*12.3 A FLOCKING SIMULATION

It was once assumed that the collective movements of flocks of birds, schools of fish, and herds of animals arise from many individuals following a designated leader. But we now know that this is not the case; instead each individual is independently carrying out an identical algorithm based on local interactions with its neighbors. The adaptive swarming and V-like patterns that result from these local interactions between individuals are known as *emergent* behaviors. This particular type of emergent behavior is, for obvious reasons, commonly known as ***swarm intelligence***.

The *boids model*, originally conceived by Craig Reynolds, is a simulation of emergent flocking behavior. In this model, a "boid" is a representation of a bird, fish or other animal. Each boid independently follows three rules:

1. Avoid collisions with obstacles and nearby flockmates.
2. Attempt to match the velocity (heading plus speed) of nearby flockmates.
3. Attempt to move toward the center of the flock to avoid predators.

In this section, we will write an object-oriented agent-based simulation to visualize the flocking behavior that results from the boids model. As in the epidemic simulation from Section 12.1, we will need two main components: the agents (in our case, boids) and a "world" in which the agents live. Each of these components can be conceived as an abstract data type, and implemented as a class.

The World

The simplest kind of world for agents to inhabit is a rectangular two-dimensional plane, like those we used in Chapter 8 and Section 12.1. However, we will think of the world as a continuous space rather than a grid. In other words, each boid may reside at any (x,y) position within its boundaries, not just those where x and y are integers. (In Project 12.4, you will have the opportunity to extend this to a three-dimensional world.) An abstract data type for this flat world will contain attributes for its dimensions and a list of agents inhabiting it, with their positions, as shown below.

Instance Variable	Description
agents	a list of agents, with their positions
width	the number of columns in the grid
height	the number of rows in the grid

The operations for a World ADT include getting its dimensions, getting, setting and deleting

Copyright Taylor and Francis, 2021

the agent in a particular position, and querying the neighborhood of any position. We will represent a position with a (x, y) tuple, where x is a column number and y is a row number.

Method	Arguments	Description
create	width, height	create a new World instance with the given size
getWidth	—	return the width of the world
getHeight	—	return the height of the world
get	position	return the agent in the given position
set	position, agent	place an agent in the given position
delete	position	delete the agent in the given position
neighbors	position, distance	return all agents within distance of position
stepAll	—	advance all agents one step in the simulation

As in the epidemic simulation from Section 12.1, the program that implements this simulation will repeatedly call stepAll to simulate the progression of time.

A class that implements the constructor, plus the first two accessor operations, is shown below. (We will use abbreviated docstrings to save space.)

```
class World:
    """A two-dimensional world class."""

    def __init__(self, width, height):
        """Construct a new flat world with the given dimensions."""

        self._width = width
        self._height = height
        self._agents = { }

    def getWidth(self):
        """Return the width of self."""

        return self._width

    def getHeight(self):
        """Return the height of self."""

        return self._height
```

As an alternative to the list-of-lists grid implementation from Chapter 8, we will store the agents in a dictionary with keys equal to the agents' positions (tuples). We will start with an empty dictionary and add entries as agents are added to the world. This way, we only use as much space as we need to store the agents and assume that the rest of the world is empty. In contrast, the implementation that we used in Chapter 8 explicitly stores every cell, whether or not it is occupied. If most cells are always occupied, then this implementation is perfectly reasonable. However, in situations where most cells are not occupied at any particular time, as will be the case in our boids simulation, the dictionary implementation is more efficient. In these situations, we say that the space is *sparse* or a *sparse matrix*.

