LiveLetters: Writing with Emergence

Michael J. Korman

May 15, 2003

Abstract

This paper will describe *LiveLetters*, a model for constructing letters from emergent behaviors. The general principles behind emergent systems will be discussed, and examples will be drawn from nature and technology.

1 Introduction

In this section, we will answer the question of what emergence is, and present examples, both natural and artificial.

1.1 What is Emergence?

Emergent behaviors are those phenomena that occur as large systems of individual agents produce complex global behavior through simple local interactions. Agents in an emergent system may not be aware of their participation in the greater whole, having only their own limited concerns. They are completely autonomous.

1.2 Principles of Emergence

Let us examine the qualities that characterize emergent systems. These qualities will henceforth be referred to as the *Principles of Emergence*.

1.2.1 Decentralization

The power of decentralization has long been ignored. Systems without centralized control exist everywhere: in cities, economies, insect societies, and biological organisms.

Adam Smith's "invisible hand" gave explanation[10] as to how economies could function without government control. His reasoning was that an economy is an *adaptive system*, composed of interacting agents, each modifying its behavior based on personal gain. While this was previously believed to cause the decay and collapse of order, Smith showed that it only strengthened the system, as each individual fighting for itself made the whole much more robust than any centralized authority ever could.

In the 19th century, Charles Darwin argued[2] that centralized design was not requisite for the creation of life. His ideas of biological evolution shook the world when he claimed that species' adapting to their changing environments was responsible for the diversity and flexibility of life.

Agents in such adaptive systems rely on a relatively simple local set of rules to govern their behavior, rather than a central control.

1.2.2 Randomness

Given that emergent systems have no central command, it is left to probability to dictate their behaviors. The strong law of large numbers[8] illustrates this.

The Strong Law of Large Numbers. *Let $X_1, X_2, \ldots, X_n$ be a sequence of independent and identically distributed random variables, each having a finite mean μ. Then,*

$$P\left\{\lim_{n\to\infty} \frac{X_1 + X_2 + \cdots + X_n}{n} = \mu\right\} = 1$$

Simply put, with probability 1, the average of a collection of similar events will converge to the average behavior of each event as the number of events increases. If I flip ten coins, each one will land heads up with probability 1/2, and tails up with probability 1/2. This does not mean that flipping ten coins will result in five coming up heads and five coming up tails. However, as I increase the number of coins to infinity, I am guaranteed that the distribution will become half-and-half.

This principle plays an important role in emergent systems. Recall that we are concerned with the behavior of the whole system, with the number of individual agents some large arbitrary figure. Emergent systems must be immune to random mutations in its agents. As long as the population is normal "overall," the system should function perfectly.

1.2.3 Localization

The principle of localization states that in an emergent system, agents can only rely on their own, internal database of information. There is no "global repository" of data that everyone can tap into. §2.1 describes how ants gather food, which makes heavy use of localization.

1.2.4 Uniformity and Roles

Agents in emergent systems can take on one or more roles. However, the number of agents is always significantly greater than the number of roles. That is, with n agents, and k roles,

$$\lim_{n\to\infty} \frac{k}{n} = 0$$

All agents in a given role must be identical. They all "execute the same code." However, it is possible for there to be states within a given role, and this can lead to variety even among agents in the same role.

Roles can help create diverse behaviors, but it is more important that there is interaction between the agents. Overall, the agents will appear to be rather uniform. See §2.1 for an example of how roles work.

2 Examples of Emergence

In this section, we will explore examples of emergent systems.

2.1 Emergence in Nature: Ants

Many insect societies exhibit remarkable emergent behavior. We will look at several characteristics of ant societies[3] that rely on decentralized attributes.

One astonishing quality of ant colonies is the limited number of roles in relation to the wide variety of tasks that colonies perform. Some ants are used for reproductive purposes; these are the queens and the males. The vast majority of the ants in a colony, however, can be classified as *workers*. Workers are responsible for several tasks, including patrolling, foraging, and nest maintenance. It is not known exactly how the ants decide which workers are assigned to which tasks, but it is widely believed that younger workers labor inside the nest, and move on to more dangerous outdoor tasks as they get older.

