Mesa Documentation

Release .1

Project Mesa Team

Aug 21, 2017

Contents

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

It 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 3-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 an IPython notebook.

CHAPTER 1

Features

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

CHAPTER 2

Using Mesa

Getting started quickly:

```
$ pip install mesa
```

To launch an example model, open any of the directories in the examples folder and launch the `run.py` file there, e.g.:

```
schelling $ python run.py
```

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

- Mesa Introductory Tutorial
- Mesa Advanced Tutorial
- GitHub Issue Tracker
- Email list
- PyPI

CHAPTER 3

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

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.

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.

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 random
from mesa import Agent, Model
from mesa.time import RandomActivation
from mesa.space import MultiGrid

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

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

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

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:

```
from mesa.datacollection import DataCollector

# ...

class MyModel(Model):
    def __init__(self, n_agents):
        # ...
        self.dc = 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 batchrunner:

```
from mesa.batchrunner import BatchRunner

parameters = {"n_agents": range(1, 20)}
batch_run = BatchRunner(MyModel, parameters, max_steps=10,
                        model_reporters={"n_agents": lambda m: m.schedule.get_agent_
↪count()})
batch_run.run_all()
```

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()
```

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:

```
from mesa.visualization.modules import CanvasGrid
from mesa.visualization.ModularVisualization import ModularServer
```

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

grid = CanvasGrid(agent_portrayal, 10, 10, 500, 500)
server = ModularServer(MyModel,
                       [grid],
                       "My Model",
                       100, 10, 10)
server.launch()
```

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

Introductory Tutorial

Tutorial Description

Mesa is a Python framework for agent-based modeling. Getting started with Mesa is easy. In this tutorial, we will walk through creating a simple model and progressively add functionality which will illustrate Mesa's core features.

Note: This tutorial is a work-in-progress. If you find any errors or bugs, or just find something unclear or confusing, let us know!

The base for this tutorial is a very simple model of agents exchanging money. Next, we add *space* to allow agents move. Then, we'll cover two of Mesa's analytic tools: the *data collector* and *batch runner*. After that, we'll add an *interactive visualization* which lets us watch the model as it runs. Finally, we go over how to write your own visualization module, for users who are comfortable with JavaScript.

You can also find all the code this tutorial describes in the **examples/Tutorial-Boltzmann_Wealth_Model** directory of the Mesa repository.

Sample Model Description

The tutorial model is a very simple simulated agent-based economy, drawn from econophysics and presenting a statistical mechanics approach to wealth distribution [Dragulescu2002]_. The rules of our tutorial model:

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.

Despite its simplicity, this model yields results that are often unexpected to those not familiar with it. For our purposes, it also easily demonstrates Mesa's core features.

Let's get started.

Installation

To start, install Mesa. We recommend doing this in a virtual environment, but make sure your environment is set up with Python 3. Mesa requires Python3 and does not work in Python 2 environments.

To install Mesa, simply:

```
$ pip install mesa
```

When you do that, it will install Mesa itself, as well as any dependencies that aren't in your setup yet. Additional dependencies required by this tutorial can be found in the **examples/Tutorial-Boltzmann_Wealth_Model/requirements.txt** file, which can be installed by running:

```
$ pip install -r requirements.txt
```

Building a sample model

Once Mesa is installed, you can start building our model. You can write models in two different ways:

1. Write the code in its own file with your favorite text editor, or
2. Write the model interactively in Jupyter Notebook cells.

Either way, it's good practice to put your model in its own folder – especially if the project will end up consisting of multiple files (for example, Python files for the model and the visualization, a Notebook for analysis, and a Readme with some documentation and discussion).

Begin by creating a folder named *money_model*, and either launch a Notebook or create a new Python source file. We will use the name `model.py` here.

Setting up the model

To begin writing the model code, we start with two core classes: one for the overall model, the other for the agents. The model class holds the model-level attributes, manages the agents, and generally handles the global level of our model. Each instantiation of the model class will be a specific model run. Each model will contain multiple agents, all of which are instantiations of the agent class. Both the model and agent classes are child classes of Mesa's generic `Model` and `Agent` classes.

Each agent has only one variable: how much wealth it currently has. (Each agent will also have a unique identifier (i.e., a name), stored in the `unique_id` variable. Giving each agent a unique id is a good practice when doing agent-based modeling.)

There is only one model-level parameter: how many agents the model contains. When a new model is started, we want it to populate itself with the given number of agents.

The beginning of both classes looks like this:

```
# model.py
from mesa import Agent, Model

class MoneyAgent(Agent):
    """An agent with fixed initial wealth."""
    def __init__(self, unique_id, model):
        super().__init__(unique_id, model)
        self.wealth = 1

class MoneyModel(Model):
    """A model with some number of agents."""
    def __init__(self, N):
        self.num_agents = N
        # Create agents
```

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

Adding the scheduler

Time in most agent-based models moves in steps, sometimes also called **ticks**. 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.

The **scheduler** is a special model component which controls the order in which agents are activated. For example, all the agents may activate in the same order every step; their order might be shuffled; we may try to simulate all the agents acting at the same time; and more. 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. This may not seem important, but scheduling patterns can have an impact on your results [Comer2014]_.

For now, let's use one of the simplest ones: `RandomActivation`, which activates all the agents once per step, in random order. Every agent is expected to have a `step` method, which takes a model object as its only argument – this is the agent's action when it is activated. We add an agent to the schedule using the `add` method; when we call the schedule's `step` method, it shuffles the order of the agents, then activates them all, one at a time.

With that in mind, the model code with the scheduler added looks like this:

```
# model.py
from mesa import Agent, Model
from mesa.time import RandomActivation

class MoneyAgent(Agent):
    """ An agent with fixed initial wealth."""
    def __init__(self, unique_id, model):
        super().__init__(unique_id, model)
        self.wealth = 1

    def step(self):
        # The agent's step will go here.
        pass

class MoneyModel(Model):
    """A model with some number of agents."""
    def __init__(self, N):
        self.num_agents = N
        self.schedule = RandomActivation(self)
        # Create agents
        for i in range(self.num_agents):
            a = MoneyAgent(i, self)
            self.schedule.add(a)

    def step(self):
        '''Advance the model by one step.'''
        self.schedule.step()
```

At this point, we have a model which runs – it just doesn't do anything. You can see for yourself with a few easy lines. If you've been working in an interactive session, you can create a model object directly. Otherwise, you need to open an interactive session in the same directory as your source code file, and import the classes. For example, if your code is in `model.py`:

```
from model import MoneyModel
```

Then create the model object, and run it for one step:

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

Exercise

Try modifying the code above to have every agent print out its `unique_id` when it is activated. Run a few steps of the model to see how the agent activation order is shuffled each step.

Agent Step

Now we just need to have the agents do what we intend for them to do: check their wealth, and if they have the money, give one unit of it away to another random agent. Since we want to use randomness, don't forget to import Python's `random` library:

```
# model.py
import random
```

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's `step` method, like this:

```
# model.py
class MoneyAgent(Agent):
    """ An agent with fixed initial wealth."""
    def __init__(self, unique_id, model):
        super().__init__(unique_id, model)
        self.wealth = 1

    def step(self):
        if self.wealth == 0:
            return
        other_agent = random.choice(self.model.schedule.agents)
        other_agent.wealth += 1
        self.wealth -= 1
```

Running your first model

With that last piece in hand, it's time for the first rudimentary run of the model. This time, let's write a `run.py` to run the script instead of using the command-line interpreter. If you wrote the code in a Notebook, you can write this in the same notebook.

```
# run.py
from model import *  # omit this in jupyter notebooks

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 matplotlib (or another graphics library) to visualize the data in a histogram.