Two-dimensional indexing

To get or change the agent in a particular position in the world, we can define indexing for the `World` class using the `__getitem__` and `__setitem__` methods. In our two-dimensional world, an index needs to be a (x,y) pair. So we can define the `__getitem__` and `__setitem__` methods to interpret the index parameter as a two-element tuple that we directly use as a key in our dictionary.

```
def __getitem__(self, position):
    """Return the agent at the given position."""

    if position in self._agents:
        return self._agents[position]
    return None

def __setitem__(self, position, agent):
    """Set the given position to contain agent."""

    if (position not in self._agents) and \
       (position[0] >= 0) and (position[0] < self._width) and \
       (position[1] >= 0) and (position[1] < self._height):
        self._agents[position] = agent
```

With these methods, we can get and set positions in the world by simply indexing with tuples. As a simple example, we can place an integer value (in lieu of an agent for now) in position (2,1) and then print the value at that position like this:

```
myWorld = World(10, 10)
myWorld[2, 1] = 5
print(myWorld[2, 1])
```

Notice that we did not need to put parentheses around the tuple values that we used in the square brackets; Python will automatically wrap the pair in parentheses and interpret it as a tuple. If we try to get an agent from a position that is empty, the `__getitem__` method returns `None`. The `__setitem__` method does nothing if we try to insert an agent into a position that is occupied or out of bounds.

When an agent moves, we will need to delete it from its current position in the world. To delete an item from a Python list or dictionary, we use the `del` operator. We can enable this behavior in a `World` object by defining the `__delitem__` method.

```
    def __delitem__(self, position):
        """Delete the agent at the given position."""

        if position in self._agents:
            del self._agents[position]
```

With this method defined, we can delete an agent from some position like this:

```
del myWorld[2, 1]
```

Meeting the neighbors

In an agent-based simulation in general, and our boids simulation in particular, we commonly need to query the neighborhood of an agent. In our boids simulation, each boid will repeatedly adjust its velocity based on the positions and velocities of nearby boids. The following method, which is very similar to the `within` method from Section 12.1, returns a list of agents within some distance of a particular position by iterating over all positions in the dictionary and checking whether each is within range.

```
    def neighbors(self, position, distance):
        """Return a list of agents within distance of tuple position."""

        neighbors = []
        for otherPosition in self._agents:
            if (position != otherPosition) and \
               (_distance(position, otherPosition) <= distance):
                neighbors.append(self._agents[otherPosition])
        return neighbors
```

The `_distance` function (not shown) is a private function defined outside the class that returns the distance between two positions.

> **Reflection 1** *Why do you think we didn't include the* `_distance` *function as a method of* `World` *instead?*

We decided to not make `_distance` a method because it does not need to access any attributes of the class. The leading underscore in its name prevents the function from being imported into other modules.

Simulating one step

Finally, the last method in the `World` class iterates over every agent in the world, and moves that agent one step forward in the simulation.

```
    def stepAll(self):
        """All agents advance one step in the simulation."""

        agents = list(self._agents.values())
        for agent in agents:
            agent.step()
```

This method assumes that an agent's actions are implemented in a method named `step`. In the case of our boid simulation, a boid will look at its neighbors in each step and adjust its

velocity accordingly. We will tackle this next. The complete `World` class is also available on the book website.

Boids

The design of an agent in an agent-based simulation depends quite a bit on the particular application, but the relationship between the agent and the world tends to share some common characteristics. As in the epidemic simulation, because agents can only interact indirectly with other agents through their shared world, an agent must have access to the world in which it resides. Also, the agent must "know" where in the world it resides. In our boid simulation, each boid will also have a velocity, which combines both its speed and heading.

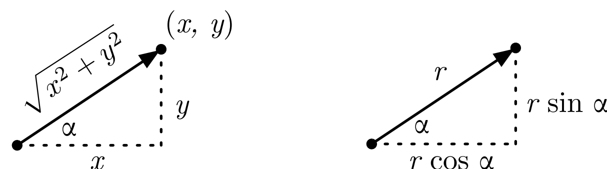
Instance Variable	Description
world	the world in which the boid resides
position	the boid's (x,y) position in its world
velocity	the boid's velocity (speed and heading)

In each step of an agent-based simulation, each agent carries out some application-specific tasks, such as querying its neighbors and moving to a new location. We encapsulate this activity in a function named step. In our boids simulation, in each step, a boid will look at its neighbors, adjust its velocity according to the three rules of the boids model, and then move to its new position. These intermediate steps will be handled by the neighbors and move operations below.

