Agent-Based Modeling: Week 4

Winter 2023

Jean Clipperton

Agenda:

- Review of server file
- Data Collector
- Model: Schelling

Review: Server file

Relevant elements:

- Header: what we import
- Agent portrayal
- Any reporting elements to display
- Canvas / grid
- Elements in the vizualization
- Set up the server

Data collection and reporting

This is where things will get a bit complicated. We're going to work backwards by starting with what information we WANT to display / have available, and then we'll go over how to get it.

The short of it is that you are able to report / generate charts based on the underlying data you have already tracked / collected using the **data collector**.

Reporting

Here, we're talking about general ways to display what and how we want to do things.

The main options you have are some combination of

- **Charts**: thinking about tracking statistics in real-time
- **Values / text elements**: here, you might put up a counter, for example, of how many of X are alive / available, etc

Reporters during the simulation

Reporters: Text Element

You can generate text elements (I think of this as value reporters) to give live / immediate feedback on some value in the model.

When to use this?

If you have some kind of metric or stat that you want to include. Maybe there's a measure or indext that relates to the analysis -- that would be great to include.

Reporters: Example pseudocode

To include your reporter, consider whether you want just one value, the value next to another, any text, etc.

```
class ElementName(TextElement):
    def render(self, model):
        return (whatever you want it to return)
```

Reporters: Example code

Here's a simple reporter followed by a more complicated one:

```
class SimilarElement(TextElement):
    def render(self, model):
        return "Avg. % similar neighbors: " + str(model.pct_neighbors) + "%"
```

```
class SimilarElement_g(TextElement):
    def render(self, model):
        return "Groups avg. % similar neighbors: (A) " + str(model.pct_neighbors0) + "%"
            + " (B) " + str(model.pct_neighbors1) + "%"
```

But wait, there's more!

You ALSO need to call the new element later in your server file.

Step 1: you refer to your text elements

```
similar_element = SimilarElement()
```

Step 2: you call these elments in setting up the server

```
server = ModularServer(
    SegModel,
    [canvas_element, happy_element,
     similar_element, similar_element_g,
     happy_chart, ],
    "Schelling's Segregation Model",
    model_params
)
```

Reporters: ChartModule

These modules are for tracking something over time -- it can be the same statistic you're choosing to report in the text element reports or something different entirely.

When to use this?

This is valuable when you're trying to not only look at some value but how it changes and/or interacts with other elements.

How to use this:

Two elements: in the server file and in the model file

Chartmodule Example: pseudocode (model file)

```
(inside __init__)

self.datacollector = DataCollector(
    model_reporters = {
        "LABEL YOU WANT": (what that thing is)
    },
    agent_reporters = {
        "LABEL YOU WANT": (what that thing is)
    }
)
```

Chartmodule Example: pseudocode (model file)

```
Chart_name = ChartModule(
    [{"Label": "NAME YOU WANT", "Color": "COLOR YOU WANT"}])
```

Chartmodule Example: code (model file)

```
        # somewhat extensive data collection
        self.datacollector = DataCollector(
            model_reporters={"Pct Happy": lambda m: round(100 * m.happy / m.num_agents, 1),
                             "Pct Happy Group A": lambda m: round(100 * m.happy0 / m.num_agents0, 1),
                             "Pct Happy Group B": lambda m: round(100 * m.happy1 / m.num_agents1, 1),
                             ...
                             "Intolerance": lambda m: m.intolerance},
                             #note: can do as follows:
                             #"Intolerance": "intolerance"

            # Model-level count of happy agents  + subgroup counts
            agent_reporters={"Similar_empty": lambda a: round(100 * a.similar / 8, 1),
                             "Similar_no_empty": lambda a: a.a_pct_similar,
                             "Agent type": lambda a: a.type}
            # Agent-level reporters can allow for individual measures
```

Chartmodule Example: code (server file)

```
happy_chart = ChartModule([{"Label": "Pct Happy", "Color": "Black"}])
happy_chart0 = ChartModule([{"Label": "Pct Happy Group A", "Color": "Black"}])
happy_chart1 = ChartModule([{"Label": "Pct Happy Group B", "Color": "Gray"}])
```

Server file

Relevant elements:

- Header: what we import
- Agent portrayal
- Any reporting elements to display
- Canvas / grid
- Elements in the vizualization
- Set up the server

How nice should it be?

