The ABM Template Models
A Reformulation with Reference Implementations

Alan G. Isaac
Department of Economics
American University
Washington, DC 20016

30 September 2009

Abstract

This paper refines a well-known set of template models for agent-based modeling and offers new reference implementations. It also addresses issues of design, flexibility, and ease of use that are relevant to the choice of an agent-based modeling platform.

Contents

This paper has two core objectives: to refine a well-known set of template models for agent-based modeling, and to offer a new reference implementation. In pursuing these goals, we address issues of design, flexibility, and ease of use that are relevant to the choice of an agent-based modeling platform.

The use of agent-based models (ABMs) has been growing rapidly in many fields, including evolutionary biology, ecology, economics, epidemiology, game theory, political science, and sociology. Popular ABM platforms include NetLogo (`http://ccl.northwestern.edu/netlogo/`), Swarm and Java Swarm (`http://www.swarm.org`), Repast (`http://repast.sourceforge.net`), and MASON (`http://www.cs.gmu.edu/%7Eeclab/projects/mason/`). Each of these platforms has benefits and limitations that have been explored in the literature (Tobias and Hofmann, 2004; Railsback et al., 2006). The most popular of these platforms remains NetLogo, which is renowned for its combination good graphical capabilities, reasonable flexibility, and excellent of ease of use. Railsback et al. (2006) also judge NetLogo to be the most professional platform in terms of appearance and documentation. The most common complaint against NetLogo is its failure to adopt an open source license, which some researchers view as a crucial component of scientific ABM research.

Railsback et al. (2006) discuss a collection of sixteen template models, which they designed as tools to introduce and explore agent-based modeling (ABM) platforms. (Railsback et al. (2005) provide the full model specifications.) The template models are intended to be "ridiculously simplified". They illustrate general modeling considerations required by many different real-world applications and do not constitute an attempt to implement a specific real-world application. These template models have already proved their usefulness. They are used in teaching and are chosen as a point of reference for introductory presentations of new ABM platforms. 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.

Railsback et al. (2005) provide reference implementations of the template models for a few popular platforms. The template-model 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 are the NetLogo reference implementations. Robertson (2005) notes that, as an ABM platform, NetLogo excels at ease of use and compactness of representation. This makes the NetLogo reference implementations particularly relevant as a point of comparison for other ABM platforms.

In this paper we provide new reference implementations. The reference implementations use the Python programming language. After modest initial start-up costs, our Python implementations of the template models prove simple, readable, and short. That is, they are useful as reference implementations. We show how use of a simple `gridworld` module allows for Python implementations that usually approach and often exceed the simplicity of NetLogo, while providing improved readability and power. Following the intent of the original reference implementations, our code emphasizes readability and ease of use (rather than speed or generality). We also suggest a few advantages and disadvantages of using Python as an ABM platform. Since NetLogo consistently receives high marks on these criteria, we take it as our primary point of comparison.

Template Model 1: Spatial Movement of Agents

The first template model provides a basic introduction to ABM platforms. The core programming goal is to implement the random movement of agents in a simple two-dimensional space. That is, we must implement a specified behavior for a base agent type that is spatially situated.

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.[1] With this background, we can attempt the specification of the first template model.

Specification: Template Model 1

World

- a 100 x 100 toroidal grid of possible agent positions

Setup

- create *an iterable collection of* 100 agents
- *give each agent a unique, random position on the grid*

Iteration

- each time step, each agent moves to a random, unoccupied neighboring location

Stopping Condition

- Not part of the specification.

Supplementary Detail

- *agent positions are characterized by integer pairs (locations on the grid)*
- no two agents share a position, *not even initially*
- agent movement is constrained to a Moore neighborhood of radius 4
  - an agent must change position unless all neighboring locations are occupied
  - the new position is a "random" selection from the unoccupied neighboring locations (no particular algorithm or PRNG is specified)
  - *the move action must terminate, even if all neighboring locations are occupied*
- *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*

Display *Suggestions*

- position: display each agent on screen at a position corresponding to its grid position
- shape: display agents as circles
- color: use a red fill color for agents
- *size: make sure adjacent agents are easily distinguishable*
- update: display once each iteration

Despite its simplicity, the original specification contained some minor ambiguities. The emphasized text in our specification constitutes clarifying additions. We specify that the initial positions of agents are chosen randomly subject to the constraint that locations are not shared. This addition is not very consequential (and matches the NetLogo reference implementation). 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, and as Kahn (2007) notes, violation of this constraint can affect outcomes in later template models.

Finally, we explicitly specify that the move action must terminate. While this is the natural reading of the original specification, it conflicts with the reference implementations.[2] As Kahn (2007) notes, the reference implementations use a simple loop that makes repeated random draws from the entire neighborhood until a vacant cell is found. If there are no vacant cells, the program never escapes this loop. (Admittedly, this is a low probability event in the first template model.) However we do not augment the specification to require a specific algorithm, since the ease of any particular approach can be platform specific.

It is notable that the display suggestions have no essential relationship to the model specification. In fact, one drawback of the original template model specifications is the lack of a clean separation of core modeling goals and visual display considerations. We cannot address this fully without diverging substantially from the original specifications, but a natural supplement to the template models is a cleaner separation of these concerns. We will provide this on a template by template basis. One other drawback or the original specification is the unnecessary narrowness of original interpretive framework. The authors of the original template model specifications presumed an ecological emphasis and somewhat 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

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. With such limited needs, we have many different options. Starting from scratch is entirely feasible.[3] Free and open source game development toolkits are also options: pygame (`http://www.pygame.org`) and pyglet (`http://www.pyglet.org`) are fairly simple and very powerful. For the purposes of this paper, however, we take a minimalist approach: we use a small `gridworld` module that provides a basic grid topology, "patches", and agents. Since Python 2.6, this approach provides ABM facilities that approach the simplicity of NetLogo.

The properties of an `Agent` resemble those of a NetLogo "turtle", which is the base NetLogo agent type.[4] We will 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. (Generally, the coordinates can be floating point numbers, but for the template models they will be integers.)

To facilitate comparison with NetLogo, we consider a preliminary version of the first template model that is procedurally oriented. Our world will be a `GridWorldGUI` from the `gridworld` module; this class handles a little grid-related accounting for us. For example, if `myworld` is a `GridWorldGUI` 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 are including the entire file.)

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

import random
from gridworld import Agent, GridWorldGUI
from gridworld import moore_neighborhood, Torus

def initialize(agent):
  agent.shape('circle')
  agent.shapesize(0.25, 0.25)
  agent.fillcolor('red')

def move(agent):
  choice = choose_location(agent)
  agent.position = choice

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)

#create a 100x100 grid and a world based on this grid
mytorus = Torus(shape=(100,100))
myworld = GridWorldGUI(grid=mytorus)
#create 100 agents, located in our world, and then initialize them
myagents = myworld.create_agents(AgentType=Agent, number=100)
for agent in myagents:
  initialize(agent)
```

