables arg accepts a dictionary mapping names of tables to lists of columns. For example, if we want to allow agents to write their age when they are destroyed (to keep track of lifespans), it might look like:

{"Lifespan": ["unique_id", "age"]}

Args:
model_reporters: Dictionary of reporter names and attributes/funcs/methods. agent_reporters: Dictionary of reporter names and attributes/funcs/methods. tables: Dictionary of table names to lists of column names.

Notes:

- If you want to pickle your model you must not use lambda functions.
- If your model includes a large number of agents, it is recommended to use attribute names for the agent reporter, as it will be faster.

collect(*model*)

Collect all the data for the given model object.

add_table_row(*table_name*, *row*, *ignore_missing=False*)

Add a row dictionary to a specific table.

Args:
table_name: Name of the table to append a row to. row: A dictionary of the form {column_name: value...} ignore_missing: If True, fill any missing columns with Nones;

if False, throw an error if any columns are missing

`get_model_vars_dataframe()`

Create a pandas DataFrame from the model variables.

The DataFrame has one column for each model variable, and the index is (implicitly) the model tick.

`get_agent_vars_dataframe()`

Create a pandas DataFrame from the agent variables.

The DataFrame has one column for each variable, with two additional columns for tick and agent_id.

`get_table_dataframe(table_name)`

Create a pandas DataFrame from a particular table.

Args:
table_name: The name of the table to convert.

`batch_run(model_cls: type[Model], parameters: Mapping[str, Any | Iterable[Any]], number_processes: int | None = 1, iterations: int = 1, data_collection_period: int = -1, max_steps: int = 1000, display_progress: bool = True) → list[dict[str, Any]]`

Batch run a mesa model with a set of parameter values.

4.6.5 Parameters

model_cls
[Type[Model]] The model class to batch-run

parameters
[Mapping[str, Union[Any, Iterable[Any]]],] Dictionary with model parameters over which to run the model. You can either pass single values or iterables.

number_processes
[int, optional] Number of processes used, by default 1. Set this to None if you want to use all CPUs.

iterations
[int, optional] Number of iterations for each parameter combination, by default 1

data_collection_period
[int, optional] Number of steps after which data gets collected, by default -1 (end of episode)

max_steps
[int, optional] Maximum number of model steps after which the model halts, by default 1000

display_progress
[bool, optional] Display batch run process, by default True

4.6.6 Returns

List[Dict[str, Any]]
[description]

4.6.7 Visualization

4.6.7.1 Modules

Initialize self. See help(type(self)) for accurate signature.

4.6.7.1.1 Modular Canvas Rendering

Module for visualizing model objects in grid cells.

4.6.7.1.2 Chart Module

Module for drawing live-updating line charts using Charts.js

4.7 "How To" Mesa Packages

The Mesa core functionality is just a subset of what we believe researchers creating Agent Based Models (ABMs) will use. We designed Mesa to be extensible, so that individuals from various domains can build, maintain, and share their own packages that work with Mesa in pursuit of "unifying algorithmic theories of the relation between adaptive behavior and system complexity (Volker Grimm et al 2005)."

DRY Principle

This decoupling of code to create building blocks is a best practice in software engineering. Specifically, it exercises the DRY principle (or don't repeat yourself) (Hunt and Thomas 2010). The creators of Mesa designed Mesa in order for this principle to be exercised in the development of agent-based models (ABMs). For example, a group of health experts may create a library of human interactions on top of core Mesa. That library then is used by other health experts. So, those health experts don't have to rewrite the same basic behaviors.

Benefits to Scientists

Besides a best practice of the software engineering community, there are other benefits for the scientific community.

1. **Reproducibility and Replicability.** Decoupled shared packages also allows for reproducibility and replicability. Having a package that is shared allows others to reproduce the model results. It also allows others to apply the model to similar phenomenon and replicate the results over a diversity of data. Both are essential part of the scientific method (Leek and Peng 2015).
2. **Accepted truths.** Once results are reproduced and replicated, a library could be considered an accepted truth, meaning that the community agrees the library does what the library intends to do and the library can be trusted to do this. Part of the idea behind 'accepted truths' is that subject matter experts are the ones that write and maintain the library.
3. **Building blocks.** Think of libraries like Legos. The researcher can borrow a piece from here or there to pull together the base of their model, so they can focus on the value add that they bring. For example, someone might pull from a human interactions library and a decision-making library and combine the two to look at how human cognitive function effects the physical spread of disease.

Mesa and Mesa Packages

Because of the possibilities of nuanced libraries, few things will actually make it into core Mesa. Mesa is intended to only include core functionality that everyone uses. However, it is not impossible that something written on the outside is brought into core at a later date if the value to everyone is proven through adoption.

An example that is analogous to Mesa and Mesa packages is Django and Django Packages. Django is a web framework that allows you to build a website in Python, but there are lots of things besides a basic website that you might want. For example, you might want authentication functionality. It would be inefficient for everyone to write their own authentication functionality, so one person writes it (or a group of people). They share it with the world and then many people can use it.

