Abstract

We refine a prominent set of template models for agent-based modeling, and we offer new reference implementations. We also address some issues of design, flexibility, and ease of use that are relevant to the choice of an agent-based modeling platform.

Keywords: Template Models, Reference Implementations, Spatially Situated Agents, Spatially Distributed Resources

Introduction

1.1 Railsback et al. (2006) discuss sixteen template models, which they designed as tools to introduce and explore agent-based modeling (ABM) platforms. (Railsback et al. (2005) provide full model specifications.) These template models are intentionally "ridiculously simplified". They illustrate general modeling considerations required by many different real-world applications but do not constitute an attempt to implement a specific real-world application.

1.2 These template models have already proved their usefulness. They are a common point of reference for introductory presentations of new ABM platforms (Standish 2008). And because of their simplicity, they are also used in teaching. Nevertheless, in this paper we propose to remove certain ambiguities, tighten certain specifications, remove from the specification some implementation details, highlight additional learning goals, and draw a clearer distinction between general programming goals and goals focused on the visual display of information.

1.3 Railsback et al. (2005) provide reference implementations of the template models for a few popular platforms. These reference implementations are intended to be "simple and intuitive" rather than clever or fast. As might be expected, the reference implementations clarify the intent of certain parts of the specifications and highlight some of their limitations. The simplest and least verbose of the reference implementations are written in NetLogo (an ABM platform written in Java and Scala). This is perhaps unsurprising: Robertson (2005) and Sklar (2007) emphasize NetLogo's reputation for ease of use and compactness of representation. The NetLogo reference implementations are therefore particularly relevant as a point of comparison for other ABM platforms.

1.4 We provide new reference implementations (as a zip archive that may be downloaded and decompressed to yield 16 Python programs and a 'ReadMe' file). Readers may also find it useful to download and review the original reference implementations for at least one platform. (We recommend the NetLogo implementations.) For our new reference implementations, we use the Python programming language, but our emphasis is not language choice but rather the refinement of the template models. Our choice of programming language for the reference implementations is driven by two main considerations: readability, and compactness of presentation. Following the intent of the original reference implementations, our code emphasizes readability and ease of use
This paper has two core objectives: to refine the Railsback et al. (2005) template model specifications, and to offer heavily-commented new reference implementations. In the pursuit of these goals, we address some issues of design, flexibility, and ease of use that are relevant to the choice of an agent-based modeling platform.

## Template Model 1: Spatial Movement of Agents

### 2.1 The first template model provides a very basic introduction to ABM platforms. The core programming goal is to implement relocatable agents in a simple two-dimensional space. Spatially situated agents are at the core of a huge variety of agent-based models in the biological and social sciences, including the famous Sugarscape model (Epstein and Axtell 1996). As in the Sugarscape model, agents in the first template model have positions on a finite discrete grid, and two different agents cannot simultaneously occupy the same location. Behavior in this model will be very simple: an agent moves by randomly picking a new location.

### 2.2 The background needed to attempt the first template model is platform dependent, but generally it requires a modest introduction to programming on the chosen ABM platform. Required skills include the use of basic built-in functions, definition and use of new functions, some understanding of variable scope (local vs. global), use of a looping construct, iteration over a collection (of agents), and simple randomization (using built-in facilities). With this background, we can attempt the specification of the first template model.

**Specification: Template Model 1**

<table>
<thead>
<tr>
<th>World</th>
</tr>
</thead>
<tbody>
<tr>
<td>a toroidal grid of possible agent positions</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Setup</th>
</tr>
</thead>
<tbody>
<tr>
<td>create an iterable collection of agents</td>
</tr>
<tr>
<td>give each agent a unique, random position on the grid</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>world_shape: the grid dimensions (100 × 100)</td>
</tr>
<tr>
<td>n_agents: the number of agents (100)</td>
</tr>
<tr>
<td>maxiter: the maximum number of iterations (unspecified)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>each time step, each agent moves to a random, unoccupied neighboring location</td>
</tr>
</tbody>
</table>

### Stopping Condition
- Not part of the specification.

### Supplementary Detail
- agent positions are characterized by integer pairs (locations on the grid)
- unique-location constraint: no two agents share a position, not even initially
- movement is random in a specified neighborhood
  - the new position is a “random” selection from the unoccupied neighboring locations (no particular algorithm or PRNG is specified)
  - neighborhood: a Moore neighborhood of radius 4
  - an agent must change position unless all neighboring locations are occupied
  - the move action must terminate, even if all neighboring locations are occupied
- sequential movement: the order in which agents move is unspecified, but agents must move sequentially (i.e., one at a time)
- a "time step" is one iteration through the model schedule

### GUI and display suggestions
- position: display each agent on screen at a position corresponding to its grid position
- shape: display agents as circles
- size: make sure adjacent agents are easily distinguishable
- color: use a red fill color for agents
- background color: display agents against a white or black background
- update display once each iteration

Despite its simplicity, the original specification of the first template model contained some minor ambiguities. The emphasized text in our specification (above) constitutes additions to the specification. These are simple
clarifications except for the required use of parameters, which we add to the specification as a matter of good practice. We specify that the initial positions of agents are chosen randomly subject to the constraint that locations are not shared. This addition matches the NetLogo reference implementation and is not very consequential. We also add that agents should be members of an iterable collection. Most ABM platforms support iterable collections for each agent type, so this too is not very consequential. More substantively, we explicitly specify that agents move sequentially. (This matches the reference implementations.) Sequential movement ensures that two agents do not move to the same location, thereby ensuring satisfaction of the unique-location constraint. As Kahn (2007) notes, violation of this constraint can affect outcomes in later template models. (Meeting the unique-location constraint in the absence of sequential movement can pose challenges for parallelization (Standish 2008.).)

2.4 The reference implementations randomly draw from the entire neighborhood until a vacant cell is found. As Kahn (2007) notes, if there are no vacant cells, this loop never terminates. (Admittedly, this is a low probability event in the first template model.) We address this problem by specifying that the move action must terminate. However we do not augment the specification to require a specific algorithm, since the ease of any particular implementation is platform specific.

2.5 One drawback of the original template model specifications is the lack of a clean separation of core modeling goals and visual display considerations (including GUI considerations). We cannot address this fully without diverging substantially from the original specifications, and our goal is to refine rather than replace these specifications. Nevertheless, a natural supplement to the template models is a cleaner separation of these concerns, and we will provide this on a template by template basis. One other drawback of the original specification is the unnecessary narrowness of the original interpretive framework. The authors of the original template model specifications presumed an ecological emphasis and light-heartedly referred to their agents as "bugs". Our interest encompasses the social sciences, and we therefore stick with the more generic term ‘agents’.

Procedural Implementation

2.6 To implement the first template model, we need a description of the agents and a description of the world in which the agents reside. The template models also make visual display suggestions, and we will implement these as well. Given that we will use the Python language, we have many different options. Starting from scratch is entirely feasible, and free and open source game development toolkits are also options. For the purposes of this paper, however, we take a minimalist approach: we import a small gridworld module that provides a basic grid topology as well as agents and patches.

2.7 The properties of a gridworld Agent resemble those of a NetLogo turtle, which is the mobile NetLogo agent type. For the purposes of the template models, we care primarily about the ability of these agents to report a current position and move to a new location. We will access and set the position of an agent via its position attribute. A position is an x,y pair of numbers, interpreted as Cartesian coordinates in a plane. (The coordinates may be floating point numbers, but coordinates in the template models are integers.)

2.8 To facilitate comparison with the NetLogo reference implementations, which are necessarily procedurally oriented, we now present a procedurally-oriented version of the first template model. (We will say a model is procedurally-oriented when we build functions to do things to our objects rather than providing these objects with methods that allow them to act.) Our world will be an instance of gridworld.GridWorld, a class that handles a little grid-related accounting. For example, if myworld is a GridWorld instance, we can determine whether a location is unoccupied as myworld.is_empty(location). With this background, we get the following implementation of the first template model. (For completeness, we include the entire file.)

""" Template Model 1 (procedural): Random Movement on a Toroidal Grid """

import random
from gridworld import Agent, TorusGrid, GridWorld
from gridworld import moore_neighborhood, GridWorldGUI

params = dict(world_shape=(100,100), n_agents=100, maxiter=100)

def move(agent):
    choice = choose_location(agent)
def choose_location(agent):
    old_position = agent.position
    hood = moore_neighborhood(radius=4, center=old_position)
    random.shuffle(hood)
    for location in hood:
        if agent.world.is_empty(location):
            return location
    return old_position

def schedule():
    for agent in myagents:
        move(agent)

def run(maxiter):
    for ct in range(maxiter):
        myobserver.off()
        schedule()
        myobserver.on()

# create a grid and a world based on this grid
mygrid = TorusGrid(shape=params['world_shape'])
myworld = GridWorld(topology=mygrid)
# create agents, located in `myworld`, and then set display suggestions
myagents = myworld.create_agents(AgentType=Agent, number=params['n_agents'])
for agent in myagents:
    agent.display(shape='circle', fillcolor='red', shapesize=(0.25, 0.25))
# add an observer
myobserver = GridWorldGUI(myworld)
# run the simulation by repeatedly executing the schedule
run(maxiter=params['maxiter'])
2.13 The choose_location function does the real work of the first template model. We generate a Moore neighborhood of radius 4 around the agent's current position as moore_neighborhood(radius=4, center=old_position), which returns a list of eighty (x,y) locations. An agent must move to a random location in its neighborhood, so we shuffle the locations with the random module's shuffle function. (The shuffle function does in-place shuffling of list elements.) We sequentially consider each location as a possible new position (thereby avoiding an error in the original reference implementations, discussed above). If the location is empty, we choose it. Note that an Agent has a world attribute. (When a world creates agents it sets this attribute.) To determine whether a location is empty, an agent queries its world's is_empty method. If no new location is empty, the agent chooses its current position. The choose_location function returns a location, and the move function sets the agent's position to this location.

2.14 We have completed the hardest and most essential part of the first template model. The schedule function is relatively simple: it applies the move function to each of our agents. The run function calls schedule once per iteration as long as the model is running. (It also turns the observer off and on, to reduce the number of screen updates by our observer, as suggested in the specification.)

2.15 We are now ready to set up and run the model. Recall that we had set params['world_shape'] = (100,100), so mygrid=TorusGrid(shape=params['world_shape']) creates the 100 by 100 grid called for by the template specification. We then create our world as myworld = GridWorld(topology=mygrid), initializing our world with our torus topology. We then populate the world with 100 agents: a GridWorld has a create_agents convenience method, so we need only specify the number and type of agents that we want. (We could additionally specify positions, but since we do not, the agents are positioned randomly when they are created.)

At this point we could run the model, but we would not gather any evidence of its behavior. Monitoring a world is usually done by an observer. Our observer will be a GridWorldGUI: a graphical observer that provides a visual display of our agent locations. But what will be the display characteristics of our agents? Agents have a display method that allows us to make visual display suggestions to graphical observers. 7 Our display implements the suggestions in the first template model specification. We create our observer with myobserver = GridWorldGUI(myworld).

2.17 To run the simulation, we call run(maxiter=params['maxiter']). The running model calls the schedule function repeatedly. (This has been arbitrarily parameterized to 100 iterations.) Since we chose a GUI observer, the behavior of the simulation will be displayed on screen.