```
for ct in range(250): #run the model (250 iterations)
  myworld.screen_updating = False
  schedule()
  myworld.screen_updating = True
```

Since this is our first model implementation, we will explicate the code. Readers with a little Python programming experience may be able to simply read through the code and then skip ahead to the more object-oriented implementation. (This invitation should not be taken for granted: it is a reasonable suggestion *because* readability is an explicit Python language design goal.)

The source file begins with a documentation string. (Python docstrings are conventionally triple quoted; triple quoted strings can include line breaks.) Immediately following the docstring are three import statements. The first imports the `random` module from the standard library, giving us access to Python's random number facilities. The other two import useful objects from the `gridworld` module. We import `Agent` and `GridWorldGUI`, two classes that will be useful for our simulation. After import, names are available for use anywhere in our program (i.e., they are in our module's global namespace). We also import the `moore_neighborhood` function, which can produce a Moore neighborhood of any specified radius, along with the `Torus` class, which characterizes a rectangular grid that wraps at its boundaries.

Our approach will be to define a collection of functions that we can use to implement the simulation.[5] Our first function simply initializes an agent. This initialization focuses on the visual display of the agent: we set its shape, size, and color.[6] We then attack the core of the first template model: the detailed description of how an agent moves. We define a `move` function, which chooses a new location and then moves the agent there. Location choice is delegated to a `choose_location` function. Movement to a new location is handled simply by assigning a new value to the agent's `position` attribute.

Let us consider the body of the `choose_location` function, which does the real work of the first template model. We generate a Moore neighborhood of radius 4 around the agent's current position as 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 (in place) the locations with the `random.shuffle` function. We sequentially consider each location as a possible new position (thereby avoiding the error in the reference implementations, discussed above). If the location is empty, we choose it. Note that a `gridworld.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 agents position to this location.

We have completed the hardest and most essential part of the first template model. Our `schedule` function is relatively simple: it applies our move function to each of our agents. We will call `schedule` once per iteration as long as the model is running.

We are now ready to set up and run the model. Following the specification, we create our 100 by 100 grid as `mytorus = Torus(shape=(100,100))`. We then create our world as `myworld = GridWorldGUI(grid=mytorus)`, initializing our world with our torus topology. We then populate the world with 100 agents: a `GridWorldGUI` also has a `create_agents` convenience method, so we need only specify the number and type of agents that we want. (Since we do not specify positions, the agents are positioned randomly when they are created.) We then use our `initialize` function to prepare the agents for our simulation.

Running the model is essentially a matter of calling the `schedule` function repeatedly. However we also perform a subsidiary, display-related task: we turn off screen updating before we start moving our agents, and we turn screen updating back on after we are done moving our agents. This was requested by the specification, and it substantially speeds the simulation. We arbitrarily stop after 250 iterations. (The specification does not include a stopping criterion.)

Object-Oriented Implementation

For illustrative contrast, we reimplement the first template model with more object-oriented code. We can again use Python, since it combines simplicity and power by supporting both procedural and object-oriented programming paradigms.

This time we use the `GridWorldGUI` and `Agent` classes of the `gridworld` module as base classes for two new classes, `World01` and `Agent01`. Instead of defining four global functions to manipulate the objects in

our model, we provide our objects with behavior by defining four corresponding methods. We initialize the shape, size, and color of our agents with an `initialize` method. We add a `move` method to our agents, which will be supported by a `choose_location` method. These are analogues for three of the four functions we defined in the procedural version. The fourth function was our schedule, which we now implement as method of our world class.

Before proceeding, we should note two features of the `gridworld` module. `Agent` defines an `initialize` method, intended to be overridden, which is automatically called as part of object initialization. Also, `GridWorldGUI` defines a `schedule` method, intended to be overridden, which is called repeatedly by its `run` method. So we propose the following design.

Agent01 extends gridworld.Agent

- New Methods: `move`, `choose_location`
- Overridden Methods: `initialize`

World01 extends gridworld.GridWorldGUI

- Overridden Methods: `schedule`

Here we use the term 'extends' to indicate inheritance: `Agent01` is a subclass of `Agent`. To say that `Agent01` extends `Agent` is to say that `Agent01` inherits data and methods from `Agent`. Specifically, we will use the `position` and the `world` data attributes that `Agent01` inherits from `Agent`. Similarly, we will use the `is_empty` method that `World01` inherits from `GridWorldGUI`, which reports whether a given location is empty or occupied by another agent.

Explicit class definition has costs and benefits relative to other possible designs. The 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.) Object-oriented design requires an understanding of class definition. In Python, this is a minimal barrier: understanding how NetLogo handles instance variables (as required by the second template model) is a comparable cost.[7] 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. As we develop our object-oriented approach to the first template model, we will not find large advantages off-setting this initial cost. However, as we work through additional template models, substantial advantages of a more object-oriented approach become evident.

For most researchers choosing an ABM platform, the issue is not whether but rather when to use inheritance. For example, NetLogo is strongly procedurally oriented and is designed for simple use, so the NetLogo reference implementations initially can dodge the issue by relying on a kind of dynamic attribute creation for the base agent class (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]

The NetLogo reference models do not introduce breeds until template model 16. We could similarly postpone the introduction of inheritance, as illustrated by our procedural version of the first template model. 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 procedural implementation with the following permits an assessment of the barrier to entry is raised by early use of a more object-oriented approach.

```
""" Template Model 1: Random Movement on a Toroidal Grid """

import random
from gridworld import Agent, GridWorldGUI
from gridworld import moore_neighborhood, Torus

class Agent01(Agent):
  def initialize(self):
    self.shape('circle')
    self.shapesize(0.25, 0.25)
    self.fillcolor('red')
  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(GridWorldGUI):
  def schedule(self):
    for agent in self.agents:
      agent.move()

if __name__ == '__main__': #setup and run the simulation
  #create a 100x100 grid and a world based on this grid
  mytorus = Torus(shape=(100,100))
  myworld = World01(grid=mytorus)
  #set up agents (create 100 agents, automatically initialized)
  myagents = myworld.create_agents(AgentType=Agent01, number=100)
  #run the world's schedule repeatedly
  myworld.run(maxiter=250)
```

A reasonable first reaction is that this more object-oriented implementation appears nearly identical to our procedural implementation. The four functions defined in the procedural version are clearly evident as the three methods defined in the `Agent01` class and the one method defined in the `World01` class. (A method definition is just a normal function definition that is part of a class definition.) A closer look reveals some small differences, and we focus on those.

Again the module begins with a short docstring. The three import statements are unchanged. We then define a new class, `Agent01`, which inherits from `Agent` and defines three methods. The `initialize` method has one difference from our earlier function of the same name: the parameter name is `self` rather than `agent`. This is 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.)

The `move` method also mirrors our previous `move` function. There is one notable difference: instead of the function call `choose_location(agent)`, we have the method call `self.choose_location()`.[9] Similarly, the `choose_location` method is also essentially identical to our previous `choose_location` function.

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.

Our remaining changes are just as minor. Instead of defining a global `schedule` function, we subclass `GridWorldGUI` 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 `screen` updating on and off. (We

will use the `run` method that our `World01` inherits from `GridWorldGUI`, which handles this for us.) Second, we will not access `myagents` as a global variable: instead we use the `agents` attribute of our `World01`, which it inherits from `GridWorldGUI`. Finally, we replace the function call `move(agent)` with the method call `agent.move()`.

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 `myworld.run(maxiter=250)` will run our simulation for 250 iterations. Our more object-oriented implementation of the first template model is complete.

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__`, which 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.[10] This ensures that we can freely import our module into other scripts.

Template Model 2: Agent Growth

The second template model assumes completion of the first template model. (The template models are generally sequential.) The new requirement is the addition of agent state in the form of data attributes. An associated visual-display goal is the provision of visual clues to the state of each agent.[11] 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

Iteration (*sequential*)

- each agent moves
- each agent grows

Supplementary Detail

- movement: agents move as in model 1
- growth: each agent "grows" at a fixed rate
  - each agent has an data attribute named `size`
  - *growth is a change in the "size" attribute*
  - `size` is initialized to 1.0; the growth increment is *0.1*

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: ensure that the agents' color changes during the model's first ninety iterations are not too rapid for viewing*

We clear up one ambiguity in the original specification, which says that the growth action "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. We also cut the growth increment from 1.0 to 0.1. The growth increment is inessential to the model, aside from visual display considerations. Even after our

change, on many platforms the color transition will take place too quickly to be easily seen, unless the model iterations are radically delayed. We therefore add a new display suggestion: schedule a delay if needed for comfortable viewing of the color transitions.

Note that the notion of `size` here is very abstract, and correspondingly so is the notion of `growth`. A biologist might conceive of `size` as the physical size of the agent, while a social scientist might conceive of `size` as the monetary value of the agent's wealth.

With this background, we propose a basic design. As we take up the second template model, we would like to take advantage of our work on the first template model. One approach to this is to simply copy our `Agent01` code into the file that implements the second template model, modifying it where appropriate. Indeed, 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.[12] We therefore subclass `Agent01` to define a new agent class (`Agent02`). that has a new data attribute (`size`) and two new methods (`grow` and `change_color`). Additionally, we create a new class, named `World02`, for our new schedule of actions. Since `World01` does not offer much here, we again subclass `GridWorldGUI` in order to override its `schedule` method.

Agent02 extends Agent01

- New data attribute: `size`
- New Methods: `grow`, `change_color`
- Overridden Methods: `initialize`

World02 extends GridWorldGUI

- Overridden Methods: `schedule`

With this design in hand, we are now ready to implement the second template model. We begin with three import statements. We import the `time` module, which is in the Python standard library, so that we can use its `sleep` function to delay our model iterations. From the `gridworld` module we import two familiar classes, `GridWorldGUI` and `Torus`, along with an inessential but convenient function, `ask`. Analogously to the `ask` command in NetLogo, this `ask` function applies a specified method to each object in an iterable collection. (Thus `ask(myagents, 'move')` is equivalent to `for agent in myagents: agent.move()`.) Finally, from our (obviously names) `template01` module we import `Agent01`, which we intend to subclass.