Patrolling is an important task, and must be carried out before foraging for food begins. Ants appear to have created an intricate network for determining the safety conditions outside the nest. In the early morning, patrollers set out from the nest, and begin wandering the surrounding area. Patrollers move in special patterns, so that they cover as much area as they can. When they randomly meet with other patrollers, they touch antennae, and move on. Outside the nest, the patrollers appear to have an "interaction rate" programmed into them, so that they are aware of how often they should be meeting up with other patrollers. This way, if too many patrollers have been killed, the others will find out almost immediately, *without the use of any communication*. Every so often, they return to the nest, where they meet with foragers who are waiting there. The foragers in the nest will know if the area is unsafe, because patrollers will not return as often. Disturbing only a few patrollers in the field is enough to send the entire colony into a state of caution.

Also of interest is the manner in which many species of ants gather food. Foragers scout the area around the nest, looking for food. Ants cannot see very well, so they basically wander around randomly. When they find food, they return it to the nest, leaving behind a chemical pheromone as they travel. As the foragers arrive at the nest, other ants follow the pheromone to the food, and carry pieces back. As more ants travel the path, the chemical increases in strength, and attracts more ants. When the food runs out, ants will stop reinforcing the trail, and it will die out. If the food source was not large to

begin with, then it will run out quickly. This way, the colony appears to seek out the strongest food sources as if by centralized command, when actually it is all emergent.

2.2 Emergence in Technology: Freenet

Freenet[1] is a distributed network that was designed with decentralization as its primary goal. Its designer, Ian Clarke, wanted to create a network that would be resistant to censorship, sabotage, and failure of individual nodes. This contrasts with the Internet, which uses a centralized mechanism, known as the Domain Name System[5] (DNS), for locating data. With DNS, and the protocols built on top of it[1], the *domain name* of the server on which the information resides must be known. To retrieve the information, a request is sent to a central DNS server, which attempts to resolve the domain name into an Internet Protocol (IP) address, by querying servers lower in the DNS hierarchy.

This system works, but it is highly centralized. The operators of the central DNS servers have total control of all information flow on the Internet. If servers contain objectionable material, they can be censored out of the DNS databases. Too much power is in one place.

In contrast, Freenet employs a decentralized strategy. It lies on top of the Internet, but uses a much different method for storing and retrieving information than typical Internet protocols. Every computer on Freenet is equal; there are no central servers. Nodes are connected to other nodes, and contain information about their neighbors. Documents are inserted into the network anonymously, referenced by keys. As documents pass through nodes, they are placed onto a stack that is internal to each node. The stack is of limited size, so once it becomes full, items on the bottom are deleted. To retrieve a document, the key is sent to neighboring nodes, which provide information on where the document is most likely to be, based on their knowledge of the network.

From this simple description of Freenet, several striking emergent behaviors are noticed. First, posting of information is completely anonymous; the network itself decides how to distribute files. Second, information is made available based on popularity. As keys are requested more often, the associated files become resident on more and more nodes, and thus become easier to attain. Consequently, the keys that are not requested often get kicked out. The resulting behavior is that important files are quickly accessed, and unimportant files are lost. *This is all done without human control.*

3 Software Simulation Tools

In the words of Herbert A. Simon, "it is the organization of components, and not their physical properties, that largely determines behavior."[9] Therefore, computer simulations play an important role in the study of multi-agent systems.

[1] Especially, the HyperText Transfer Protocol, the underlying protocol of the World Wide Web.

In this section, we will examine StarLogo and related software, which are used to model emergent systems.

3.1 StarLogo

StarLogo[6] was developed by Mitchel Resnick in 1989 at the MIT Media Lab. It is described in [7]. StarLogo is based on an earlier programming language called Logo. Logo was designed as language to teach programming to children. Typical Logo systems involve the use of robots, often constructed from LEGO bricks. Children can program these robots to move across the floor, making use of their light and touch sensors.

Other versions of Logo do not use robots, but are completely virtual. The user can control an on-screen creature, called a *turtle*. Graphics can be created by having the turtle paint the screen as it moves across. As exciting as this sounds, it is fairly limited in complexity. There is only one turtle, and it doesn't do much except move around.

StarLogo improves on Logo drastically. Instead of one turtle, it allows thousand of turtles. Whereas the environment in Logo is inert, the StarLogo world is composed of *patches* that are just as alive as the turtles themselves. They can have their own state, and act on the turtles in addition to being acted on. StarLogo provides a much richer environment for exploring multi-agent systems.