Object-Oriented Implementation

2.18 We now reimplement the first template model with more object-oriented code. One cost of a more object-oriented approach is that it immediately requires some understanding of class definition. Python class definition proves to be a minimal barrier. Understanding how NetLogo handles instance variables, as required by the second template model, is a comparable cost. As a somewhat more substantial cost, object-oriented design immediately requires a minimal understanding of inheritance: that a derived class behaves like its base class, in the sense the it can receive the same method calls. These costs accrue primarily to programming novices, which make them largely of pedagogical relevance. (This matters, since pedagogy is a common use of the template models.) As we develop our object-oriented approach to the first template model, we will not find large advantages off-setting these initial costs. However, as we work through additional template models, substantial advantages of a more object-oriented approach become evident.

2.19 For most agent-based modeling projects, the issue is not whether but rather when to use inheritance-related concepts. For example, NetLogo is strongly procedurally oriented and is designed for simple use, so the NetLogo reference implementations initially dodge inheritance-related issues by relying on (in essence) dynamic attribute creation for base agent classes (using the turtles-own and patches-own keywords). But once a model requires different types of agents, a discussion of the NetLogo concept of breed becomes unavoidable. Any reasonable introduction to breeds will strongly overlap elementary discussions of class definition and inheritance. 8

2.20 The NetLogo reference models do not introduce breeds until template model 16. As illustrated by our procedural version of the first template model, Python implementations could similarly postpone the introduction of inheritance. Postponement slightly decreases the background required to implement the first template model,
but it does not change the requirements for the set of template models. Additionally, an object-oriented approach substantially simplifies the set of models. A comparison of our previous procedural implementation with the following object-oriented implementation permits an assessment of the barrier to entry raised by early use of a more object-oriented approach.

2.21 We again utilize the Agent and GridWorld classes of the gridworld module, but now they become base classes for two new classes, Agent01 and World01. Instead of defining global functions to manipulate the objects in our model, we provide our objects with behavior by defining corresponding methods. We add a move method to our agents, which will be supported by a choose_location method. These are analogues of the move and choose_location functions in the procedural version. Similarly, the schedule function in the procedural version becomes the schedule method of our new world class. We should note that a GridWorld already defines schedule and run methods. The run method repeatedly calls the schedule method, which in turn is intended to be overridden in subclasses. So we propose the following design.

```python
Agent01 extends gridworld.Agent
New Methods: move, choose_location

World01 extends gridworld.GridWorld
Overridden Methods: schedule
```

2.22 Here we use the term ‘extends’ to indicate inheritance: Agent01 inherits data and methods from Agent. Specifically, we utilize the position and the world data attributes that Agent01 inherits from Agent. Similarly, we utilize the is_empty method that World01 inherits from GridWorld, which reports whether a given location is empty or occupied by another agent.

```python
class Agent01(Agent):
    def move(self):
        choice = self.choose_location()
        self.position = choice
    def choose_location(self):
        old_position = self.position
        hood = moore_neighborhood(radius=4, center=old_position)
        random.shuffle(hood)
        for location in hood:
            if self.world.is_empty(location):
                return location
        return old_position

class World01(GridWorld):
    def schedule(self):
        for agent in self.agents:
            agent.move()

if __name__ == '__main__':
    # setup and run the simulation
    mygrid = TorusGrid(shape=params['world_shape'])
    myworld = World01(topology=mygrid)
    myagents = myworld.create_agents(AgentType=Agent01, number=params['n_agents'])
    for agent in myagents:
        agent.display(shape='circle', fillcolor='red', shapesize=(0.25, 0.25))
    observer = GridWorldGUI(myworld)
    myworld.run(maxiter=params['maxiter'])
    observer.mainloop()
```

2.23 A reasonable first reaction is that this more object-oriented implementation appears nearly identical to our procedural implementation. The move and choose_location functions defined in the procedural version are clearly evident as the new methods defined in the Agent01 class. (A method definition is just a normal function definition that is part of a class definition.) The schedule function defined in the procedural version similarly becomes a method of our World01 class. The code to set up and run the model is also almost identical. A
closer look reveals some small differences, and we now focus on those.

2.24 Again the module begins with a short docstring. The three import statements are unchanged, as is the parameter specification. We then define a new class, `Agent01`, which inherits from `Agent` and defines two new methods. The `move` method differs very slightly from our earlier function of the same name. First of all, the parameter name is `self` rather than `agent`. This is inessential: the choice of the name `self` is a standard Python convention to emphasize that when we call this method on an instance, it will act on that instance itself. (We will return to this.) Related to this, we have the method call `self.choose_location()`, instead of the function call `choose_location(agent)`. The `choose_location` method is essentially identical to our previous `choose_location` function.

2.25 So far we have introduced only very minor changes in program logic, but they all reflect a more fundamental change in perspective: behavior is now an attribute of an agent. To give this a slightly misleading but nevertheless suggestive phrasing, in the procedural design we do things to agents, whereas in the object-oriented design our agents do things.

2.26 Our remaining changes are just as minor. Instead of defining a global `schedule` function, we subclass `GridWorld` and override its `schedule` method. The `schedule` method has three notable changes from our earlier function of the same name. First, we do not have to turn our observer on and off. (The `run` method that our `World01` inherits from `GridWorld` handles this for us.) Second, we do not access `myagents` as a global variable: instead we use the `agents` attribute of our `World01`, which it inherits from `GridWorld`. Finally, we replace the function call `move(agent)` with the method call `agent.move()`.

2.27 As before, we must take four steps to run our first simulation: create a grid of locations for our agents, create a world based on that grid, populate the world with initialized agents, and run the scheduled actions repeatedly. Each of these steps is familiar, and only the last changes substantively from our procedural implementation. A `World01` has a run method (inherited from `GridWorldGUI`) that repeatedly calls its own `schedule` method, so we run our simulation by calling `myworld.run(maxiter=params['maxiter'])`. This completes our more object-oriented implementation of the first template model.

2.28 We add one modification that is not needed to implement the model: we run the simulation only if `__name__ == 'main'`. This is a purely forward looking change: we want to import `Agent01` and `World01` into other modules without running our first template model. However, a module’s code is executed when the module is imported. We could put the code to set up and run the model in a separate file, but for the purposes of this paper, we wish to group our class definitions beside the code that creates and runs our simulations. Our solution is to use the special module attribute, `__name__`. This is just the filename (as a string) for an `imported` module. In contrast, the module executed as the main program is always assigned the string `__main__` as its name. We can therefore condition on the value of `__name__` in order to prevent some code from being executed when a module is imported. The code in the body of this `if` statement will not be executed if the module is imported, but it will be executed if the module is executed as a script. This ensures that we can freely import this module, and we will do so in our implementation of the second template model.

Template Model 2: Dynamic Agent State

3.1 The second template model assumes completion of the first template model. (The template models are usually sequential.) The new requirement is the addition of dynamically changing agent state. The model addresses this by endowing our agents with a new attribute, which for concreteness we will call `size`, along with a specification of how agent state changes over time. A subsidiary visual-display goal is the provision of visual clues to the state of each agent. Once again, the emphasized text in our specification constitutes additions or changes to the original.

**Specification: Template Model 2**

World and Setup

unchanged from model 1

**Parameters**

as in model 1, plus

- `agent_initial_size`: the initial size of an agent (0.0)
• **extraction_rate**: determines the change in agent size (0.1)

**Iteration (sequential)**
- each agent moves
- each agent changes size

**Supplementary Detail**
- **movement**: agents move as in model 1
- **size change**: the agent's `size` attribute is augmented by the `extraction_rate`

**GUI and display suggestions**
- position, shape, and update: as in model 1
- **color**: each agent’s fill color should represent its "size". (Specifically, use white if size=0.0, red if size>=10.0, and increasingly chromatic tints of red as "size" increases from 0 to 10.)
- **delay**: slow the iterations enough to ensure attribute changes are observable

### 3.2
We clear up one ambiguity in the original specification, which says that growth “is scheduled after the move action”. In accord with the NetLogo reference implementation, we interpret this to mean that all agents move, and after that all agents grow. (Although such sequencing decisions can matter substantially in other settings, they are inessential for the second template model.)

### 3.3
Our only substantive change is to explicitly parameterize the model, but we make two minor changes: we change the initial size of agents to 0.0, and we reduce the `extraction_rate` parameter from 1.0 to 0.1. (We refer to this as an “extraction rate” rather than say a “growth increment” in order to better match subsequent template models.) These changes are inessential to this template model. The first eliminates an inconsistency in agent initialization across the template models, and the second merely supports our visual display goals. Even with this smaller extraction rate, on modern hardware the color transitions and even agent movements during the first 100 iterations will take place far too quickly to be easily seen. We therefore add a new display suggestion: schedule a delay if needed for comfortable viewing of the color transitions. (Some might prefer to leave this to user discretion. For example, the NetLogo GUI automatically provides a slider allowing user control of iteration speed.)

### 3.4
The notion of “size” in this model is intentionally very abstract, and we introduce a correspondingly abstract notion of “extraction”. A biologist might conceive of the `size` attribute as representing the physical size of the agent, while for a social scientist it might represent the value of the agent’s wealth. (For example, in the Sugarscape model, our `size` attribute would represent the agent’s “sugar wealth” (Epstein and Axtell 1996).) In either case, it is natural to link `size` changes to the agent’s ability to extract resources from its environment.

### 3.5
With this background, we are ready to propose a basic design. Naturally, we would like to take advantage of our work on the first template model. One approach would be to literally copy our `Agent01` code into the file that implements the second template model, modifying it where appropriate. (Examination of the NetLogo reference implementations reveals the use of this copy-and-paste approach.) For languages that support inheritance, a much better approach is to import into our second template model any useful objects from the first template model. We therefore subclass `Agent01` to define a new agent class: `Agent02` has a new data attribute (size) and three new methods (`change_size`, `extract`, and `change_color`). Note that `Agent` defines an `initialize` method, which it automatically calls as part of object initialization. This method is intended to be overridden by users who want to provide object initializations in subclasses. We will override this `initialize` method in order to set the agent’s display suggestions and initial `size`. Finally, we create another new class, named `World02`, to implement our new schedule of actions. Since `World01` does not offer much here, we once again subclass `GridWorld` in order to override its `schedule` method.

```plaintext
Agent02 extends Agent01
New data attribute: size
New Methods: change_size, extract, change_color
Overridden Methods: initialize

World02 extends GridWorldGUI
Overridden Methods: schedule
```

### 3.7
With this design in hand, we are now ready to implement the second template model. We begin with three import statements. From the Python standard library’s `time` module we import the `sleep` function, which we
will use to delay our model iterations. From the gridworld module we import an inessential but convenient function (ask). Analogously to the ask command in NetLogo, this ask function makes a specified method call on each object in an iterable collection. (Thus ask(myagents, 'move') is equivalent to for agent in myagents: agent.move().) Finally, from our (obviously named) template01 module we import the entire namespace, using the asterisk (*) wild-card syntax. This gives us access to Agent01, which we intend to subclass, and the params dictionary of parameter values, to which we add our two new parameters (agent_initial_size and extraction_rate). It also gives us access to TorusGrid, GridWorld, and GridWorldGUI, which template01 imported from gridworld.