```
""" Template Model 2: Agent Growth """

import time
from gridworld import ask, GridWorldGUI, Torus
from template01 import Agent01

class Agent02(Agent01):
  def initialize(self):
    self.shape('circle')
    self.shapesize(0.25, 0.25)
    self.size = 1.0
    self.change_color()
  def grow(self):
    self.size += 0.1
    self.change_color()
  def change_color(self):
    g = b = max(0.0, 1.0 - self.size/10)
    self.fillcolor(1.0, g, b)

class World02(GridWorldGUI):
  def schedule(self):
    time.sleep(0.2)
    myagents = self.agents
    ask(myagents, 'move')
    ask(myagents, 'grow')
```

```
if __name__ == '__main__':
  myworld = World02(grid=Torus(shape=(100,100)))
  myagents = myworld.create_agents(AgentType=Agent02, number=100)
  myworld.run(maxiter=100) # run simulation
  myworld.mainloop() # keep GUI open after `run` completes
```

Our new agent class, `Agent02`, has two new methods (`grow` and `change_color`) and a new data attribute (`size`). We initialize each agent's `shape`, `shapesize`, and `size`, and then we initialize its color by calling its `change_color` method. The change color method sets the agent's fillcolor based on its `size`, following the display suggestions in the specification.

The `grow` method is almost trivial: as required by the specification, an agent adds 0.1 to its `size` each time its `grow` method is called. Since the agent's color should always reflect its size, the `grow` method then calls the agent's `change_color` method. Using the default RGB color model, we specify colors as (red,green,blue) triplets of floating point numbers between 0 and 1. The intensity of red is always 1.0. The intensity of green and blue decreases from 1.0 to 0.0, as `size` increases from 0 to 10, and remains at 0 thereafter.

`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 `GridWorldGUI` 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 retrieves an iterable collection of the agents, sequentially calls the `move` method on each of these agents, and then sequentially calls the `grow` method on each of these agents.

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 world. For now, this is just to keep our display visible after our simulation is done running. (We will return to this.)

We emphasize that our code listing 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, our Python code remains clear and readable. As early as the second template model, we discover an evident strength of Python as an ABM platform.[13]

Template Model 3: Spatially Distributed Resources

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 spatial resources in the form of spatially distributed "cells" with which agents interact. This interaction changes the state of the cell and the state of the agent. To make this concrete, we say that cells produce a resource, and agent growth is determined by the resource it extracts from its cell. A secondary goal is to introduce a random change in state: cells produce randomly.

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 implementation details that users would have trouble confirming on many platforms.[14] 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.)

Furthermore, the rest of the specification (and its interpretive framework) 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 `grow` procedure is applied to each agent, which consumes from its cell and grows.[15] We can conceive a more patch-centered approach—say, applying a `supply-agent` procedure to each patch—but this is not natural, necessary, or efficient.

Specification: Template Model 3

Setup

- same as model 2, plus
- create a "cell" *for* each grid location

Iteration (sequential)

- each cell produces, which adds to its available supply
- each agent moves
- each agent grows

Supplementary Detail

- production: random, *uniform* between 0 and the cell's maximum production rate
  - a cell has an available supply of the produced resource (data attribute, initialized to 0.0) and a maximum production rate (data attribute, initialized to 0.01)
  - 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 (a data attribute, initialized to 1.0)
  - an agent extracts whichever is smaller: its maximum extraction rate, or its cell's supply
  - agent growth equals the quantity it extracts
  - a cell's supply is reduced by the amount extracted

Display *Suggestions*

- unchanged from model 2

As a minor matter, we have adopted 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 possible. (For example, itinerant laborers solicit paid jobs, which add to their wealth.) Again, the interpretive framework of the template models is secondary: any serious application to interactions between an agent and its spatial environment will require many more details.

With this background, we are ready to propose a design. We need to describe a cell that produces something that can be taken by our agents. We also need to make some changes to our agent description, so that it can appropriately consume and grow.

Cell03 extends gridworld.Patch

- New Data: `supply`, `max_produce`
- New Methods: `produce`, `give`
- Overridden Methods: `initialize`

Agent03 extends Agent02

- New Data: `max_take`
- New Methods: `extract`
- Overridden Methods: `initialize`, `grow`

World03 extends gridworld.GridWorldGUI

- Overridden Methods: `schedule`

At this point, our basic strategy should be growing familiar. We inherit as much as we can from past work, and we add any new behavior we need. For example, `Agent03` will inherit the `patch_here` method from `Agent02`, who inherited it from `Agent01`, who inherited in from `gridworld.Agent`. However, it must override the `grow` method rather than inherit it from `Agent02`. We now move to discussion of the code.

```
""" Template Model 3: Spatially Distributed Resources """

import random
from gridworld import ask, GridWorldGUI, Patch, Torus
from template02 import Agent02

class Cell03(Patch):
  def initialize(self):
    self.supply = 0.0
    self.max_produce = 0.01
  def produce(self):
    self.supply += random.uniform(0, self.max_produce)
  def give(self, amount):
    amount = min(self.supply, amount)
    self.supply -= amount
    return amount

class Agent03(Agent02):
  def initialize(self):
    Agent02.initialize(self)
    self.max_take = 1.0
  def grow(self):
    self.size += self.extract()
    self.change_color()
  def extract(self):
    mycell = self.patch_here()
    mytake = mycell.give(self.max_take)
    return mytake

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


if __name__ == '__main__':
  myworld = World03(grid=Torus(shape=(100,100)))
  mypatches = myworld.create_patches(PatchType=Cell03) #setup patches
  myagents = myworld.create_agents(AgentType=Agent03, number=100)
  myworld.run(maxiter=100)
  myworld.mainloop()
```

For now, the only important thing we get by having `Cell03` inherit from `gridworld.Patch` is that a `GridWorldGUI` knows how to create instances of `Patch`. (We will use this when we set up the model.) All the important cell behavior is new. We begin by overriding the `initialize` method, which is automatically called during instance creation. This initializes the `supply` and `max_produce` attributes of each cell. We define a `produce` method to augment supply by a random amount, which is uniform between 0 and `max_produce`. And the `give` method will give from the cell's current supply the `amount` requested (up to the entire supply), while reducing the cell's `supply` by the amount given.

An `Agent03` has all the initializations of an `Agent02`, and we invoke by calling `Agent02.initialize`. (This is for illustrative purposes: the gain from reusing the `Agent02` initializations is tiny in this simple class.) An `Agent03` is an `Agent02` that knows how to extract resources from its cell and to grows based on this extraction. The `grow` method calls the agent's `extract` method, and growth equals the amount extracted. Given a call to its `extract` method, an `Agent03` will try to extract its `max_take` (e.g., its maximum capacity). (Of course it can only extract up to its cell's current `supply`.) The `max_take` of an `Agent03` is set at initialization to 1.0.

As usual, we create a schedule this by overriding the `schedule` method of `GridWorldGUI`. First we ask each patch to `produce`, then we ask each agent to `move`, and finally we ask each agent to `grow` (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 environment (i.e., the cells). (This step has no corollary in NetLogo, which always conveniently creates a patch environment for its agents, but it is required by most ABM platforms.) Note that the specification still does not include a stopping condition, so our decision to run the simulation for 100 iterations is in this sense arbitrary.

Template Models 4, 5, 6, 7: Probes, Parameters, Histograms, and Stopping

The next three template models emphasize the 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

The core task of the fourth template model is to implement click monitors, or "probes", fore 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 (with new supplementary detail)

User Interaction

- 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

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.[16] We add probes explicitly using our world's `add_clickmonitor` method, which takes as arguments a label for the display, an object type to monitor, and an attribute (or attributes) whose values are to be monitored.[17]