This process isn't perfect. Just because you write something doesn't mean people are going to use it. Sometimes two different packages will be created that do similar things, but one of them does it better or is easier to use. That is the one that will get more adoption. In the world of academia, often researchers hold on to their content until they are ready to publish it. In the world of open source software, this can backfire. The sooner you open source something the more likely it will be a success, because you will build consensus and engagement. Another thing that can happen is that while you are working on perfecting it, someone else is building in the open and establishes the audience you were looking for. So, don't be afraid to start working directly out in the open and then release it to the world.

What is in this doc

There are two sections in this documentation. The first is the User Guide, which is aimed at users of packages. The section is a package development guide, which is aimed at those who want to develop packages. Without further ado, let's get started!

4.7.1 User Guide

- Note: MESA does not endorse or verify any of the code shared through MESA packages. This is left to the domain experts of the community that created the code.*

Step 1: Select a package

Currently, a central list of compatible packages is located on the Mesa Wiki Packages Page.

Step 2: Establish an environment

Create a virtual environment for the ABM you are building. The purpose of a virtual environment is to isolate the packages for your project from other projects. This is helpful when you need to use two different versions of a package or if you are running one version in production but want to test out another version. You can do with either virtualenv or Anaconda.

- Why a virtual environment
- Virtualenv and Virtualenv Wrapper
- Creating a virtual environment with Anaconda

Step 3: Install the packages

Install the package(s) into your environment via pip/conda or GitHub. If the package is a mature package that is hosted in the Python package repository, then you can install it just like you did Mesa:

```
pip install package_name
```

However, sometimes it takes a little bit for projects to reach that level of maturity. In that case to use the library, you would install from GitHub (or other code repository) with something like the following:

```
pip install https://github.com/<path to project>
```

The commands above should also work with Anaconda, just replace the `pip` with `conda`.

4.7.2 Package Development: A "How-to Guide"

The purpose of this section is help you understand, setup, and distribute your Mesa package as quickly as possible. A Mesa package is just a Python package or repo. We just call it a Mesa package, because we are talking about a Python package in the context of Mesa. These instructions assume that you are a little familiar with development, but that you have little knowledge of the packaging process.

There are two ways to share a package:

1. Via GitHub or other service (e.g. GitLab, Bitbucket, etc.)
2. Via PyPI, the Python package manager

Sharing a package via PyPI make it easier to install for users but is more overhead for whomever is maintaining it. However, if you are truly intending for a wider/longer-term adoption, then PyPI should be your goal.

Most likely you created an ABM that has the code that you want to share in it, which is what the steps below describe.

Sharing your package

1. Layout a new file structure to move the code into and then make sure it is callable from Mesa, in a simple, easy to understand way. For example, `from example_package import foo`. See Creating the Scaffolding.
2. Pick a name.
3. Create a repo on GitHub.
    - Enter the name of the repo.
    - Select a license (not sure– click the blue 'i' next to the i for a great run down of licenses). We recommend something permissive Apache 2.0, BSD, or MIT so that others can freely adopt it. The more permissive the more likely it will gain followers and adoption. If you do not include a license, it is our belief that you will retain all rights, which means that people can't use your project, but it should be noted that we are also not lawyers.
    - Create a readme.md file (this contains a description of the package) see an example: Bilateral Shapley
4. Clone the repo to your computer.
5. Copy your code directory into the repo that you cloned one your computer.
6. Add a requirements.txt file, which lets people know which external Python packages are needed to run the code in your repo. To create a file, run: `pip freeze > requirements.txt`. Note, if you are running Anaconda, you will need to install pip first: `conda install pip`.
7. `git add` all the files to the repo, which means the repo starts to track the files. Then `git commit` the files with a meaningful message. To learn more about this see: Saving changes. Finally, you will want to `git push` all your changes to GitHub, see: Git Push.
8. Let people know about your package on the MESA Wiki Page and share it on the email list. In the future, we will create more of a directory, but at this point we are not there yet.

From this point, someone can clone your repo and then add your repo to their Python path and use it in their project. However, if you want to take your package to the next level, you will want to add more structure to your package and share it on PyPI.

Next Level: PyPI

You want to do even more. The authoritative guide for python package development is through the Python Packaging User Guide. This will take you through the entire process necessary for getting your package on the Python Package Index.

The Python Package Index is the main repository of software for Python Packages and following this guide will ensure your code and documentation meets the standards for distribution across the Python community.

4.8 References

Grimm, Volker, Eloy Revilla, Uta Berger, Florian Jeltsch, Wolf M. Mooij, Steven F. Railsback, Hans-Hermann Thulke, Jacob Weiner, Thorsten Wiegand, and Donald L. DeAngelis. 2005. "Pattern-Oriented Modeling of Agent Based Complex Systems: Lessons from Ecology." American Association for the Advancement of Science 310 (5750): 987–91. doi:10.1126/science.1116681.

Hunt, Andrew, and David Thomas. 2010. The Pragmatic Programmer: From Journeyman to Master. Reading, Massachusetts: Addison-Wesley.

Leek, Jeffrey T., and Roger D. Peng. 2015. "Reproducible Research Can Still Be Wrong: Adopting a Prevention Approach." Proceedings of the National Academy of Sciences 112 (6): 1645–46. doi:10.1073/pnas.1421412111.