""" Template Model 2: Dynamic Agent State """
from time import sleep
from gridworld import ask
from template01 import *
params.update(agent_initial_size=0.0, extraction_rate=0.1)
class Agent02(Agent01):
    def initialize(self):
        self.size = params['agent_initial_size']
        self.display(shape="circle", shapesize=(0.25,0.25))
        self.change_color()
    def change_size(self):
        self.size += self.extract()
        self.change_color()
    def extract(self):
        return params['extraction_rate']
    def change_color(self):
        g = b = max(0.0, 1.0 - self.size/10)
        self.display(fillcolor=(1.0, g, b))
class World02(GridWorld):
    def schedule(self):
        sleep(0.2)
        ask(self.agents, 'move')
        ask(self.agents, 'change_size')
if __name__ == '__main__':
    myworld = World02(topology=TorusGrid(shape=params['world_shape']))
    myagents = myworld.create_agents(Agent02, number=params['n_agents'])
    myobserver = GridWorldGUI(myworld)
    myworld.run(maxiter=params['maxiter'])
    myobserver.mainloop() # keep GUI open after `run` completes

3.8 Our new agent class, Agent02, has new methods (change_size, extract, and change_color) and a new data attribute (size). In addition we define an initialize method (which overrides the method of that name provided by Agent). Here we initialize each agent’s size based on the agent_initial_size parameter. We also make shape and shapesize display suggestions. By calling change_color we also make a fill-color display suggestion: this method sets the agent’s fill color based on its size, following the display suggestions in the specification.

3.9 The change_size method is almost trivial: an agent adds the extracted value to its size each time its change_size method is called. We retrieve this value via an extract method, which in the present model does no more than return the extraction_rate parameter value. Since we want our graphical observer to adjust the agent’s color to reflect its new size, the change_size method then calls the agent’s change_color method, which makes the appropriate display suggestion.

3.10 The specification’s display suggestions call for a fairly fine discrimination among tints of red. We represent these with the default RGB color model: colors are (red,green,blue) triplets of floating point numbers between 0 and 1. We produce tints of red by keeping the intensity of red at 1.0 while decreasing the intensity of green and blue from 1.0 to 0.0 as size increases from 0 to 10.

3.11 Agent02 is ready for use. We just need to create a collection of these agents and schedule their movement and
growth according to the specification. As before, we approach this by defining a new class (World02), which subclasses GridWorld in order to override its schedule method. We first schedule a brief sleep (i.e., a suspension of the program execution), which provides us with time to examine the current state of the visual display. The schedule method then sequentially calls the move method on each agent, and then sequentially calls the change_size method on each agent.

3.12 With our schedule in place, we can set up and run the simulation as in the first template model. We introduce one slight change: after we run the model, we call the mainloop method of our observer. For now, this is just to keep our display visible after the simulation is done running. (We will return to this.)

3.13 We emphasize that the code listing above is again complete. Although the second template model is slightly more complex, its code is no lengthier than that for the first template model. (Of course this is due to our import of Agent01.) This is a stark contrast to the NetLogo reference implementation of the second template model, where we find that code size increases rapidly with the complexity of the model. While brevity is no virtue if it sacrifices clarity, the code remains clear and readable.14

Template Model 3: Spatially Distributed Resources

4.1 In comparison to the modest goals of the first two template models, the third template model is somewhat ambitious. The core programming goal is to introduce spatially distributed resources in an environment with which agents interact. A secondary goal is to introduce a new random change in the model state: cells produce randomly.

4.2 Resources are location specific, so this template model introduces a “cell” (or “site”) object at each location. An agent at a location can interact with the cell at that location, thereby changing the state of the cell and the state of the agent. To make this concrete, we say that cells produce a resource, and that changes in an agent's size are determined by its extraction from its current cell.

Specification: Template Model 3

Setup
same as model 2, plus
• create a “cell” for each grid location

Parameters
as in model 2, plus
• agent_max_extract: the maximum extraction rate of an agent (1.0)
• cell_initial_supply: initial resource availability in a cell (0.0)
• cell_max_produce: maximum production rate of a cell (0.01)

Iteration (sequential)
each cell produces
• each agent moves
• each agent grows

Supplementary Detail
production: random, uniform between 0 and cell_max_produce
• a cell’s attributes include its current supply and its maximum production rate
• a cell’s production is added to its supply
• movement: agents move as in model 1
• growth: an agent “grows” by “extracting” from its cell
  • an agent can interact with the cell at its position (specifically, it can extract the cell’s supply)
  • an agent has a maximum extraction rate (equal to agent_max_extract)
  • an agent extracts whichever is smaller: its maximum extraction rate, or its cell’s supply
  • an agent’s size changes by the quantity it extracts
  • a cell’s supply is reduced by the amount extracted

GUI and display suggestions
unchanged from model 2

4.3 We remove two requirements from the original specification: we do not require that a “grid space object” hold the cells, and we do not require that each cell store its occupant in an instance variable. We consider these to be

http://jasss.soc.surrey.ac.uk/14/2/5.html 10 08/10/2015
implementation details that users would have trouble confirming on many platforms.\textsuperscript{15} Even when a model is implemented from the ground up, these are not sensible requirements: there are many competitive designs. (For example, a cell might delegate the determination of its occupants to an intermediary, which might maintain a mapping from cells to occupants.)

4.4 Furthermore, the rest of the specification (and its interpretive frameworks) make it more natural that an agent “know” its cell than that a cell “know” its agent. For example, in the NetLogo reference implementation a procedure is applied to each agent, which causes it to consume from its cell and grow.\textsuperscript{16} We can conceive a more patch-centered approach—say, applying a supply-agent procedure to each NetLogo patch—but this is not natural, necessary, or efficient.

4.5 As usual, we lightly modify the specification to provide for explicit parameterization of the model. As a minor matter, in line with the reference implementations, we specify that a cell’s random production is a draw from a uniform distribution. We also adopt a somewhat more general description of the agent and cell activities. The original specification suggested the following interpretation: cells grow food, which bugs eat to grow. However many other interpretations are available. For example, in the basic Sugarscape model, each agent extracts resources from its site, which adds to the agent’s wealth (Epstein and Axtell 1996). Of course, an interpretive framework for the template models remains supererogatory: any serious application to interactions between an agent and its spatial environment will require many more details.

4.6 With this background, we are ready to propose a design. We need to describe a cell that can produce and accumulate a supply, along with an agent that can extract this supply. We also need to make some changes to our agent description, so that it can appropriately consume and grow. A basic strategy should be growing evident. We inherit as much as we can from past work, and we add any new behavior we need. For example, an Agent03 is an Agent02 that knows how to extract resources from its cell. An Agent03 needs a different extract method than an Agent02, so we override this method.

\begin{verbatim}
Cell03 extends gridworld.Patch
   New Data: supply, max_produce
   New Methods: produce, provide
Agent03 extends Agent02
   New Data: max_extract
   Overridden Methods: extract
World03 extends gridworld.GridWorld
   Overridden Methods: schedule
\end{verbatim}

4.7 An Agent03 has a max_extract data attribute. Since this should always have the same value for every Agent03, it is a class variable. An Agent03 has a change_size method inherited from Agent02, which calls its new extract method. The extract method will try to extract max_extract (e.g., its maximum capacity) from its cell, but it cannot extract more than its cell’s current supply.

\begin{verbatim}
""" Template Model 3: Spatially Distributed Resources """
from gridworld import Patch
from template02 import *
p想办法.update(cell_initial_supply=0.0, cell_max_produce=0.01)
p想办法.update(agent_max_extract=1.0)
class Cell03(Patch):
   max_produce = params['cell_max_produce']
   supply = params['cell_initial_supply']
   def produce(self):
      self.supply += random.uniform(0, self.max_produce)
   def provide(self, amount):
      amount = min(self.supply, amount)
      self.supply -= amount
      return amount

class Agent03(Agent02):
   max_extract = params['agent_max_extract']
   def extract(self):
      mytake = self.patch.provide(self.max_extract)
\end{verbatim}
return mytake

class World03(GridWorld):
    def schedule(self):
        ask(self.patches, 'produce')
        ask(self.agents, 'move')
        ask(self.agents, 'change_size')

if __name__ == '__main__':
    myworld = World03(topology=TorusGrid(shape=params['world_shape']))
    mypatches = myworld.create_patches(Cell03)  # setup patches
    myagents = myworld.create_agents(Agent03, number=params['n_agents'])
    myobserver = GridWorldGUI(myworld)
    myworld.run(maxiter=params['maxiter'])
    myobserver.mainloop()

4.8 Cell03 is our new “cell” (or “site”) class. All the important cell behavior is new. However Cell03 subclasses gridworld.Patch, because a GridWorld knows how to create instances of Patch. (We use this when we set up the model.) In this model, every cell should always have the same value of max_produce, so it is naturally a class variable. In the template models every cell has the same initial supply, so we can also make this a class variable. (Each instance will automatically reference a supply instance variable when we change its supply.) We define a produce method to augment supply by a random amount, which is uniform between 0 and max_produce. And the provide method will yield the amount requested, up to the entire supply, while reducing the cell’s supply by the amount provided.

4.9 As usual, we create a schedule by overriding the schedule method of GridWorld. First we ask each patch to produce, then we ask each agent to move, and finally we ask each agent to change_size (by extracting from its cell). With our schedule in place, we can set up and run our simulation as usual, with one small change: Before we create our agents, we must create their new environment (i.e., the cells). This step is required by most ABM platforms, but it has no corollary in NetLogo, which automatically creates a patch environment for its agents.

Template Models 4, 5, and 6: Probes, Parameters, and Histograms

5.1 The next three template models emphasize the graphical user interface (GUI). In principle the GUI is a mere adjunct to ABM platforms. In practice it has proved an important tool for model exploration and understanding. It therefore receives a strong emphasis in the template models.

Template Model 4: Click Monitors

5.2 The core task of the fourth template model is to implement click monitors, or “probes”, for both agents and cells. A click monitor displays the state of an object type when the object receives a mouse click. The specification is correspondingly simple.

Specification: Template Model 4

World, Setup, and Iteration
unchanged from model 3

GUI and display suggestions
report agent size and cell supply in response to mouse clicks in the visual display

  • display suggestions as in model 3, plus: display click monitor reports in the GUI

5.3 The ease of producing such probes varies substantially by platform. On some platforms it is a substantial effort to probe both agents and cells (Railsback et al. 2006). NetLogo once again makes things easy: probes for agents and cells are provided automatically and displayed in the GUI.\(^\text{17}\) In this section, we add probes explicitly using our observer’s add_clickmonitor method, which takes as arguments a label for the display, an object type to monitor, and an attribute (or attribute list) whose values are to be monitored.\(^\text{18}\)
5.4 The resulting code is very simple. We simply reuse `Agent03` and `Cell103`. We subclass `GridWorldGUI` in order to override its `gui` method. (The `gui` method is special: it is automatically called when our observer instance is initialized.) This is where one adds probes, using the `add_clickmonitor` method. We add two click monitors: one to report the size of a clicked agent, and another to report the supply of a clicked cell.

5.5 The code for creating our world, adding cells and agents, and running the simulation is by now familiar. When we create our world, the two click monitors appear in the GUI (see Figure 1), and they display the state of the objects at the moment we clicked them. This highlights a new reason for calling the `mainloop` method of our world: the “main loop” is an event loop, which waits for and handles GUI events. In this case, it handles our mouse click events.