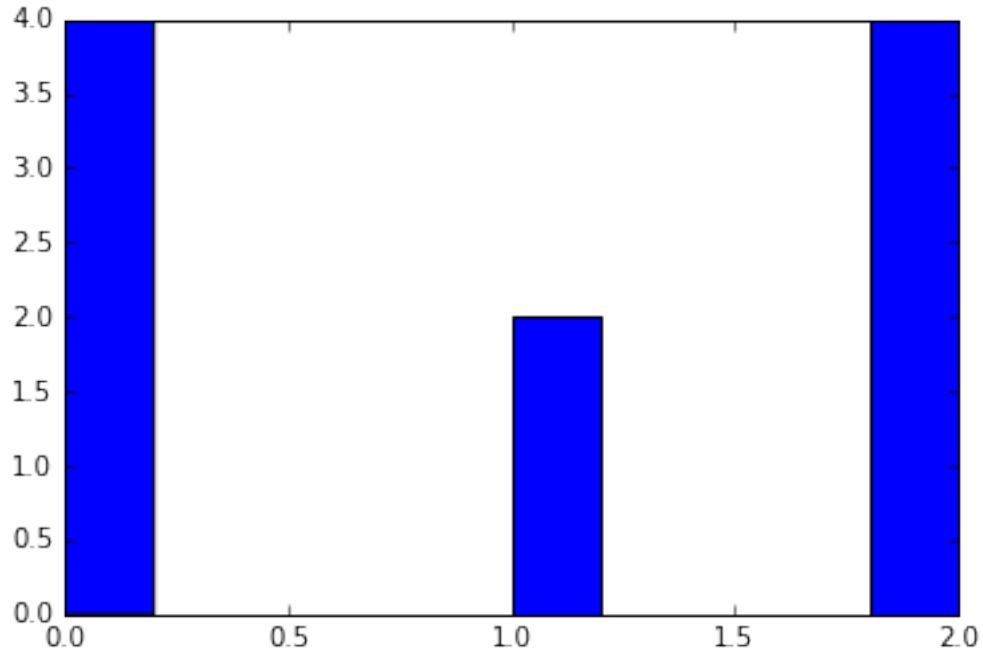
```
# run.py
import matplotlib.pyplot as plt

agent_wealth = [a.wealth for a in model.schedule.agents]
plt.hist(agent_wealth)
plt.show()
```

Or, in a Jupyter notebook:

```
%matplotlib inline
import matplotlib.pyplot as plt
agent_wealth = [a.wealth for a in model.schedule.agents]
plt.hist(agent_wealth)
```

```
(array([ 4.,  0.,  0.,  0.,  0.,  2.,  0.,  0.,  0.,  4.]),
 array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ,  1.2,  1.4,  1.6,  1.8,  2. ]),
 <a list of 10 Patch objects>)
```

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

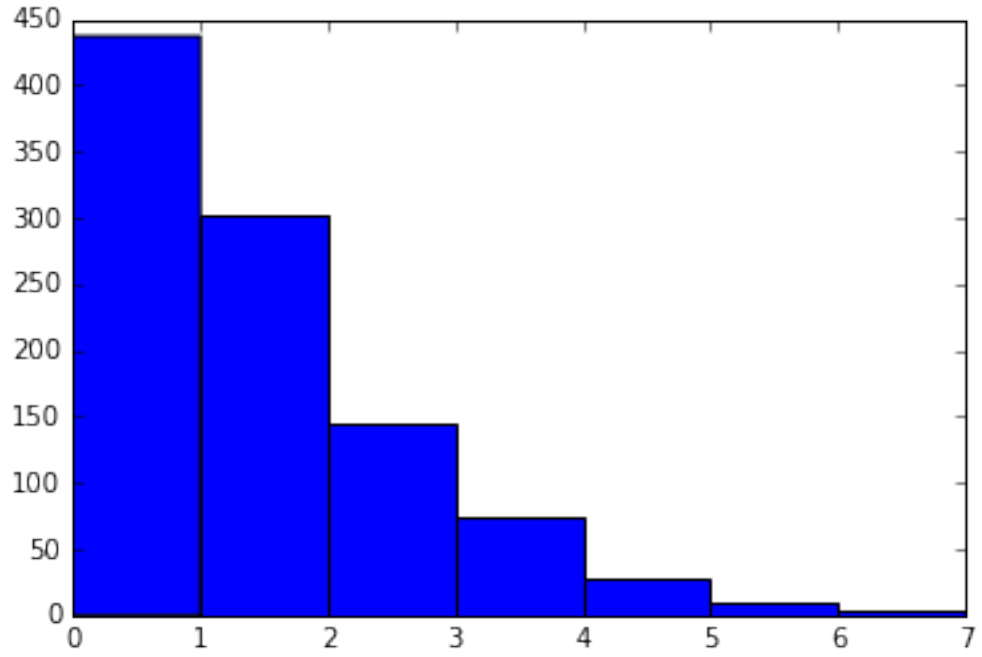


To get a better idea of how a model behaves, we can create multiple model runs and see the distribution that emerges from all of them. We can do this with a nested for loop:

```
# run.py
all_wealth = []
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)

plt.hist(all_wealth, bins=range(max(all_wealth)+1))
```

```
(array([ 437.,  303.,  144.,   75.,   28.,    9.,    4.]),
 array([0, 1, 2, 3, 4, 5, 6, 7]),
 <a list of 7 Patch objects>)
```



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.

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`.

```
# model.py
from mesa.space import MultiGrid
```

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.

```
# model.py
class MoneyModel(Model):
    """A model with some number of agents."""
    def __init__(self, N, width, height):
        self.num_agents = N
        self.grid = MultiGrid(width, height, True)
        self.schedule = 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 = random.randrange(self.grid.width)
            y = 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 (including diagonals), 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:

```
# model.py
class MoneyAgent(Agent):
    #...
```

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

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

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

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

Now, putting that all together should look like this:

```
# model.py
from mesa.space import MultiGrid
from mesa import Agent, Model
from mesa.time import RandomActivation

class MoneyModel(Model):
    """A model with some number of agents."""
    def __init__(self, N, width, height):
        self.num_agents = N
        self.grid = MultiGrid(width, height, True)
        self.schedule = 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 = random.randrange(self.grid.width)
            y = random.randrange(self.grid.height)
            self.grid.place_agent(a, (x, y))

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

class MoneyAgent(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 = 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 = random.choice(cellmates)
            other.wealth += 1
            self.wealth -= 1

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

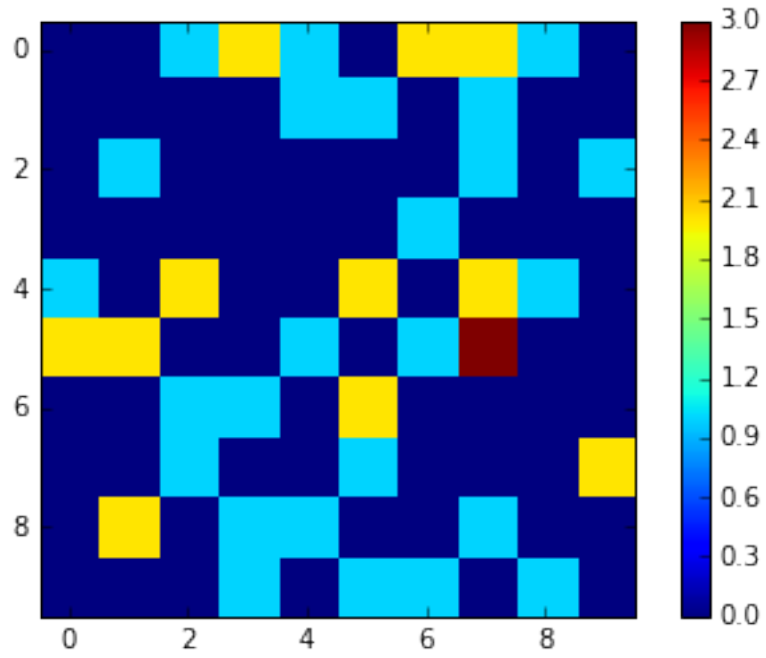
Let's create a model with 50 agents on a 10x10 grid, and run it for 20 steps.

```
# run.py
model = MoneyModel(50, 10, 10)
for i in range(20):
    model.step()
```

Now let's use matplotlib 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 coordinates and contents in turn.

```
# run.py
import numpy as np

agent_counts = np.zeros((model.grid.width, model.grid.height))
for cell in model.grid.coord_iter():
    cell_content, x, y = cell
    agent_count = len(cell_content)
    agent_counts[x][y] = agent_count
plt.imshow(agent_counts, interpolation='nearest')
plt.colorbar()

# If running from a text editor or IDE, remember you'll need the following:
plt.show()
```

```
<matplotlib.colorbar.Colorbar at 0x10fb446a0>
```