4.9 Advanced Tutorial

This is the legacy version of the advanced tutorial. We recommend you to read the newer (current) version because it is easier.

4.9.1 Adding visualization

So far, we've built a model, run it, and analyzed some output afterwards. However, one of the advantages of agent-based models is that we can often watch them run step by step, potentially spotting unexpected patterns, behaviors or bugs, or developing new intuitions, hypotheses, or insights. Other times, watching a model run can explain it to an unfamiliar audience better than static explanations. Like many ABM frameworks, Mesa allows you to create an interactive visualization of the model. In this section we'll walk through creating a visualization using built-in components, and (for advanced users) how to create a new visualization element.

Note for Jupyter users: Due to conflicts with the tornado server Mesa uses and Jupyter, the interactive browser of your model will load but likely not work. This will require you to use run the code from .py files. The Mesa development team is working to develop a Jupyter compatible interface.

First, a quick explanation of how Mesa's interactive visualization works. Visualization is done in a browser window, using JavaScript to draw the different things being visualized at each step of the model. To do this, Mesa launches a small web server, which runs the model, turns each step into a JSON object (essentially, structured plain text) and sends those steps to the browser.

A visualization is built up of a few different modules: for example, a module for drawing agents on a grid, and another one for drawing a chart of some variable. Each module has a Python part, which runs on the server and turns a model state into JSON data; and a JavaScript side, which takes that JSON data and draws it in the browser window. Mesa comes with a few modules built in, and let you add your own as well.

4.9.1.1 Grid Visualization

To start with, let's have a visualization where we can watch the agents moving around the grid. For this, you will need to put your model code in a separate Python source file. For now, let us use the `MoneyModel` created in the Introductory Tutorial saved to `MoneyModel.py` file provided. Next, in a new source file (e.g. `MoneyModel_Viz.py`) include the code shown in the following cells to run and avoid Jupyter compatibility issue.

```
# If MoneyModel.py is where your code is:
from MoneyModel import mesa, MoneyModel
```

Mesa's `CanvasGrid` visualization class works by looping over every cell in a grid, and generating a portrayal for every agent it finds. A portrayal is a dictionary (which can easily be turned into a JSON object) which tells the JavaScript side how to draw it. The only thing we need to provide is a function which takes an agent, and returns a portrayal object. Here's the simplest one: it'll draw each agent as a red, filled circle which fills half of each cell.

```
def agent_portrayal(agent):
    portrayal = {
        "Shape": "circle",
        "Color": "red",
        "Filled": "true",
        "Layer": 0,
        "r": 0.5,
    }
    return portrayal
```

In addition to the portrayal method, we instantiate a canvas grid with its width and height in cells, and in pixels. In this case, let's create a 10x10 grid, drawn in 500 x 500 pixels.

```
grid = mesa.visualization.CanvasGrid(agent_portrayal, 10, 10, 500, 500)
```

Now we create and launch the actual server. We do this with the following arguments:

- The model class we're running and visualizing; in this case, `MoneyModel`.
- A list of module objects to include in the visualization; here, just `[grid]`
- The title of the model: "Money Model"
- Any inputs or arguments for the model itself. In this case, 100 agents, and height and width of 10.

Once we create the server, we set the port (use default 8521 here) for it to listen on (you can treat this as just a piece of the URL you'll open in the browser).

```
server = mesa.visualization.ModularServer(
    MoneyModel, [grid], "Money Model", {"N": 100, "width": 10, "height": 10}
)
server.port = 8521  # the default
```

Finally, when you're ready to run the visualization, use the server's launch() method.

The full code for source file `MoneyModel_Viz.py` should now look like:

```
from MoneyModel import mesa, MoneyModel


def agent_portrayal(agent):
    portrayal = {"Shape": "circle",
                 "Filled": "true",
                 "Layer": 0,
                 "Color": "red",
                 "r": 0.5}
    return portrayal

grid = mesa.visualization.CanvasGrid(agent_portrayal, 10, 10, 500, 500)
server = mesa.visualization.ModularServer(MoneyModel,
                       [grid],
                       "Money Model",
```

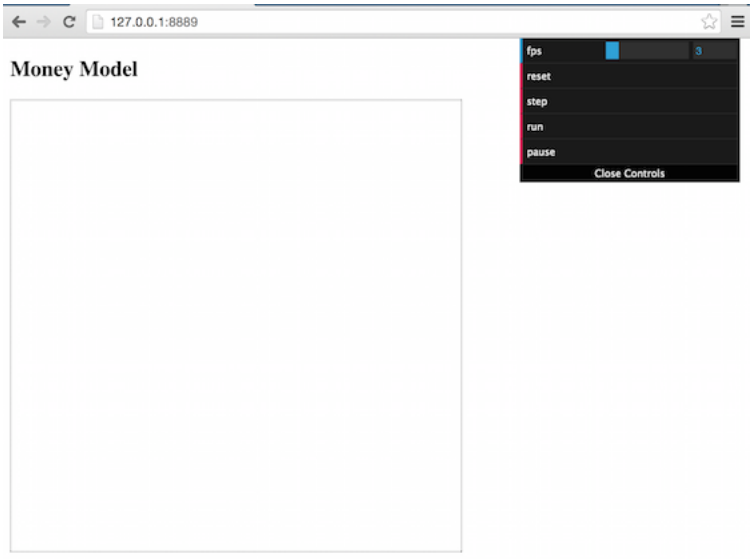
(continues on next page)