Template Model 5: Parameter Interface

5.6 The fifth template models addresses user manipulation of model parameters. The original specification focuses on the GUI, where a natural approach is to add sliders to control the parameters. This approach is often quite useful for early exploration of a model. However, the use of a GUI for parameter setting interferes with replicability, is an inefficient approach to testing model robustness, and is not a good ultimate practice in a research setting. We therefore slightly modify the original specification by demoting this to a display suggestion.

5.7 We make two additional changes to the specification. First, we add two display suggestions: the provision of `SetUp` and `Run` buttons. This is a natural implication of allowing parameter setting in the GUI: the parameter values must be determined before the model can be set up and run. (Thus, for example, we find `setup` and `go` buttons in the NetLogo reference implementation.) Second, we do not require an interface for setting of the maximum production rate of cells. This is for two reasons: it is redundant since the technique is exactly the same as for agent parameters, and it conflicts with data-based model initialization in the fifteenth template model.

**Specification: Template Model 5**

Setup and Iteration

unchanged from model 4

Parameters

facilitate user setting of *two* model parameters: the initial number of agents, and the maximum extraction rate of agents

GUI and display *suggestions*

as in model 4, plus:

- enable GUI setting of specified model parameters (*e.g.*, with sliders)
- include buttons in the GUI to set up and run the simulation, where set up includes the creation of agents (and patches if needed).

5.8 As Railsback et al. (2006) note, model parameters are often implemented as attributes of a model class. (Another approach, adopted by NetLogo, is to use global variables.) For the two specified model parameters, we add two class variables to our `World05` class. Since objects created by a `GridWorld` (such as agents or patches) know their world, they have access to these parameters as attributes of their world. During their initializations, agents need to access the parameter values stored by their world. Correspondingly, we now move patch and agent creation into a `setup` method of our world, which is to be called *after* any parameter
An Agent05 should have all the initializations of an Agent03 (as inherited from Agent02), and we accomplish this by calling Agent03.initialize. (This is for illustrative purposes: the gain from reusing the Agent03 initializations is trifling in this simple class.) The new initialization is to set the maximum extraction rate, as determined by the agent’s world.

Let us focus on the gui method of GUI05. We begin in a familiar way, invoking all the GUI04 initializations. Then we add two sliders, one for each of model parameters that we introduced as a class variable in World05. The sliders allow users to set new values for these parameters. (This approach matches the NetLogo reference implementation.) A slider is created with the add_slider method, which needs a label, the name of the target attribute (of our world), minimum and maximum possible values for the attribute, and a resolution (i.e., minimum increment) for the slider. The values for these are not part of the specification, but clearly they should encompass the default parameter values.

We then add the two specified buttons. As a convenience, we also add a Stop button, although this is not required by the specification. Each button is created with the add_button method, which requires as arguments a label and a function. This function (called a "callback" or "command") is called when the button is clicked. These buttons call the setup, run, and stop methods of the observer’s subject (i.e., our world).

World05 inherits run and stop methods from GridWorld. We do not override these. It also inherits a setup dummy method, intended to be overridden. We override setup to create our patches and agents.

Recall that in template model 4 we saw that the mainloop method prepared our GUI’s event handling, so that it could respond to mouse clicks. Now these GUI events include button clicks and slider adjustments. For example, we now can use button clicks in the GUI to set up and run the model.
Template Model 6: Animated Histogram

5.14 Agent-based models can generate a lot of data. Graphical summaries of the data can be helpful in understanding the model evolution and outcomes. The sixth template model addresses this need by requiring the production of a histogram to summarize the distribution of an agent attribute. (We choose the size attribute for monitoring.) Furthermore, we clarify that the histogram is to be “animated”, in the sense that it displays the changes in this distribution across model iterations.

5.15 As a minor change in the original specification, we do not require that the histogram be displayed in the GUI. We thereby remove an implicit requirement that animated histogram generation be synchronous with the simulation run. This reflects our view that the ability to synchronously view the histogram during a simulation run is more often a convenience rather than a fundamental feature of an ABM platform. (For example, MASON has no integrated graphing facilities, and lack of graphics documentation in Java Swarm suggests this is not yet a priority.) Indeed, saving data from simulation runs and analyzing it with separate tools is a reasonable and flexible approach to ABM assessment. We therefore demote GUI presentation of the histogram to a display suggestion.

Specification: Template Model 6

Setup, Iteration, and Parameters
as in model 5 ●

Data Display
produce an “animated” histogram that represents the evolution of the distribution of an agent attribute ●

Supplementary Detail
● the histogram should be of the size attribute of agents ●
  ● the sampling frequency for the histogram is not specified, but should be high enough to be informative about the evolution of this distribution

GUI and display suggestions
● as in model 6, plus: display a histogram chart in the GUI, and update the histogram as the simulation runs ●
  ● use 10 bins for the histogram, with a minimum size of 0 and a maximum size of 10

5.16 Fine graphics control will always be complex, but even the ease of basic graph creation varies substantially among platforms (Railsback et al. 2006). Once again NetLogo sets the standard for ease of basic use: to add a continually updated histogram of turtle size to the NetLogo GUI, just place histogram [size] of turtles in a procedure that will be called each iteration. The approach used by gridworld.py is only slightly more complex and is very flexible.

""" Template Model 6: Animated Histogram """

from template05 import *

class GUI06(GUI05):
    def gui(self):
        GUI05.gui(self)
    def get_agent_sizes():
        agents = self.subject.get_agents(self.subject.AgentType)
        return list(agent.size for agent in agents)
        self.add_histogram('Agent Sizes', get_agent_sizes, bins=range(11))

if __name__ == '__main__':
    myworld = World05(topology=TorusGrid(shape=params['world_shape']))
    myobserver = GUI06(myworld)
    myobserver.mainloop()

5.17 We add a dynamic histogram to the GUI with the add_histogram method of our observer. This method needs as arguments a title for the graph and a function that will be recomputed each iteration, along with a list of increasing numbers for bin “edges”. (The Python built-in function range returns a list of integers; here range(11) returns the first 11 non-negative integers.) When called, the function must provide the data for the
histogram. We therefore define a `get_agent_sizes` function, which returns a list of agent sizes. In doing so, we add one forward looking refinement: we illustrate the `get_agents` method our world inherited from `GridWorld`. (The agents belong to our world, which is the `subject` of our observer, so we call `self.subject.get_agents`.) This returns a list of the agents of the specified type. (Currently all of our agents are of type `Agent05`, so we get a list of all agents.) From this, we generate a list of agent sizes (using a convenient generator expression). This is the data needed for our histogram. We create our world, set up the model, and run the simulation as before. Our histogram appears in the GUI, where it is updated each iteration.

Template Models 7 and 8: Stopping and Logging

6.1 The next two template models address common ABM-modeling needs: criterion based stopping of the simulation, and recording of data generated by the simulation.

**Template Model 7: Stopping Condition**

6.2 Template model 7 simply adds a stopping condition to template model 6. The condition is that the model iterations should stop when any agent reaches a size of 100 or larger. We scarcely change the original specification. As usual, we demote the GUI aspects to display suggestions. We also remove a reference to “clean up steps” to be done upon termination, because these steps were not specified (Railsback et al. 2006).

**Specification: Template Model 7**

**Setup, Iteration, Parameters, and Data Display**
unchanged from model 6

**Stopping Condition**
terminate iterations based on model state

**Supplementary Detail**
- stopping condition is a cutoff for the agent size attribute
  - the suggested cutoff is that any agent reaches size >= 100
  - after stopping, completely exit the simulation

**GUI and display suggestions**
as in model 6, plus: close GUI when the simulation terminates

6.3 The most straightforward approach to the stopping-condition problem is to add a conditional check to the schedule. (This is essentially the approach taken by the NetLogo reference implementation, but it is an implementation detail, and is not required by the specification.) We test for the stopping criterion, and once it is satisfied, we stop the simulation and exit the program. To stop the iterative process we use the `stop` method that our world inherited from `GridWorld`. To exit our observer’s mainloop (and thus completely terminate the program) we set the `exit` keyword argument of the `stop` method to `True`, which will send an `exit` notification to our observer.

```python
""" Template Model 7: Stopping Condition """

from template06 import *

class World07(World05):
    def schedule(self):
        World05.schedule(self)
        if max(agent.size for agent in self.agents) >= 100:
            self.stop(exit=True)

if __name__ == '__main__':
    myworld = World07(topology=TorusGrid(shape=params['world_shape']))
    myobserver = GUI06(myworld)
    myobserver.mainloop()
```

Template Model 8: Output Files

http://jasss.soc.surrey.ac.uk/14/2/5.html

08/10/2015
Railsback et al. (2006) emphasize that producing data for subsequent analysis is a core facility for ABM platforms. For the eighth template model, we can break this into two parts: the production of summary statistics, and file input-output operations. In template model 8, each iteration, a summary of the agent states must be computed and written to an output file.

We make a single substantive change in the original requirements: we drop the (implicit) requirement that output be written as plain text. Plain text output has the great advantage of being human readable, but when we generate a large amount of data this advantage is rapidly outweighed by speed (of reading and writing) and eventually even file-size considerations. (See Vaingast (2009) for a good introductory discussion.)

Even if we settle on a plain text file format, many remaining issues are not addressed by the specification. Will the data be written in a well-known format, such as the comma-separated values (CSV) format? (CSV is the most obvious format to use for plain text data storage, since it is a standard spreadsheet format, but it is not the choice of the reference implementations.) Will the data be documented in any way, at least with a file header (as is a common first row in CSV files)? These are general concerns for data generation and maintenance. While we do not attempt to force decisions into the model specification, as the needs and resources of users vary widely, we do introduce some output suggestions into the specification.

**Specification: Template Model 8**

| Setup, Parameters, Display, and Stopping Condition |  
|--------------------------------------------------|---|
| unchanged from model 7 |  
| **Iteration (sequential)** |  
| as in model 7 |  
| • compute some summary statistics for the model |  
| • write model summary to a file |  
| **Supplementary Detail** |  
| • summary statistics are for the size of agents |  
| • summary statistics should include the minimum, mean, and maximum size |  
| **Output Suggestions** |  
| open, append to, and close the output file each iteration |  
| • write output as plain text |  
| • output from a single iteration is a single line in the text file |  
| • use CSV format for the output file |  
| • write a header as the first line in the log file |  

Implementations of input-output facilities vary substantially: some platforms provide special classes to facilitate file output, but Java-based platforms usually expect users to use the basic Java file-handling facilities (e.g., the FileWriter class). As one expects, NetLogo provides fairly easy access to very basic facilities. In contrast, Python’s input-output facilities are not only easy to use but also very powerful. Additionally, they can be coupled with Python’s powerful string manipulation facilities to easily produce nicely formatted plain text output.

We will use Python’s basic file handling abilities, as implemented in the built-in `open` function. We provide two arguments to `open`: a filename (as a string) and a mode (as a string). The basic text modes are ‘r’ (read), ‘w’ (write), or ‘a’ (append). For example, we can open the file `sizes.csv` for writing with the statement `open('sizes.csv','w')` or for appending with the statement `open('sizes.csv','a')`. The `open` function returns a file object, to which our with statement binds the name `fout`. We can then write a string `mystring` to the file as `fout.write(mystring)`.

