Agent Based Modeling With RePast Py

By Jeff Heaton

Agent Based Modeling (ABM) can be used to simulate highly complex systems. ABM's are designed to model how participants in a complex system interact with each other. This puts ABM into the category of unsupervised learning. Unlike linear regression and generalized linear models (GLM) you do not fit past data and receive a prediction on future data. Rather, you set a model into motion and observe how different parts of the model interact over time. Additionally, ABM is always time series, as ABM always occurs over a defined time range.

ABM models attempt to deal with the chaos theory concept of the "butterfly effect" by modeling how a small change in one part of the model might have a huge change in another. In 1972, Philip Merilees stated the theory as, "Does the flap of a butterfly's wings in Brazil set off a tornado in Texas?" Consider if we simply wanted to model the profitability of an active and passive real estate investment strategy. The difference between the two approaches could be compared using historic data. Similarly, forecasts of the two models might use analyst predictions about market value and volatility. However, such predictions assume there is no interaction between the buyers and sellers. If many investors follow the more active investment strategy, does the increase in transactions affect real estate prices paid by practitioners of both investment strategies?

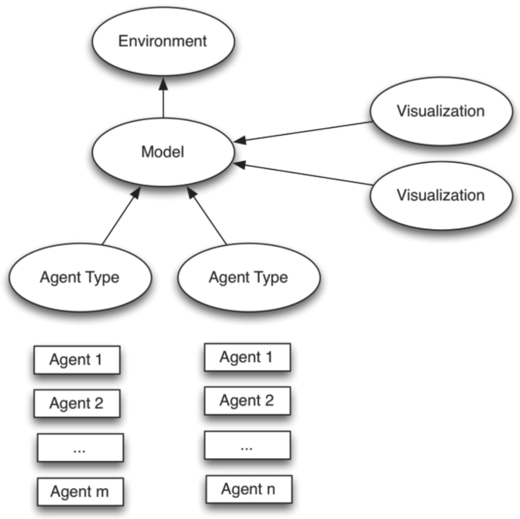
Agent based modeling attempts to consider these interactions. To construct an agent based model you create several different types of agent. To model real estate investment you would create agent types of investors following active and passive investment strategies. However, you would likely also create agent types for properties, builders, lending institutions, real estate agents/brokers, and many others. State variables are defined for each of these agent types. These variables would include cash on hand, property value, properties owned, and others. Additionally, the agents would form relationships, or links. An investor agent would form a link to every property that they owned. Finally, you would define actions. Actions are usually implemented in a programming language, such as Python, Java, C++, C# or R. Actions define how the agents affect each other. The action allows you to specify the effect that an agent purchasing a property has on all of the other agent types.

INTRODUCING REPAST

This article will introduce you to the Recursive Porous Agent Simulation Toolkit for Python (Repast Py), a free open-source ABM platform that is part of a suite of products that includes Repast Simphony, Repast for High Performance Computing and several language-specific instances of Repast. Repast Py allows the agent actions to be defined using a specialized scripting language based on Python 2.7. Repast was originally developed by David Sallach, Nick Collier, Tom Howe, Michael North and others at the University of Chicago. Repast is widely used. Google Scholar lists more than 50,000 citations and references. Additionally, Repast was used by Alan Mills for the "Simulating health behavior" SOA research project, sponsored by the SOA Health section.

Repast's architecture is composed of an Environment, Model, Agent Types and Visualizations. The Environment allows your agents to access supplemental information, such as files or a database. The model holds global model values and actions. This structure is shown in Figure 1.

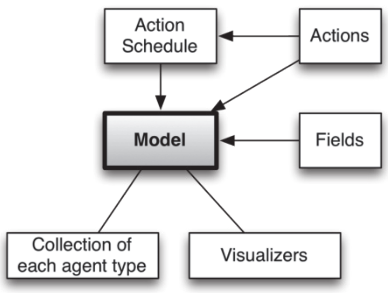
Figure 1: Repast Py Architecture



CONTINUED ON **PAGE 32**

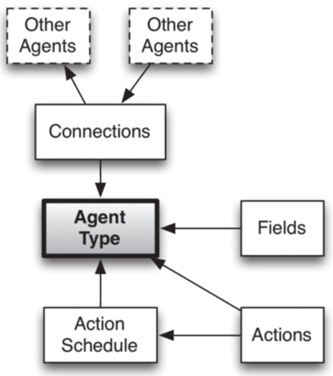
The model can hold actions and properties that are global to the agents. For example, you might store the prime interest rate at the model level. Agent types can also hold properties and these properties will be unique to each agent instance that is created. This structure is shown in Figure 2.