This is a tough question to ask and depends on the following components:

1. Your own preferences (some people want things to be always polished)
2. The audience for the model (will the model be used by anyone other than you, for example)
3. The kind of project (is this a proof of concept approach, is this going to be how you generate findings, etc)

DataCollector: getting the data you want

We are trying to determine what happens over time / overall. To do this, we want to look at multiple runs.

We may also have questions about how parameter values interact with one another.

DataCollector: FixedBatchRunner pseudocode

(I do this in a separate file because I like things tidy)

```
from model import SegModel
from mesa.batchrunner import FixedBatchRunner

# parameters that will remain constant
fixed_parameters = {
}

#these vary
parameters_list = {
batch_run = FixedBatchRunner(SegModel, parameters_list,
                             fixed_parameters, iterations=10,
                             model_reporters={ list},
                             agent_reporters={list},
                             max_steps=NUMBER
}

batch_run.run_all()

#export
# extract data as a pandas Data Frame
batch_df = batch_run.get_model_vars_dataframe()

# export the data to a csv file for graphing/analysis
batch_df.to_csv("data/seg_model_batch_run_data.csv")
```

DataCollector: FixedBatchRunner example code

(from schelling complex)

```
from model import SegModel
from mesa.batchrunner import FixedBatchRunner

# parameters that will remain constant
fixed_parameters = {
    "height": 20,
    "width": 20,
    "num_agents": 350,
    "minority_pc": 0.4,
}

# parameters you want to vary
# can also include combinations here
parameters_list = [{"intolerance": 0.125},
                   {"intolerance": 0.25},
                   {"intolerance": 0.375},
                   {"intolerance": 0.5},
                   {"intolerance": 0.625},
                   {"intolerance": 0.75}]

# what to run and what to collect
# iterations is how many runs per parameter value
# max_steps is how long to run the model
batch_run = FixedBatchRunner(SegModel, parameters_list,
                             fixed_parameters, iterations=10,
                             model_reporters={"Pct Happy": lambda m: round(100 * m.happy / m.num_agents, 1),
                                              "Pct Happy Group A": lambda m: round(100 * m.happy0 / m.num_agents0, 1),
                                              "Pct Happy Group B": lambda m: round(100 * m.happy1 / m.num_agents1, 1),
                                              "Avg pct similar neighbors": lambda m: m.pct_neighbors,
                                              "Avg pct similar neighbors (A)": lambda m: m.pct_neighbors0,
                                              "Avg pct similar neighbors (B)": lambda m: m.pct_neighbors1,
                                              "Avg pct similar neighbors (count empty)": lambda m: m.pct_neighbors_e,
                                              "Avg pct similar neighbors (A) (count empty)": lambda
                                                  m: m.pct_neighbors_e0,
                                              "Avg pct similar neighbors (B) (count empty)": lambda
                                                  m: m.pct_neighbors_e1,
                                              "Num Agents": lambda m: m.num_agents,
                                              "Num Agents (A)": lambda m: m.num_agents0,
                                              "Num Agents (B)": lambda m: m.num_agents1},
                            agent_reporters={"Type": "type"},
                             max_steps=100)

# run the batches of your model with the specified variations
batch_run.run_all()


## NOTE: to do data collection, you need to be sure your pathway is correct to save this!
# Data collection
# extract data as a pandas Data Frame
batch_df = batch_run.get_model_vars_dataframe()
batch_df_a = batch_run.get_agent_vars_dataframe()

# export the data to a csv file for graphing/analysis
batch_df.to_csv("schelling_complex/data/seg_model_batch_run_data.csv")
batch_df_a.to_csv("schelling_complex/data/seg_agent_batch_run_data.csv")
```

DataCollector: graphing it!!!

HOW YOU DISPLAY YOUR DATA MATTERS!!!

(see additional slides / resources)

Questions?

TASK: Set up a single and batch run

- What parameters did you decide to vary? (be specific about the values you chose and include the code)
- Why did you choose these parameters?
- How long did it take to run?
- How are you planning to analyze your results?

*Include a graphic of your results and an interpretation -- how will you use this information?