Chapter 19

Agent-Based Models

19.1 What Are Agent-Based Models?

At last, we have reached the very final chapter, on *agent-based models (ABMs).* ABMs are arguably the most generalized framework for modeling and simulation of complex systems, which actually include both cellular automata and dynamical networks as special cases. ABMs are widely used in a variety of disciplines to simulate dynamical behaviors of systems made of a large number of entities, such as traders' behaviors in a market (in economics), migration of people (in social sciences), interaction among employees and their performance improvement (in organizational science), flocking/schooling behavior of birds/fish (in behavioral ecology), cell growth and morphogenesis (in developmental biology), and collective behavior of granular materials (in physics). Figure 19.1 shows a schematic illustration of an ABM.

It is a little challenging to define precisely what an agent-based model is, because its modeling assumptions are wide open, and thus there aren't many fundamental constraints that characterize ABMs. But here is what I hope to be a minimalistic definition of them:

Agent-based models are computational simulation models that involve many discrete *agents.*

There are a few keywords in this definition that are important to understand ABMs.

The first keyword is "computational." ABMs are usually implemented as simulation models in a computer, where each agent's behavioral rules are described in an algorithmic fashion rather than a purely mathematical way. This allows modelers to implement complex internal properties of agents and their nontrivial behavioral rules. Such representations of complex individual traits are highly valued especially in social, organizational

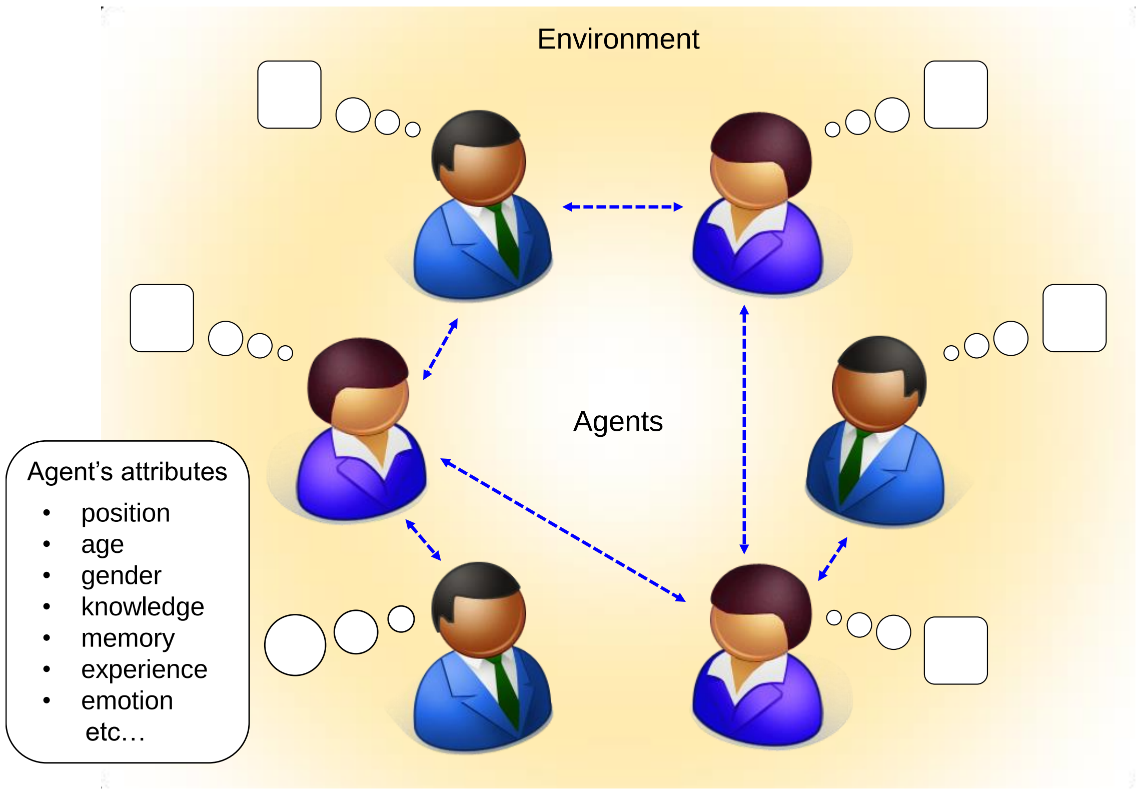

Figure 19.1: Schematic illustration of what an agent-based model (ABM) looks like.

and management sciences, where the modelers need to capture complex, realistic behaviors of human individuals. This is why ABMs are particularly popular in those research areas.

This, of course, comes at the cost of analytical tractability. Since agents can have any number of complex properties and behavioral rules, it is generally not easy to conduct an elegant mathematical analysis of an ABM (which is why there is no "Analysis" chapter on ABMs after this one). Therefore, the analysis of ABMs and their simulation results are usually carried out using more conventional statistical analysis commonly used in social sciences, e.g., by running Monte Carlo simulations to obtain distributions of outcome measurements under multiple experimental conditions, and then conducting statistical hypothesis testing to see if there was any significant difference between the different experimental conditions. In this sense, ABMs could serve as a virtual replacement of experimental fields for researchers.

The second keyword in the definition above is "many." Although it is technically possible to create an ABM made of just a few agents, there would be little need for such a model, because the typical context in which an ABM is needed is when researchers want to study the *collective behavior* of a large number of agents (otherwise it would be sufficient to use a more conventional equation-based model with a small number of variables). Therefore, typical ABMs contain a population of agents, just like cells in CA or nodes in dynamical networks, and their dynamical behaviors are studied using computational simulations.

The third keyword is "discrete." While there are some ambiguities about how to rigorously define an agent, what is commonly accepted is that an agent should be a discrete individual entity, which has a clear boundary between self and the outside. CA and network models are made of discrete components, so they qualify as special cases of ABMs. In the meantime, continuous field models adopt continuous spatial functions as a representation of the system's state, so they are not considered ABMs.

There are certain properties that are generally assumed in agents and ABMs, which collectively define the "agent-ness" of the entities in a model. Here is a list of such properties:

Typical properties generally assumed in agents and ABMs

- Agents are discrete entities.
- Agents may have internal states.
- Agents may be spatially localized.

- Agents may perceive and interact with the environment.
- Agents may behave based on predefined rules.
- Agents may be able to learn and adapt.
- Agents may interact with other agents.
- ABMs often lack central supervisors/controllers.
- ABMs may produce nontrivial "collective behavior" as a whole.

Note that these are not strict requirements for ABMs (perhaps except for the first one). Some ABMs don't have internal states of agents; some don't have space or environment; and some *do* have central controllers as special kinds of agents. Therefore, which model properties should be incorporated into an ABM is really up to the objective of your model.

Before we move on to actual ABM building, I would like to point out that there are a few things we need to be particularly careful about when we build ABMs. One is about coding. Implementing an ABM is usually much more coding-intense than implementing other simpler models, partly because the ABM framework is so open-ended. The fact that there aren't many constraints on ABMs also means that you have to take care of all the details of the simulation yourself. This naturally increases the amount of coding you will need to do. And, the more you code, the more likely an unexpected bug or two will sneak into your code. It is thus very important to keep your code simple and organized, and to use the best practices in computer programming (e.g., modularizing sections, adding plenty of comments, implementing systematic tests, etc.), in order to minimize the risks of having bugs in your simulation. If possible, it is desirable to have multiple people test and thoroughly check your code.

Another issue we should be aware of is that, since ABMs are so open-ended and flexible, modelers are tempted to add more and more complex settings and assumptions into their ABMs. This is understandable, as ABMs are such nice playgrounds to try testing "what-if" scenarios in a virtual world, and the results can be obtained quickly. I have seen many people who became fascinated by such an interactive modeling experience and tried adding more and more details into their own ABMs to make them more "realistic." But beware—the increased model complexity means the increased difficulty of analyzing and justifying the model and its results. If we want to derive a useful, reliable conclusion from our model, we should resist the temptation to unnecessarily add complexity to our ABMs. We need to strike the right balance between simplicity, validity, and robustness, as discussed in Section 2.4.

$\boxed{\textit{Exercise 19.1}}$ Do a quick online literature search to learn how ABMs are used in various scientific disciplines. Choose a few examples of your interest and learn more about how researchers developed and used their ABMs for their research.