3.2 NetLogo

NetLogo[11] is basically an improved version of StarLogo. Developed at the Center for Connected Learning and Computer-Based Modeling at Northwestern University, it provides a more expressive command language, and a more powerful interface. I am discussing it because it is the environment I chose to implement the model described in §4. Please refer to Appendix A for a brief NetLogo tutorial.

4 *LiveLetters*

This section will describe the *LiveLetters* model. The goal of *LiveLetters* is to create English letters through emergent behavior. The system is decentralized; the turtles only have a basic idea of what they're doing, governed by a set of simple rules. *LiveLetters* was built in NetLogo (see §3.2). As such, we will use NetLogo parlance when describing it, namely the terms "turtles" and "patches."

Before we look at the *LiveLetters* model, let us consider a more traditional, centralized approach of solving the problem.

4.1 The Centralized Model

This model uses the tried and true software engineering principle of top-down design. The code is listed in Appendix B.

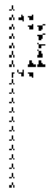

Figure 1: The centralized 'P'

4.1.1 Description

In the centralized method, all of the turtles will be identical. Letters will be composed of lines, and the turtles will be instructed specifically how to create those lines. To accomplish this, a set of procedural commands is created.

Suppose we wanted to create a line beginning at the point (x, y), with length ℓ, and at an angle θ. We could do this by dropping a large amount of turtles at (x, y), have them choose a random number of steps between 0 and ℓ, and order them to march that number of steps in the direction given by θ. If there are enough turtles, this will create the illusion of a solid line.

Now, imagine we wanted to create an arc at (x, y) with radius r and angle θ. This will be done by placing several turtles at (x, y), having them choose a random direction from 0 to θ, and having them march r steps. As an example, the letter 'P' is composed of a single straight line and an arc (see Figure 1).

Let us suppose our final goal was to display the word "AI." To do this, we will first create procedures to draw the letters 'A' and 'I.' This is easy, as each of these letters is composed of straight lines, so it reduces to a straight-forward geometry problem. The code for these letters therefore consists of calls to our `line` procedure. To draw the two letters, we must place them correctly on the field. The final output is shown in Figure 2.

4.1.2 Discussion

Though the centralized model works, it violates the Principles of Emergence (discussed in §1.2) in the following ways:

- It is highly *centralized*. The user, a god-like being, directs the turtles in their every action. There is no adaptiveness.
- Though the turtles have no global knowledge, they have no local knowledge, either. Therefore, the principle of localization is violated, as they are not autonomous. The turtles do not make any decisions.

The centralized model is simplistic, and does not capture everything we want in an emergent system.

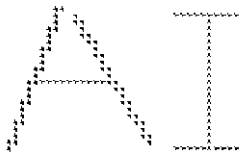

Figure 2: The centralized "AI"

4.2 The Decentralized Model

Now that we have considered the centralized model, and its problems, let's see how we can modify it by shifting some of the power from the user to the turtles.

4.2.1 Description

In the decentralized approach, there are two types of turtles:

Seekers are the normal, "worker" turtles. They roam the field, ultimately becoming the creatures that form letters.

Leaders are the guideposts of letters. Letters are described by placing a few leaders in a proper position. Leaders are divided into *line-leaders*, which create lines, and *arc-leaders*, which create arcs.

To create a line, two line-leaders are placed, one at each endpoint of the line. All leaders have a *charge*, and the charges of leaders at the opposite ends of a line are opposite. That is, if one leader has a charge of 5, then the opposite leader will have a charge of -5. The field is then flooded with seekers. The seekers will move about randomly, occasionally bumping into a leader. When this happens, the seeker will be bound to the leader, and will acquire the opposite charge of the leader. It will then seek out the leader that has its new charge[2]. When it finally reaches that leader, it will reverse charges again, and seek out the first leader. As this process continues, the seeker will move back and forth in a straight line. Enough seekers, and we have a solid line.

Arcs must be drawn for the creation of certain letters. This can be accomplished by placing arc-leaders on the field, one per arc. We will require arc-leaders to have charge 0. As seekers collide with the arc-leaders, they will acquire charge 0, jump away, and then attempt to reach the arc-leader again.

[2]Like electric charges repel, but like turtle charges attract!

However, they will be repelled, and forced to keep a certain distance. This will have the effect of the seekers being frozen in place, and they will form an arc.