One might reasonably implement the eighth template model by modifying our observer (or creating an additional observer). However we adopt the simplest implementation, which simply adds data logging to our schedule. We add two new methods (`header2logfile` and `log2logfile`), and override two methods (`setup` and `schedule`). For convenience, we store the file name and data format in two new parameters (`logfile` and `logformat`).

```python
""" Template Model 8: Log Model State Information to File """

from gridworld import describe
from template07 import *
```
params.update(logfile='c:/temp/sizes.csv', logformat='n\(\min\), \{mean\}, \{max\}')

class World08(World05):
    def setup(self):
        World05.setup(self)
        self.header2logfile()  # write header to logfile
    def header2logfile(self):
        with open(params['logfile'], 'w') as fout:
            fout.write('minimum, mean, maximum')
    def log2logfile(self):
        agents = self.get_agents(self.AgentType)
        sizes = list(agent.size for agent in agents)
        stats = describe(sizes)
        with open(params['logfile'], 'a') as fout:
            fout.write(params['logformat'].format(**stats))
    def schedule(self):
        self.log2logfile()
        World05.schedule(self)
        if max(agent.size for agent in self.agents) >= 100:  # from model 7
            self.log2logfile()  # log final agent state
            self.stop(exit=True)

if __name__ == '__main__':
    myworld = World08(topology=TorusGrid(shape=params['world_shape']))
    myobserver = GUI06(myworld)
    myobserver.mainloop()

6.10 Our setup keeps the World05 setup but appends logging of a header to our log file. Data logging is handled by the new log2logfile method, which retrieves a list of agent sizes, gets the summary statistics, and appends them to our output file. We handle data description with the gridworld.describe convenience function. This has functionality similar to Swarm's Averager class. It takes a list of numbers as its argument and returns a Python dictionary of descriptive statistics, with keys including the self-explanatory min, max, and mean. The expression params['logformat'].format(**stats) uses our logformat string to format the corresponding values in the stats dictionary. This is all it takes to log our data in CSV format.

6.11 We have one last change to make, and that is to the schedule. We keep the schedule of World07, but we prepend to it the data logging we want to do for each iteration. Additionally, we make sure we log the final model state before we exit the model. With that adjustment, we are done with the eighth template model. We set up and run the simulation as before.

Template Models 9, 10, and 11: Randomization, Hierarchy, and Optimization

7.1 In the template models, each agent moves once each iteration. However, until now we have not specified the order in which they move. Neglecting this has different implications on different platforms. For example, the most commonly used iterable collection in NetLogo is the “agentset”. Iteration over a NetLogo agentset retrieves agents in random order. (For example, when we ask a NetLogo agentset to take an action, the individual agents are asked in random order to take this action.) In contrast, the basic iterable collection of gridworld is a Python list, which is not randomized. However, the gridworld module provides separate ask and askrandomly functions, where the latter shuffles our agents before making the method calls.

7.2 In the first eight template models, the one place order might matter is when we ask agents to move. (Since location is unique, the movement of one agent changes the possible moves of other agents.) Randomizing the order in which agents move can help us avoid artifacts, such as unintended first mover advantages. The ninth template model specifies that agents move in random order. We make no changes to the original specification.

**Specification: Template Model 9**

- Setup, Parameters, Iteration, Data Display, and Stopping Condition: unchanged from model 8 (but with a new supplementary detail)
- Supplementary Detail: each iteration, the order in which agents move is randomized
After the eighth template model, implementation of the ninth is trivial: in our schedule we simply change
`ask(self.agents, 'move')` to `askrandomly(self.agents, 'move')`. So we will not repeat the code here. Instead we will focus on our implementation of the tenth template model, which requires a slightly larger change.

The tenth template model drops randomization of agent moves in favor of hierarchical priority: agents with a larger `size` attribute get to move earlier. (This will matter when we get to the subsequent template models.) If we adopt a “biological” interpretation, this hierarchy might represent an advantage of physical size. If we adopt a “social” interpretation, in the sense of Epstein and Axtell (1996), the hierarchy might represent an advantage of wealthier individuals when choosing among the available economic opportunities.

### Specification: Template Model 10

**Setup, Parameters, Iteration, Display, and Stopping Condition**

unchanged from model 8 (but with new supplementary detail)

**Supplementary Detail**

each iteration, sort agents by `size` to determine the move order: a larger `size` implies an earlier move

Sorting is straightforward on most ABM platforms. (Most use the built-in sorting facilities of their implementation languages.) Some require the user to implement a boolean comparison, and others want a sort key (i.e., a function whose values determine the sort order). For example, NetLogo’s `sort-by` command requires a user implemented boolean comparison, while the `sort` method of a Python list takes a sort key.

```python
from template09 import *

class World10(World08):
    def sortkey10(self, agent):
        return agent.size
    def schedule(self):
        self.log2logfile()
        ask(self.patches, 'produce')
        agents = self.get_agents(self.AgentType)
        agents.sort(key=self.sortkey10, reverse=True)
        ask(agents, 'move')
        ask(agents, 'change_size')
        if max(agent.size for agent in agents) >= 100:
            self.log2logfile()
            self.stop(exit=True)

if __name__ == '__main__':
    myworld = World10(topology=TorusGrid(shape=params['world_shape']))
    myobserver = GUI06(myworld)
    myobserver.mainloop()
```

Implementation of the tenth template model is a very small modification of the eighth: we just need to sort agents by size before they move. `World10` therefore extends `World08` by adding a new method, `sortkey10`. This sort key takes an agent as its argument and returns the agent’s `size`. We also override `schedule` in order to sort agents before they move. (The rest of the schedule should be familiar after our work on the previous template models.) Since `get_agents` returns a list, we can use the `sort` method to handle the sorting for us, once we specify our sort key. (Note that by default sorting is lowest to highest, so we set the `reverse` keyword argument to `True`.) With our `World10` definition in place, we set up and run the simulation as usual.

### Template Model 11: Optimization

Recall that we defined a `choose_location` method for `Agent01`. Subsequent to the first template model, location choice has remained random. Such agents have very limited agency. In the eleventh template model, we add interest to our agents by basing their movement on a local optimization.
We remove a couple ambiguities in the original specification. Substantively, we make it clear that the hierarchical movement of model 10 still applies. We stick with the original general specification of optimization behavior, while recognizing that there are many simple alternatives that might equally well be used for the eleventh template model. (E.g., Sugarscape movement rule M (Epstein and Axtell 1996).) We add some clarifications of the optimization details, but we demote these details to suggestions. Based on the NetLogo reference implementation, we clarify that an agent does not move if its current cell is one of the best. As a more minor matter, we resolve “ties” in a specific way: an agent’s choice among multiple best cells is random. Of course, since each cell’s supply is a random floating point number, the probability that tie resolution will be required is very low. (However, during the initial iterations of later template models, ties become much more likely.)

**Specification: Template Model 11**

**Setup, Parameters, Iteration, Display, and Stopping Condition**

*unchanged from model 10 (but with new supplementary detail)*

**Supplementary Detail**

each agent moves to a best available neighboring cell.

Suggested optimization details:

- an agent’s neighborhood remains a *Moore neighborhood of radius 4*, but now the agent’s current cell is included
- a cell is “available” if it is unoccupied
- a cell is “best” if it has greatest supply
- if the agent’s current cell is not a best cell, the agent changes location to a best available cell, resolving any ties with a random choice from the best cells

The substance of this template model lies in the `choose_location` method. An `Agent11` must choose the unoccupied neighboring cell with the greatest supply. We therefore retrieve a list of all the cells in a Moore neighborhood of radius four and then filter out the occupied patches. (We use a Python list comprehension, which is equivalent here to using a NetLogo filter. A cell’s `get_agents` method returns a list of its agents of the specified type; an empty list has a boolean value of `False`.) We want the agent to consider its own cell as well,
so we retrieve it from the agent’s patch attribute (inherited from gridworld.Agent) and append it to the list of available cells. The best cell is the one with the greatest supply. We can use the maximizers function of gridworld.py to find the best cells: this function takes as arguments an objective function and a collection of items to be evaluated. We use our sortkey11 as the maximand. If the current cell is among the best, the agent stays put. Otherwise, we use the choice function from the random module to choose randomly from the best available cells and return the position of our choice. This is our agent’s choice of the best location for its move.

Template Models 12 and 13: Entry, Exit, and Time-Series Plots

8.1 In some agent-based models, agents remain in the model during the entire simulation. Other models require that agents exit or enter the simulation as it runs. For example, organisms in a biological model or Sugarscape agents may exit through death and enter via birth, or firms in an economic model may exit through bankruptcy and enter as new start-ups (Gatti et al. 2001). The twelfth template model illustrates this common need of agent-based simulations: existing agents have a fixed probability of exit, and new agents enter when existing agents successfully propagate.

8.2 We make the following changes to the original specification. We addresses a potential first mover advantage in propagation by specifying that agents propagate in random order. We also remove some ambiguities in the specification, relying on the NetLogo reference implementation for guidance. Specifically, we specify that an agent attempts to propagate and then exits (implying that its position remains occupied during the split action), we clarify that testing for agent exit takes place after all agents have moved and changed size, and we clarify that new entrants are also tested for exit. As a minor matter, for reasons presented in our discussion of the second template model, we do not specify the size of new agents explicitly. Instead we simply specify a new agent has the same type as its “parent”, so that its initial size is determined by the agent’s ordinary initialization. (As usual, emphasized text indicates alterations of the original specification.)

Specification: Template Model 12

Setup and Display
unchanged from model 11 (with new supplementary detail)

Parameters
as in model 11, plus
● agent_exit_probability: the probability of agent exit (0.05)

Iteration (sequential)
as in model 11, then
● each agent with size > 10 attempts to propagate and then exits
● each remaining agent (including new entrants) exits with a fixed probability

Supplementary Detail
propagate: agents attempt propagation in random order
● a propagating agent (“parent”) makes five sequential propagation attempts
● a propagation attempt succeeds by finding an unoccupied location, which becomes the position of a new agent, where for each attempt
  ■ search is restricted to a Moore neighborhood of radius 3
  ■ for each propagation attempt, cells to search are selected by random sampling without replacement of the neighborhood
  ■ search is sequential and terminates if an unoccupied cell is found
  ■ no more than 5 cells are searched (if all five are occupied, search terminates, and the propagation attempt fails)
● a new agent is the same type as its “parent” (the propagating agent)
● exit: each remaining agent (including the new entrants) exits with fixed probability (agent_exit_probability)

Stopping Condition:
the number of agents reaches 0, or the number of iterations exceeds 1000

8.3 We approach the twelfth template model by very slightly extending World11 to implement our new schedule and by extending Agent11 to handle propagation (via a new split_if_ready method) and to accommodate randomized exit (via a new venture method). The new split_if_ready method will simply delegate to a new propagate method, which contains the propagation implementation details, and to the die method inherited from gridworld.Agent. This leads to the following design.
Agent12 extends Agent11
New Methods: split_if_ready, propagate, venture

World12 extends World11
Overridden Methods: schedule

8.4 As usual, we begin our implementation of the twelfth template model by importing all our earlier work. We then add our new parameter, agent_exit_probability. World12 is a very small extension of World11: we must append two agent actions (split_if_ready and venture) to the World11 schedule, along with our new stopping condition.