19.2 Building an Agent-Based Model

Let's get started with agent-based modeling. In fact, there are many great tutorials already out there about how to build an ABM, especially those by Charles Macal and Michael North, renowned agent-based modelers at Argonne National Laboratory [84]. Macal and North suggest considering the following aspects when you design an agent-based model:

1. Specific problem to be solved by the ABM
2. Design of agents and their static/dynamic attributes
3. Design of an environment and the way agents interact with it
4. Design of agents' behaviors
5. Design of agents' mutual interactions
6. Availability of data
7. Method of model validation

Among those points, 1, 6, and 7 are about fundamental scientific methodologies. It is important to keep in mind that just building an arbitrary ABM and obtaining results by simulation wouldn't produce any scientifically meaningful conclusion. In order for an ABM to be scientifically meaningful, it has to be built and used in either of the following two complementary approaches:

A. Build an ABM using model assumptions that are derived from empirically observed phenomena, and then produce previously unknown collective behaviors by simulation.

B. Build an ABM using hypothetical model assumptions, and then reproduce empirically observed collective phenomena by simulation.

The former is to use ABMs to make predictions using validated theories of agent behaviors, while the latter is to explore and develop new explanations of empirically observed phenomena. These two approaches are different in terms of the scales of the known and the unknown (A uses micro-known to produce macro-unknown, while B uses micro-unknown to reproduce macro-known), but the important thing is that one of those scales should be grounded on well-established empirical knowledge. Otherwise, the simulation results would have no implications for the real-world system being modeled. Of course, a free exploration of various collective dynamics by testing hypothetical agent behaviors to generate hypothetical outcomes is quite fun and educational, with lots of intellectual benefits of its own. My point is that we shouldn't misinterpret outcomes obtained from such exploratory ABMs as a validated prediction of reality.

In the meantime, items 2, 3, 4, and 5 in Macal and North's list above are more focused on the technical aspects of modeling. They can be translated into the following design tasks in actual coding using a programming language like Python:

Design tasks you need to do when you implement an ABM

1. Design the data structure to store the attributes of the agents.
2. Design the data structure to store the states of the environment.
3. Describe the rules for how the environment behaves on its own.
4. Describe the rules for how agents interact with the environment.
5. Describe the rules for how agents behave on their own.
6. Describe the rules for how agents interact with each other.

Representation of agents in Python It is often convenient and customary to define both agents' attributes and behaviors using a *class* in object-oriented programming languages, but in this textbook, we won't cover object-oriented programming in much detail. Instead, we will use Python's dynamic class as a pure data structure to store agents' attributes in a concise manner. For example, we can define an empty `agent` class as follows:

Code 19.1:

```
class agent:
    pass
```

The class command defines a new class under which you can define various attributes (variables, properties) and methods (functions, actions). In conventional object-oriented programming, you need to give more specific definitions of attributes and methods available under this class. But in this textbook, we will be a bit lazy and exploit the dynamic, flexible nature of Python's classes. Therefore, we just threw pass into the class definition above. pass is a dummy keyword that doesn't do anything, but we still need something there just for syntactic reasons.

Anyway, once this agent class is defined, you can create a new empty agent object as follows:

Code 19.2:

```
>>> a = agent()
```

Then you can dynamically add various attributes to this agent object a:

Code 19.3:

```
>>> a.x = 2
>>> a.y = 8
>>> a.name = 'John'
>>> a.age = 21
>>> a.x
2
>>> a.name
'John'
```

This flexibility is very similar to the flexibility of Python's dictionary. You don't have to predefine attributes of a Python object. As you assign a value to an attribute (written as "object's name"."attribute"), Python automatically generates a new attribute if it hasn't been defined before. If you want to know what kinds of attributes are available under an object, you can use the dir command:

Code 19.4:

```
>>> dir(a)
['__doc__', '__module__', 'age', 'name', 'x', 'y']
```

The first two attributes are Python's default attributes that are available for any objects (you can ignore them for now). Aside from those, we see there are four attributes defined for this object a.

In the rest of this chapter, we will use this class-based agent representation. The technical architecture of simulator codes is still the same as before, made of three components: initialization, visualization, and updating functions. Let's work on some examples to see how you can build an ABM in Python.

Example: Schelling's segregation model There is a perfect model for our first ABM exercise. It is called *Schelling's segregation model*, widely known as the very first ABM proposed in the early 1970s by Thomas Schelling, the 2005 Nobel Laureate in Economics [85]. Schelling created this model in order to provide an explanation for why people with different ethnic backgrounds tend to segregate geographically. Therefore, this model was developed in approach B discussed above, reproducing the macro-known by using hypothetical micro-unknowns. The model assumptions Schelling used were the following:

- Two different types of agents are distributed in a finite 2-D space.

- In each iteration, a randomly chosen agent looks around its neighborhood, and if the fraction of agents of the same type among its neighbors is below a threshold, it jumps to another location randomly chosen in the space.

As you can see, the rule of this model is extremely simple. The main question Schelling addressed with this model was how high the threshold had to be in order for segregation to occur. It may sound reasonable to assume that segregation would require highly homophilic agents, so the critical threshold might be relatively high, say 80% or so. But what Schelling actually showed was that the critical threshold can be much lower than one would expect. This means that segregation can occur even if people aren't so homophilic. In the meantime, quite contrary to our intuition, a high level of homophily can actually result in a mixed state of the society because agents keep moving without reaching a stationary state. We can observe these emergent behaviors in simulations.

Back in the early 1970s, Schelling simulated his model on graph paper using pennies and nickels as two types of agents (this was still a perfectly computational simulation!). But here, we can loosen the spatial constraints and simulate this model in a continuous space. Let's design the simulation model step by step, going through the design tasks listed above.

1. Design the data structure to store the attributes of the agents. In this model, each agent has a type attribute as well as a position in the 2-D space. The two types can be represented by 0 and 1, and the spatial position can be anywhere within a unit square. Therefore we can generate each agent as follows:

Code 19.5:

```
class agent:
    pass

ag = agent()
ag.type = randint(2)
ag.x = random()
ag.y = random()
```

To generate a population of agents, we can write something like:

Code 19.6:

```
n = 1000 # number of agents

class agent:
    pass

def initialize():
    global agents
    agents = []
    for i in xrange(n):
        ag = agent()
        ag.type = randint(2)
        ag.x = random()
        ag.y = random()
        agents.append(ag)
```

2. Design the data structure to store the states of the environment, *3. Describe the rules for how the environment behaves on its own,* & *4. Describe the rules for how agents interact with the environment.* Schelling's model doesn't have a separate environment that interacts with agents, so we can skip these design tasks.

5. Describe the rules for how agents behave on their own. We assume that agents don't do anything by themselves, because their actions (movements) are triggered only by interactions with other agents. So we can ignore this design task too.

6. Describe the rules for how agents interact with each other. Finally, there is something we need to implement. The model assumption says each agent checks who are in its neighborhood, and if the fraction of the other agents of the same type is less than a threshold, it jumps to another randomly selected location. This requires *neighbor detec-*

tion, which was easy in CA and networks because the neighborhood relationships were explicitly modeled in those modeling frameworks. But in ABM, neighborhood relationships may be implicit, which is the case for our model. Therefore we need to implement a code that allows each agent to find who is nearby.

There are several computationally efficient algorithms available for neighbor detection, but here we use the simplest possible method: Exhaustive search. You literally check all the agents, one by one, to see if they are close enough to the focal agent. This is not computationally efficient (its computational amount increases quadratically as the number of agents increases), but is very straightforward and extremely easy to implement. You can write such exhaustive neighbor detection using Python's list comprehension, e.g.:

Code 19.7:

```
neighbors = [nb for nb in agents
             if (ag.x - nb.x)**2 + (ag.y - nb.y)**2 < r**2 and nb != ag]
```

Here `ag` is the focal agent whose neighbors are searched for. The `if` part in the list comprehension measures the distance squared between `ag` and `nb`, and if it is less than `r` squared (`r` is the neighborhood radius, which must be defined earlier in the code) `nb` is included in the result. Also note that an additional condition `nb != ag` is given in the `if` part. This is because if `nb == ag`, the distance is always 0 so `ag` itself would be mistakenly included as a neighbor of `ag`.