```
""" Template Model 4: Probe Object State """

from gridworld import Torus
from template03 import Agent03 as Agent04, Cell03 as Cell04, World03

class World04(World03):
  def initialize(self):
    self.add_clickmonitor('Agent', Agent04, 'size')
    self.add_clickmonitor('Cell', Cell04, 'supply')

if __name__ == '__main__':
  myworld = World04(grid=Torus(shape=(100,100)))
  mypatches = myworld.create_patches(PatchType=Cell04)
  myagents = myworld.create_agents(AgentType=Agent04, number=100)
  myworld.run(maxiter=100)
  myworld.mainloop()
```

The resulting code is very simple. We simply reuse `Agent03` and `Cell03` (renamed as `Agent04` and `Cell04`). We subclass `World03` in order to override its `initialize` method. (Recall that the `initialize` method is special: it is called when our world 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.

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 current state of the objects we click. 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: Model Parameters

The fifth template model introduces user settable model parameters. While the concept is not precisely defined, the idea is that certain variables that govern the simulation should be easily settable by users of the model.

Specification: Template Model 5

Setup and Iteration

- unchanged from model 4

Model Parameters

- initial number of agents (*suggested* default: 100)
- maximum extraction rate of agents (*suggested* default: 1.0)
- maximum production rate of cells (*suggested* default: 0.01)

Supplementary Detail

- model parameters should be easily settable by users of the model.

Display *Suggestions*

- as in model 4, plus
- enable GUI setting of model parameters *(e.g., with sliders)*
- *include SetUp and Run buttons in the GUI (where "SetUp" creates the agents (and patches if needed), and "Run" should run the simulation*

We make two changes in the original specification. First, we do not *require* GUI parameter setting. While this can be quite useful for experimentation with the model, use of the 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 demote this to a display suggestion. Second, we add to the display suggestions that provision of `SetUp` and `Run` buttons. This suggestion is a natural implication of allowing parameter setting in the GUI: the parameters 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.)

As Railsback et al. (2006) note, model parameters are often implemented as attributes of a model class. For the three specified model parameters, we introduce three class variables to our `World05` class. Since patches and agents created by a `GridWorldGUI` know their world, they have access to these parameters as attributes of their world.

With this background, we expect that the primary change from the third template model is a change in the `initialize` methods of our cell, agent, and world classes. We add button and slider creation to the initialization of our world. During their initializations, cells and 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.

```
""" Template Model 5: Parameters """

from gridworld import Patch, Torus
from template04 import Cell04, Agent04, World04

class Cell05(Cell04):
  def initialize(self):
    self.supply = 0.0
    self.max_produce = self.world.cell_max_produce

class Agent05(Agent04):
  def initialize(self):
    Agent04.initialize(self)
    self.max_take = self.world.agent_max_take

class World05(World04):
  n_agents = 100
  cell_max_produce = 0.01
  agent_max_take = 1.0
  def initialize(self):
    World04.initialize(self)
    self.add_slider('Initial Number of Bugs', 'n_agents', 10, 500, 10)
    self.add_slider('Agent Max Extraction', 'agent_max_take', 0.0, 2.0, 0.1)
    self.add_slider('Cell Max Production', 'cell_max_produce', 0.0, 0.1, 0.01)
    self.add_button('Set Up', self.setup)
    self.add_button('Run', self.run)
    self.add_button('Stop', self.stop)
  def setup(self):
    mypatches = self.create_patches(PatchType=Cell05)
    myagents = self.create_agents(AgentType=Agent05, number=self.n_agents)

if __name__ == '__main__':
  myworld = World05( grid=Torus(shape=(100,100)) )
  myworld.mainloop()
```

Let us focus on the `initialize` method of `World05`. We begin by invoking all the `World04` initializations. Then we add three sliders, one for each of the model parameters that we introduced as class variables. 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, an attribute (of our world) to be set, 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 is called a "callback" (or "command"): it is called when the button is clicked. Our buttons call the `setup`, `run`, and `stop` methods of our world.

Our `World04` inherited `run` and `stop` methods from `GridWorldGUI`. We do not override these. It also inherited 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 world to receive mouse clicks. Here too it sets up event handling, and now these events include button clicks and slider adjustments. We now use button clicks to set up and run the model.

Template Model 6: Dynamic Histogram

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 `size` among the agents.

We make a minor change in the original specification: we do not require that the histogram be displayed in the GUI, and we thereby remove the implicit requirement that 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 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 an 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 Model Parameters

- as in model 5

Data Display

- produce a series of histograms 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*

Display *Suggestions*

- use 10 bins for the histogram, with a minimum size of 0 and a maximum size of 10
- *display a histogram chart in the GUI, and update the histogram as the simulation runs*

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 very flexible.

```
""" Template Model 6: Histogram """

from gridworld import Torus
from template05 import Agent05 as Agent06, World05

class World06(World05):
  def initialize(self):
    World05.initialize(self)
    def get_sizes():
      agents = self.get_agents(Agent06)
      return list(agent.size for agent in agents)
    self.add_histogram('Agent Sizes', get_sizes, bins=range(11))

if __name__ == '__main__':
  myworld = World06(grid=Torus(shape=(100,100)))
  myworld.mainloop()
```

We add a dynamic histogram to the GUI with the `add_histogram` method of our world. This method needs as arguments a title for the graph and a function that will be recomputed each iteration. When called, this function must provide the data for the histogram. We therefore define a `get_sizes` function, which returns a list of agent sizes.[18] In doing so, we add one forward looking refinement: we illustrate the `get_agents` method our world inherited from `GridWorldGUI`. This returns a list of the agents of the specified type. (Of course currently all of our agents are of type `Agent06`, so we get a list of all agents.) From this, we produce a list of the agent sizes. 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 Model 7: Stopping Condition

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.[19]

Specification: Template Model 7

Setup, Iteration, Model Parameters, and Data Display

- unchanged from model 6

Stopping Condition

- terminate iteration based on model state

Supplementary Detail

- base stopping condition on a cutoff for the agent `size` attribute
- the suggested cutoff is that any agent reaches `size >= 100`
- after stopping, completely exit the simulation

Display *Suggestions*

- as in model 6, plus
- close GUI when the simulation terminates

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).[20]

There are many ways to approach this problem, but the most straightforward is to add a conditional check to the schedule. (This is essentially the approach taken by the NetLogo reference implementation.) We test for the stopping criterion, and once it is satisfied, we stop the simulation and exit the program. Note that this approach is an implementation detail, and is not required by the specification. To stop the iterative process we use the `stop` method that our world inherited from `GridWorldGUI`; to exit the mainloop (and thus completely terminate the program) we use its `exit` method.[21]

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

from gridworld import Torus
from template06 import Agent06 as Agent07, World06

class World07(World06):
  def schedule(self):
    World06.schedule(self)
    if max(agent.size for agent in self.agents) >= 100:
      self.stop()
      self.exit()

if __name__ == '__main__':
  myworld = World07(grid=Torus(shape=(100,100)))
  myworld.mainloop()
```

Template Model 8: Output Files

Railsback et al. (2006) point out 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 is 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.[22] Plain text output has the great advantage of being human readable, but when we generate a large amount of data this advantage is rapidly outweighted by speed (of reading and writing) and eventually even file-size considerations. (Vaingast (2009) offers a good introductory discussion.)

Even if we settle on a plain text file format, many issues are unaddressed by the specification. Will the data be written in a well-known format, such as the 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 specification, as the needs and resources of users vary widely, we do introduce some output *suggestions* into the specification.

Specification: Template Model 8

Setup, Model Parameters, Data Display, and Stopping Condition

- unchanged from model 7

Iteration (sequential)

- as in model 7
- compute 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*

Display *Suggestions*

- as in model 7

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.[23] 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 produce nicely formatted plain text output.

We will use Python's basic file handling abilities, as implemented in the built-in `open` command.[24] 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 `fh=open('sizes.csv','w')`. (The `open` command returns a file object, which we bind to the name `fh`.) We can then write a string `mystring` to the file as `fh.write(mystring)`.

We can implement the eighth template model simply by extending `World07`. We add two new class variables (`logfile` and `logformat`), add two new methods (`header2logfile` and `log2logfile`), and override two methods (`setup` and `schedule`).