Figure 2: Models & Environments



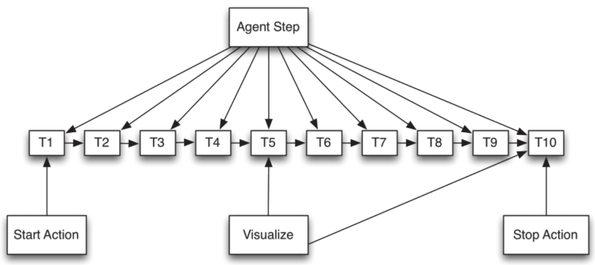
Agents are the real workhorse of an ABM model. One or more agent types are defined for a model. The model can have many instances of an agent type. Larger numbers of agents allow closer approximations to reality. The example provided for this article makes use of 10,000 agents. Agents contain actions that define how the agents interact. Agents can be competitive, cooperative or oblivious to other agents, depending on how their action code is constructed. Agents also form links to each other and maintain private and public properties. Figure 3 shows an agent type.

Figure 3: Agents



Agent based modeling is a simulation that runs in time. ABM time is expressed in a series of uniform intervals called ticks. Schedules are created that define the intervals that agent and model actions occur. Additionally, actions can be scheduled to only occur at the beginning or end of the simulation run. Visualizers are tied to intervals to define the granularity of their display. Figure 4 shows how actions are scheduled with the passage of time.

Figure 4: Passage of Time



CREATING A SIMPLE REPAST MODEL

For this article I created a simple ABM with Repast Py. You can download the source code to this article from GitHub, at the following URL:

https://github.com/jeffheaton/soa

The example is stored under the folder annual-2014 because this is an example that I presented at the SOA 2014 annual meeting. This example simulation can be used as a starting point for other simulations of your own.

This example seeks to model the following very simple scenario.

- Simple model of insurer response times to meet varying consumer demand for five insurance products.
- Two agent types: consumer and insurer.
- Consumers demand one of five products. Once demand is satisfied, consumer will cycle to the next product. (e.g., 1->2,…,4->5, 5->1)

- Insurers supply one product, and may retool when half of requests are unfilled.
- Initial set of insurers offer random products chosen uniformly.
- Initial set of consumers demand random products chosen uniformly.
- Model will track the rise and fall of the demand of each product on a linear plot.
- Initial setup will be 10,000 consumer agents and 10 insurer agents.
- Experiment with different insurer counts.

To implement this we create a consumer and insurer agent. The consumer agent has a property that defines what product the consumer is currently demanding. The insurer has properties that define both the product currently supplied, as well as the current cash on hand. Cash is not really used by the current model, however, it could be used as a performance visualization. The model is setup with each consumer agent demanding a different product. Likewise, each insurer agent is set to providing a random product.

Listing 1: Model Setup

```
# Cause the customers to demand random
products.
for consumer as ConsumerAgent in self.
consumers:
 product_num = Random.uniform.nextInt-
FromTo(0, 4)
 consumer.setProductDesired(product_num)
# Cause the insurers to offer random
products.
for insurer as InsurerAgent in self.
insurers:
 product_num = Random.uniform.nextInt-
FromTo(0, 4)
 insurer.setCurrentProduct(product_num)
```

Step actions occur at time intervals specified by the schedules. The step actions for the agents specify their interaction with the other agent. The step action for the consumer agent selects a random insurer and demands a product. If the insurer is unable to process this demand, then the failure is recorded, and the consumer demands no further products this tick. This is accomplished by the step action shown in Listing 2.