Method	Arguments	Description
create	the world	create a new Boid instance with random position and velocity in the world
neighbors	distance, angle	return all boids within distance and viewing angle
move	—	move to a new position based on current velocity
step	—	adjust the boid's velocity following the three rules

Vectors

Velocity is represented by a ***vector***. A vector is simply an ordered pair, like a geometric point, but it represents a quantity with both magnitude and direction (e.g., velocity, force, or displacement). A vector $\langle x,y \rangle$ is often represented by a directed line segment that extends from the origin (0,0) to the point (x,y), as shown below on the left.



The angle α that the vector makes with the horizontal axis is the direction of the vector and the length of the line segment is the vector's magnitude. If a vector represents velocity, then α is the direction of movement and the magnitude is speed. We know from Pythagorean

theorem that the magnitude of the vector is $\sqrt{x^2 + y^2}$. If you know some trigonometry, you also know that the angle $\alpha = \tan^{-1}(y/x)$. Also, if you only know the magnitude r and angle α, you can find the vector $\langle x, y \rangle$ with $x = r\cos\alpha$ and $y = r\sin\alpha$, as illustrated above on the right.

To facilitate some of the computations that will be necessary in our boids simulation, we have written a new `Vector` class, based on the `Pair` class, but with a few additional methods that apply specifically to vectors. You can download the `vector.py` module from the book website.

The Boid class

In the constructor of the `Boid` class below, a reference to the `World` object in which the boid resides is passed as the parameter `myWorld`, and stored in the instance variable named `self._world`. Each boid needs to have access to the `World` object so that the boid can change its position in the world and call the `World` object's `neighbors` method.

```
class Boid:
    """A boid in a agent-based flocking simulation."""

    def __init__(self, myWorld):
        """Construct a boid at a random position in the given world."""

        self._world = myWorld
        (x, y) = (random.randrange(self._world.getWidth()),
                  random.randrange(self._world.getHeight()))
        while self._world[x, y] != None:
            (x, y) = (random.randrange(self._world.getWidth()),
                      random.randrange(self._world.getHeight()))
        self._position = Pair(x, y)
        self._world[x, y] = self
        self._velocity = Vector((random.uniform(-1, 1),
                                 random.uniform(-1, 1))).unit()
        self._turtle = turtle.Turtle()
        self._turtle.speed(0)
        self._turtle.up()
        self._turtle.setheading(self._velocity.angle())
```

The constructor assigns the new `Boid` object a random, unoccupied position in `self._world`. Notice that this involves three steps: finding an unoccupied position using a `while` loop, assigning this position (a `Pair` object) to the instance variable `self._position`, and placing the new `Boid` object (`self`) in `self._world` at that position. Next, the instance variable `self._velocity` is assigned a `Vector` object with random value between $\langle -1, -1 \rangle$ and $\langle 1,1 \rangle$ (which covers every angle between 0 and 360 degrees). The `unit` method scales the velocity vector so that it has magnitude (speed) 1. Finally, we add an instance variable for a `Turtle` object to visualize the boid, and set the turtle's initial heading to the angle of the velocity vector.

Reflection 2 *What angle does the vector $\langle 1, 1 \rangle$ represent? What about $\langle -1, -1 \rangle$ and $\langle 0, -1 \rangle$?*

Reflection 3 *How many instance variables does each* `Boid` *object have?*

Moving a boid

In each step of the simulation, a boid will move to a new location based on its current velocity (which will change periodically based on its interaction with neighboring boids). The `move` method below moves the `Boid` object by adding the x and y coordinates of its velocity to the x and y coordinates of its current position.

```
def move(self):
    """Move self to a new position in its world."""

    self._turtle.setheading(self._velocity.angle())

    width = self._world.getWidth()
    height = self._world.getHeight()

    newX = self._position[0] + self._velocity[0]
    newX = min(max(0, newX), width - 1)
    newY = self._position[1] + self._velocity[1]
    newY = min(max(0, newY), height - 1)

    if self._world[newX, newY] == None:
        self._world[newX, newY] = self          # place in new pos
        del self._world[self._position.get()]  # and del from old
        self._position = Pair(newX, newY)       # set new pos
        self._turtle.goto(newX, newY)           # move turtle

    if (self._velocity[0] < 0 and newX < 5) or \
       (self._velocity[0] > 0 and newX > width - 5) or \
       (self._velocity[1] < 0 and newY < 5) or \
       (self._velocity[1] > 0 and newY > height - 5):
        self._velocity.turn(TURN_ANGLE)
```