Our `setup` keeps the `World07` set up but appends logging of a header and an initial data summary to our log file. Data logging is handled by a 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 `describe`, which is a `gridworld` convenience function.[25] This has functionally 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 `self.logformat.format(**stats)` uses our `logformat` string to format the corresponding values in the `stats` dictionary. (This offers a nice illustration of the power and flexibility of using a Python format string). This is all it takes to log our data in CSV format.

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

from gridworld import Torus, describe
from template07 import Agent07 as Agent08, World07

class World08(World07):
  logfile = 'c:/temp/sizes.csv'
  logformat = '\n{min}, {mean}, {max}'
  def setup(self):
    World07.setup(self)
    self.header2logfile() # write header to logfile
    self.log2logfile() # log initial state to logfile
  def schedule(self):
    World07.schedule(self)
    self.log2logfile()
  def header2logfile(self):
    logheader = "minimum, mean, maximum"
    fh = open(self.logfile, 'w')
    fh.write(logheader)
    fh.close()
  def log2logfile(self):
    agents = self.get_agents(AgentType=Agent08)
    sizes = list(agent.size for agent in agents)
    stats = describe(sizes)
    fh = open(self.logfile, 'a')
    fh.write( self.logformat.format(**stats) )
    fh.close()

if __name__ == '__main__':
  myworld = World08(grid=Torus(shape=(100,100)))
  myworld.mainloop()
```

We have one last change to make, and that is to the schedule. We begin with the `schedule` of `World07`, but we append to it the data logging we want to do for each iteration. 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

In the template models, each agent moves once each iteration. However, we have not specified the order in which agents they move. Neglecting this has different implications on different platforms. For example, the most commonly used iterable collection in NetLogo is the "agent set". Iteration over a NetLogo agent set retrieves agents in random order. (For example, when we `ask` a NetLogo agent set 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, `gridworld` provides separate `askrandomly` and `ask` functions, allowing us to explicitly state whether or not we want to shuffle our agents before making our method calls. This has some advantages: shuffling the agents is a computational expense, which we should incur on an as needed basis.

In the first eight template models, the one place order might matter is when we ask agents to move. (Since occupation is unique, when one agent moves to a cell, the next agent cannot move to that cell.) 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, Model 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 is 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: "bigger" agents get to move before "smaller" agents. (This will matter when we get to our next template model.) On a biological interpretation, this hierarchy might suggest an advantage of physical size. On a economic interpretation, the hierarchy might suggest an advantage of the wealthiest individuals when choosing from the available economic opportunities.

Specification: Template Model 10

Setup, Model Parameters, Iteration, Data Display, and Stopping Condition

- unchanged from model 8 (but with new supplementary detail)

Supplementary Detail

- the order in which agents move is determined by their `size` attributes: bigger agents move before smaller agents

Sorting is fairly simple 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 Python's `sorted` function takes a sort key.

```
""" Template Model 10: Hierarchy """

from gridworld import ask, Torus
from template08 import World08

class World10(World08):
  def sortkey10(self, agent):
    return agent.size
  def schedule(self):
    ask(self.patches, 'produce') #from model 3
    #version 10: agents move in size order
    agents = sorted(self.agents, key=self.sortkey10, reverse=True)
    ask(agents, 'move') #from model 1
    ask(agents, 'grow') #from model 3
    self.log2logfile() #from model 8
    if max(agent.size for agent in agents) >= 100: #from model 7
      self.stop()
      self.exit()

if __name__ == '__main__':
  myworld = World10(grid=Torus(shape=(100,100)))
  myworld.mainloop()
```

Implementation is 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, our sort key `sortkey10`. This 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.) Python's built-in `sorted` function handles 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 out `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`, which we did not subsequently override. Location choice has remained random. Such agents have very limited agency. In the eleventh template model, we add interest to our agents by adding optimization to their movement.

We remove a couple ambiguities in the original specification. Substantively, we make it clear that the hierarchical movement of model 10 still applies. As a 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 models, ties will be much more likely.)

Specification: Template Model 11

Setup, Model Parameters, Iteration, Data Display, and Stopping Condition

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

Supplementary Detail

- agent movement is no longer random
  - an agent explores cell its neighborhood and moves to the cell that has the largest `supply`
  - an agent's neighborhood is still a *Moore neighborhood of radius 4*, but now the agent's current cell is included
  - *in case of a tie, the agent moves to a random choice of the best cells*

We will reuse our `Cell05` class, renaming it to `Cell11`. We will essentially reuse our `World10` class: `World11` just overrides `setup` to use our new `Agent11`. The only substantial changes are to our agent class. `Agent11` has two new methods: the new `sortkey11` method, which provides a key for sorting cells by their `supply`, and an overridden `choose_location` method, which picks the best location in the agent's neighborhood.

```
""" Template Model 11: Optimization """

import random
from gridworld import Torus
from template05 import Agent05, Cell05 as Cell11
from template10 import World10


class Agent11(Agent05):
  def sortkey11(self, cell):
    return cell.supply
  def choose_location(self):
    hood = self.neighborhood('moore', 4)
    MyType = self.__class__
    hood4move = [ cell for cell in hood if not cell.get_agents(MyType) ]
    hood4move.append(self.patch_here())
    random.shuffle(hood4move)
    best_cell = max(hood4move, key=self.sortkey11)
    return best_cell.position

class World11(World10):
  def setup(self):
    mypatches = self.create_patches(PatchType=Cell11)
    myagents = self.create_agents(Agent11, number=self.n_agents)
```

```
if __name__ == '__main__':
  myworld = World11(grid=Torus(shape=(100,100)))
  myworld.mainloop()
```

We focus on this `choose_location` method. An `Agent11` must choose the unoccupied neighboring cell with the greatest `supply`. So first we retrieve a list of all the cells in a Moore neighborhood of radius four. Then we filter out the occupied patches. (The expression `cell.get_agents(AgentType)` returns a list of the agents of type `AgentType` on `cell`, which has a boolean value of `False` if the list is empty.) We want the agent to consider its own cell as well, so we retrieve it with the agent's `patch_here` method (inherited from `gridworld.Agent` and append it to the list. Then we shuffle the list as a way to ensure that any "ties" will be resolved randomly. The best cell is the one with the greatest supply. We can use Python's built-in `max` function to do the maximization, as long as we supply an appropriate key (`sortkey11`) on which to base the comparison. Having discovered the best patch, `choose_location` returns its position. This is our agent's choice of the best location for its move.

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

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, "bugs" in a biological model may exit through death and enter via birth, or "firms" in an economic model may exit through bankruptcy and enter as new start ups. 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 "split".

We make two changes to the original specification. We addresses a potential first mover advantage to the `split` action by specifying that agents split in random order. We also remove some ambiguities by clarifying that new agents will not immediately exit and by specifying that an agent makes its attempts to split and *then* dies (implying that its position remains occupied during the split action).

Specification: Template Model 12

Setup and Data Display

- unchanged from model 11 (with new supplementary detail)

Model Parameters

- as in model 11, plus
- probability of agent exit (*suggested* default: 0.05)

Iteration

- as in model 11, then
- each "big" agent splits
- each "smaller" agent exits with a fixed probability

Supplementary Detail

- split: each agent of `size` > 10.0 makes five attempts to produce a new agent, and then exits
  - *agents take the split action in random order*
  - to produce a new agent requires finding an unoccupied location in a randomly searched *Moore neighborhood* of radius 3, where no more than 5 cells may be searched for each attempt to produce a new agent
    - when an unoccupied cell is found, it becomes the position of a new agent, and search terminates
    - if all five cells are occupied, search terminates, and the attempt to bring a new agent into the world fails
  - each new agent has initial size of 0.0
- exit: each "smaller" agent (*except the new entrants*) exits the simulation with fixed probability
  - the probability of agent exit is a model parameter

Stopping Condition:

- the number of agents reaches 0, or the number of iterations exceeds 1000

We approach the twelfth template model by very slightly extending `World11` to implement our new schedule and by extending `Agent11` to handle the `split` action and to accommodate a chance of exit. This leads to the following design.

Agent12 extends Agent11

- New Methods: `split`, `propagate`, `chance`

World12 extends World11

- New Data: agent_exit_probability
- Overridden Methods: `schedule`, `setup`

`World12` is a very small extension of `World11`. As usual, we modify the `setup` method so that it creates agents and cells of from most recent classes. We also add a new class variable, `agent_exit_probability`, for our new model parameter.[26] Finally, we must append two agent actions (`split` and `chance`) to the `World11` schedule, along with our new stopping condition.[27]

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

import random
from gridworld import ask, askrandomly, Torus
from template11 import Agent11, Cell11 as Cell12, World11

class Agent12(Agent11):
  def split(self):
    if self.size > 10:
      self.propagate()
      self.die()
  def propagate(self):
    SplitClass = self.__class__ #splits share agent class
    hood4split = self.neighborhood('moore', radius=3)
    cells4split = list()
    for i in range(5): #5 attempts
      for cell in random.sample(hood4split,5): #5 tries per attempt
        if cell not in cells4split and not cell.get_agents(SplitClass):
          cells4split.append(cell)
          break
    splitlocs = list(cell.position for cell in cells4split)
    splits = self.world.create_agents(SplitClass, locations=splitlocs)
    for agent in splits:
      agent.size = 0.0
    return splits
  def chance(self):
    if self.is_alive:
      if random.uniform(0,1) < self.world.agent_exit_probability:
        self.die()

class World12(World11):
  agent_exit_probability = 0.05
  def schedule(self):
    World11.schedule(self)
    agents = askrandomly(self.agents, 'split')
    ask(agents, 'chance')
    if (self.iteration==1000) or (len(self.agents)==0):
      print self.iteration, len(self.agents)
      self.stop()
  def setup(self):
    mypatches = self.create_patches(PatchType=Cell12)
    myagents = self.create_agents(Agent12, number=self.n_agents)
```

```
if __name__ == '__main__':
  myworld = World12(grid=Torus(shape=(100,100)))
  myworld.mainloop()
```