Once we obtain `neighbors` for `ag`, we can calculate the fraction of the other agents whose type is the same as `ag`'s, and if it is less than a given threshold, `ag`'s position is randomly reset. Below is the completed simulator code, with the visualization function also implemented using the simple `plot` function:

Code 19.8: segregation.py

```
import matplotlib
matplotlib.use('TkAgg')
from pylab import *

n = 1000 # number of agents
r = 0.1 # neighborhood radius
th = 0.5 # threshold for moving

class agent:
    pass
```

```
def initialize():
    global agents
    agents = []
    for i in xrange(n):
        ag = agent()
        ag.type = randint(2)
        ag.x = random()
        ag.y = random()
        agents.append(ag)

def observe():
    global agents
    cla()
    white = [ag for ag in agents if ag.type == 0]
    black = [ag for ag in agents if ag.type == 1]
    plot([ag.x for ag in white], [ag.y for ag in white], 'wo')
    plot([ag.x for ag in black], [ag.y for ag in black], 'ko')
    axis('image')
    axis([0, 1, 0, 1])

def update():
    global agents
    ag = agents[randint(n)]
    neighbors = [nb for nb in agents
                 if (ag.x - nb.x)**2 + (ag.y - nb.y)**2 < r**2 and nb != ag]
    if len(neighbors) > 0:
        q = len([nb for nb in neighbors if nb.type == ag.type]) \
            / float(len(neighbors))
        if q < th:
            ag.x, ag.y = random(), random()

import pycxsimulator
pycxsimulator.GUI().start(func=[initialize, observe, update])
```

When you run this code, you should set the step size to 50 under the “Settings” tab to speed up the simulations.

Figure 19.2 shows the result with the neighborhood radius `r=0.1` and the threshold for moving `th = 0.5`. It is clearly observed that the agents self-organize from an initially random distribution to a patchy pattern where the two types are clearly segregated from each other.

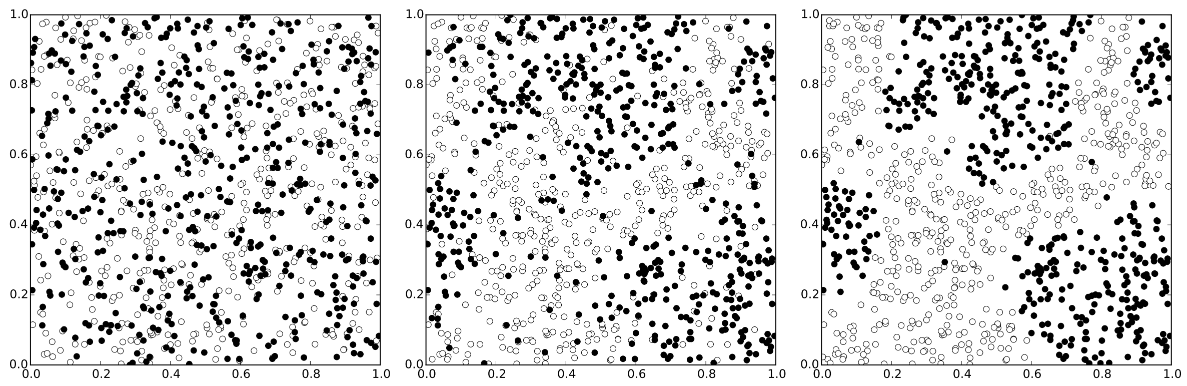

Figure 19.2: Visual output of Code 19.8. Time flows from left to right.

Exercise 19.2 Conduct simulations of Schelling's segregation model with `th` (threshold for moving), `r` (neighborhood radius), and/or `n` (population size = density) varied systematically. Determine the condition in which segregation occurs. Is the transition gradual or sharp?

Exercise 19.3 Develop a metric that characterizes the level of segregation from the positions of the two types of agents. Then plot how this metric changes as parameter values are varied.

Here are some other well-known models that show quite unique emergent patterns or dynamic behaviors. They can be implemented as an ABM by modifying the code for Schelling's segregation model. Have fun!

Exercise 19.4 **Diffusion-limited aggregation** *Diffusion-limited aggregation (DLA)* is a growth process of clusters of aggregated particles driven by their random diffusion. There are two types of particles, like in Schelling's segregation model, but only one of them can move freely. The movable particles diffuse in a 2-D space by random walk, while the immovable particles do nothing; they just remain where they are. If a movable particle "collides" with an immovable particle (i.e., if they come close enough to each other), the movable particle becomes immovable and stays there forever. This is the only rule for the agents' interaction.

Implement the simulator code of the DLA model. Conduct a simulation with all particles initially movable, except for one immovable "seed" particle placed at the center of the space, and observe what kind of spatial pattern emerges. Also carry out the simulations with multiple immovable seeds randomly positioned in the space, and observe how multiple clusters interact with each other at macroscopic scales.

For your information, a completed Python simulator code of the DLA model is available from `http://sourceforge.net/projects/pycx/files/`, but you should try implementing your own simulator code first.

Exercise 19.5 **Boids** This is a fairly advanced, challenging exercise about the collective behavior of animal groups, such as bird flocks, fish schools, and insect swarms, which is a popular subject of research that has been extensively modeled and studied with ABMs. One of the earliest computational models of such collective behaviors was proposed by computer scientist Craig Reynolds in the late 1980s [86]. Reynolds came up with a set of simple behavioral rules for agents moving in a continuous space that can reproduce an amazingly natural-looking flock behavior of birds. His model was called bird-oids, or *"Boids"* for short. Algorithms used in Boids has been utilized extensively in the computer graphics industry to automatically generate animations of natural movements of animals in flocks (e.g., bats in the Batman movies). Boids' dynamics are generated by the following three essential behavioral rules (Fig. 19.3):

Cohesion Agents tend to steer toward the center of mass of local neighbors.

Alignment Agents tend to steer to align their directions with the average velocity of local neighbors.

Separation Agents try to avoid collisions with local neighbors.

Design an ABM of collective behavior with these three rules, and implement its simulator code. Conduct simulations by systematically varying relative strengths of the three rules above, and see how the collective behavior changes.

You can also simulate the collective behavior of a population in which multiple types of agents are mixed together. It is known that interactions among kinetically distinct types of swarming agents can produce various nontrivial dynamic patterns [87].

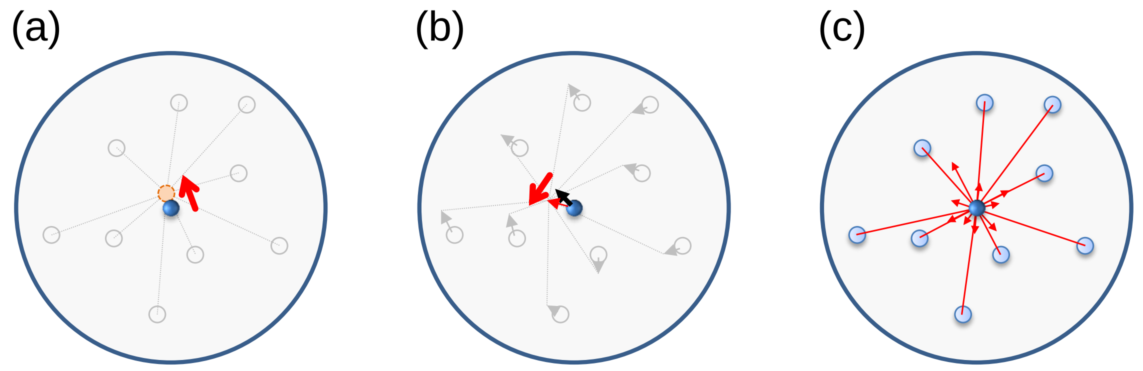

Figure 19.3: Three essential behavioral rules of Boids. (a) Cohesion. (b) Alignment. (c) Separation.