(continued from previous page)

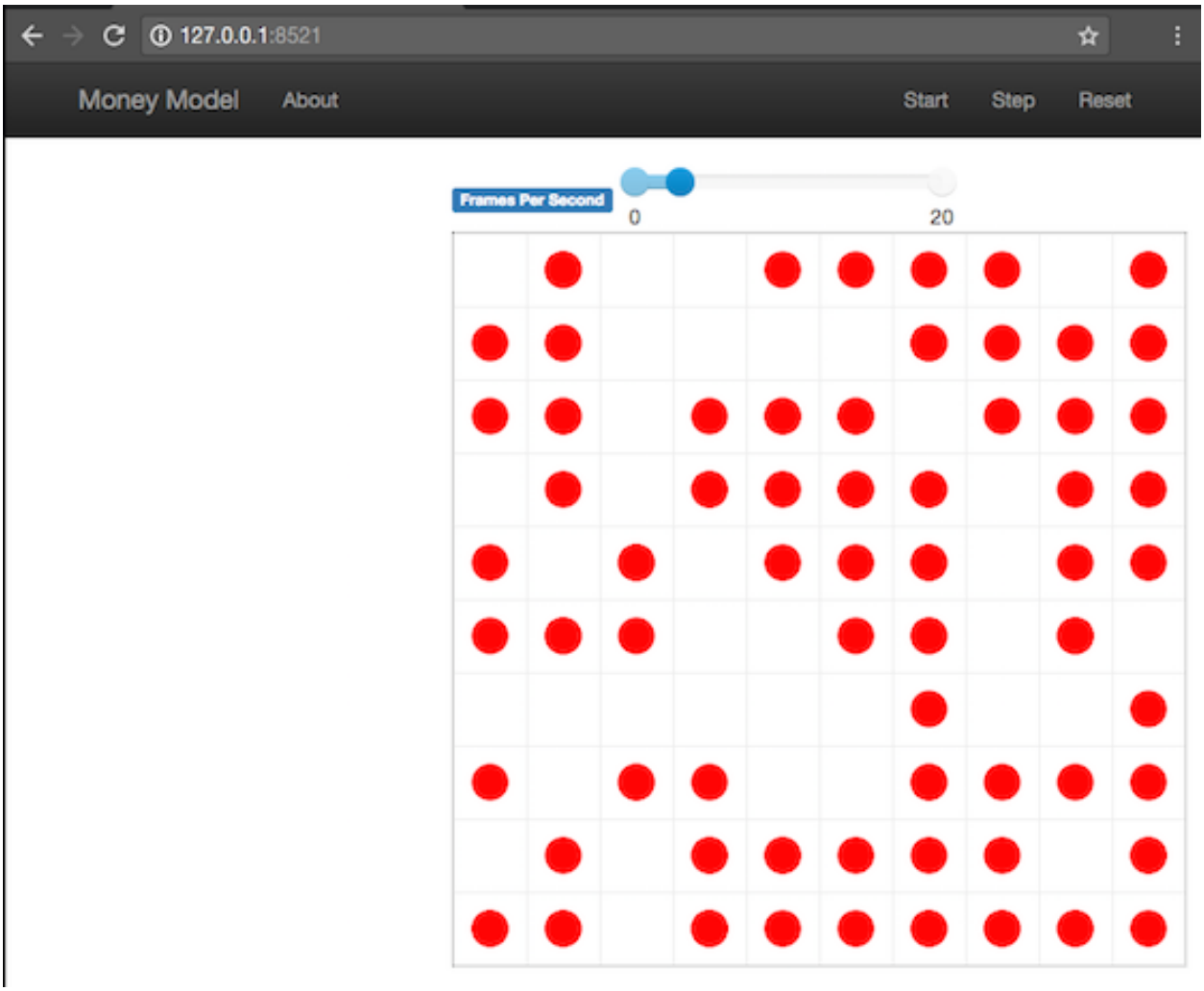
```
                    {"N":100, "width":10, "height":10})
server.port = 8521 # The default
server.launch()
```

Now run this file; this should launch the interactive visualization server and open your web browser automatically. (If the browser doesn't open automatically, try pointing it at http://127.0.0.1:8521 manually. If this doesn't show you the visualization, something may have gone wrong with the server launch.)

You should see something like the figure below: the model title, an empty space where the grid will be, and a control panel off to the right.



Click the `Reset` button on the control panel, and you should see the grid fill up with red circles, representing agents.



Click `Step` to advance the model by one step, and the agents will move around. Click `Start` and the agents will keep moving around, at the rate set by the 'fps' (frames per second) slider at the top. Try moving it around and see how the speed of the model changes. Pressing `Stop` will pause the model; presing `Start` again will restart it. Finally, `Reset` will start a new instantiation of the model.