Turning to `Agent12`, consider the new `chance` method. Each agent has an `is_alive` attribute which is set to `False` if the agent's `dies` method is called. ("Dying" is a model specific concept, which might represent physical death, failure of a firm, or even retirement of a worker.) The `chance` method draws from a standard uniform distribution and compares the draw to the `agent_exit_probability` model parameter. A small draw removes a "living" agent from the simulation; by default this happens five per cent of the time.

The real work of the twelfth template model is in the `split` action, where "bigger" agents propagate themselves and then die. Our very simple `split` method handles this by delegating propagation to a `propagate` method, which is where the real work takes place. Here we begin by specifying that successors (or "splits") will share the agent's type. Following the specification, we consider a Moore neighborhood of radius 3 as possible locations for these successors. 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., 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 implement this, we use the `sample` function of the `random` module to retrieve five random cells, which we check sequentially until we either find an empty cell or run out of options. This approach yields a list of empty cells, one for each successful split. We determine the position of each of these cells, and create the new agents at those locations. All that is left to do is to set the size of each new agent to zero, according to the specification.

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. 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.[28] 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 Model 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 is not specified, but the update frequency should be high enough to be informative about the evolution of this distribution*

Display *Suggestions*

- *display the plot in the GUI, and update it 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 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.[29] 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.

```
""" Template Model 13: Time-Series Plot """

from gridworld import Torus
from template12 import Agent12 as Agent13, World12

class World13(World12):
  def initialize(self):
    World12.initialize(self)
    def number_living():
      return len(self.get_agents(Agent13))
    self.add_plot('Number of Agents', number_living)

if __name__ == '__main__':
  myworld = World13(grid=Torus(shape=(100,100)))
  myworld.mainloop()
```

To implement the thirteenth template model, we make no changes to `Agent12`, which we rename as `Agent13`. Our `World13` is just a `World12` with two items appended to its initialization: the addition of a time series plot (using `addplot`), and the definition of the function `number_living`. This function, which we provide as an argument to `addplot`, 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 Initializations

One of the great strengths of agent-based modeling is that we have no need for a representative agent. Randomization can be a natural way to introduce initial heterogeneity. The object of the fourteenth template model is to illustrate the randomization of agent initialization. Each initial agent draws an initial `size` from a normal distribution, with mean and standard deviation that are user settable. The specification then requires that `size` be truncated at a minimum of zero.

Specification: Template Model 14

Model Parameters

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

World, Setup, Iteration and Stopping Condition

- as in model 13, plus
- the initial agents are initialized with a random `size`
  - `size` is based on a draw from a normal distribution
  - mean and standard deviation of this distribution are model parameters
  - *if the size draw is less than 0.0, it is set to 0.0*

Display *Suggestions*

- as in model 13, plus
- allow GUI setting of the 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.[30] Our only other change is the usual demotion of GUI parameter setting to a display suggestion.

Our agent type and world type will be largely identical to those in template model 13. `Word14` has two new parameters, which we name `agent_size_mean` and `agent_size_sd`. As usual, we introduce these parameters as class variables, which are initialized with the specified default values. `World14` reuses the `initialize` method of `World13` and then appends two sliders corresponding to the new parameters. (We discussed the `add_slider` method in the fifth template model.)

```
""" Template Model 14: Randomized Initializations """

import random
from gridworld import Torus
from template13 import Agent13, World13

class Agent14(Agent13):
  def initialize(self):
    Agent13.initialize(self)
    mean = self.world.agent_size_mean
    sd = self.world.agent_size_sd
    size_drawn = random.normalvariate(mean, sd)
    self.size = max(0.0, size_drawn)

class World14(World13):
  agent_size_mean = 0.1
  agent_size_sd = 0.03
  def initialize(self):
    World13.initialize(self)
    self.add_slider('Init. Size Mean', 'agent_size_mean', 0.0, 1.0, 0.1)
    self.add_slider('Init. Size SD', 'agent_size_sd', 0.0, 0.1, 0.01)

if __name__ == '__main__':
  myworld = World14(grid=Torus(shape=(100,100)))
  myworld.mainloop()
```

Most ABM platforms provide a fairly extensive collection of distributions for (pseudo) random number generation. The `random` module of the Python standard library provides a variety of distributions including the normal. We use the `normalvariate` function to initialize our first collection of agents.

`Agent14` overrides only the `initialize` method of `Agent13`. We keep all of the `Agent13` initializations, and we then set the agent's size randomly. An agent retrieves the mean (`agent_size_mean`) and standard deviation (`agent_size_sd`) from its world, draws from a normal distribution with these parameters, and then sets its size to the maximum of this draw or 0. Once we have defined our new world and agent classes, we are ready to create a `World14` instance and enter its main loop as usual.

Template Model 15: Data-Based Model Initialization

The fifteenth template model makes no fundamental changes in the characterization of agents or cells. However it requires that cell production rates be based on a data file. The data in this file provides a production rate at each location, and the shape of the grid must be determined by the listed locations.

Making the model set-up depend on data read from a file is the core change in the model, but the specification includes a few additional changes: agents must choose randomly among their best possible moves, 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. While these are fairly minor changes, including them all 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
- create a rectangular grid (*not* a torus) based on the shape implied by the cell data (see below)
- create a patch for each grid location
- set a production rate for each patch based on the cell data
- create agents as in model 14

Model 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: locations off the grid are now unavailable (instead of wrapping)*
- movement: an agent randomly resolves "ties" for best cell
- production: each iteration a cell produces the amount specified by the file of cell data (i.e., production is no longer random)

Display *Suggestions*

- as in model 14, plus
- display cells in the GUI
- base a cell's color on its `supply`
- 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 easily 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)*

We make few changes to the original specification. Most substantively, we remove the implementation details for random "tie" resolution, as we consider implementation details extraneous to the specification. The randomization itself is important to avoid movement artifacts (Railsback et al., 2005). Similarly, 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, since agents might have direct access to an appropriately constrained neighborhood. Finally, we add a few details about the cell data file format, and we demote display specifications to suggestions.[31]

We are ready to propose a design using new world and cell types. `Cell15` extends `Cell03` by adding a `change_color` method and overrides `initialize` to set an initial color. It also overrides the `produce` method to make production deterministic. `World15` is just `World14` with a slightly more complex `setup` method, since our set up depends on the cell data we must read from a file.