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, 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.

```
# model.py
from mesa.datacollection import DataCollector

class MoneyAgent(Agent):
    # ...

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(Model):
    """A model with some number of agents."""
    def __init__(self, N, width, height):
        self.num_agents = N
        self.grid = MultiGrid(width, height, True)
        self.schedule = 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 = random.randrange(self.grid.width)
            y = random.randrange(self.grid.height)
            self.grid.place_agent(a, (x, y))

        self.datacollector = DataCollector(
            model_reporters={"Gini": compute_gini},
            agent_reporters={"Wealth": lambda a: a.wealth})

    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.

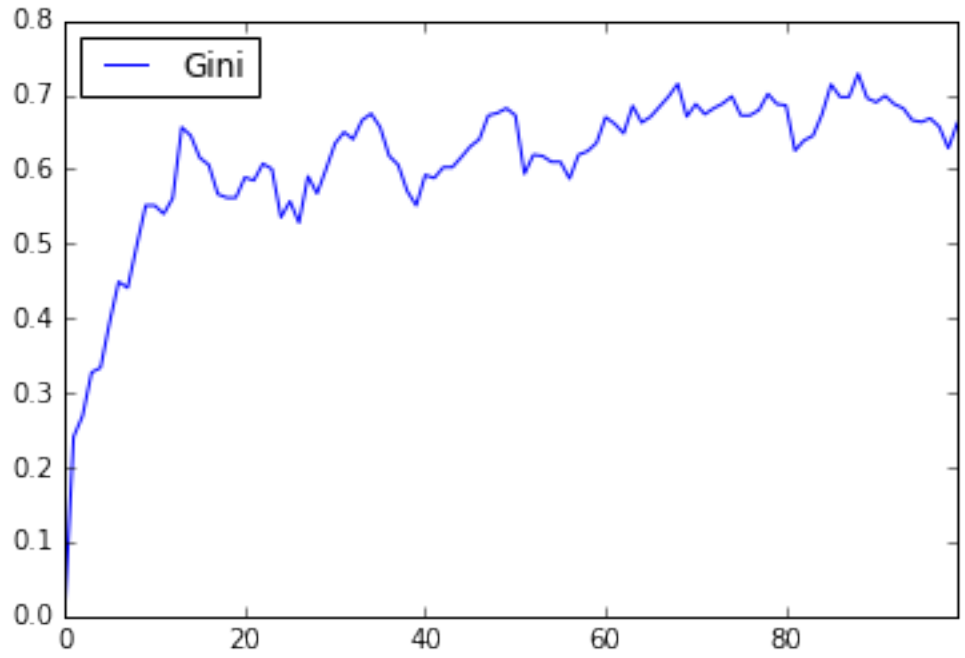
We run the model just as we did above. Now is when an interactive session, especially via a Notebook, comes in handy: the DataCollector can export the data it's collected as a pandas DataFrame, for easy interactive analysis.

```
# run.py
model = MoneyModel(50, 10, 10)
for i in range(100):
    model.step()
```

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

```
# run.py
gini = model.datacollector.get_model_vars_dataframe()
gini.plot()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x10fa4b278>
```



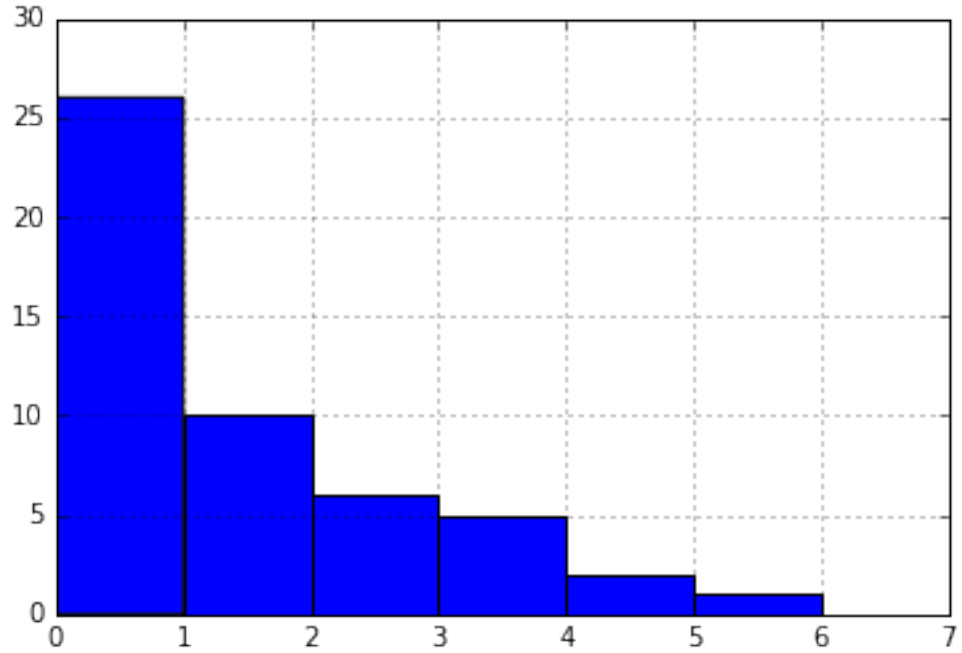
Similarly, we can get the agent-wealth data:

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

You'll see that the DataFrame's index is pairings of model step and 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:

```
end_wealth = agent_wealth.xs(99, level="Step")["Wealth"]
end_wealth.hist(bins=range(agent_wealth.Wealth.max()+1))
```

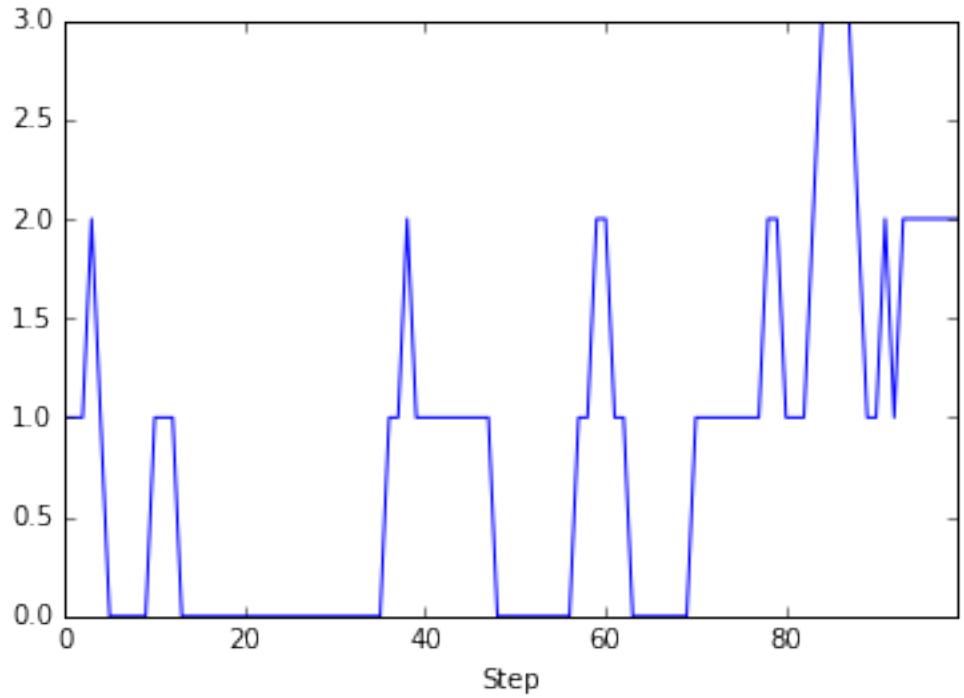
```
<matplotlib.axes._subplots.AxesSubplot at 0x10fa5a978>
```



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

```
one_agent_wealth = agent_wealth.xs(14, level="AgentID")
one_agent_wealth.Wealth.plot()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x10f311438>
```



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 BatchRunner class which automates it for you.

The BatchRunner also requires an additional instance attribute `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 in `__init__`.