19.3 Agent-Environment Interaction

One important component you should consider adding to your ABM is the interaction between agents and their environment. The environmental state is still part of the system's overall state, but it is defined over space, and not associated with specific agents. The environmental state dynamically changes either spontaneously or by agents' actions (or both). The examples discussed so far (Schelling's segregation model, DLA, Boids) did not include such an environment, but many ABMs explicitly represent environments that agents act on and interact with. The importance of agent-environment interaction is well illustrated by the fact that NetLogo [13], a popular ABM platform, uses "turtles" and "patches" by default, to represent agents and the environment, respectively. We can do the same in Python.

A good example of such agent-environment interaction is in the *Keller-Segel slime mold aggregation model* we discussed in Section 13.4, where slime mold cells behave as agents and interact with an environment made of cAMP molecules. The concentration of cAMP is defined everywhere in the space, and it changes by its own inherent dynamics (natural decay) and by the actions of agents (secretion of cAMP molecules by agents). This model can be implemented as an ABM, designed step by step as follows:

1. Design the data structure to store the attributes of the agents. If the slime mold cells are represented by individual agents, their concentration in the original Keller-Segel model is represented by the density of agents, so they will no longer have any attributes other than spatial position in the 2-D space. Therefore `x` and `y` are the only attributes of agents in this model.

2. Design the data structure to store the states of the environment. The environment in this model is the spatial function that represents the concentration of cAMP molecules at each location. We can represent this environment by discretizing space and assigning a value to each discrete spatial cell, just like we did for numerical simulations of PDEs. We can use the `array` data structure for this purpose.

Here is a sample `initialize` part that sets up the data structures for both agents and the environment. Note that we prepare two arrays, `env` and `nextenv`, for simulating the dynamics of the environment.

Code 19.9:

```
n = 1000 # number of agents
w = 100 # number of rows/columns in spatial array

class agent:
    pass

def initialize():
    global agents, env, nextenv

    agents = []
    for i in xrange(n):
        ag = agent()
        ag.x = randint(w)
        ag.y = randint(w)
        agents.append(ag)

    env = zeros([w, w])
    nextenv = zeros([w, w])
```

3. Describe the rules for how the environment behaves on its own. The inherent dynamics of the environment in this model are the diffusion and spontaneous decay of the cAMP concentration. These can be modeled by using the discrete version of the Laplacian operator, as well as an exponential decay factor, applied to the environmental states everywhere in the space in every iteration. This is no different from what we did for the numerical simulations of PDEs. We can implement them in the code as follows:

Code 19.10:

```
k = 1 # rate of cAMP decay
Dc = 0.001 # diffusion constant of cAMP
Dh = 0.01 # spatial resolution for cAMP simulation
Dt = 0.01 # time resolution for cAMP simulation

def update():
    global agents, env, nextenv

    # simulating diffusion and evaporation of cAMP
    for x in xrange(w):
        for y in xrange(w):
            C, R, L, U, D = env[x,y], env[(x+1)%w,y], env[(x-1)%w,y], \
                            env[x,(y+1)%w], env[x,(y-1)%w]
            lap = (R + L + U + D - 4 * C)/(Dh**2)
            nextenv[x,y] = env[x,y] + (- k * C + Dc * lap) * Dt
    env, nextenv = nextenv, env
```

Here we adopt periodic boundary conditions for simplicity.

4. Describe the rules for how agents interact with the environment. In this model, agents interact with the environment in two different ways. One way is the secretion of cAMP by the agents, which can be implemented by letting each agent increase the cAMP concentration in a discrete cell where it is located. To do this, we can add the following to the `update` function:

Code 19.11:

```
    f = 1 # rate of cAMP secretion by an agent

    # simulating secretion of cAMP by agents
    for ag in agents:
        env[ag.x, ag.y] += f * Dt
```

The other way is the chemotaxis, which can be implemented in several different ways. For example, we can have each agent look at a cell randomly chosen from its neighborhood, and move there with a probability determined by the difference in cAMP concentration (Δc) between the neighbor cell and the cell where the agent is currently located. A *sigmoid function*

$$P(\Delta c) = \frac{e^{\Delta c/c_0}}{1 + e^{\Delta c/c_0}} \tag{19.1}$$

would be suitable for this purpose, where c_0 is a parameter that determines how sensitive this probability is to Δc. $P(\Delta c)$ approaches 1 with $\Delta c \to \infty$, or 0 with $\Delta c \to -\infty$. This part can be implemented in the `update` function as follows:

Code 19.12:

```
    # simulating chemotaxis of agents
    for ag in agents:
        newx, newy = (ag.x + randint(-1, 2)) % w, (ag.y + randint(-1, 2)) % w
        diff = (env[newx, newy] - env[ag.x, ag.y]) / 0.1
        if random() < exp(diff) / (1 + exp(diff)):
            ag.x, ag.y = newx, newy
```

Here `diff` corresponds to Δc, and we used $c_0 = 0.1$.

5. Describe the rules for how agents behave on their own. All the actions taken by the agents in this model are interactions with the environment, so we can skip this design task.

6. Describe the rules for how agents interact with each other. In this model, agents don't interact with each other directly; all interactions among them are indirect, mediated by environmental variables (cAMP concentration in this case), so there is no need to implement anything for the agents' direct interaction with each other. Indirect agent-agent interaction through informational signals written in the environment is called *stigmergy*, which is an essential coordination mechanism used by many social organisms [88].

By putting everything together, and adding the `observe` function for visualization, the entire simulator code looks like this:

Code 19.13: keller-segel-abm.py

```
import matplotlib
matplotlib.use('TkAgg')
from pylab import *
```

```
n = 1000 # number of agents
w = 100 # number of rows/columns in spatial array

k = 1 # rate of cAMP decay
Dc = 0.001 # diffusion constant of cAMP
Dh = 0.01 # spatial resolution for cAMP simulation
Dt = 0.01 # time resolution for cAMP simulation

f = 1 # rate of cAMP secretion by an agent

class agent:
    pass

def initialize():
    global agents, env, nextenv

    agents = []
    for i in xrange(n):
        ag = agent()
        ag.x = randint(w)
        ag.y = randint(w)
        agents.append(ag)

    env = zeros([w, w])
    nextenv = zeros([w, w])

def observe():
    global agents, env, nextenv
    cla()
    imshow(env, cmap = cm.binary, vmin = 0, vmax = 1)
    axis('image')
    x = [ag.x for ag in agents]
    y = [ag.y for ag in agents]
    plot(y, x, 'b.') # x and y are swapped to match the orientation of env

def update():
    global agents, env, nextenv
    # simulating diffusion and evaporation of cAMP
    for x in xrange(w):
        for y in xrange(w):
            C, R, L, U, D = env[x,y], env[(x+1)%w,y], env[(x-1)%w,y], \
                            env[x,(y+1)%w], env[x,(y-1)%w]
            lap = (R + L + U + D - 4 * C)/(Dh**2)
            nextenv[x,y] = env[x,y] + (- k * C + Dc * lap) * Dt
    env, nextenv = nextenv, env

    # simulating secretion of cAMP by agents
    for ag in agents:
        env[ag.x, ag.y] += f * Dt

    # simulating chemotaxis of agents
    for ag in agents:
        newx, newy = (ag.x + randint(-1, 2)) % w, (ag.y + randint(-1, 2)) % w
        diff = (env[newx, newy] - env[ag.x, ag.y]) / 0.1
        if random() < exp(diff) / (1 + exp(diff)):
            ag.x, ag.y = newx, newy

import pycxsimulator
pycxsimulator.GUI().start(func=[initialize, observe, update])
```

As you see, the code is getting longer and more complicated than before, which is a typical consequence when you implement an ABM.

Figure 19.4 shows a typical simulation result. The tiny blue dots represent individual cells (agents), while the grayscale shades on the background show the cAMP concentration (environment). As time goes by, the slightly more populated areas produce more cAMP, attracting more cells. Eventually, several distinct peaks of agent populations are spontaneously formed, reproducing self-organizing patterns that are similar to what the PDE-based model produced.