To stop the visualization server, go back to the terminal where you launched it, and press Control+c.

4.9.1.2 Changing the agents

In the visualization above, all we could see is the agents moving around – but not how much money they had, or anything else of interest. Let's change it so that agents who are broke (wealth 0) are drawn in grey, smaller, and above agents who still have money.

To do this, we go back to our `agent_portrayal` code and add some code to change the portrayal based on the agent properties and launch the server again.

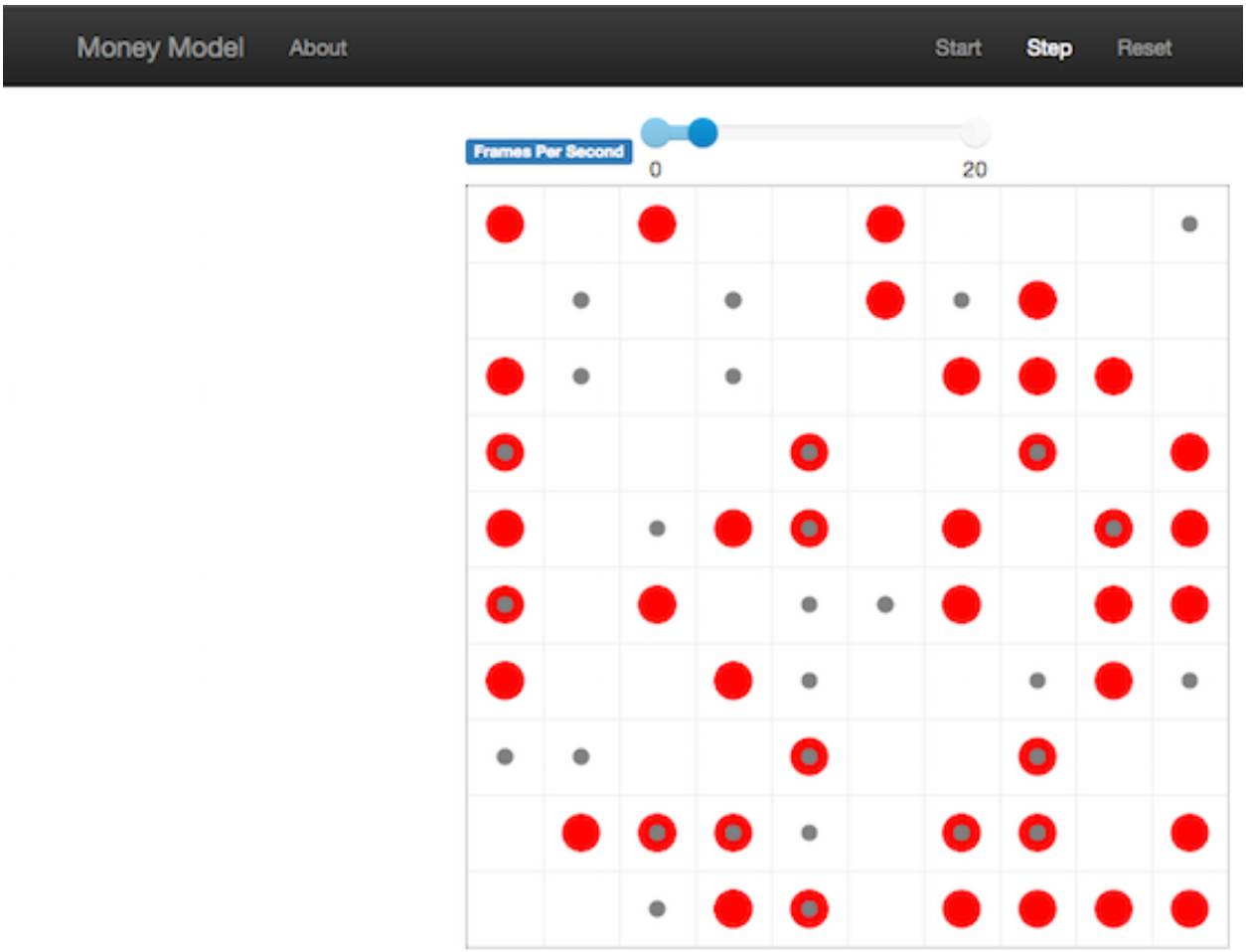
```
def agent_portrayal(agent):
    portrayal = {"Shape": "circle", "Filled": "true", "r": 0.5}

    if agent.wealth > 0:
        portrayal["Color"] = "red"
```

(continues on next page)

(continued from previous page)

```
        portrayal["Layer"] = 0
    else:
        portrayal["Color"] = "grey"
        portrayal["Layer"] = 1
        portrayal["r"] = 0.2
    return portrayal
```

This will open a new browser window pointed at the updated visualization. Initially it looks the same, but advance the model and smaller grey circles start to appear. Note that since the zero-wealth agents have a higher layer number, they are drawn on top of the red agents.