```
# model.py
class MoneyModel(Model):
    """A model with some number of agents."""
    def __init__(self, N, width, height):
        self.running = True
        # ...
```

We instantiate a BatchRunner with a model class to run, and two dictionaries: one of the fixed parameters (mapping model arguments to values) and one of varying parameters (mapping each parameter name to a sequence of values for it to take). The BatchRunner also takes an argument for how many model instantiations to create and run at each combination of parameter values, and how many steps to run each instantiation for. Finally, like the DataCollector, it takes dictionaries of model- and agent-level reporters to collect. Unlike the DataCollector, it won't collect the data every step of the model, but only at the end of each run.

In the following example, we hold the height and width fixed, and vary the number of agents. We tell the BatchRunner to run 5 instantiations of the model with each number of agents, and to run each for 100 steps. We have it collect the final Gini coefficient value.

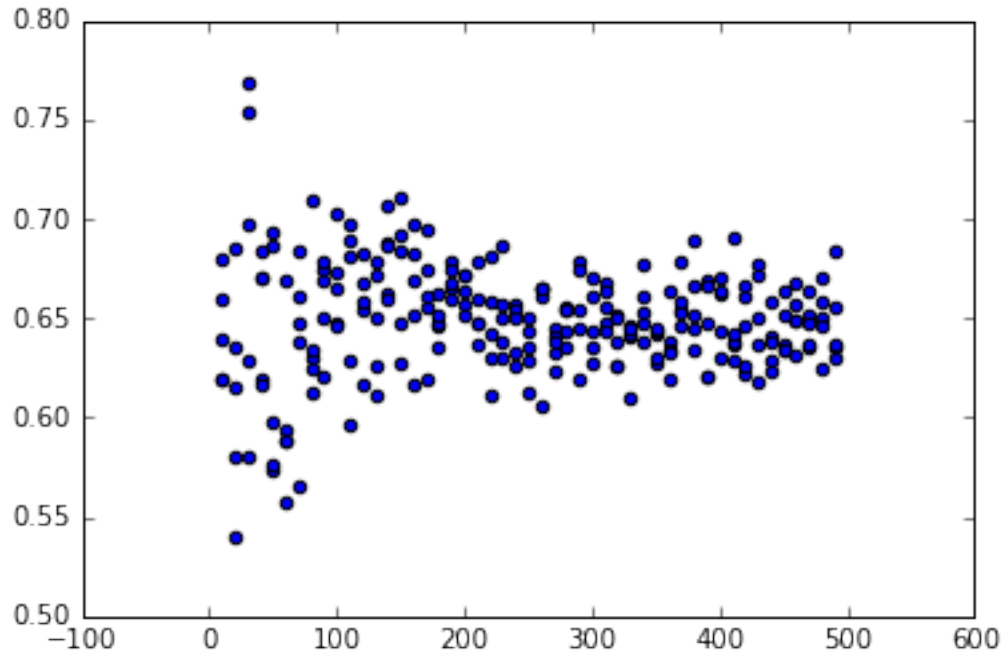
Now, we can set up and run the BatchRunner:

```
# run.py
from mesa.batchrunner import BatchRunner

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

batch_run = BatchRunner(MoneyModel,
                        fixed_parameters=fixed_params,
                        variable_parameters=variable_params,
                        iterations=5,
                        max_steps=100,
                        model_reporters={"Gini": compute_gini})
batch_run.run_all()
```

Like the DataCollector, we can extract the data we collected as a DataFrame.

```
run_data = batch_run.get_model_vars_dataframe()
run_data.head()
plt.scatter(run_data.N, run_data.Gini)
```

```
<matplotlib.collections.PathCollection at 0x114ab80f0>
```



Notice that each row is a model run, and gives us the parameter values associated with that run. We can use this data to view a scatter-plot comparing the number of agents to the final Gini.

Model Best Practices

If you would like to share your model with other people, or to remind yourself of its details when you return to it, you will want to add a few extra bits.

The *Best Practices* document describes the recommended layout for models, including a README and `requirements.txt`.

Happy Modeling!

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

`virtual environment`: http://docs.python-guide.org/en/latest/dev/virtualenvs/

[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://gradworks.umi.com/36/23/3623940.html.

[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.

Advanced Tutorial

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. 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.

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 create a server that will support visualization in a web browser. Set that up in `server.py`.

Import the server class and the Canvas Grid class (so-called because it uses HTML5 canvas to draw a grid). If you're in a new file, you'll also need to import the actual model object.

```
# server.py
from mesa.visualization.modules import CanvasGrid
from mesa.visualization.ModularVisualization import ModularServer
from model import MoneyModel
```

`CanvasGrid` 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.

```
# server.py
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 = 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"
- A dictionary of arguments for the model itself. In this case, `{"N": 100, "width": 10, "height": 10}`

Once we create the server, we set the port for it to listen on (you can treat this as just a piece of the URL you'll open in the browser).

```
# server.py
server = ModularServer(MoneyModel,
                       [grid],
                       "Money Model",
                       {"N": 100, "width": 10, "height": 10})
```

Finally, when you're ready to run the visualization, use the server's `launch()` method, in `run.py`. In this arrangmenet, `run.py` is very short!

```
# run.py
from server import server
server.port = 8521 # The default
server.launch()
```

The full code should now look like:

```
# server.py
from mesa.visualization.modules import CanvasGrid
from mesa.visualization.ModularVisualization import ModularServer
from model import MoneyModel

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

grid = CanvasGrid(agent_portrayal, 10, 10, 500, 500)
server = ModularServer(MoneyModel,
                       [grid],
                       "Money Model",
                       {"N": 100, "width": 10, "height": 10})
```

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, a grid with red circles respresenting the agents, and the model controls at the top.

Click 'step' to advance the model by one step, and the agents will move around. Click 'run' 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 'pause' will (as you'd expect) pause the model; presing 'run' 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.

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.

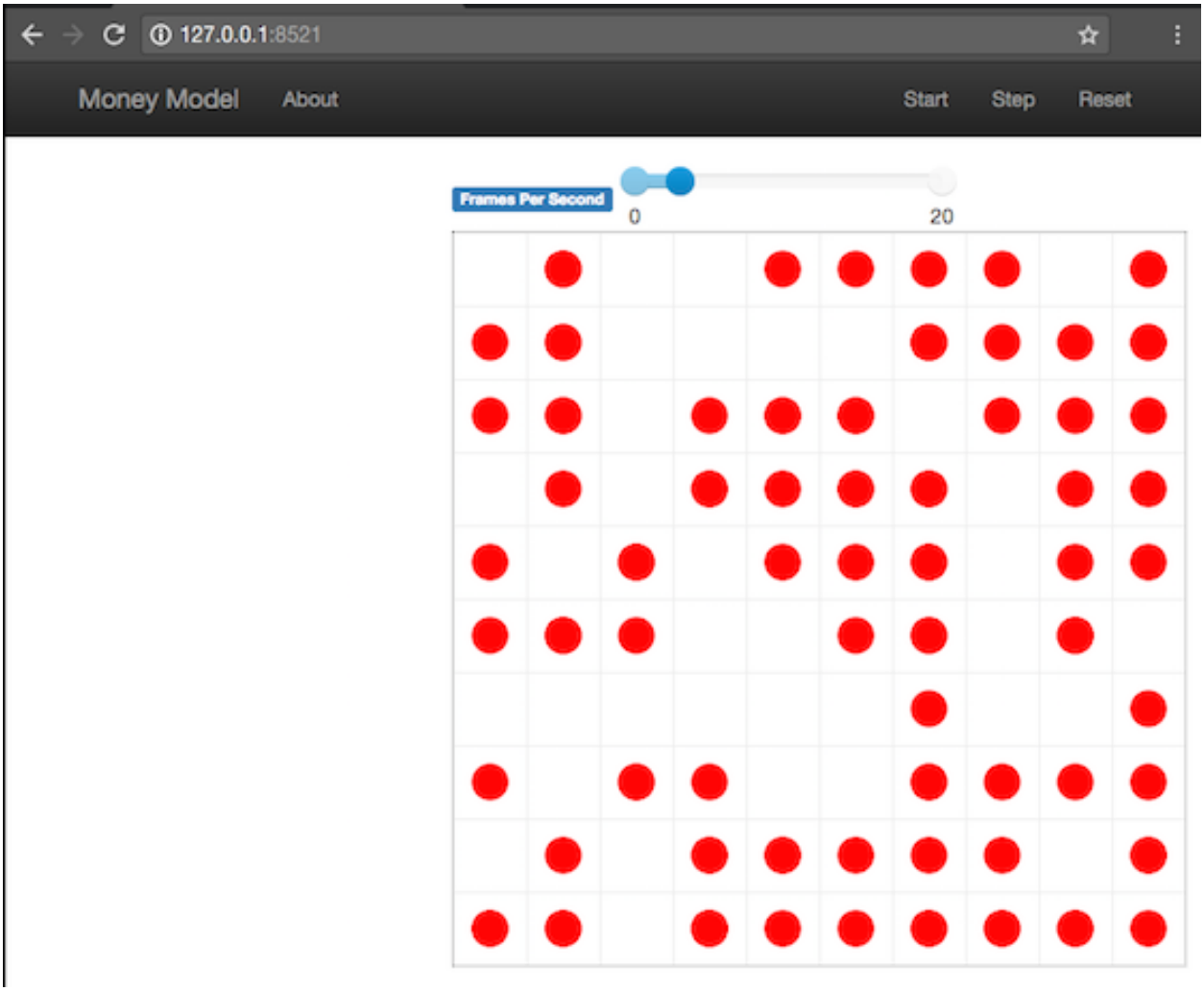


Fig. 3.1: Redcircles Visualization

```
# server.py
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
```

Now launch the server again - 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.