After the boid's new position is assigned to `newX` and `newY`, if this new position is not occupied in the boid's world, the `Boid` object moves to it. Finally, if the boid is approaching a boundary, we rotate its velocity by some angle, so that it turns in the next step. The constant value `TURN_ANGLE` along with some other named constants will be defined at the top of the boid module. To avoid unnaturally abrupt turns, we use a small angle like

```
TURN_ANGLE = 30
```

Reflection 4 *What is the purpose of the* `min(max(0, newX), width - 1)` *expression (and the analogous expression for* `newY`*)?*

Reflection 5 *In the last* `if` *statement, why do we check both the new position and the velocity?*

Implementing the boids' rules

We are finally ready to implement the three rules of the boids model. In each step of the simulation, we will compute a new velocity based on each of the rules, and then assign the boid a weighted sum of these and the current velocity, scaled to a magnitude of one so that

all boids maintain the same speed. (A vector with magnitude one is called a *unit vector*.) Once the velocity is set, we call `move` to move the boid to its new position.

```
    def step(self):
        """Advance self one step in the flocking simulation."""

        newVelocity = (self._velocity * PREV_WEIGHT +
                       self._avoid() * AVOID_WEIGHT +   # rule 1
                       self._match() * MATCH_WEIGHT +   # rule 2
                       self._center() * CENTER_WEIGHT)  # rule 3

        self._velocity = newVelocity.unit()
        self.move()
```

The private `_match`, `_center`, and `_avoid` methods will compute each of the three individual velocities. The boids model suggests that the weights assigned to the rules decrease in order of rule number. So avoidance should have the highest weight, velocity matching the next highest weight, and centering the lowest weight. For example, the following values follow these guidelines.

```
PREV_WEIGHT = 0.5
AVOID_WEIGHT = 0.25
MATCH_WEIGHT = 0.15
CENTER_WEIGHT = 0.1
```

Note that because we always scale the resulting vector to a unit vector, these weights need not always sum to 1. Once we have the complete simulation, we can modify these weights to induce different behaviors.

In each of the `_match`, `_center` and `_avoid` methods, we will need to get a list of boids within some distance and viewing angle. This is accomplished by the `Boid` method named `neighbors`, shown below.

```
    def neighbors(self, distance, angle):
        """Return a list of boids within distance and viewing angle."""

        neighbors = self._world.neighbors(self._position.get(), distance)
        visibleNeighbors = []
        for boid in neighbors:
            neighborDir = Vector((boid._position - self._position).get())
            if self._velocity.diffAngle(neighborDir) < angle:
                visibleNeighbors.append(boid)
        return visibleNeighbors
```

The method begins by calling the `neighbors` method of the `World` class to get a list of boids within the given distance. Then we iterate over these neighbors, and check whether each one is visible within the given viewing angle. Doing this requires a little vector algebra. In a nutshell, `neighborDir` is the vector pointing in the direction of the neighbor named `boid`, from the point of view of this boid (i.e., `self`). The `diffAngle` method of the `Vector` class computes the angle between `neighborDir` and the velocity of this boid. If this angle is within the boid's viewing angle, we add the neighboring boid to the list of visible neighbors to return.

With this infrastructure in place, methods that follow the three rules are relatively straightforward. Let's review them before continuing:

1. Avoid collisions with obstacles and nearby flockmates.
2. Attempt to match the velocity (heading plus speed) of nearby flockmates.
3. Attempt to move toward the center of the flock to avoid predators.

Since rule 2 is slightly easier than the other two, let's implement that one first.

```
def _match(self):
    """Return the average velocity of neighboring boids."""

    neighbors = self.neighbors(MATCH_DISTANCE, MATCH_ANGLE)
    if len(neighbors) == 0:
        return Vector()
    sumVelocity = Vector()
    for boid in neighbors:
        sumVelocity = sumVelocity + boid._velocity
    return (sumVelocity / len(neighbors)).unit()
```

The method first gets a list of visible neighbors, according to a distance and viewing angle that we will define shortly. If there are no such neighbors, the method returns the zero vector $\langle 0,0 \rangle$. Otherwise, it returns the average velocity of the neighbors, normalized to a unit vector. Since we have overloaded the addition and division operators for `Vector` objects, as we did for the `Pair` class, finding the average of a list of vectors is no harder than finding an average of a list of numbers!