4.9.1.3 Adding a chart

Next, let's add another element to the visualization: a chart, tracking the model's Gini Coefficient. This is another built-in element that Mesa provides.

The basic chart pulls data from the model's DataCollector, and draws it as a line graph using the Charts.js JavaScript libraries. We instantiate a chart element with a list of series for the chart to track. Each series is defined in a dictionary, and has a `Label` (which must match the name of a model-level variable collected by the DataCollector) and a `Color` name. We can also give the chart the name of the DataCollector object in the model.

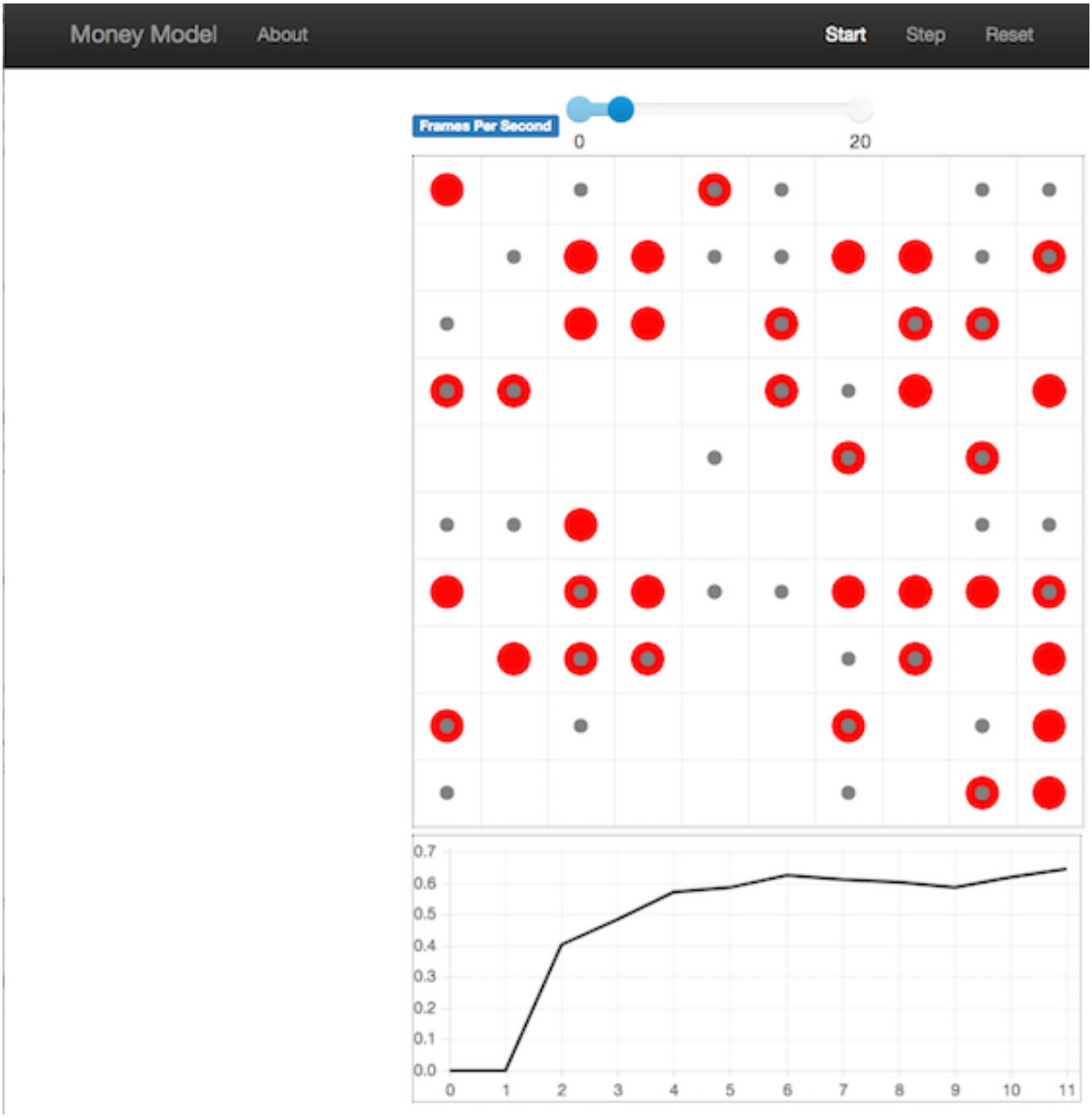
Finally, we add the chart to the list of elements in the server. The elements are added to the visualization in the order they appear, so the chart will appear underneath the grid.

```
chart = mesa.visualization.ChartModule(
    [{"Label": "Gini", "Color": "Black"}], data_collector_name="datacollector"
)

server = mesa.visualization.ModularServer(
    MoneyModel, [grid, chart], "Money Model", {"N": 100, "width": 10, "height": 10}
)
```