Cell15

- inherits data and methods from `Cell03`
- New Methods: `change_color`
- Overridden Methods: `initialize`, `produce`

World15

- inherits data and methods from `World14`
- Overridden Methods: `setup`

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 tie resolution. The other reason is a convenience of the `gridworld` module: an `Agent` has a `neighbors` method that queries its world, correctly handling 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 Initialization """

from gridworld import RectangularGrid
from template03 import Cell03
from template14 import Agent14 as Agent15, World14

def read_celldata(filename):
  location2value = dict()
  maxx, maxy = 0, 0
  fh = open(filename,'r')
  for _ in range(3):
    trash = next(fh) #discard 3 lines
  for line in fh:
    x, y, prodrate = line.split()
    x, y = int(x), int(y)
    maxx, maxy = max(x,maxx), max(y,maxy)
    prodrate = float(prodrate)
    location2value[(x,y)] = prodrate
  location2value['shape'] = (maxx+1,maxy+1)
  return location2value

class Cell15(Cell03):
  def initialize(self):
    self.supply = 0.0
    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.set_fillcolor(r, g, b)

class World15(World14):
  def setup(self):
    celldata = read_celldata('Cell.Data')
    shape = celldata.pop('shape')
    self.set_grid(RectangularGrid(shape=shape))
    patches = self.create_patches(PatchType=Cell15)
    for (x,y), prodrate in celldata.items():
      patches[x][y].max_produce = prodrate
    myagents = self.create_agents(Agent15, number=self.n_agents)

if __name__ == '__main__':
  myworld = World15(grid=None)
  myworld.mainloop()
```

We do not gain much in having `Cell15` inherit from our previous cell types. We reuse the `give` method we implemented for `Cell03`, but we override the `initialize` and `produce` methods. Our new `initialize` method simply sets an initial `supply` of zero, as before, but then changes the cell's color to match its supply. Our new `produce` method deterministically (instead of randomly) augments the cell's supply and then

changes the cell's color to match its new supply. (Note that `max_produce` now denotes this deterministic production level.) The new `change_color` method is used to set the cell's color to a shade of green, as specified. Once again we set the colors via the default RGB color model, which we discussed during the specification of the second template model.

`World15` is identical to `World14` except for the new `setup` method. As part of `setup`, we read the cell data from file using the helper function `read_celldata`.[32] This reads each line of the cell data file, discarding the first three and collecting the data from the others. It returns a Python dictionary mapping locations to production rates. It also computes the implicit shape of the grid but keeping track of the maximum values for each coordinate. We set our world's grid to a `RectangularGrid` (no longer a `Torus`), with dimensions based on the discovered shape. We then create our patches and set `max_produce` for each patch based on the data. As the final step in our `setup`, we create our agents. At this point we are read to create our world and enter its main loop, as usual.

Template Model 16: Interacting Agents of Different Types

The core programming goal of the sixteenth template model is to introduce a new agent type that can interact with our previous agent type. This interaction changes the state of one or more cells and the state of the agents. To lend concreteness, we will say our agents are hunters and prey.

Specification: Template Model 16

Model 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 prey
- hunting: hunters randomly search for prey and "kill" the first found
  - search randomly samples (without replacement) a *Moore neighborhood of radius 1*, center included
  - if *another* hunter is found in a cell, search terminates, and the hunter and remains at its current location.
  - if *another* hunter is not found in a cell, but a prey is found, search terminates, and the hunter moves to the cell and "kills" the prey
  - if no neighboring cell contains *another hunter* or a prey, the hunter moves randomly to an unoccupied cell
  - *prey are removed from the simulation as soon as they are "killed"*

Display *Suggestions*

- as in model 15, plus
- *hunters are colored yellow with the classic (arrowhead) shape*

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 prey is encountered *and* there is already a hunter in its cell? We follow the reference implementations and adopt the first interpretation.[33] 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 found during the search.[34] 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 prey be created before hunters, we add a display suggestion for the color and shape of the hunters.

Our implementation reuses `Agent15`, now renamed as `Prey`. Our `World16` overrides the `schedule` and `setup` methods of `World15` and make no other changes. And we introduce a `Hunter` class that extends our basic `Agent` class with a new `hunt` method.

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

import random
from gridworld import Agent, ask, askrandomly, Torus
from template15 import Agent15 as Prey, Cell15 as Cell16, World15

class Hunter(Agent):
  def initialize(self):
    self.fillcolor('yellow')
    self.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)
      prey = patch.get_agents(AgentType=Prey)
      if hunters and not self in hunters:
          change_position = False
          break
      elif prey:
        prey.pop().die()
        self.position = patch.position
        change_position = False
        break
    if change_position:
      newcell = random.choice(hunthood)
      self.position = newcell.position

class World16(World15):
  def schedule(self):
    ask(self.patches, 'produce')
    prey = self.get_agents(AgentType=Prey)
    prey = sorted(prey, key=self.sortkey10, reverse=True)
    ask(prey, 'move')
    ask(prey, 'grow')
    askrandomly(prey, 'split')
    ask(prey, 'chance')
    hunters = self.get_agents(Hunter) #model 16
    askrandomly(hunters, 'hunt') #model 16
    self.log2logfile()
  def setup(self):
    World15.setup(self)
    hunter_locations = self.get_random_locations(200)
    hunters = self.create_agents(Hunter, locations=hunter_locations)

if __name__ == '__main__':
  myworld = World16(grid=None)
  myworld.mainloop()
```

Our new `Hunter` class overrides the `initialize` method of `Agent` simply to set a color and size. The real work is in the new `hunt` method. A hunter starts by retrieving a neighborhood for hunting. This

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 prey. The hunter sequentially considers each patch in the shuffled neighborhood, retrieving a list of hunters and a list of prey 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 `prey` list contains a prey, search terminates and the hunter moves to that position and "kills" the prey. If no hunters are encountered and no prey found in the entire neighborhood, the hunter moves to a random cell in the neighborhood.

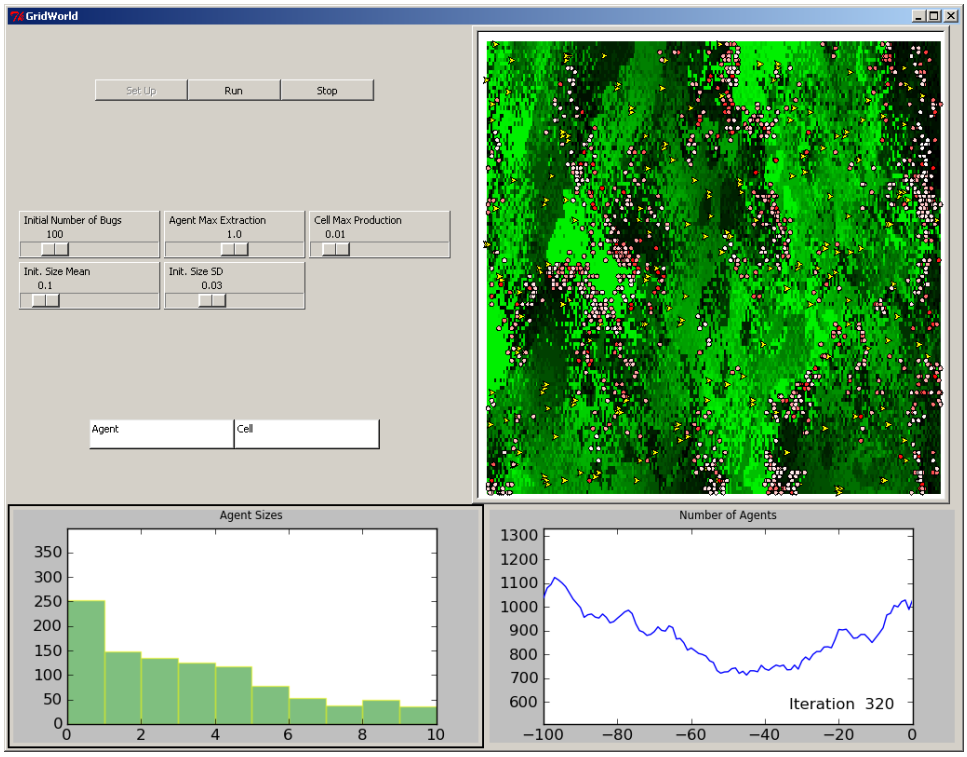

Figure 1: Figure 1: Template Model 16, GUI Display

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 still have patches produce and agents (prey) move, grow, split, and (possibly) exit. Only the next two statements are truly new: we need to 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

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 they contain some ambiguities, weaknesses, and errors. Rather than propose a new set of template models, we attempt to refine the existing set, and we provide a new, supporting reference implementation.