Listing 2: Consumer Step Action

```
# Choose a random insurer to obtain the
product from.
insurer_num = Random.uniform.nextIntFrom-
To(0, self.model.insurers.size()-1)
insurer = (InsurerAgent)self.model.insur-
ers.get(insurer_num)
# If the insurer has the product, then
obtain it.
if insurer.getCurrentProduct() == self.
productDesired:
 insurer.setCash(insurer.getCash()+1)
 self.requestFilled=self.productDesired
 self.productDesired = self.productDesired
+ 1
 insurer.setFilledRequests(insurer.get-
FilledRequests()+1)
 # Change our product desired, simply
cycle between 0 and 4.
 if self.productDesired>=5:
  self.productDesired=0
else:
# If the insurer does not have the prod-
uct, then record that.
 insurer.setFailedRequests(insurer.get-
FailedRequests()+1)
 self.requestFilled=-1
```

CONTINUED ON **PAGE 34**

AGENT BASED MODELING … | FROM PAGE 33

The insurer step action evaluates failed requests from customers. If the insurer failed to fulfil 50 percent of the requests, then the insurer might retool and offer a different product. There is a cost associated with retooling. This process is shown in Listing 3.

Listing 3: Insurer Step Action

```
# Did we fail to fulfill any orders (pre-
vent div/0)?
if self.failedRequests>0:
 ratio = self.filledRequests / self.faile-
dRequests
 # Did we fail to fulfill 50% of the re-
quests?
 if ratio < 0.5:
  refit = Random.uniform.nextDouble()
  # Do we want to retool?
  if refit<self.model.probNewProduct:
   self.currentProduct =
      Random.uniform.nextIntFromTo(0, 4)
   self.cash = self.cash - 5
```

Python code is used to perform visualizations. Most visualizers in Repast are time-series line charts. The visualizer for this example shows the demand of all five of the products. Each line on the chart requires its own step action. Listing 4 shows the step action for Listing?? 1's (index 0) line.

Listing 4: Visualizer Step Action

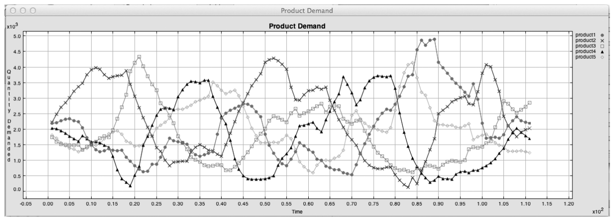
```
sum = 0
for consumer as ConsumerAgent
   in self.consumers:
 if consumer.getProductDesired()
   ==0:
  sum = sum + 1
return sum
```

The above code works by looping over all consumers and counting the number of consumers demanding product #0.

INTERPRETING THE RESULTS OF THE MODEL

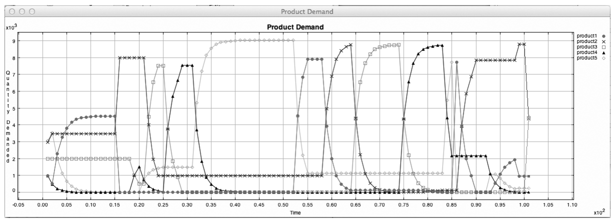
To see some of the insights that the simulation will provide, I created two visualizations. Each of these visualizations has 10,000 consumer agents. Figure 5 shows the simulation with 10 insureds.

Figure 5: Visualizer with 10 Insurers



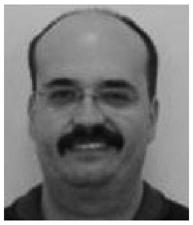
As you can see from the above, the demand for each product rises and falls somewhat smoothly. Because there are enough insurers, each product can be offered by one or two insurers at a time. If we force the model down to only two insurers we get much different results, as seen in Figure 6.

Figure 6: Visualizer with 2 Insurers



With fewer products being offered, the consumers are forced into narrow bands of demand. The transitions between products being offered are very sharp. Though the duration of demand for each product is somewhat random, the order in which the products are demanded in Figure 6 is mostly consistent.

ABM's are a great tool for forecasting the future through simulation. There are many considerations for building your own models. Often you will start the ABM somewhere in the past and tweak the parameters so that prediction matches reality up to the current date. The model then runs into the future providing predictions. Increasing the number of agents can provide more accurate results; however, it is important to ensure that the ratio of agent types makes sense. ABM's are a technology where you can start simple and increase the complexity of your model to handle increasingly complex situations. ABM can be an important part of your toolbox. ▼


Jeff Heaton

Jeff Heaton, is data scientist at RGA Reinsurance Company and author of several books on artificial intelligence. He can be reached at *jheaton@rgare.com.*