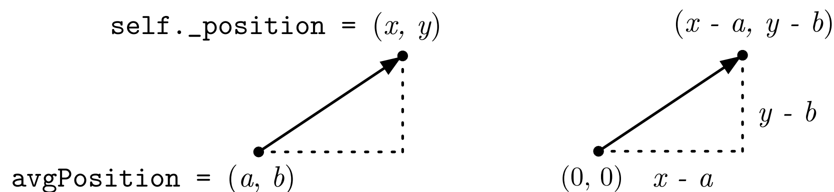
The method to implement the first rule is similar, but now we want to find the velocity vector that points *away* from the average position of close boids.

```
def _avoid(self):
    """Return a velocity away from close neighbors."""

    neighbors = self.neighbors(AVOID_DISTANCE, AVOID_ANGLE)
    if len(neighbors) == 0:
        return Vector()
    sumPosition = Pair()
    for boid in neighbors:
        sumPosition = sumPosition + boid._position
    avgPosition = sumPosition / len(neighbors)

    avoidVelocity = Vector((self._position - avgPosition).get())
    return avoidVelocity.unit()
```

The first part of the method finds the average position (rather than velocity) of neighboring boids. Then it finds the vector that points away from this average position by subtracting it from the position of the boid. This is illustrated below.



On the left is an illustration of the vector we want, pointing from avgPosition, which we will call (a,b), to the boid's position (x,y). But vectors always start at $(0,0)$, so the correct vector has horizontal distance $x - a$ and vertical distance $y - b$, as shown on the right. This is precisely the vector $\langle x - a, y - b\rangle$ that we get by subtracting avgPosition from self._position.

Finally, the method that implements rule 3 is almost identical to the _avoid method, except that we want a vector that points *toward* the average position of the flock, which we define to be a group of neighboring boids within a larger radius of the boid than those it is trying to avoid.

```
    def _center(self):
        """Return a velocity toward center of neighboring flock."""

        neighbors = self.neighbors(CENTER_DISTANCE, CENTER_ANGLE)
        if len(neighbors) == 0:
            return Vector()
        sumPosition = Pair()
        for boid in neighbors:
            sumPosition = sumPosition + boid._position
        avgPosition = sumPosition / len(neighbors)

        centerVelocity = Vector((avgPosition - self._position).get())
        return centerVelocity.unit()
```

In the _center method, we perform the subtraction the other way around to produce a vector in the opposite direction. The distance and viewing angle will also be different from those in the _avoid method. We want AVOID_DISTANCE to be much smaller than CENTER_DISTANCE so that boids avoid only other boids that are very close to them. We will use the following values to start.

```
AVOID_DISTANCE = 3     # avoid only close neighbors
AVOID_ANGLE = 300

MATCH_DISTANCE = 10    # match velocity of intermediate neighbors
MATCH_ANGLE = 240

CENTER_DISTANCE = 15   # move toward center of farther neighbors
CENTER_ANGLE = 180
```

The main simulation

We are finally ready to run the flocking simulation! The following program sets up a turtle graphics window, creates a world named sky, and then creates several Boid objects. The simulation is set in motion by the last for loop, which repeatedly calls the step method of the World class.

```
import turtle
from world import *
from boid import *

WIDTH = 100
HEIGHT = 100
NUM_BIRDS = 30
ITERATIONS = 2000
```