""" Template Model 12: Entry and Exit """

from template11 import *
params.update(agent_exit_probability=0.05)

class Agent12(Agent11):
    def split_if_ready(self):
        if self.size > 10:
            self.propagate()
            self.die()
    def propagate(self):
        MyType = self.__class__
        hood4split = self.neighborhood('moore', radius=3)
        cells4split = list()
        for i in range(5):
            for cell in random.sample(hood4split, 5):
                if cell not in cells4split and not cell.get_agents(MyType):
                    cells4split.append(cell)
                    break
        splitlocs = list(cell.position for cell in cells4split)
        splits = self.world.create_agents(MyType, locations=splitlocs)
        return splits
    def venture(self):
        if random.uniform(0,1) < params['agent_exit_probability']:
            self.die()

class World12(World11):
    AgentType = Agent12
    def schedule(self):
        World11.schedule(self)
        agents = self.get_agents(self.AgentType)
        askrandomly(agents, 'split_if_ready')
        agents = self.get_agents(self.AgentType)
        ask(agents, 'venture')
        if (self.iteration==1000) or (len(agents)==0):
            #model 12
            self.log2logfile() #from model 8
            self.stop()
    if __name__ == '__main__':
        myworld = World12(topology=TorusGrid(shape=params['world_shape'])))
        myobserver = GUI06(myworld)
        myobserver.mainloop()
agent's type. Following the specification, we consider a Moore neighborhood of radius 3 as possible locations for these successors. (We retrieve the cells in this neighborhood with the neighborhood method inherited from gridworld.Agent.) We create an empty list to hold the cells that can accept successors — i.e., unoccupied neighboring cells. An agent gets five attempts to split (i.e., to produce a successor), but for each potential successor the agent must find an empty cell within five tries. Each try is a random choice from the neighborhood. To make these choices, we use the sample function of Python's standard random module to retrieve five random cells, which we check sequentially until we either find an empty cell or run out of options. After five propagation attempts, we end up with a (possibly empty) list of empty cells, one for each successful attempt. We determine the position of each of these cells, and create the new agents at those locations.

Railsback et al. (2006) note that the twelfth template model is substantially more complex than the preceding models. They found themselves forced to resort to “clumsy and complex methods” on most platforms. The exception, once again, was NetLogo, and even so their NetLogo reference implementation is nearly 300 lines of code. Additionally, the closed-source nature of NetLogo proved problematic. (See Polhill and Edmonds (2007) for a discussion of the importance of open access for social simulations.) While adding and removing agents during a simulation is very simple in NetLogo, they found the documentation inadequate to determine whether an agent will execute its actions in the same iteration that it was created. Our Python implementation of the twelfth template model relies completely on open source code, is not “clumsy”, and avoids disheartening complexity (despite a nested loop that in turn nests a conditional branch).

Template Model 13: Time-Series Plot

The time-series behavior of the state of a simulation model is often interesting and informative. For example, starting with template model 12, we may wish to examine how the number of agents changes over time. The thirteenth template model requires the production of a time-series plot of the number of agents. The focus is visual data analysis: such plots can help us to understand the evolution of the simulation.

**Specification: Template Model 13**

**Setup, Iteration, and Parameters**
as in model 12

**Data Display**
as in model 12, plus: produce a time-series plot that displays the evolution of a model statistic over time

**Supplementary Detail**
the plot should display the number of agents “alive” at each iteration

- update frequency for the plot: not specified, but should be high enough to be informative about the evolution of this distribution

**GUI and display suggestions**
as in model 12, plus: display the plot in the GUI, and update it in real time as the simulation runs

As a minor relaxation of the specification, we do not require that the plot be displayed in a GUI. This removes an implicit requirement that plot generation be synchronous with the model run and demotes GUI presentation of the plot to a display suggestion. Our reasons are unchanged from those given for model 6.

With that background, let us add a real-time time-series plot to our GUI. NetLogo again epitomizes ease of use for very basic plots: to add a plot of the number of turtles to the NetLogo display, just place plot count turtles in a procedure that will be called each iteration. The gridworld approach is similar and very flexible: use the add_plot method of our world (as inherited from GridWorldGUI), which needs as arguments a title for the graph and a function that will be recomputed each iteration. This function must provide a single data point for the plot.

```python
""" Template Model 13: Time-Series Plot """
from template12 import *

class GUI13(GUI06):
def gui(self):
    GUI06.gui(self)
```

Railsback et al. (2006) note that the twelfth template model is substantially more complex than the preceding models. They found themselves forced to resort to “clumsy and complex methods” on most platforms. The exception, once again, was NetLogo, and even so their NetLogo reference implementation is nearly 300 lines of code. Additionally, the closed-source nature of NetLogo proved problematic. (See Polhill and Edmonds (2007) for a discussion of the importance of open access for social simulations.) While adding and removing agents during a simulation is very simple in NetLogo, they found the documentation inadequate to determine whether an agent will execute its actions in the same iteration that it was created. Our Python implementation of the twelfth template model relies completely on open source code, is not “clumsy”, and avoids disheartening complexity (despite a nested loop that in turn nests a conditional branch).

Template Model 13: Time-Series Plot

The time-series behavior of the state of a simulation model is often interesting and informative. For example, starting with template model 12, we may wish to examine how the number of agents changes over time. The thirteenth template model requires the production of a time-series plot of the number of agents. The focus is visual data analysis: such plots can help us to understand the evolution of the simulation.

**Specification: Template Model 13**

**Setup, Iteration, and Parameters**
as in model 12

**Data Display**
as in model 12, plus: produce a time-series plot that displays the evolution of a model statistic over time

**Supplementary Detail**
the plot should display the number of agents “alive” at each iteration

- update frequency for the plot: not specified, but should be high enough to be informative about the evolution of this distribution

**GUI and display suggestions**
as in model 12, plus: display the plot in the GUI, and update it in real time as the simulation runs

As a minor relaxation of the specification, we do not require that the plot be displayed in a GUI. This removes an implicit requirement that plot generation be synchronous with the model run and demotes GUI presentation of the plot to a display suggestion. Our reasons are unchanged from those given for model 6.

With that background, let us add a real-time time-series plot to our GUI. NetLogo again epitomizes ease of use for very basic plots: to add a plot of the number of turtles to the NetLogo display, just place plot count turtles in a procedure that will be called each iteration. The gridworld approach is similar and very flexible: use the add_plot method of our world (as inherited from GridWorldGUI), which needs as arguments a title for the graph and a function that will be recomputed each iteration. This function must provide a single data point for the plot.

```python
""" Template Model 13: Time-Series Plot """
from template12 import *

class GUI13(GUI06):
def gui(self):
    GUI06.gui(self)
```
To implement the thirteenth template model, we continue to use Agent12 and World12. Our new GUI13 is a GUI06 with two items appended to its initialization: the addition of a time series plot (using add_plot), and the definition of the function number_living. This function, which we provide as an argument to add_plot, simply returns the number (len) of agents in the simulation at the time it is called. This completes the thirteenth template model.

Template Model 14: Randomized Model Initialization

A strength of agent-based modeling is that it dispenses with the “representative agent”. Randomization can be a natural way to introduce heterogeneity among agents in the initial model state, and such randomization is the object of the fourteenth template model. During the model setup, for each agent we draw an initial size from a normal distribution, with mean and standard deviation that are user settable. The specification requires that size be truncated at a minimum of zero.

Specification: Template Model 14

Parameters

- as in model 13, plus
- mean of the size distribution of initial agents (default: 0.1)
- standard deviation of the size distribution of initial agents (default: 0.03)

World, Setup, Iteration and Stopping Condition

- as in model 13, plus: each initial agent is assigned a random size during the model set up
  - each initial size is drawn independently from a normal distribution
  - mean and standard deviation of the size distribution are model parameters
  - if the size draw is less than 0.0, it is set to 0.0

GUI and display suggestions

- as in model 13, plus: allow GUI setting of the two new model parameters

We make one substantive clarification of the original specification, which said only that a check is introduced to limit size to a minimum of zero. This could mean that negative draws are set to zero or alternatively that negative draws are discarded in favor of a new draw (i.e., a truncated normal distribution). We remove this ambiguity by adopting the first interpretation as the simplest reading of the original specification. Our only other change is the usual demotion of GUI parameter setting to a display suggestion.

We retain our previous agent type, but we need to make a modest change in our world type. World14 has two new class variables based on our two new parameters, which we name agent_size_mean and agent_size_sd. Recall from World05 the setup method that creates patches and agents by delegating to setup_patches and setup_agents. We now override the setup_agents method so that it not only creates the initial agents but also implements the new randomization of their initial state.

```python
from template13 import *
params.update(agent_size_mean=0.1, agent_size_sd=0.03)

class World14(World12):
    agent_size_mean = params['agent_size_mean']
    agent_size_sd = params['agent_size_sd']
    def setup_agents(self):
        #random size for initial agents
```

http://jasss.soc.surrey.ac.uk/14/2/5.html
9.4 Most ABM platforms have pseudo random number generators (PRNGs). The `random` module of the Python standard library implements a fairly extensive collection of distributions, which naturally includes the normal. We use the `normalvariate` function to initialize our first collection of agents. We retrieve the mean (`agent_size_mean`) and standard deviation (`agent_size_sd`) of the initial-agent size distribution. For each agent we draw from a normal distribution using these parameters, and we set the agent’s `size` attribute to the maximum of this draw or 0.

9.5 The change to our GUI is modest: we retain all the features of `GUI13`, but we append the addition of two sliders corresponding to the new parameters. (We discussed the `add_slider` method in the fifth template model.) Once we have defined our new world and GUI classes, we are ready to create a `World14` instance and an observer for it, and enter the GUI’s main loop as usual.

**Template Model 15: Data-Based Model Initialization**

10.1 The fifteenth template model makes no fundamental changes in the characterization of agents or cells. As with the fourteenth template model, the focus is on the initial model state. But this time, instead of adding randomization, we read data for the initial model state from a file. This can be useful when simulations must be interrupted and restarted, and it can be crucial when the initial state of a simulation is empirically anchored. The fifteenth template model requires that cell production rates be read from a data file. The data in this file specifies a production rate at each location, and the shape of the grid must be determined by the listed locations.

10.2 Implementing a data-dependent model set-up is the core change in the model, but the original specification includes a few additional changes.35 agents are located on a bounded rectangular grid (instead of a torus), cell production is no longer random, and cells should change color to reflect their supply. In terms of the implementation, these are fairly minor additions, but including them does lengthen the specification of the fifteenth template model.

**Specification: Template Model 15**

- **World**
  - a rectangular grid of possible agent locations

- **Setup**
  - read in a given file of cell data
  - determine the grid shape based on the cell data (but see other suggestions, below)
  - create a rectangular grid (not a torus) based on the shape implied by the cell data (see below)
  - create a cell for each grid location
  - set a production rate for each cell based on the cell data
  - set up the initial agents as in model 14

- **Parameters, Iteration, and Stopping Condition**
  - as in model 14 (but with new supplementary detail)

- **Supplementary Detail**
  - cell data file (from http://condor.depaul.edu/~slytinen/abm/):
- the unzipped file format is plain text
- the first three lines are header information (to be discarded)
- each additional line provides three space-delimited numbers: an integer x coordinate, an integer y coordinate, and a floating point production rate.