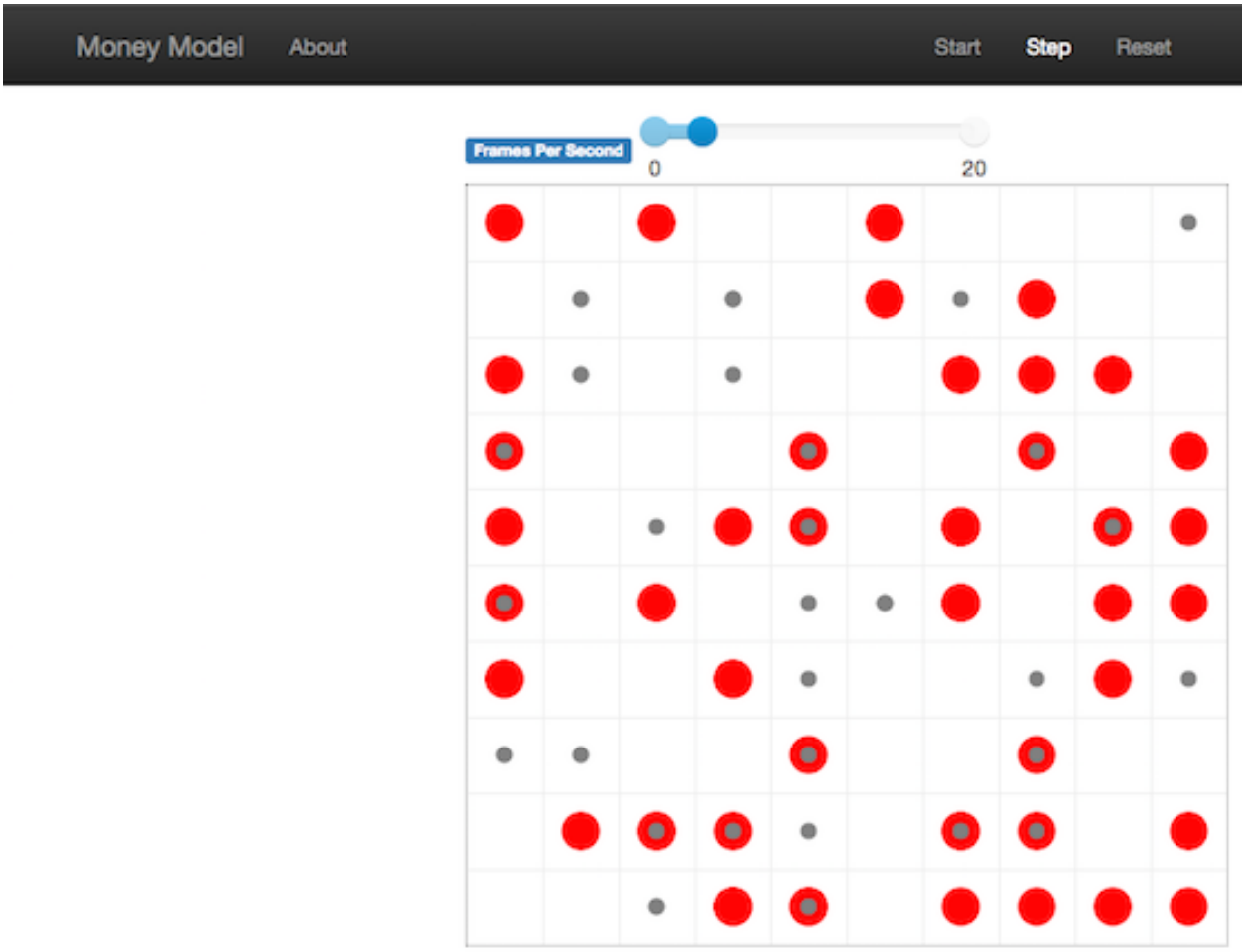


Fig. 3.2: Greycircles Visualization

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.

```
# server.py
from mesa.visualization.modules import ChartModule
```

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.

```
# server.py
chart = ChartModule([{"Label": "Gini",
                      "Color": "Black"}],
                    data_collector_name='datacollector')

server = ModularServer(MoneyModel,
                       [grid, chart],
                       "Money Model",
                       {"N": 100, "width": 10, "height": 10})
```

Launch the visualization and start a model run, and 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.

Making a parameter interactive

One of the reasons we want to be able to watch a model run is to conduct ad-hoc experiments – for example, to get an idea of how the model changes with different parameter values. Having to stop the simulation, edit a parameter value, and relaunch isn't an ideal way to go about it. That's why Mesa lets you set any parameter to be interactive, using the `UserSettableParameter` class.

For this example, we'll add a slider that controls how `N`, many agents there are in the model. To do this, we need to choose the starting value (let's keep this at 100); the minimum parameter value we'll allow (let's do 2, since one agent alone will have nobody to trade with) and the maximum (we'll say 200); and the increment the slider will go in (set this to 1, since there's no such thing as a fraction of an agent). This looks like this:

```
from mesa.visualization.UserParam import UserSettableParameter

n_slider = UserSettableParameter('slider', "Number of Agents", 100, 2, 200, 1)
```

To incorporate it into the model visualization interface, we make the slider one of the model inputs, replacing the static parameter:

```
# server.py
chart = ChartModule([{"Label": "Gini",
                      "Color": "Black"}],
                    data_collector_name='datacollector')

server = ModularServer(MoneyModel,
                       [grid, chart],
                       "Money Model",
                       {"N": n_slider, "width": 10, "height": 10})
```

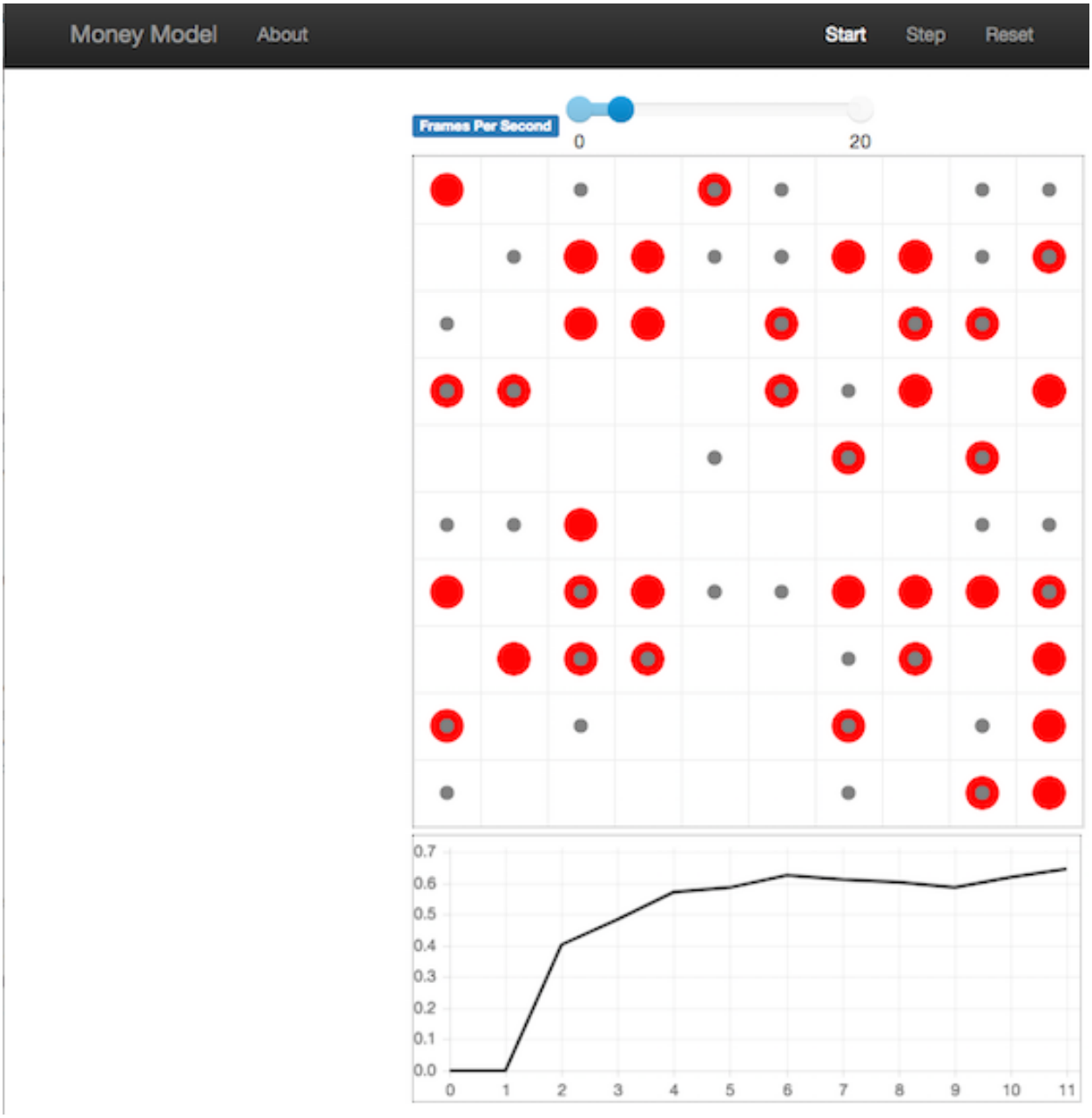


Fig. 3.3: Chart Visualization

When you launch the model, you'll see a slider, labeled "Number of Agents", on the left side of the interface. Try moving the slider around, then press Reset to restart the model with the number of agents you set. Parameter changes don't take effect until you reset the model.