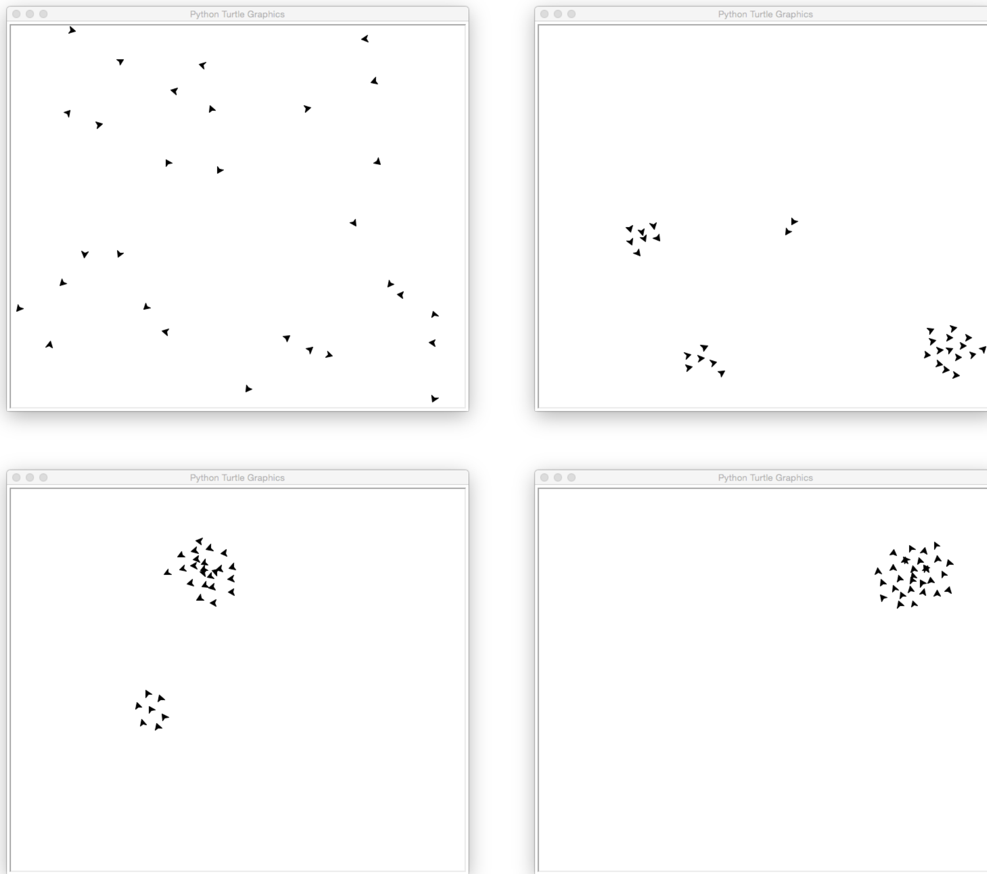


Figure 1 A progression of flocking boids.

```
def main():
    worldScreen = turtle.Screen()
    worldScreen.setworldcoordinates(0, 0, WIDTH - 1, HEIGHT - 1)
    worldScreen.tracer(NUM_BIRDS)

    sky = World(WIDTH, HEIGHT)
    for index in range(NUM_BIRDS):
        bird = Boid(sky)

    for step in range(ITERATIONS):
        sky.stepAll()

    worldScreen.update()
    worldScreen.exitonclick()

main()
```

The complete program is available on the book website. Figure 1 shows a sequence of four screenshots of the simulation.

Reflection 6 *Run the program a few times to see what happens. Then try changing the following constant values. What is the effect in each case?*

(a) `TURN_ANGLE = 90`
(b) `PREV_WEIGHT = 0`
(c) `AVOID_DISTANCE = 8`
(d) `CENTER_WEIGHT = 0.25`

Exercises

12.3.1. Remove each of the three rules from the simulation, one at a time, by setting its corresponding weight to zero. What is the effect of removing each one? What can you conclude about the importance of each rule to successful flocking?

12.3.2* Implement the `__eq__` method for the `Vector` class. Then modify the `step` method so that it slightly alters the boid's heading with some probability if the new velocity is the same as the old velocity (which will happen if it has no neighbors).

12.3.3. Modify the `move` method so that a boid randomly turns either left or right when it approaches a boundary. What is the effect?

Selected Exercise Solutions

12.3.2
```
if newVelocity == self._velocity:
    if random.random() < 0.2:
        newVelocity = self._velocity \
                    + Vector((random.uniform(-0.1, 0.1),
                              random.uniform(-0.1, 0.1)))
```