- Moore neighborhood: on a rectangular grid (instead of a torus) our neighborhood definition must change: coordinates off the grid are now unavailable (instead of wrapping)
- production: each iteration a cell produces the amount specified by the file of cell data (i.e., production is no longer random)

GUI and display suggestions
as in model 14, plus: display cells in the GUI and color-code the cell state
- base a cell's color on the value of its supply attribute: a cell is green when supply is 0.5 or higher and shades to black as supply goes to 0

Other Suggestions
if the platform does not readily allow dynamic setting of the grid size, preset the grid size using the grid shape implicit in the data file, which is 251 (x' values of 0 to 250) by 113 (y' values from 0 to 112)

10.3 We make small changes to the original specification. We do not require that the change in the grid (from torus to rectangle) be implemented as a change in the move method of agents. That is just an implementation detail, and agents might instead be constrained by the topology in which they move. We also add a few details about the cell data file format, and we demote display specifications to suggestions.

10.4 We are ready to propose a design using new world and cell types. The changes in the cell type are very small. **Cell15 extends Cell03 by adding a change_color method. It also overrides the initialize method to add the setting of an initial color for each cell. (The initialize method is inherited from Patch, which automatically calls it as part of object initialization. This method is intended to be overridden by users who want to provide object initializations in subclasses.) The initial color will be black, since cells still have a zero initial supply. We also override the produce method to make production deterministic and to reset the cell color after production. A World15 is a World14 with a slightly more complex set up. This manifests in the overridden setup_patches method: model set up depends on the cell data, which we must read from a file.**

```python
from gridworld import RectangularGrid
from template14 import *
params.update(cell_data_file='Cell.Data')
```

```python
def read_celldata(filename):
    location2value = dict()
    maxx, maxy = 0, 0
    fh = open(filename, 'r')
    for _ in range(3): #discard 3 lines
        trash = next(fh)
    for line in fh:
        x, y, prodrate = line.split()
        x, y, prodrate = int(x), int(y), float(prodrate)
        location2value[(x,y)] = prodrate
```

10.5 Note that we need not change the move method of our agent. One reason is that our implementation of template model 11 already correctly handles optimal movement, including tie resolution. The other reason is a convenience of the gridworld module: an Agent has a neighbors method that queries its world, so it correctly handles the topology of that world. (E.g., for a Torus, it returns a list of patches that includes locations that “wrap” on the torus, while for a RectangularGrid patches beyond the boundary are not returned.) As a result, our agent description does not change at all.

""" Template Model 15: Data-Based Model Initialization """

http://jasss.soc.surrey.ac.uk/14/2/5.html 08/10/2015
maxx, maxy = max(x, maxx), max(y, maxy)
location2value['shape'] = (maxx+1, maxy+1)
return location2value

class Cell15(Cell03):
def initialize(self):
    self.change_color()
def produce(self):
    self.supply += self.max_produce # no longer random
    self.change_color()
def change_color(self):
    r = b = 0
g = min(2*self.supply, 1.0)
    self.display(fillcolor=(r, g, b))

class World15(World14):
    PatchType = Cell15
    def setup_patches(self):
        celldata = read_celldata(params['cell_data_file'])
        shape = celldata.pop('shape')
        self.set_topology(RectangularGrid(shape=shape))
        patches = self.create_patches(self.PatchType)
        for (x,y), prodrate in celldata.items():
            patches[x][y].max_produce = prodrate

if __name__ == '__main__':
    myworld = World15(topology=None)
    myobserver = GUI14(myworld)
    myobserver.mainloop()
Parameters, World, and Setup
as in model 15, plus: create 200 randomly distributed “hunters”

Iteration (sequential)
as in model 15, then
● each hunter hunts

Stopping Condition
as in model 15

Supplementary Detail
a cell may contain a hunter and a gatherer
● hunting: hunters randomly search for gatherers and “kill” the first found
  ○ search: randomly sample (without replacement) a Moore neighborhood of radius 1, center included
  ○ if another hunter is found in a cell, search terminates, and the hunter remains at its current location.
  ○ if another hunter is not found in a cell, but a gatherer is found, search terminates, and the hunter moves to the cell and “kills” the gatherer
  ○ if no neighboring cell contains another hunter or a gatherer, the hunter moves randomly to an unoccupied cell
  ○ gatherers are removed from the simulation as soon as they are “killed”

GUI and display suggestions
as in model 15, plus: hunters are colored yellow with a classic arrowhead shape

11.2 Our changes to the original specification are primarily an effort to reduce ambiguity. Kahn (2007) points out an important ambiguity in the original specification: does search terminate whenever a hunter is encountered during cell search, or only if a gatherer is encountered and there is already a hunter in its cell? We follow the reference implementations and adopt the first interpretation.38 Also, the original specification simply refers to the “immediately neighboring cells” of a hunter, but we specify that this is a Moore neighborhood of radius 1. This is the apparent intent of the original wording, and it matches the NetLogo reference implementation. A substantive clarification is that we specify that search terminates if another hunter is encountered.39 As a very minor change, we remove the implementation detail that random search be implemented with a shuffled list and replace it with the underlying requirement that sampling from the neighborhood be done without replacement. Finally, we remove the pointless requirement that gatherers be created before hunters, and we add display suggestions for the color and shape of hunters.

11.3 Our World16 is a World15 with new schedule and setup methods. We reuse our previous agent type (Agent12) for our gatherers, but we also introduce a new Hunter class that extends our basic Agent class with a new hunt method.

""" Template Model 16: Interacting Agents of Different Types """

from template15 import *

class Hunter(Agent):
    def initialize(self):
        self.display(fillcolor='yellow', shapesize=(0.75,0.75))
    def hunt(self):
        hunthood = self.neighborhood('moore', radius=1, keepcenter=True)
        random.shuffle(hunthood)
        change_position = True
        for patch in hunthood:
            hunters = patch.get_agents(AgentType=Hunter)
            gatherers = patch.get_agents(Agent12)
            if hunters and not self in hunters:
                change_position = False
                break
            elif gatherers:
                gatherers.pop().die()
                self.position = patch.position
                change_position = False
                break
            if change_position: #encountered no gatherers nor hunters
                newcell = random.choice(hunthood)
            if another hunter is found in a cell, search terminates, and the hunter remains at its current location.
11.4 Our new Hunter class overrides the initialize method of Agent simply to set display suggestions (color and size). The real work of the sixteenth template model is in the new hunt method. A hunter starts by retrieving a neighborhood for hunting, which is shuffled. The hunter begins searching with the variable change_position set to True; we will negate this upon encountering another hunter or upon moving to the position of a discovered gatherer. The hunter sequentially considers each patch in the shuffled neighborhood, retrieving a list of hunters and a list of gatherers from that patch. Each list might be empty. If the hunters list contains another hunter, search terminates and the hunter does not change position. Otherwise, if the gatherers list contains a gatherer, search terminates and the hunter moves to that position and “kills” the gatherer. (I.e., the gatherer is popped from the list, and its die method is called.) If no other hunters are encountered and no gatherer is found in the entire neighborhood, the hunter moves to a random cell in the neighborhood.

11.5 Our new World16 class retains all the initializations and parameters of World15. It also reuses the setup method, but appends the creation of 200 hunters at random locations. The real change is in the schedule, although that too is largely familiar. We retain the schedule from World15 and then append a new action: we retrieve a list of hunters and ask them to hunt. Since there can be a first mover advantage in hunting, the hunters hunt in random order. We create our world and run the simulation as in model 15. Figure 1 captures the state of the GUI during a model run that keeps the default parameter values.
Conclusion

12.1 This paper has two core objectives: to refine a well-known set of template models for agent-based modeling, and to offer useful and heavily-commented new reference implementations. In pursuing these goals, we also address some issues of design, flexibility, and ease of use that are relevant to the choice of an agent-based modeling platform. However, deciding among agent-based modeling platforms is not a goal of this paper. (In addition to Railsback et al. (2006), discussed above, see Nikolai and Madey (2009) or Tobias and Hofmann (2004) for recent discussions of platform choice.) Our reference implementations use the Python programming language, since reference implementations particularly profit from the Python virtues of readability and compactness.

The template models discussed by Railsback et al. (2006) have proven useful to instructors and researchers for ABM platform introduction and comparison. So have the reference implementations of Railsback et al. (2005). However the template models contain some ambiguities, and the existing reference implementations contain some weaknesses and errors. Rather than propose a new set of template models, we attempt a refinement of the existing set, and we provide new, supporting reference implementations.

12.2 Reference implementations are important for removing remaining ambiguities in the specification and for providing concrete illustrations of implementation strategies. We present readable reference implementations that can serve these purposes. Our reference implementations retain the original emphasis on being "simple and intuitive" rather than being clever, fast, or illustrative of best practices. However we address some issues of design, flexibility, and ease of use that are relevant to instructors and researchers choosing an agent-based modeling platform. Specifically, we explore some of the costs and advantages of taking an object-oriented approach to the template models. We provisionally conclude that the primary cost is a slight front-loading of the skill set needed to begin modeling and that the (somewhat language specific) benefits can include reusability, maintainability, compactness, clarity, and readability.

12.3 Railsback et al. (2005) conveniently provide reference implementations for a variety of platforms. Their NetLogo...
This sometimes feels a little peculiar to programmers coming from other languages, but most Python provided, and it is the instance itself. The first parameter name is (conventionally) called `myagent.move()`.

Breeds. (So a breed is essentially an instance attribute of a NetLogo `turtle` module, which is part of the Python standard library. Like NetLogo, but unlike many other ABM platforms, the `turtle` module provides for automatic visual display of its agents. The `gridworld` module depends (for graphs only) on Matplotlib, a popular free and open source Python graphics library. Matplotlib in turn depends on NumPy, a popular free and open source scientific computing library. These libraries are very simple to install on common platforms and are widely used in scientific computing. It is worth noting that the `turtle` module implements separate classes for the movement and positioning methods and for the drawing methods, making it easy to implement agent-based models without visual displays. (Interested readers may wish to examine the excellent online `turtle.py` documentation, especially the documentation of the motion methods.)

This display method can set fillcolor, shape, and shapesize. For a `GridWorldGUI` observer, these attributes have the meanings given to them by a `turtle.Turtle`, since Python's `turtle` module is used to provide a visual display of agents. Note that this "visual display information" is just a set of attributes, which can have meaning outside a GUI context. (E.g., agents could interact differently based on having different "colors", regardless of whether these "colors" are ever visually displayed. This is a common practice among NetLogo programmers.)

See http://ccl.northwestern.edu/netlogo/docs/dictionary.html#breed. Note that a NetLogo `turtle` can change breeds. (So a breed is essentially an instance attribute of a NetLogo `turtle`.)

Note that no argument is passed by the method call. If `myagent` is an instance of `Agent01`, then `myagent.move()` and `Agent01.move(myagent)` mean the same thing. This means that the first parameter of a method definition is special: when the method is called on an instance, the first argument is automatically provided, and it is the instance itself. The first parameter name is (conventionally) called `self` for this reason. This sometimes feels a little peculiar to programmers coming from other languages, but most Python

---

**Notes**

1 These features of Python are so compelling that the authors of Repast, a Java library for agent-based modeling, implement a subset of Python as a scripting language for their Repast Py user interface to Repast library (Collier and North 2004).

2 We therefore assume readers have perused an introduction to Python. Examples include Kuhlman (2008) for a quick introduction, Isaac (2008) for a narrower but more applied introduction, or Zelle (2003) for a basic textbook introduction. We also refer to some basic NetLogo, for which Izquierdo (2007) provides a quick introduction. Note that we use the term 'function' to refer to callable subroutines of any type, even when a specific language or platform adopts a different terminology.