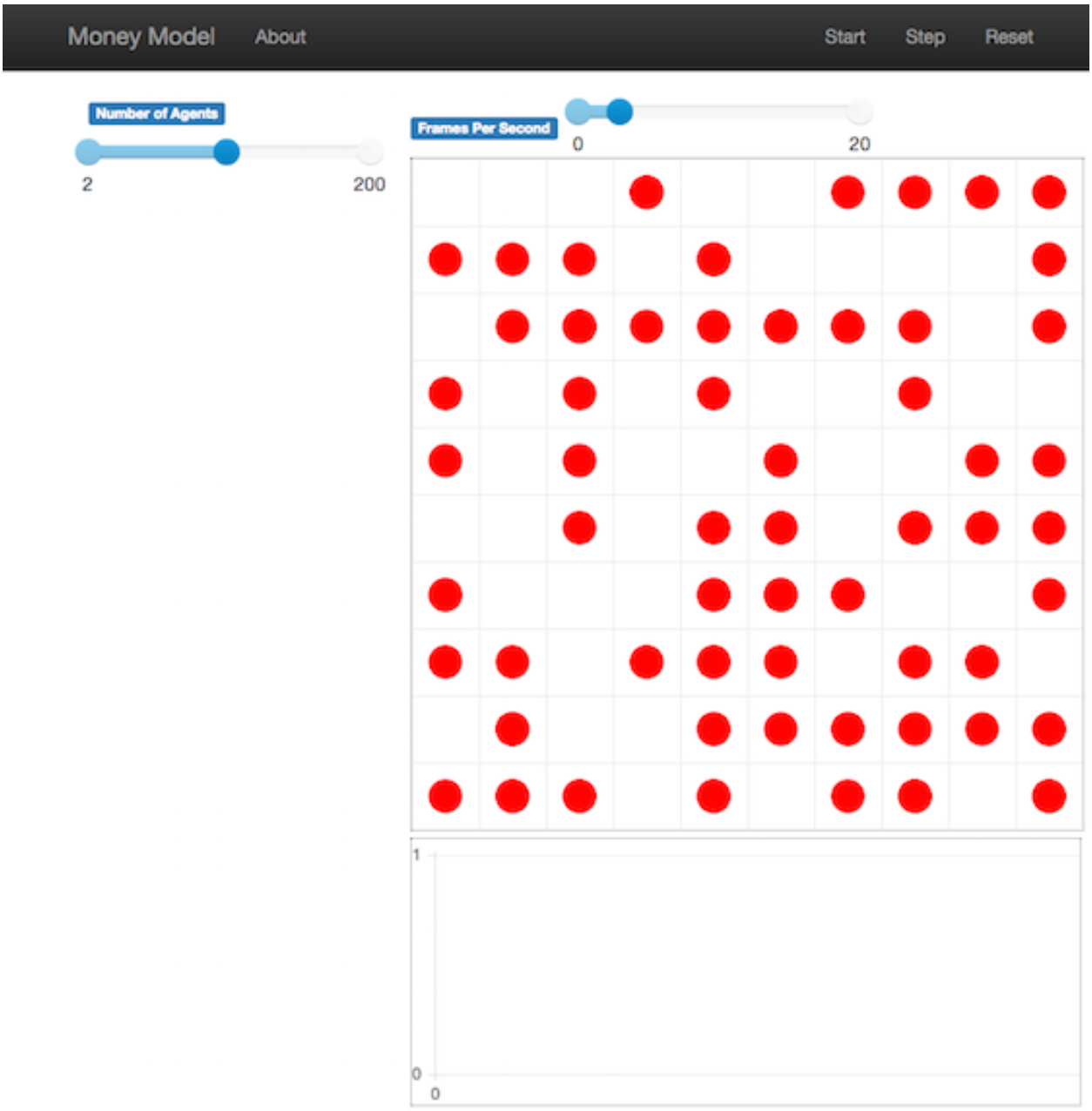


Fig. 3.4: User-Settable Parameter Slider

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.

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:

```
// HistogramModule.js
var 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, using JQuery's dollar-sign syntax (JQuery is already included with Mesa). It also gets the canvas' context, which is required for doing anything with it.

```
// HistogramModule.js
var HistogramModule = function(bins, canvas_width, canvas_height) {
    // Create the tag:
    var canvas_tag = "<canvas width='" + canvas_width + "' height='" + canvas_height␣
↪+ "' ";
    canvas_tag += "style='border:1px dotted'></canvas>";
    // Append it to body:
    var canvas = $(canvas_tag)[0];
    $("body").append(canvas);
    // Create the context and the drawing controller:
    var 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.

```
// HistogramModule.js
var HistogramModule = function(bins, canvas_width, canvas_height) {
    // Create the elements

    // Create the tag:
    var canvas_tag = "<canvas width='" + canvas_width + "' height='" + canvas_height␣
↪+ "' ";
    canvas_tag += "style='border:1px dotted'></canvas>";
    // Append it to body:
    var canvas = $(canvas_tag)[0];
    $("body").append(canvas);
    // Create the context and the drawing controller:
    var context = canvas.getContext("2d");

    // Prep the chart properties and series:
    var 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);

    var data = {
        labels: bins,
        datasets: datasets
    };

    var options = {
        scaleBeginsAtZero: true
    };

    // Create the chart object
    var chart = new Chart(context).Bar(data, options);

    // 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:

```
// HistogramModule.js
var HistogramModule = function(bins, canvas_width, canvas_height) {
    // ...Everything from above...
    this.render = function(data) {
        for (var i in data)
            chart.datasets[0].bars[i].value = data[i];
        chart.update();
    };

    this.reset = function() {
        chart.destroy();
        chart = new Chart(context).Bar(data, 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.

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.

```
# server.py
from mesa.visualization.ModularVisualization import VisualizationElement

class HistogramModule(VisualizationElement):
    package_includes = ["Chart.min.js"]
    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,
                                         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.

```
# server.py
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:

```
# server.py
histogram = HistogramModule(list(range(10)), 200, 500)
server = ModularServer(MoneyModel,
                       [grid, histogram, chart],
                       "Money Model",
                       {"N": n_slider, "width": 10, "height": 10})
```

Run this code, and you should see your brand-new histogram added to the visualization and updating along with the model!

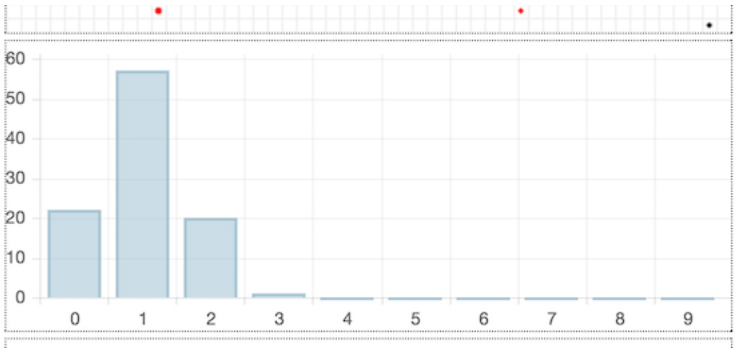


Fig. 3.5: Histogram Visualization

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.

Happy Modeling!

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

Best Practices

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

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.
- `requirements.txt` contains any additional Python distributions, beyond Mesa itself, required to run the model.
- `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 as `python run.py`.

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.

Randomization

If your model involves some random choice, you can use either `random` (Python's built-in random number generator) or `numpy.random` (the generator included with Numpy).