Launch the visualization and start a model run, either by launching the server here or through the full code for source file `MoneyModel_Viz.py`.

```
from MoneyModel import mesa, MoneyModel


def agent_portrayal(agent):
    portrayal = {"Shape": "circle", "Filled": "true", "r": 0.5}

    if agent.wealth > 0:
        portrayal["Color"] = "red"
        portrayal["Layer"] = 0
    else:
        portrayal["Color"] = "grey"
        portrayal["Layer"] = 1
        portrayal["r"] = 0.2
    return portrayal


grid = mesa.visualization.CanvasGrid(agent_portrayal, 10, 10, 500, 500)
chart = mesa.visualization.ChartModule(
    [{"Label": "Gini", "Color": "Black"}], data_collector_name="datacollector"
)

server = mesa.visualization.ModularServer(
    MoneyModel, [grid, chart], "Money Model", {"N": 100, "width": 10, "height": 10}
)
server.port = 8521  # The default
server.launch()
```

You'll see a line chart underneath the grid. Every step of the model, the line chart updates along with the grid. Reset the model, and the chart resets too.



Note: You might notice that the chart line only starts after a couple of steps; this is due to a bug in Charts.js which will hopefully be fixed soon.

4.9.2 Building your own visualization component

Note: This section is for users who have a basic familiarity with JavaScript. If that's not you, don't worry! (If you're an advanced JavaScript coder and find things that we've done wrong or inefficiently, please let us know!)

If the visualization elements provided by Mesa aren't enough for you, you can build your own and plug them into the model server.

First, you need to understand how the visualization works under the hood. Remember that each visualization module has two sides: a Python object that runs on the server and generates JSON data from the model state (the server side), and a JavaScript object that runs in the browser and turns the JSON into something it renders on the screen (the client side).

Obviously, the two sides of each visualization must be designed in tandem. They result in one Python class, and one JavaScript `.js` file. The path to the JavaScript file is a property of the Python class.

For this example, let's build a simple histogram visualization, which can count the number of agents with each value of wealth. We'll use the Charts.js JavaScript library, which is already included with Mesa. If you go and look at its documentation, you'll see that it had no histogram functionality, which means we have to build our own out of a bar chart. We'll keep the histogram as simple as possible, giving it a fixed number of integer bins. If you were designing a more general histogram to add to the Mesa repository for everyone to use across different models, obviously you'd want something more general.

4.9.2.1 Client-Side Code

In general, the server- and client-side are written in tandem. However, if you're like me and more comfortable with Python than JavaScript, it makes sense to figure out how to get the JavaScript working first, and then write the Python to be compatible with that.

In the same directory as your model, create a new file called `HistogramModule.js`. This will store the JavaScript code for the client side of the new module.

JavaScript classes can look alien to people coming from other languages – specifically, they can look like functions. (The Mozilla Introduction to Object-Oriented JavaScript is a good starting point). In `HistogramModule.js`, start by creating the class itself:

```
const HistogramModule = function(bins, canvas_width, canvas_height) {
    // The actual code will go here.
};
```

Note that our object is instantiated with three arguments: the number of integer bins, and the width and height (in pixels) the chart will take up in the visualization window.

When the visualization object is instantiated, the first thing it needs to do is prepare to draw on the current page. To do so, it adds a canvas tag to the page. It also gets the canvas' context, which is required for doing anything with it.

```
const HistogramModule = function(bins, canvas_width, canvas_height) {
  // Create the canvas object:
    const canvas = document.createElement("canvas");
    Object.assign(canvas, {
      width: canvas_width,
      height: canvas_height,
      style: "border:1px dotted",
    });
  // Append it to #elements:
  const elements = document.getElementById("elements");
```

(continues on next page)

(continued from previous page)

```
  elements.appendChild(canvas);

  // Create the context and the drawing controller:
  const context = canvas.getContext("2d");
};
```

Look at the Charts.js bar chart documentation. You'll see some of the boilerplate needed to get a chart set up. Especially important is the `data` object, which includes the datasets, labels, and color options. In this case, we want just one dataset (we'll keep things simple and name it "Data"); it has `bins` for categories, and the value of each category starts out at zero. Finally, using these boilerplate objects and the canvas context we created, we can create the chart object.

```
const HistogramModule = function(bins, canvas_width, canvas_height) {
  // Create the canvas object:
    const canvas = document.createElement("canvas");
    Object.assign(canvas, {
      width: canvas_width,
      height: canvas_height,
      style: "border:1px dotted",
    });
  // Append it to #elements:
  const elements = document.getElementById("elements");
  elements.appendChild(canvas);

  // Create the context and the drawing controller:
  const context = canvas.getContext("2d");

  // Prep the chart properties and series:
  const datasets = [{
    label: "Data",
    fillColor: "rgba(151,187,205,0.5)",
    strokeColor: "rgba(151,187,205,0.8)",
    highlightFill: "rgba(151,187,205,0.75)",
    highlightStroke: "rgba(151,187,205,1)",
    data: []
  }];

  // Add a zero value for each bin
  for (var i in bins)
    datasets[0].data.push(0);

  const data = {
    labels: bins,
    datasets: datasets
  };

  const options = {
    scaleBeginsAtZero: true
  };

  // Create the chart object
  let chart = new Chart(context, {type: 'bar', data: data, options: options});
```

(continues on next page)

(continued from previous page)

```
  // Now what?
};
```

There are two methods every client-side visualization class must implement to be able to work: `render(data)` to render the incoming data, and `reset()` which is called to clear the visualization when the user hits the reset button and starts a new model run.