3 We do not specify the PRNG because availability differs across platforms. Naturally users should attend to the properties of their chosen PRNG (Niel and Laffan 2003). Python uses the rigorously tested Mersenne Twister (MT19937) as its core generator, which has a period of $2^{19937} - 1$.

4 Although it conflicts with the reference implementations, this is a natural reading of the original specification, which says that if an occupied location is selected, "then another new location is chosen". However, the reference implementations draw with replacement, instead of requiring a new (i.e., different) random draw from the neighborhood.

5 Examples of free and open source game development toolkits include pygame and pyglet, both of which are fairly simple and very powerful. To start entirely from scratch, we could implement basic agents, patches, and worlds as Python classes. We could also implement a visual display environment using the Tkinter graphical user interface (GUI) toolkit, which is part of the Python standard library. Although Tkinter is simple to use, presenting a GUI toolkit is an unnecessary burden for the present paper. Those interested in examining such an approach should look at Swampy (Downey 2009). Another possible toolkit is SLAPP (Terna 2009). SLAPP is a recent Python implementation of the Swarm protocol (Langton 1996).

6 Download the gridworld module from http://econpy.googlecode.com/svn/trunk/abm/gridworld/gridworld.py. The agents provided by the gridworld module are represented graphically by instances of the `turtle` class of the `turtle` module, which is part of the Python standard library. Like NetLogo, but unlike many other ABM platforms, the `turtle` module provides for automatic visual display of its agents. The `gridworld` module depends (for graphs only) on Matplotlib, a popular free and open source Python graphics library. Matplotlib in turn depends on NumPy, a popular free and open source scientific computing library. These libraries are very simple to install on common platforms and are widely used in scientific computing. It is worth noting that the `turtle` module implements separate classes for the movement and positioning methods and for the drawing methods, making it easy to implement agent-based models without visual displays. (Interested readers may wish to examine the excellent online `turtle.py` documentation, especially the documentation of the motion methods.)

7 This display method can set fillcolor, shape, and shapesize. For a `GridWorldGUI` observer, these attributes have the meanings given to them by a `turtle.Turtle`, since Python's `turtle` module is used to provide a visual display of agents. Note that this "visual display information" is just a set of attributes, which can have meaning outside a GUI context. (E.g., agents could interact differently based on having different "colors", regardless of whether these "colors" are ever visually displayed. This is a common practice among NetLogo programmers.)

8 See http://ccl.northwestern.edu/netlogo/docs/dictionary.html#breed. Note that a NetLogo `turtle` can change breeds. (So a breed is essentially an instance attribute of a NetLogo `turtle`.)

9 Note that no argument is passed by the method call. If `myagent` is an instance of `Agent01`, then `myagent.move()` and `Agent01.move(myagent)` mean the same thing. This means that the first parameter of a method definition is special: when the method is called on an instance, the first argument is automatically provided, and it is the instance itself. The first parameter name is (conventionally) called `self` for this reason. This sometimes feels a little peculiar to programmers coming from other languages, but most Python
programmers come to feel that it is an excellent, explicit design choice. Python uses `self` roughly like `*this` in C++ or the `this` reference in Java and C#, except that it is never implicit.

10 Rossum (2003) provides an extended discussion of this strategy.

11 Color coding of agent states is common when agent-based simulations are displayed. (E.g., Epstein and Axtell (1996) color code agent vision and metabolism.) We give agents attributes intended to prompt a certain visual display from an observer instance, but naturally one can design an observer to ignore or reinterpret these attributes. Note that the NetLogo reference implementations delay adding these color changes until template model 3.

12 In template model 12, agents propagate new agents of the same type, which are specified to have an initial size of 0.0. There is a reason for this inconsistency: it is often desirable to initialize a simulation with agent states that differ from the normal agent initialization. (For example, a simulation of a population that ages might set up by constructing a distribution of ages in an initial population, but new agents always enter the population with an age of zero.) We consider it appropriate to postpone this consideration until template model 14, where it is taken up explicitly.

13 Here ‘useful’ means reusable or extensible. When appropriate, we would like to honor the so-called DRY principle ("don’t repeat yourself"), which provides an important rule for the production of maintainable code (Hunt and Thomas 1999).

14 NetLogo does not offer convenient mechanisms for class definition and reuse. However, improved compactness of the code base may be achievable since NetLogo 4.1 by using the experimental `__includes` keyword. (For example, procedure definitions can be grouped together for inclusion.)

15 Even when relatively easy to confirm, as with the `gridworld` module, the information is fundamentally irrelevant to the user. Railsback et al. (2006) discuss how their specified design was picked as “easiest” for the MASON, Repast, and Swarm implementations, again confirming that it is just an implementation detail. Note that neither of the deleted requirements would make much sense to a user of the NetLogo platform. NetLogo users do not worry about creating a grid space, since NetLogo automatically creates patches, and users can simply apply the `turtles-here` command to a patch (or turtle) without worrying how this is implemented.

16 The `grow` procedure in the NetLogo reference implementation does not use the `turtles-here` command nor even the `patch-here` command. It relies instead on a convenient but very implicit NetLogo feature: a procedure call on a turtle that references data attributes owned by patches will access, and alter, that turtle’s patch’s values.

17 As a result, there is no separate NetLogo reference implementation for the fourth template model.

18 One may also include any keyword arguments appropriate to a Tkinter label widget, which gives substantial control over the appearance of the click-monitor display. By default, monitors are updated each iteration, but click monitors always display the value obtained at the last click.

19 Here we define `get_agent_sizes` in the body of our `gui` method, taking advantage of Python’s support of closure. Alternatively, we could have added a new `get_agent_sizes` method to `GUI06`.

20 Kahn (2007) notes that the Netlogo reference implementation erroneously stops when an agent’s size reaches 1000. This affects nothing of substance in this template model.

21 In the present context, we can imagine two interpretations of possible clean up: resetting the model in order to begin a new simulation run, or housekeeping prior and subsequent to termination of the process running the simulation. The NetLogo reference implementation addresses neither of these. The first interpretation conflicts with the instruction to close the graphics windows. The second should be otiose with any modern operating system. For example, memory used by the simulation should be released back to the operating system when the process running the simulation terminates.

22 Note that the `stop` method of a `GridWorld` simply sets that world’s `_stop` attribute to `True`, and this attribute is tested each iteration by the `run` method. One implication is that we can call stop anywhere in the schedule, and the schedule will still be completely executed before the the model run terminates. (If stopping with only partial execution of the schedule is desired, we can of course add a `return` statement to our conditional branch.)

23 We also suggest opening, appending to, and closing the file each iteration, so that a model crash is less likely to cause data loss. A minor change is that we make it clearer that the minimal amount of information to be written each iteration is the minimum, mean, and maximum size. More information can be written at the user’s discretion. For example, the NetLogo reference implementation also writes `currentTime`, which is simply the iteration number, and also writes a header at the top of the file.

24 A traditional difficulty for new NetLogo users is distinguishing between `file-write`, `file-type`, and `file-print`. This is mitigated in more recent versions, since many users may prefer to use NetLogo’s
BehaviorSpace tool, which uses CSV as its output format.

This is the best match to the reference implementations. However it is worth noting that the Python standard library includes the csv module, which makes it particularly simple to write CSV files. In this paper we do not introduce the csv module, but interested readers should consider the csv.DictWriter class.

Python’s with statement ensures that the file is properly closed after use. Note that the filename argument to open can be a fully qualified file name, and that the use of forward slashes is acceptable on all platforms (including Windows).

Our use of describe is for purposes of illustration: we are logging only the minimum, mean, and maximum agent size, which we could readily compute with Python’s built-in functions as min(sizes), sum(sizes)/len(sizes), and max(sizes). This is essentially the approach of the NetLogo reference implementation, although NetLogo additionally provides mean as a built-in command.

This footnote examines the expression params["logformat"].format(**stats) in more detail. Since params is our dictionary of parameters, the expression params["logformat"] evaluates to our logformat string. Like all strings, this one has a format method that accepts keyword arguments to determine substitutions for the named terms in braces. Python uses the double asterisk for unpacking a dictionary as keyword arguments, so the expression format(**stats) unpacks the stats dictionary into keyword arguments for the format method. This offers a nice illustration of the simplicity, power, and flexibility obtained using a Python format string.

This has some advantages: shuffling the agents is a computational expense, which we should incur on an as needed basis. Keep in mind that askrandomly is a simply gridworld convenience method. A pure Python equivalent of askrandomly(lst,'methodname') is random.shuffle(lst); for item in lst: item.methodname().

The original specification parameterized survival probability, which is the complement of our exit probability. This is largely a matter of taste, but given this model’s emphasis on exit, the change seemed appropriate. Note that we have already addressed setting parameters in the GUI, so in parallel with the NetLogo reference implementation, we do not repeat that here.

We need not worry about the model 7 stopping condition, as model 12 agents never reach the size cutoff for that to apply. Therefore for presentational simplicity, we just ignore it. The NetLogo reference implementation inexplicably diverges from the specification’s stopping codition, but we do not alter our specification to match that divergence.

The NetLogo 4 user manual now clarifies this: only agents in the agentset at the time the ask command executes will run the commands.

The default behavior of a NetLogo plot is to continually add points, which make the plots uninformative during long simulation runs. The time-series plot provided by gridworld provides a window on the most recent iterations, which is arguably more informative.

Unfortunately, the NetLogo reference implementation of the fourteenth template model does not provide guidance as it does not conform to the original template model specification. The initial agents are given size 1.0 instead of a random size, new agents are given a random size instead of a size of 0.0, and the random size is not truncated at 0.0. While the details of the model initialization are largely inconsequential, the NetLogo reference implementation fails to implement the core of the fourteenth template model, which is to randomize the initial model state.

Failing to randomize agent choice among indifferent alternatives can introduce movement artifacts (Railsback et al. 2005). The original specification of the fifteenth model therefore listed one additional requirement: agents randomly break “ties” when there are multiple best possible moves. This random resolution of indifference is really part of the specification of optimizing behavior (Ullmann-Margalit and Morgenbesser 1977). We therefore incorporate this requirement in the eleventh template model, which introduces optimization. However, the original specification’s implementation details for random “tie” resolution are extraneous to the specification, so we discard those.

Kahn (2007) notes a couple problems with the NetLogo reference implementation of the fifteenth template model. Of some importance, the NetLogo implementation neglects to drop the random determination of the production rate. A nugatory problem is that patch coloring in the Netlogo implementation does not follow the display suggestion. We note in addition that the Netlogo implementation does not set the grid size based on its reading the file of cell data, reflecting a platform limitation. Our final modification of the specification accommodates this limitation.

Note that Python can simply iterate through the distributed .zip file, but we stick the the apparent presumption in the specification that the compressed text file will be unzipped before use. We name the
decompressed file Cell.Data, and we store this name as a model parameter.

38 As Kahn notes this has some odd implications. (Adjacent hunters will remain stationary until a gatherer comes within range.) Experimenting with alternatives to this specification will be a natural extension of the sixteenth template model.

39 For example, the NetLogo reference implementation first asks if there are hunters on a cell, and then asks if there are gatherers. Since hunting stops if a hunter is found (there is no identity testing), in that implementation, a hunter will never find a gatherer on its own cell. However the NetLogo implementation also fails to include that cell in the list of neighbors, so a hunter will never search its own cell in any case.

References