The constructor for the `Model` class automatically "seeds" these random number generators using the current time, so each run will produce different random numbers. For testing purposes, it can be helpful to use the same random-number seed for multiple runs. To accomplish this, pass a value to the Model constructor:

```
class AwesomeModel(Model):
    def __init__(self, seed=None):
        super().__init__(seed)
        # ...

model = AwesomeModel(seed=1234)
# ...
```

This approach will cause `RandomActivation` to activate agents in a repeatable fashion.

APIs

init

Batchrunner

Batchrunner

A single class to manage a batch run or parameter sweep of a given model.

class batchrunner.**BatchRunner**(*model_cls*, *variable_parameters=None*, *fixed_parameters=None*, *iterations=1*, *max_steps=1000*, *model_reporters=None*, *agent_reporters=None*, *display_progress=True*)

This class is instantiated with a model class, and model parameters associated with one or more values. It is also instantiated with model and agent-level reporters, dictionaries mapping a variable name to a function which collects some data from the model or its agents at the end of the run and stores it.

Note that by default, the reporters only collect data at the *end* of the run. To get step by step data, simply have a reporter store the model's entire DataCollector object.

collect_agent_vars(*model*)
Run reporters and collect agent-level variables.

collect_model_vars(*model*)
Run reporters and collect model-level variables.

get_agent_vars_dataframe()
Generate a pandas DataFrame from the agent-level variables collected.

get_model_vars_dataframe()
Generate a pandas DataFrame from the model-level variables collected.

run_all()
Run the model at all parameter combinations and store results.

run_model(*model*)
Run a model object to completion, or until reaching max steps.

If your model runs in a non-standard way, this is the method to modify in your subclass.

batchrunner.**combinations**(**items*)
A small fix to handle dictionary type parameters in cartesian product.

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 functions for each, one for model-level data and one for agent-level data; a third dictionary provides table names and columns.

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 in the model scheduler.

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
- agent_vars maps each reporter to a list of lists, where each nested list stores (agent_id, value) pairs.
- tables maps each table to a dictionary, with each column as a key with a list as its value.

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

The default DataCollector here makes several assumptions:

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

class datacollection.**DataCollector**(*model_reporters={}*, *agent_reporters={}*, *tables={}*)
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 functions which actually collect them. When the collect(...) method is called, it executes these functions one by one and stores the results.

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

collect(*model*)
Collect all the data for the given model object.

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_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_table_dataframe(*table_name*)
Create a pandas DataFrame from a particular table.

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

Mesa Space Module

Objects used to add a spatial component to a model.

Grid: base grid, a simple list-of-lists. SingleGrid: grid which strictly enforces one object per cell. MultiGrid: extension to Grid where each cell is a set of objects.

class space.**ContinuousSpace**(*x_max*, *y_max*, *torus*, *x_min=0*, *y_min=0*, *grid_width=100*, *grid_height=100*)
Continuous space where each agent can have an arbitrary position.

Assumes that all agents are point objects, and have a pos property storing their position as an (x, y) tuple. This class uses a MultiGrid internally to store agent objects, to speed up neighborhood lookups.

get_distance(*pos_1*, *pos_2*)
Get the distance between two point, accounting for toroidal space.

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

get_neighbors(*pos*, *radius*, *include_center=True*)
Get all objects 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.

move_agent(*agent*, *pos*)
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.

out_of_bounds(*pos*)
Check if a point is out of bounds.

place_agent(*agent*, *pos*)
Place a new agent in the space.

Args: agent: Agent object to place. pos: Coordinate tuple for where to place the agent.

torus_adj(*pos*)
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.

class space.**Grid**(*width*, *height*, *torus*)
Base class for a square grid.

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

Properties: width, height: The grid's width and height. torus: Boolean which determines whether to treat the grid as a torus. grid: Internal list-of-lists which holds the grid cells themselves.

Methods: get_neighbors: Returns the objects surrounding a given cell. get_neighborhood: Returns the cells surrounding a given cell. get_cell_list_contents: Returns the contents of a list of cells

((x,y) tuples)

neighbor_iter: Iterates over position neightbors. coord_iter: Returns coordinates as well as cell contents. place_agent: Positions an agent on the grid, and set its pos variable. move_agent: Moves an agent from its current position to a new position. iter_neighborhood: Returns an iterator over cell coordinates that are in the neighborhood of a certain point. torus_adj: Converts coordinate, handles torus looping. out_of_bounds: Determines whether position is off the grid, returns the out of bounds coordinate. iter_cell_list_contents: Returns an iterator of the contents of the cells identified in cell_list. get_cell_list_contents: Returns a list of the contents of the cells identified in cell_list. remove_agent: Removes an agent from the grid. is_cell_empty: Returns a bool of the contents of a cell.

coord_iter()
An iterator that returns coordinates as well as cell contents.

static default_val()
Default value for new cell elements.

exists_empty_cells()
Return True if any cells empty else False.

find_empty()
Pick a random empty cell.

get_neighborhood(*pos*, *moore*, *include_center=False*, *radius=1*)
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_neighbors(*pos*, *moore*, *include_center=False*, *radius=1*)
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*)
Returns a bool of the contents of a cell.

iter_neighborhood(*pos*, *moore*, *include_center=False*, *radius=1*)
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: 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).

iter_neighbors(*pos*, *moore*, *include_center=False*, *radius=1*)
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*, *pos*)
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_to_empty(*agent*)
Moves agent to a random empty cell, vacating agent's old cell.

neighbor_iter(*pos*, *moore=True*)
Iterate over position neighbors.

Args: pos: (x,y) coords tuple for the position to get the neighbors of. moore: Boolean for whether to use Moore neighborhood (including

diagonals) or Von Neumann (only up/down/left/right).

out_of_bounds(*pos*)
Determines whether position is off the grid, returns the out of bounds coordinate.

place_agent(*agent*, *pos*)
Position an agent on the grid, and set its pos variable.

remove_agent(*agent*)
Remove the agent from the grid and set its pos variable to None.

torus_adj(*coord*, *dim_len*)
Convert coordinate, handling torus looping.

class space.**MultiGrid**(*width*, *height*, *torus*)
Grid where each cell can contain more than one object.

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.

Each grid cell holds a set object.

Properties: width, height: The grid's width and height.

torus: Boolean which determines whether to treat the grid as a torus.

grid: Internal list-of-lists which holds the grid cells themselves.

Methods: get_neighbors: Returns the objects surrounding a given cell.

static default_val()
Default value for new cell elements.

class space.**SingleGrid**(*width*, *height*, *torus*)
Grid where each cell contains exactly at most one object.

position_agent(*agent*, *x='random'*, *y='random'*)
Position an agent on the grid. This is used when first placing agents! Use 'move_to_empty()' when you want agents to jump to an empty cell. Use 'swap_pos()' to swap agents positions. If x or y are positive, they are used, but if "random", we get a random position. Ensure this random position is not occupied (in Grid).

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

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.

TODO: Have the schedulers use the model's randomizer, to keep random number seeds consistent and allow for replication.

class mesa.time.**BaseScheduler**(*model*)
Simplest scheduler; activates agents one at a time, in the order they were added.

Assumes that each agent added has a *step* method which takes no arguments.

(This is explicitly meant to replicate the scheduler in MASON).

add(*agent*)
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()
Returns the current number of agents in the queue.

remove(*agent*)
Remove all instances of a given agent from the schedule.

Args: agent: An agent object.

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

class mesa.time.**RandomActivation**(*model*)
A scheduler which activates each agent once per step, in random order, with the order reshuffled every step.

This is equivalent to the NetLogo 'ask agents...' and is generally the default behavior for an ABM.

Assumes that all agents have a step(model) method.

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

class mesa.time.**SimultaneousActivation**(*model*)
A scheduler to simulate the simultaneous activation of all the agents.

This scheduler requires that each agent have two methods: step and advance. step() activates the agent and stages any necessary changes, but does not apply them yet. advance() then applies the changes.

step()
Step all agents, then advance them.

class mesa.time.**StagedActivation**(*model*, *stage_list=None*, *shuffle=False*, *shuffle_between_stages=False*)
A scheduler which allows agent activation to be divided into several stages instead of a single *step* method. All agents execute one stage before moving on to the next.

Agents must have all the stage methods implemented. Stage methods take a model object as their only argument.

This schedule tracks steps and time separately. Time advances in fractional increments of 1 / (# of stages), meaning that 1 step = 1 unit of time.

step()
Executes all the stages for all agents.

Visualization

Mesa Visualization Module

TextVisualization: Base class for writing ASCII visualizations of model state.