What is unique about this ABM version of the Keller-Segel model, compared to its original PDE-based version, is that the simulation result looks more "natural"—the formation of the population peaks are not simultaneous, but they gradually appear one after another, and the spatial arrangements of those peaks are not regular. In contrast, in the PDE-based version, the spots are formed simultaneously at regular intervals (see

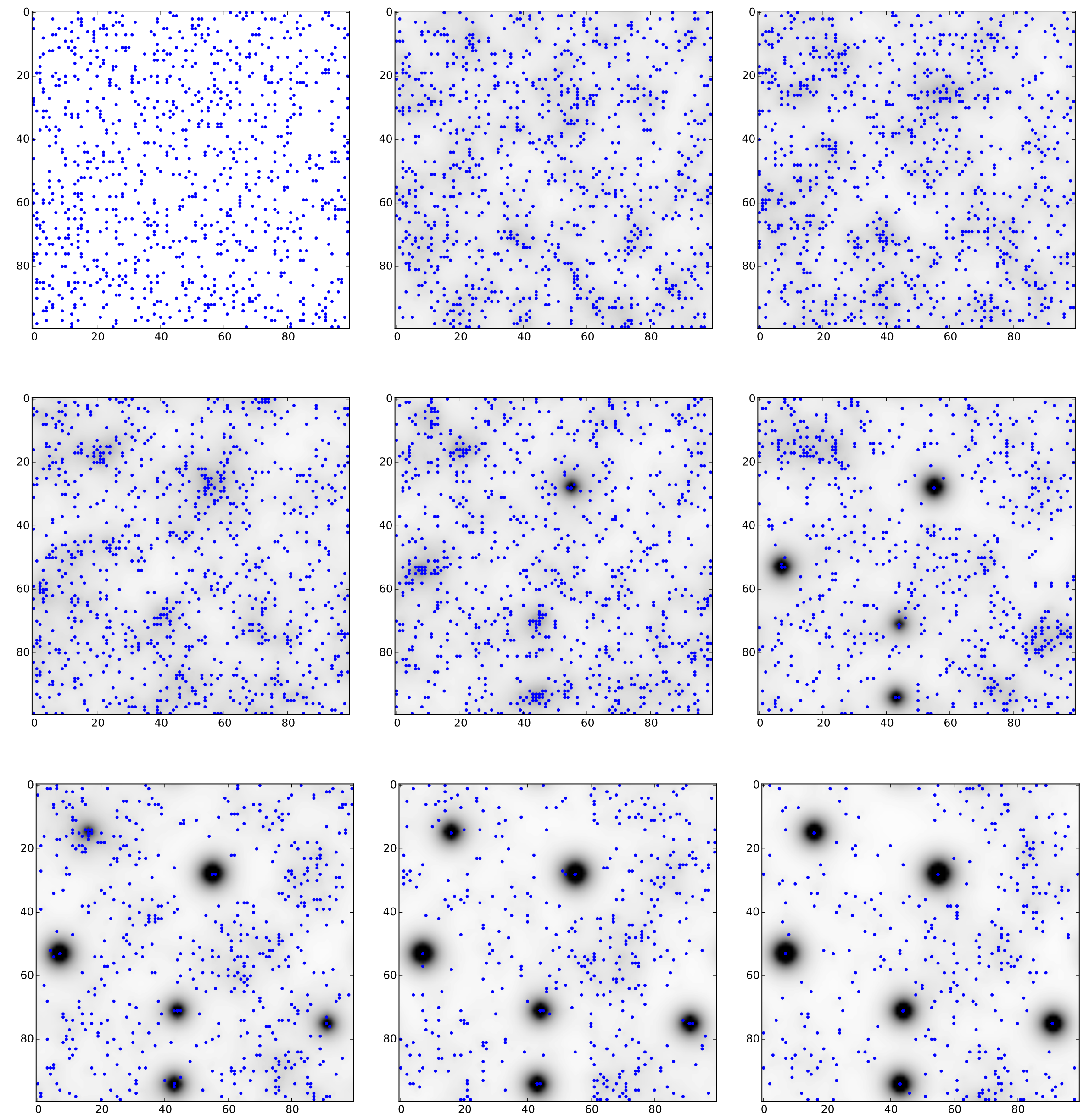

Figure 19.4: Visual output of Code 19.13. Time flows from left to right, then from top to bottom.

Fig. 13.12). Such "natural-lookingness" of the results obtained with ABMs comes from the fact that ABMs are based on the behaviors of discrete individual agents, which often involve a lot of realistic uncertainty.

Exercise 19.6 Conduct simulations of the ABM version of the Keller-Segel model with `k`, `Dc`, and `f` varied systematically. Are the effects of those parameters on pattern formation similar to those in the PDE version of the same model (see Eq. (14.74))?

Exercise 19.7 Implement an additional mechanism into the ABM above to prevent agents from aggregating too densely at a single location, so that the population density doesn't exceed a predetermined upper limit anywhere in the space. Conduct simulations to see what kind of effect this additional model assumption has on pattern formation.

Exercise 19.8 **Garbage collection by ants** This is another interesting ABM with an agent-environment interaction, which was first presented by Mitchel Resnick in his famous book "Turtles, Termites and Traffic Jams" [89]. Assume there are many tiny pieces of garbage scattered in a 2-D space, where many ants are wandering randomly. When an ant comes to a place where there is some garbage, it behaves according to the following very simple rules:

1. If the ant is holding a piece of garbage, it drops the piece.
2. If the ant isn't holding any garbage, it picks up a piece of garbage.

What would result from these rules? Are the garbage pieces going to be scattered more and more due to these brainless insects? Implement an ABM of this model and conduct simulations to find out what kind of collective behavior emerges.

If you implement the model right, you will see that these very simple behavioral rules let the ants spontaneously collect and pile up garbage and clear up the space in the long run. This model tells us how such emergent behavior of the collective is sometimes counter to our natural intuition.

19.4 Ecological and Evolutionary Models

In this very final section of this textbook, we will discuss ABMs of ecological and evolutionary dynamics. Such ABMs are different from the other examples discussed so far in this chapter regarding one important aspect: Agents can be born and can die during a simulation. This means that the number of state variables involved in a system can change dynamically over time, so the traditional concepts of dynamical systems don't apply easily to those systems. You may remember that we saw a similar challenge when we made a transition from dynamics *on* networks to dynamics *of* networks in Chapter 16. A dynamic increase or decrease in the number of system components violates the assumption that the system's behavior can be represented as a trajectory within a static phase space. In order to study the behaviors of such systems, the most general, practical approach would probably be to conduct explicit simulations on computers.

Simulating an ABM with a varying number of agents requires special care for simulating births and deaths of agents. Because agents can be added to or removed from the system at any time during an updating process, it is a little tricky to implement synchronous updating of agents (though it isn't impossible). It is much simpler and more straightforward to update the system's state in an asynchronous manner, by randomly choosing an agent to update its state and, if needed, directly remove it from the system (simulation of death) or add a new agent to the system (simulation of birth). In what follows, we will adopt this asynchronous updating scheme for the simulation of ecological ABMs.

An illustrative example of ecological ABMs is the predator-prey ecosystem, which we already discussed in Section 4.6 and on several other occasions. The basic idea of this model is still simple: Prey naturally grow but get eaten by predators, while the predators grow if they get prey but otherwise naturally die off. When we are to implement this model as an ABM, these ecological dynamics should be described at an individual agent level, not at an aggregated population level. Let's design this ABM step by step, again going through the six design tasks.

1. Design the data structure to store the attributes of the agents. The predator-prey ecosystem is obviously made of two types of agents: prey and predators. So the information about agent type must be represented in the data structure. Also, if we are to simulate their interactions in a space, the information about their spatial location is also needed. Note that these attributes are identical to those of the agents in Schelling's segregation model, so we can use the same agent design, as follows:

Code 19.14:

```
r_init = 100 # initial rabbit population
f_init = 30 # initial fox population

class agent:
    pass

def initialize():
    global agents
    agents = []
    for i in xrange(r_init + f_init):
        ag = agent()
        ag.type = 'r' if i < r_init else 'f'
        ag.x = random()
        ag.y = random()
        agents.append(ag)
```

Here, we call prey “rabbits” and predators “foxes” in the code, so we can denote them by `r` and `f`, respectively (as both prey and predators begin with “pre”!). Also, we use `r_init` and `f_init` to represent the initial population of each species. The `for` loop iterates `r_init + f_init` times, and in the first `r_init` iteration, the prey agents are generated, while the predator agents are generated for the rest.