In this case, the easiest way to pass data to the histogram is as an array, one value for each bin. We can then just loop over the array and update the values in the chart's dataset.

There are a few ways to reset the chart, but the easiest is probably to destroy it and create a new chart object in its place.

With that in mind, we can add these two methods to the class:

```
const HistogramModule = function(bins, canvas_width, canvas_height) {
  // ...Everything from above...
  this.render = function(data) {
    datasets[0].data = data;
    chart.update();
  };

  this.reset = function() {
    chart.destroy();
    chart = new Chart(context, {type: 'bar', data: data, options: options});
  };
};
```

Note the `this.` before the method names. This makes them public and ensures that they are accessible outside of the object itself. All the other variables inside the class are only accessible inside the object itself, but not outside of it.

4.9.2.2 Server-Side Code

Can we get back to Python code? Please?

Every JavaScript visualization element has an equal and opposite server-side Python element. The Python class needs to also have a `render` method, to get data out of the model object and into a JSON-ready format. It also needs to point towards the code where the relevant JavaScript lives, and add the JavaScript object to the model page.

In a Python file (either its own, or in the same file as your visualization code), import the `VisualizationElement` class we'll inherit from, and create the new visualization class.

```
    from mesa.visualization.ModularVisualization import VisualizationElement, CHART_JS_
↪FILE

    class HistogramModule(VisualizationElement):
        package_includes = [CHART_JS_FILE]
        local_includes = ["HistogramModule.js"]

        def __init__(self, bins, canvas_height, canvas_width):
            self.canvas_height = canvas_height
            self.canvas_width = canvas_width
            self.bins = bins
            new_element = "new HistogramModule({}, {}, {})"
            new_element = new_element.format(bins,
```

(continues on next page)

(continued from previous page)

```
                                            canvas_width,
                                            canvas_height)
        self.js_code = "elements.push(" + new_element + ");"
```

There are a few things going on here. `package_includes` is a list of JavaScript files that are part of Mesa itself that the visualization element relies on. You can see the included files in mesa/visualization/templates/. Similarly, `local_includes` is a list of JavaScript files in the same directory as the class code itself. Note that both of these are class variables, not object variables – they hold for all particular objects.

Next, look at the `__init__` method. It takes three arguments: the number of bins, and the width and height for the histogram. It then uses these values to populate the `js_code` property; this is code that the server will insert into the visualization page, which will run when the page loads. In this case, it creates a new HistogramModule (the class we created in JavaScript in the step above) with the desired bins, width and height; it then appends (`push`es) this object to `elements`, the list of visualization elements that the visualization page itself maintains.

Now, the last thing we need is the `render` method. If we were making a general-purpose visualization module we'd want this to be more general, but in this case we can hard-code it to our model.

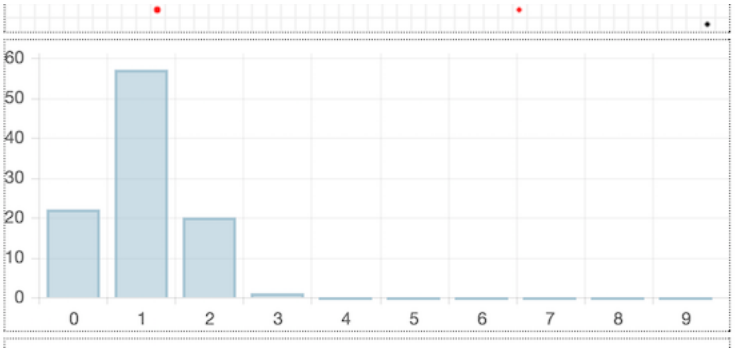
```
import numpy as np

class HistogramModule(VisualizationElement):
    # ... Everything from above...

    def render(self, model):
        wealth_vals = [agent.wealth for agent in model.schedule.agents]
        hist = np.histogram(wealth_vals, bins=self.bins)[0]
        return [int(x) for x in hist]
```

Every time the render method is called (with a model object as the argument) it uses numpy to generate counts of agents with each wealth value in the bins, and then returns a list of these values. Note that the `render` method doesn't return a JSON string – just an object that can be turned into JSON, in this case a Python list (with Python integers as the values; the `json` library doesn't like dealing with numpy's integer type).

Now, you can create your new HistogramModule and add it to the server:

```
    histogram = mesa.visualization.HistogramModule(list(range(10)), 200, 500)
    server = mesa.visualization.ModularServer(MoneyModel,
                          [grid, histogram, chart],
                          "Money Model",
                          {"N":100, "width":10, "height":10})
    server.launch()
```

Run this code, and you should see your brand-new histogram added to the visualization and updating along with the model!



If you've felt comfortable with this section, it might be instructive to read the code for the ModularServer and the modular_template to get a better idea of how all the pieces fit together.

4.9.3 Happy Modeling!

This document is a work in progress. If you see any errors, exclusions or have any problems please contact us.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

H

I

M

N

O

P

R

S

T

V