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 clever or fast. However we also address 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.

Railsback et al. (2005) conveniently provide reference implementations for a variety of platforms. In comparison, the Python implementations introduced by this paper are more compact, readable, and correct. They therefore provide a useful code resource for those who wish to utilize the template model specifications. Additionally, they add to the literature on the choice of ABM platforms. Our reference implementations show how a general purpose programming language such as Python can be readily exploited as a very powerful and yet easy to use ABM platform.

References

Downey, A. B. (2009). *Python for Software Design: How to Think Like a Computer Scientist*. Cambridge, UK: Cambridge University Press.

Hunt, A. and D. Thomas (1999). *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional.

Isaac, A. G. (2008, June). Simulating evolutionary games: A python-based introduction. *Journal of Artificial Societies and Social Simulation 11*(3), paper 8. `http://jasss.soc.surrey.ac.uk/11/3/8.html`.

Izquierdo, L. R. (2007). Netlogo 4.0 quck guide. `http://luis.izqui.org/resources/NetLogo-4-0-QuickGuide.pdf`.

Kahn, K. (2007). Comparing multi-agent models composed from micro-behaviours. In J. Rouchier, C. Cioffi-Revilla, G. Polhill, and K. Takadama (Eds.), *M2M 2007: Third International Model-to-Model Workshop*, pp. 165–177.

Kuhlman, D. (2008, August). Python 101 - introduction to python. `http://www.rexx.com/~dkuhlman/python_101/python_101.html`.

Langton, C. (1996). Simplebug swarm tutorial. Technical report, Santa Fe Institute. As updated by the Swarm Development Team. `http://ftp.swarm.org/pub/swarm/apps/objc/sdg/`.

Railsback, S., S. Lytinen, and V. Grimm (2005, December). Stupidmodel and extensions: A template and teaching tool for agent-based modeling platforms. Technical report, Swarm Development Group.

Railsback, S. F., S. L. Lytinen, and S. K. Jackson (2005). Stupidmodel implementation source code. `http://condor.depaul.edu/%7Eslytinen/abm/StupidModel/`.

Railsback, S. F., S. L. Lytinen, and S. K. Jackson (2006, September). Agent-based simulation platforms: Review and development recommendations. *Simulation 82*(9), 609–623.

Robertson, D. A. (2005, December). Agent-based modeling toolkits netlogo, repast, and swarm. *Academy of Management Learning and Education 4*(4), 525.

Terna, P. (2009). Imaginary or actual artificial worlds using a new tool in the abm perspective. Working paper, University of Torino, Department of Economics and Public Finance, University of Torino, Torino, Italy. As presented at theOrganized Sessions of the NYC Computational Economics and Complexity Workshop of the Eastern Economic Association Meetings, 2009. `http://andromeda.rutgers.edu/%7Ejmbarr/EEA2009/terna.doc`.

Tobias, R. and C. Hofmann (2004). Evaluation of free java-libraries for social-scientific agent based simulation. *Journal of Artificial Societies and Social Simulation 7*(1), Article 6. `http://jasss.soc.surrey.ac.uk/7/1/6.html`.

Vaingast, S. (2009). *Beginning Python Visualization: Crafting Visual Transformation Scripts*. Books for Professionals by Professionals. New York, NY: Apress. ISBN: 978-1430218432.

van Rossum, G. (2003, May). Python main() functions. All Things Pythonic: A Weblog by Guido van van Rossum. `http://www.artima.com/weblogs/viewpost.jsp?thread=4829`.

Zelle, J. M. (2003). *Python Programming: An Introduction to Computer Science*. Franklin Beedle and Associates. ISBN 978-1887902991.

Notes

[1] We therefore assume readers have read an introduction to Python. See Kuhlman (2008) for a quick introduction, Isaac (2008) for a narrower but more applied introduction, or Zelle (2003) for a textbook introduction. We also refer to some basic NetLogo, which is quickly introduced in (Izquierdo, 2007). Note that we use the term 'function' to refer to callable subroutines of any type, even when a specific platform may adopt a different terminology.

[2] The original specification says that if an occupied location is selected, "then another new location is chosen". However, for the reference implementations, the phrase "another new location" was evidentally interpreted as "a new random draw from the entire neighborhood" rather than "a *new* (i.e., different) random draw from the neighborhood", so we are more explicit.

[3] 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. Even though Tkinter is extremely 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). We should also mention SLAPP, a recent Python implementation of the Swarm protocol (Terna, 2009). (The standard reference for the Swarm protocol has become (Langton, 1996).)

[4] The `gridworld` module is at `http://econpy.googlecode.com/svn/trunk/abm/gridworld/gridworld.py`. The agents provided by the `gridworld` module are based on the `Turtle` class of the `turtle` module, which is part of the Python standard library. Like NetLogo, but unlike most other ABM platforms, Python provides a basic agent class (in its `turtle` module) along with automatic visual display of the agent instances. It 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 to isolate the movement and positioning methods from the drawing methods, making it easy to implement agent-based models without visual displays. Since the template models emphasize visual display, we will not explore this further. However, readers may wish to examine the excellent online turtle.py documentation, especially the documentation of the motion methods.

[6] An overview of function definition can be found in the Python tutorial: `http://docs.python.org/tutorial/controlflow.html#defining-functions`

[6] We call the `shape` method to set the shape, the `shapesize` method to set the size of this shape, and the `fillcolor` method to set the fill-color of our agent. If we were working directly with a `turtle.Turtle`, we would also want to which ensures that an instance begins life with its pen up (so that it is not drawing as it moves) and its speed set to 0 (i.e., to "jump", so that it is moving at maximum speed). However a `gridworld.Agent` handles these initializations for us.

[7] For an introduction to class definition, see the tutorial discussion: `http://docs.python.org/tutorial/classes.html#a-first-look-at-classes`

[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. 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] For an extended discussion of this strategy, see (van Rossum, 2003).

[11] Inexplicably, the Netlogo reference implementations delay adding these color changes until template model 3.

[12] When appropriate, we would like to honor the DRY principle ("don't repeat yourself"), which provides an important rule for the production of maintainable code. A good discussion of the DRY principle can be found in (Hunt and Thomas, 1999). Note that NetLogo does not offer convenient mechanisms for class definition and reuse, although recent versions allow some reuse through file inclusion.

[13] Improved compactness of the code base may be achievable in NetLogo 4.1 by using the experimental `__includes` keyword. (For example, procedure definitions can be grouped together for inclusion.)

[14] 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.

[15] 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 uses 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.

[17] 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.

[18] Here we define `get_sizes` in the body of our `initialize` method, taking advantage of Python's support of closure, but we could alternatively have added a new `get_sizes` method to `World06`.

[19] Kahn (2007) notes that the Netlogo reference implementation erroneously stops at a `size` of 1000. This affects nothing of substance in this template model.

[20] 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 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.

[21] Note that the `stop` method of a `GridWorldGUI` 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 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.)

[22] We also suggest opening, appending to, and closing the file each iteration, so that a model crash does 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.

[23] A difficulty for new users will be learning to distinguish between `file-write`, `file-type`, and `file-print`.

[24] 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.

[25] 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 also provides `mean` as a built-in command.

[27] The original specification parameterized survival probability, which is the complement of our exit probability. Note that we have already addressed setting parameters in the GUI, so in parallel with the NetLogo reference implementation, we will not do that here.

[27] We do not need to 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 can just ignore it. Although the NetLogo reference implementation inexplicably uses a different stopping codition, but we do not alter the specification to match that.

[28] The NetLogo 4 user manual now clarifies this: only agents in the agent set at the time the `ask` command executes will run the commands.

[29] 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.

[30] Unfortunately, the NetLogo reference implementation of the fourteenth template model does provide guidance as it does not conform to the original 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.

[31] 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 minor 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.

[32] Note that Python can simply iterate over 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.

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

[34] For example, the NetLogo reference implementation first asks if there are hunters on a cell, and then asks if there are prey. Since hunting stops if a hunter is found (there is no identity testing), in that implementation, a hunter will never find a prey on its own cell. However the NetLogo implementation also errs in not including that cell in the list of neighbors, so a hunter will never search its own cell in any case.