2. Design the data structure to store the states of the environment, 3. Describe the rules for how the environment behaves on its own, & *4. Describe the rules for how agents interact with the environment.* This ABM doesn't involve an environment explicitly, so we can ignore these design tasks.

5. Describe the rules for how agents behave on their own, & *6. Describe the rules for how agents interact with each other.* In this model, the agents' inherent behaviors and interactions are somewhat intertwined, so we will discuss these two design tasks together. Different rules are to be designed for prey and predator agents, as follows.

For prey agents, each individual agent reproduces at a certain reproduction rate. In equation-based models, it was possible to allow a population to grow exponentially, but in ABMs, exponential growth means exponential increase of memory use because each agent physically takes a certain amount of memory space in your computer. Therefore we need to prevent such growth of memory use by applying a logistic-type growth restriction. In the meantime, if a prey agent meets a predator agent, it dies with some probability because of predation. Death can be implemented simply as the removal of the agent from the `agents` list.

For predator agents, the rules are somewhat opposite. If a predator agent can't find any prey agent nearby, it dies with some probability because of the lack of food. But if it can consume prey, it can also reproduce at a certain reproduction rate.

Finally, both types of agents diffuse in space by random walk. The rates of diffusion can be different between the two species, so let's assume that predators can diffuse a little faster than prey.

The assumptions designed above can be implemented altogether in the `update` function as follows. As you can see, the code is getting a bit longer than before, which reflects the increased complexity of the agents' behavioral rules:

Code 19.15:

```
import copy as cp

nr = 500. # carrying capacity of rabbits

mr = 0.03 # magnitude of movement of rabbits
dr = 1.0 # death rate of rabbits when it faces foxes
rr = 0.1 # reproduction rate of rabbits

mf = 0.05 # magnitude of movement of foxes
df = 0.1 # death rate of foxes when there is no food
rf = 0.5 # reproduction rate of foxes

cd = 0.02 # radius for collision detection
cdsq = cd ** 2

def update():
    global agents
    if agents == []:
        return

    ag = agents[randint(len(agents))]

    # simulating random movement
    m = mr if ag.type == 'r' else mf
    ag.x += uniform(-m, m)
    ag.y += uniform(-m, m)
    ag.x = 1 if ag.x > 1 else 0 if ag.x < 0 else ag.x
    ag.y = 1 if ag.y > 1 else 0 if ag.y < 0 else ag.y

    # detecting collision and simulating death or birth
    neighbors = [nb for nb in agents if nb.type != ag.type
                and (ag.x - nb.x)**2 + (ag.y - nb.y)**2 < cdsq]

    if ag.type == 'r':
        if len(neighbors) > 0: # if there are foxes nearby
            if random() < dr:
                agents.remove(ag)
                return
        if random() < rr*(1-sum(1 for x in agents if x.type == 'r')/nr):
            agents.append(cp.copy(ag))
    else:
        if len(neighbors) == 0: # if there are no rabbits nearby
            if random() < df:
                agents.remove(ag)
                return
        else: # if there are rabbits nearby
            if random() < rf:
                agents.append(cp.copy(ag))
```

Here, `ag` is the agent randomly selected for asynchronous updating. Python's `copy` module is used to create a copy of each agent as offspring when it reproduces. Note that the logistic-type growth restriction is implemented for prey agents by multiplying the reproduction probability by $(1 - x/n_r)$, where x is the current population of prey and n_r is the carrying capacity. Also note that at the very beginning of this function, it is checked whether there is any agent in the `agents` list. This is because there is a possibility for all agents to die out in ecological ABMs.

Now, I would like to bring up one subtle issue that arises in ABMs with a varying number of agents that are simulated asynchronously. When the number of agents was fixed and constant, the length of elapsed time in the simulated world was linearly proportional to the number of executions of the asynchronous `update` function (e.g., in Schelling's segregation model), so we didn't have to do anything special to handle the flow of time. However, when the number of agents varies, an execution of the asynchronous `update` function on one randomly selected agent doesn't always represent the same amount of elapsed time in the simulated world. To better understand this issue, imagine two different situations: Simulating 10 agents and simulating 1,000 agents. In the former situation, each agent is updated once, on average, when the `update` function is executed 10 times. However, in the latter situation, 10 times of execution of the function means only about 1% of the agents being updated. But in the simulated world, agents should be behaving concurrently in parallel, so each agent should be updated once, on average, in one unit length of simulated time. This implies that the elapsed time per each asynchronous updating should depend on the size of the agent population. How can we cope with this additional complication of the simulation?

A quick and easy solution to this issue is to assume that, in each asynchronous updating, $1/n$ of a unit length of time passes by, where n is the size of the agent population at the time of updating. This method can naturally handle situations where the size of the agent population changes rapidly, and it (almost) guarantees that each agent is updated once, on average, in each unit time length. To implement a simulation for one unit length of time, we can write the following "wrapper" function to make sure that the time in the simulated world elapses by one unit length:

Code 19.16:

```
def update_one_unit_time():
    global agents
    t = 0.
    while t < 1.:
        t += 1. / len(agents)
        update()

import pycxsimulator
pycxsimulator.GUI().start(func=[initialize, observe, update_one_unit_time])
```

This trick realizes that the progress of time appears steady in the simulation, even if the number of agents changes over time.

Okay, now we are basically done. Putting everything together and adding the visualization function, the entire simulator code looks like this:

Code 19.17: predator-prey-abm.py

```
import matplotlib
matplotlib.use('TkAgg')
from pylab import *
import copy as cp
```

```
nr = 500. # carrying capacity of rabbits

r_init = 100 # initial rabbit population
mr = 0.03 # magnitude of movement of rabbits
dr = 1.0 # death rate of rabbits when it faces foxes
rr = 0.1 # reproduction rate of rabbits

f_init = 30 # initial fox population
mf = 0.05 # magnitude of movement of foxes
df = 0.1 # death rate of foxes when there is no food
rf = 0.5 # reproduction rate of foxes

cd = 0.02 # radius for collision detection
cdsq = cd ** 2

class agent:
    pass

def initialize():
    global agents
    agents = []
    for i in xrange(r_init + f_init):
        ag = agent()
        ag.type = 'r' if i < r_init else 'f'
        ag.x = random()
        ag.y = random()
        agents.append(ag)

def observe():
    global agents
    cla()
    rabbits = [ag for ag in agents if ag.type == 'r']
    if len(rabbits) > 0:
        x = [ag.x for ag in rabbits]
        y = [ag.y for ag in rabbits]
        plot(x, y, 'b.')
    foxes = [ag for ag in agents if ag.type == 'f']
    if len(foxes) > 0:
        x = [ag.x for ag in foxes]
        y = [ag.y for ag in foxes]
        plot(x, y, 'ro')
    axis('image')
    axis([0, 1, 0, 1])

def update():
    global agents
    if agents == []:
        return

    ag = agents[randint(len(agents))]

    # simulating random movement
    m = mr if ag.type == 'r' else mf
    ag.x += uniform(-m, m)
    ag.y += uniform(-m, m)
    ag.x = 1 if ag.x > 1 else 0 if ag.x < 0 else ag.x
    ag.y = 1 if ag.y > 1 else 0 if ag.y < 0 else ag.y

    # detecting collision and simulating death or birth
    neighbors = [nb for nb in agents if nb.type != ag.type
                 and (ag.x - nb.x)**2 + (ag.y - nb.y)**2 < cdsq]

    if ag.type == 'r':
        if len(neighbors) > 0: # if there are foxes nearby
            if random() < dr:
                agents.remove(ag)
                return
        if random() < rr*(1-sum(1 for x in agents if x.type == 'r')/nr):
            agents.append(cp.copy(ag))
    else:
        if len(neighbors) == 0: # if there are no rabbits nearby
            if random() < df:
                agents.remove(ag)
                return
        else: # if there are rabbits nearby
            if random() < rf:
                agents.append(cp.copy(ag))

def update_one_unit_time():
    global agents
    t = 0.
    while t < 1.:
        t += 1. / len(agents)
        update()

import pycxsimulator
pycxsimulator.GUI().start(func=[initialize, observe, update_one_unit_time])
```

A typical simulation result is shown in Figure 19.5. The tiny blue dots represent prey individuals, while the larger red circles represent predators. As you see in the figure, the interplay between prey and predators produces very dynamic spatial patterns, which somewhat resemble the patterns seen in the host-pathogen CA model discussed in Section 11.5. The prey population grows to form clusters (clouds of tiny blue dots), but if they are infested by predators, the predators also grow rapidly to consume the prey, leaving a deserted empty space behind. As a result, the system as a whole begins to show dynamic waves of prey followed by predators. It is clear that spatial distributions of those species are highly heterogeneous. While we observe the periodic wax and wane of these species, as predicted in the equation-based predator-prey models, the ABM version generates far more complex dynamics that involve spatial locality and stochasticity. One could argue that the results obtained from this ABM version would be more realistic than those obtained from purely equation-based models.