TextServer: Class which takes a TextVisualization child class as an input, and renders it in-browser, along with an interface.

ModularServer

A visualization server which renders a model via one or more elements.

The concept for the modular visualization server as follows: A visualization is composed of VisualizationElements, each of which defines how to generate some visualization from a model instance and render it on the client. VisualizationElements may be anything from a simple text display to a multilayered HTML5 canvas.

The actual server is launched with one or more VisualizationElements; it runs the model object through each of them, generating data to be sent to the client. The client page is also generated based on the JavaScript code provided by each element.

This file consists of the following classes:

VisualizationElement: Parent class for all other visualization elements, with the minimal necessary options.

PageHandler: The handler for the visualization page, generated from a template and built from the various visualization elements.

SocketHandler: Handles the websocket connection between the client page and the server.

ModularServer: The overall visualization application class which stores and controls the model and visualization instance.

ModularServer should *not* need to be subclassed on a model-by-model basis; it should be primarily a pass-through for VisualizationElement subclasses, which define the actual visualization specifics.

For example, suppose we have created two visualization elements for our model, called canvasvis and graphvis; we would launch a server with:

server = ModularServer(MyModel, [canvasvis, graphvis], name="My Model") server.launch()

The client keeps track of what step it is showing. Clicking the Step button in the browser sends a message requesting the viz_state corresponding to the next step position, which is then sent back to the client via the websocket.

The websocket protocol is as follows: Each message is a JSON object, with a "type" property which defines the rest of the structure.

Server -> Client: Send over the model state to visualize. Model state is a list, with each element corresponding to a div; each div is expected to have a render function associated with it, which knows how to render that particular data. The example below includes two elements: the first is data for a CanvasGrid, the second for a raw text display.

{ "type": "viz_state", "data": [{0:[{"Shape": "circle", "x": 0, "y": 0, "r": 0.5,

"Color": "#AAAAAA", "Filled": "true", "Layer": 0, "text": 'A', "text_color": "white" }]},

"Shape Count: 1"]

}

Informs the client that the model is over. {"type": "end"}

Informs the client of the current model's parameters { "type": "model_params", "params": 'dict' of model params, (i.e. {arg_1: val_1, ...}) }

Client -> Server: Reset the model. TODO: Allow this to come with parameters { "type": "reset" }

Get a given state. { "type": "get_step", "step:" index of the step to get. }

Submit model parameter updates { "type": "submit_params", "param": name of model parameter "value": new value for 'param' }

Get the model's parameters { "type": "get_params" }

class visualization.ModularVisualization.**ModularServer**(*model_cls*, *visualization_elements*, *name='Mesa Model'*, *model_params={}*)

Main visualization application.

launch(*port=None*)
Run the app.

render_model()
Turn the current state of the model into a dictionary of visualizations

reset_model()
Reinstantiate the model object, using the current parameters.

class visualization.ModularVisualization.**PageHandler**(*application*, *request*, ***kwargs*)
Handler for the HTML template which holds the visualization.

class visualization.ModularVisualization.**SocketHandler**(*application*, *request*, ***kwargs*)
Handler for websocket.

on_message(*message*)
Receiving a message from the websocket, parse, and act accordingly.

class visualization.ModularVisualization.**VisualizationElement**
Defines an element of the visualization.

Attributes:

package_includes: A list of external JavaScript files to include that are part of the Mesa packages.

local_includes: A list of JavaScript files that are local to the directory that the server is being run in.

js_code: A JavaScript code string to instantiate the element.

Methods:

render: Takes a model object, and produces JSON data which can be sent to the client.

render(*model*)
Build visualization data from a model object.

Args: model: A model object

Returns: A JSON-ready object.

Text Visualization

Base classes for ASCII-only visualizations of a model. These are useful for quick debugging, and can readily be rendered in an IPython Notebook or via text alone in a browser window.

Classes:

TextVisualization: Class meant to wrap around a Model object and render it in some way using Elements, which are stored in a list and rendered in that order. Each element, in turn, renders a particular piece of information as text.

TextElement: Parent class for all other ASCII elements. render() returns its representative string, which can be printed via the overloaded __str__ method.

TextData: Uses getattr to get the value of a particular property of a model and prints it, along with its name.

TextGrid: Prints a grid, assuming that the value of each cell maps to exactly one ASCII character via a converter method. This (as opposed to a dictionary) is used so as to allow the method to access Agent internals, as well as to potentially render a cell based on several values (e.g. an Agent grid and a Patch value grid).

class `visualization.TextVisualization.`**`TextData`**(*model*, *var_name*)
Prints the value of one particular variable from the base model.

class `visualization.TextVisualization.`**`TextElement`**
Base class for all TextElements to render.

Methods: render: 'Renders' some data into ASCII and returns. __str__: Displays render() by default.

`render`()
Render the element as text.

class `visualization.TextVisualization.`**`TextGrid`**(*grid*, *converter*)
Class for creating an ASCII visualization of a basic grid object.

By default, assume that each cell is represented by one character, and that empty cells are rendered as ' ' characters. When printed, the TextGrid results in a width x height grid of ascii characters.

Properties: grid: The underlying grid object.

static `converter`(*x*)
Text content of cells.

`render`()
What to show when printed.

class `visualization.TextVisualization.`**`TextVisualization`**(*model*)
ASCII-Only visualization of a model.

Properties:

model: The underlying model object to be visualized. elements: List of visualization elements, which will be rendered

in the order they are added.

`render`()
Render all the text elements, in order.

`step`()
Advance the model by a step and print the results.

Modules

Container for all built-in visualization modules.

Modular Canvas Rendering

Module for visualizing model objects in grid cells.

class visualization.modules.CanvasGridVisualization.**CanvasGrid**(*portrayal_method*, *grid_width*, *grid_height*, *canvas_width=500*, *canvas_height=500*)

A CanvasGrid object uses a user-provided portrayal method to generate a portrayal for each object. A portrayal is a JSON-ready dictionary which tells the relevant JavaScript code (GridDraw.js) where to draw what shape.

The render method returns a dictionary, keyed on layers, with values as lists of portrayals to draw. Portrayals themselves are generated by the user-provided portrayal_method, which accepts an object as an input and produces a portrayal of it.

A portrayal as a dictionary with the following structure: "x", "y": Coordinates for the cell in which the object is placed. "Shape": Can be either "circle", "rect" or "arrowHead"

For Circles:

"r": The radius, defined as a fraction of cell size. r=1 will fill the entire cell.

For Rectangles:

"w", "h": The width and height of the rectangle, which are in fractions of cell width and height.

For arrowHead: "scale": Proportion scaling as a fraction of cell size. "heading_x": represents x direction unit vector. "heading_y": represents y direction unit vector.

"Color": The color to draw the shape in; needs to be a valid HTML color, e.g."Red" or "#AA08F8"

"Filled": either "true" or "false", and determines whether the shape is filled or not.

"Layer": Layer number of 0 or above; higher-numbered layers are drawn above lower-numbered layers.

"text": The text to be inscribed inside the Shape. Normally useful for showing the unique_id of the agent.

"text_color": The color to draw the inscribed text. Should be given in conjunction of "text" property.

Attributes:

portrayal_method: Function which generates portrayals from objects, as described above.

grid_height, grid_width: Size of the grid to visualize, in cells. canvas_height, canvas_width: Size, in pixels, of the grid visualization

to draw on the client.

template: "canvas_module.html" stores the module's HTML template.

Chart Module

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

class visualization.modules.ChartVisualization.**ChartModule**(*series*, *canvas_height=200*, *canvas_width=500*, *data_collector_name='datacollector'*)

Each chart can visualize one or more model-level series as lines with the data value on the Y axis and the step number as the X axis.

At the moment, each call to the render method returns a list of the most recent values of each series.

Attributes:

series: A list of dictionaries containing information on series to plot. Each dictionary must contain (at least) the "Label" and "Color" keys. The "Label" value must correspond to a model-level series collected by the model's DataCollector, and "Color" must have a valid HTML color.

canvas_height, canvas_width: The width and height to draw the chart on the page, in pixels. Default to 200 x 500

data_collector_name: Name of the DataCollector object in the model to retrieve data from.

template: "chart_module.html" stores the HTML template for the module.

Example:

schelling_chart = ChartModule([{"Label": "happy", "Color": "Black"}],
data_collector_name="datacollector")

TODO: Have it be able to handle agent-level variables as well.

More Pythonic customization; in particular, have both series-level and chart-level options settable in Python, and passed to the front-end the same way that "Color" is currently.

Text Module

Module for drawing live-updating text.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index

O

P

R

S

T

V