We can further modify the simulator code of the predator-prey ABM so that it outputs the time series plot of prey and predator populations next to the visualization of the agents' positions. This can be done in a fairly simple revision: We just need to create lists to store time series of prey and predator populations, and then revise the `observe` function to count them, append the results to the lists, and visualize the lists as time series plots. Here are the updated `initialize` and `observe` functions:

Code 19.18: predator-prey-abm-with-plot.py

```
def initialize():
    global agents, rdata, fdata
```

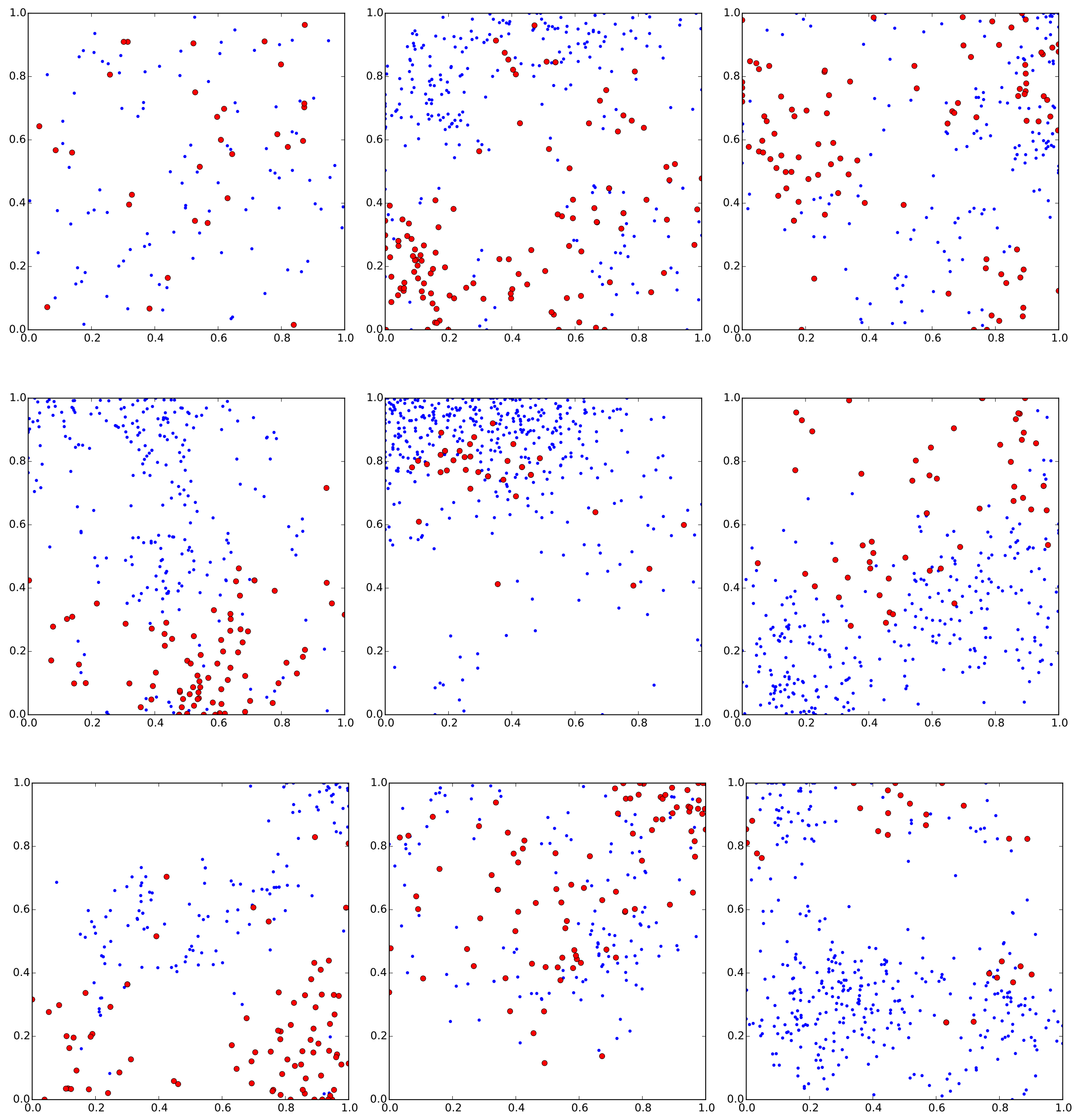

Figure 19.5: Visual output of Code 19.17. Time flows from left to right, then from top to bottom.

```
    agents = []
    rdata = []
    fdata = []

    ...

def observe():
    global agents, rdata, fdata

    subplot(2, 1, 1)
    cla()
    rabbits = [ag for ag in agents if ag.type == 'r']
    rdata.append(len(rabbits))
    if len(rabbits) > 0:
        x = [ag.x for ag in rabbits]
        y = [ag.y for ag in rabbits]
        plot(x, y, 'b.')
    foxes = [ag for ag in agents if ag.type == 'f']
    fdata.append(len(foxes))
    if len(foxes) > 0:
        x = [ag.x for ag in foxes]
        y = [ag.y for ag in foxes]
        plot(x, y, 'ro')
    axis('image')
    axis([0, 1, 0, 1])

    subplot(2, 1, 2)
    cla()
    plot(rdata, label = 'prey')
    plot(fdata, label = 'predator')
    legend()
```

A typical simulation result with this revised code is shown in Fig. 19.6. You can see that the populations of the two species are definitely showing oscillatory dynamics, yet they are nothing like the regular, cyclic ones predicted by equation-based models (e.g., Fig. 4.8). Instead, there are significant fluctuations and the period of the oscillations is not regular either. Spatial extension, discreteness of individual agents, and stochasticity in their behaviors all contribute in making the results of agent-based simulations more dynamic and "realistic."

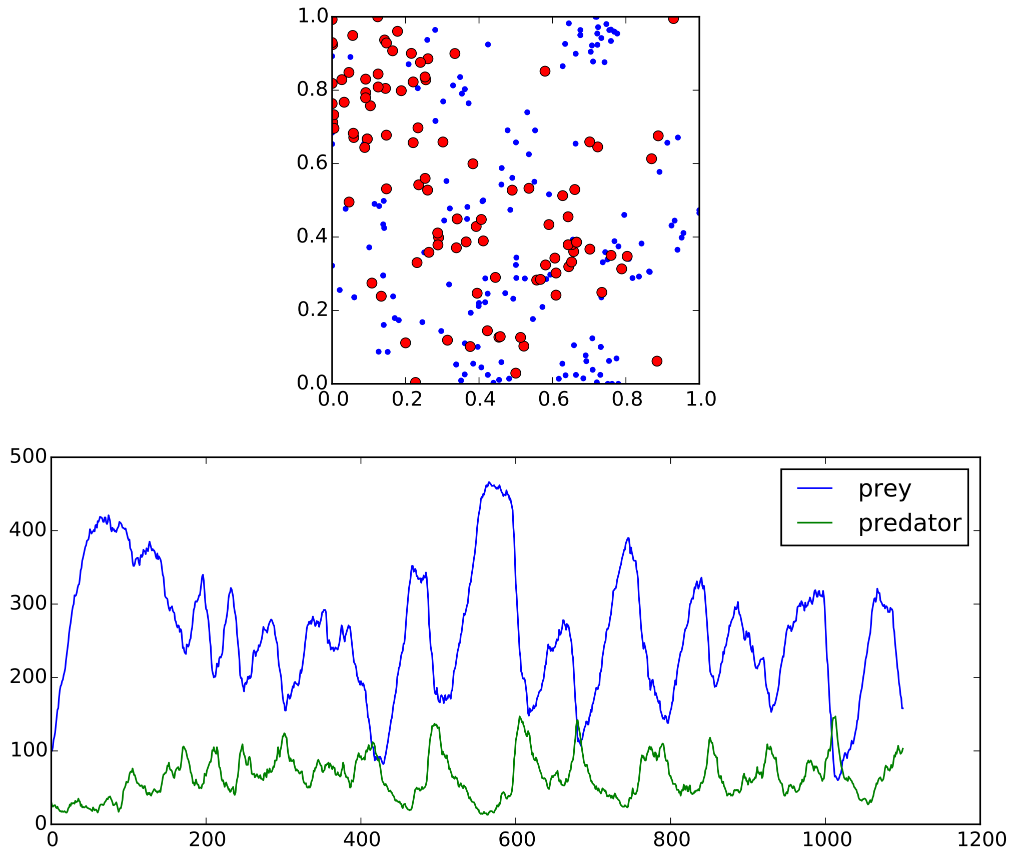

Figure 19.6: Visual output of Code 19.18.

Exercise 19.9 In the current model settings, the predator-prey ABM occasionally shows extinction of predator agents (or both types of agents). How can you make the coexistence of two species more robust and sustainable? Develop your own strategy (e.g., adjusting model parameters, revising agents' behavioral rules, adding structures to space, etc.), implement it in your simulator code, and test how effective it is. Find what kind of strategy works best for the conservation of both species.

Exercise 19.10 Revise the predator-prey ABM so that it also includes a spatially distributed food resource (e.g., grass) that the prey need to consume for their survival and reproduction. This resource can spontaneously grow and diffuse over space, and decrease when eaten by the prey. Conduct simulations of this revised model to see how the introduction of this third species affects the dynamics of the simulated ecosystem.

The final addition we will make to the model is to introduce the *evolution* of agents. Evolution is a simple yet very powerful dynamical process by which a population of organisms may spontaneously optimize their attributes for their own survival. It consists of the following three components:

Three components of evolution

Inheritance Organisms reproduce offspring whose attributes are similar to their own.

Variation There is some diversity of organisms' attributes within the population, which primarily arises from imperfect inheritance (e.g., mutation).

Selection Different attributes causes different survivability of organisms (*fitness*).

When all of these three components are present, evolution can occur in any kind of systems, not limited to biological ones but also social, cultural, and informational ones too. In our predator-prey ABM, the selection is already there (i.e., death of agents). Therefore, what we will need to do is to implement inheritance and variation in the model. For example, we can let the diffusion rates of prey and predators evolve spontaneously over time. To make them evolvable, we need to represent them as heritable traits. This can be accomplished by adding

Code 19.19:

```
        ag.m = mr if i < r_init else mf
```

to the `initialize` function, and then replacing

Code 19.20:

```
    m = mr if ag.type == 'r' else mf
```

with

Code 19.21:

```
    m = ag.m
```

in the `update` function. These changes make the magnitude of movement a heritable attribute of individual agents. Finally, to introduce variation of this attribute, we should add some small random number to it when an offspring is born. This can be implemented by replacing

Code 19.22:

```
            agents.append(cp.copy(ag))
```

with something like:

Code 19.23:

```
            newborn = cp.copy(ag)
            newborn.m += uniform(-0.01, 0.01)
            agents.append(newborn)
```

There are two places in the code where this replacement is needed, one for prey and another for predators. Note that, with the mutations implemented above, the diffusion rates of agents (`m`) could become arbitrarily large or small (they could even become negative). This is fine for our purpose, because `m` is used in the `update` function in the form of `uniform(-m, m)` (which works whether `m` is positive or negative). But in general, you should carefully check your code to make sure the agents' evolvable attributes stay within meaningful bounds.

For completeness, here is the revised code for the evolutionary predator-prey ABM (with the revised parts indicated by `###`):

Code 19.24: predator-prey-abm-evolvable.py

```
import matplotlib
matplotlib.use('TkAgg')
from pylab import *
import copy as cp

nr = 500. # carrying capacity of rabbits
```

```
r_init = 100 # initial rabbit population
mr = 0.03 # magnitude of movement of rabbits
dr = 1.0 # death rate of rabbits when it faces foxes
rr = 0.1 # reproduction rate of rabbits

f_init = 30 # initial fox population
mf = 0.05 # magnitude of movement of foxes
df = 0.1 # death rate of foxes when there is no food
rf = 0.5 # reproduction rate of foxes

cd = 0.02 # radius for collision detection
cdsq = cd ** 2

class agent:
    pass

def initialize():
    global agents
    agents = []
    for i in xrange(r_init + f_init):
        ag = agent()
        ag.type = 'r' if i < r_init else 'f'
        ag.m = mr if i < r_init else mf ###
        ag.x = random()
        ag.y = random()
        agents.append(ag)

def observe():
    global agents
    cla()
    rabbits = [ag for ag in agents if ag.type == 'r']
    if len(rabbits) > 0:
        x = [ag.x for ag in rabbits]
        y = [ag.y for ag in rabbits]
        plot(x, y, 'b.')
    foxes = [ag for ag in agents if ag.type == 'f']
    if len(foxes) > 0:
        x = [ag.x for ag in foxes]
        y = [ag.y for ag in foxes]
        plot(x, y, 'ro')
    axis('image')
    axis([0, 1, 0, 1])

def update():
    global agents
    if agents == []:
        return

    ag = agents[randint(len(agents))]

    # simulating random movement
    m = ag.m ###
    ag.x += uniform(-m, m)
    ag.y += uniform(-m, m)
    ag.x = 1 if ag.x > 1 else 0 if ag.x < 0 else ag.x
    ag.y = 1 if ag.y > 1 else 0 if ag.y < 0 else ag.y

    # detecting collision and simulating death or birth
    neighbors = [nb for nb in agents if nb.type != ag.type
                 and (ag.x - nb.x)**2 + (ag.y - nb.y)**2 < cdsq]

    if ag.type == 'r':
        if len(neighbors) > 0: # if there are foxes nearby
            if random() < dr:
                agents.remove(ag)
                return
        if random() < rr*(1-sum(1 for x in agents if x.type == 'r')/nr):
            newborn = cp.copy(ag) ###
            newborn.m += uniform(-0.01, 0.01) ###
            agents.append(newborn) ###
    else:
        if len(neighbors) == 0: # if there are no rabbits nearby
            if random() < df:
                agents.remove(ag)
                return
        else: # if there are rabbits nearby
            if random() < rf:
                newborn = cp.copy(ag) ###
                newborn.m += uniform(-0.01, 0.01) ###
                agents.append(newborn) ###

def update_one_unit_time():
    global agents
    t = 0.
    while t < 1.:
        t += 1. / len(agents)
        update()

import pycxsimulator
pycxsimulator.GUI().start(func=[initialize, observe, update_one_unit_time])
```

You can conduct simulations with this revised code to see how the prey's and predators' mobilities evolve over time. Is the result consistent with or counter to your initial prediction? I hope you find some interesting discoveries.

Exercise 19.11 Revise the `observe` function of the evolutionary predator-prey ABM developed above so that you can see the distributions of `m` among prey and predators. Observe how the agents' mobilities spontaneously evolve in a simulation. Are they converging to certain values or continuously changing?

Exercise 19.12 Conduct systematic simulations using both the original and evolutionary predator-prey ABMs to quantitatively assess whether the introduction of evolution into the model has made the coexistence of two species more robust and sustainable or not.

Exercise 19.13 Make the reproduction rates of prey and predator agents also evolvable, in addition to the magnitude of their movements. Conduct simulations to see how the reproduction rates evolve over time.

Believe it or not, we are now at the end of our over-450-page-long journey. Thanks for exploring the world of complex systems with me. I hope you enjoyed it. Needless to say, this is just the beginning for you to dive into a much larger, unexplored territory of complex systems science. I believe there are tons of new properties of various complex systems still to be discovered, and it would be my utmost pleasure if this textbook was a bit of a help for you to make such discoveries in the coming years. Bon voyage!