Appendix C lists the code for the Decentralized Model. The `add_seekers` procedure begins by creating `tnumber` seekers, based on the global variable `tnumber` that is set by the user at run-time. Next, it changes the shape and color of the seekers. Following that, it sets their locations based on a random normal distribution, so the seekers are sufficiently spread out over the field. It assigns them the `charge` 20, which will change as they come in contact with leaders. For now, the `charge` 20 represents unbound seekers.

Now that we have added seekers to the field, we need to add leaders. The `place_leader` procedure accepts three parameters: the xy-coordinates, and the charge. Remember, the charges of line-leaders must come in pairs; if I add a leader with charge 5, I must also add a leader with charge -5.

Now that we have leaders and seekers, we must have them interact with each other. A `go` procedure will run continuously, acting as the main loop of our program. It will tell the seekers to perform their actions. The procedure that the seekers continuously execute is named `hunt`.

Seekers can be in one of two states, UNBOUND or BOUND:

Unbound seekers wander the field randomly, looking for leaders.

Bound seekers have transitioned from UNBOUND seekers when they bumped into a leader. They have acquired charge because of this.

Our `hunt` procedure must specify actions for both states.

First, let us consider the case when the seeker is in state UNBOUND. It will move a random number of steps, in a random direction, and determine if there is a leader at its new position. If there is, it will set its charge equal to the charge of the leader and transition to state BOUND.

Next, we will deal with state BOUND. Since the seeker is bound at this point, it will orient itself toward a leader that has the same charge as it. If it is an arc seeker (has a charge of 0), then it will first check if it is too close to its leader. If not, or if it is not an arc seeker, it will move forward a random number of steps up to 2. Then, it will check if there is a leader of the correct charge in the patch it settled in. If so, and it is an arc seeker, it will point in a random direction between 0 and 180 degrees, and jump 10 steps in that direction. If it is not an arc seeker, it will reverse its charge and set off in the opposite direction, seeking out the new leader.

4.2.2 Discussion

Unlike the previous model, the decentralized version exploits the Principles:

- It is highly *decentralized*. Every seeker has its own program to execute, and there is no authority telling what to do at each step, other than execute its program.

- Randomness plays a large role in the system. Before they are bound, seeker wander randomly. There is no way to tell which leader will end up with which seekers.
- The turtles are divided into 2 distinct roles, seekers and leaders. As many seekers as desired can be added, and the system will still work.

4.3 Improved Decentralized Model

Now, let's try to improve the model by adding more realism. This final model represents the complete *LiveLetters* system.

4.3.1 Description

To begin with, instead of the seekers wandering around randomly before they're bound, we can have them be attracted to leaders. To this end, we will have the leaders emit an attractive chemical. If seekers are UNBOUND, they will be attracted to this chemical.

First, we will add a variable `achem` to the patches (see Appendix D for the complete code). This will represent the amount of chemical on the patch. Next, we must have the patches diffuse their chemical to their neighbors at each step. Following that, we can instruct the seekers to follow the chemical.

A new procedure, `simmer`, is added. In this procedure, the leaders will give off some of their chemical. When `go` is called, that chemical will be diffused to its neighboring patches.

What else can be added to our model? We can make the turtles die and reproduce. The mechanism will work as follows. Seekers will have some amount of life, and as time goes on, they loose life. When their life reaches 0, they die. UNBOUND seekers cannot reproduce. BOUND seekers, on the other hand, try to reproduce if the structure they're in starts to die off. Let us give every seeker a counter variable as well. This counter is initialized to some high number. As bound seekers bump into other bound seekers, their counter is reinforced. If they don't bump into anyone, their counter decreases. When it reaches 0, they create new BOUND seekers. These seekers then go and fill in the gaps in the structure.

With these modifications, we have a truly self-maintaining system. As more seekers die, the letters become sparser. As the letters become sparse, more seekers are born. As more seekers are born, the letters become less sparse, and as a result fewer turtles are born. The result is a stable population. In a *completely decentralized, emergent* manner, the letter manages to sustain itself.

Figure 3 shows a plot of the population for 10,000 clock ticks. As you can see, the population experiences an initial burst, as the letter is filled out, and then settles into a pattern of regular oscillations. Figure 4 depicts an image of the letter after this same amount of time.

Something else to note is that we do not need a large amount of seekers to start off with. In fact, a starting population of only 6 seekers is enough to create the "AI."

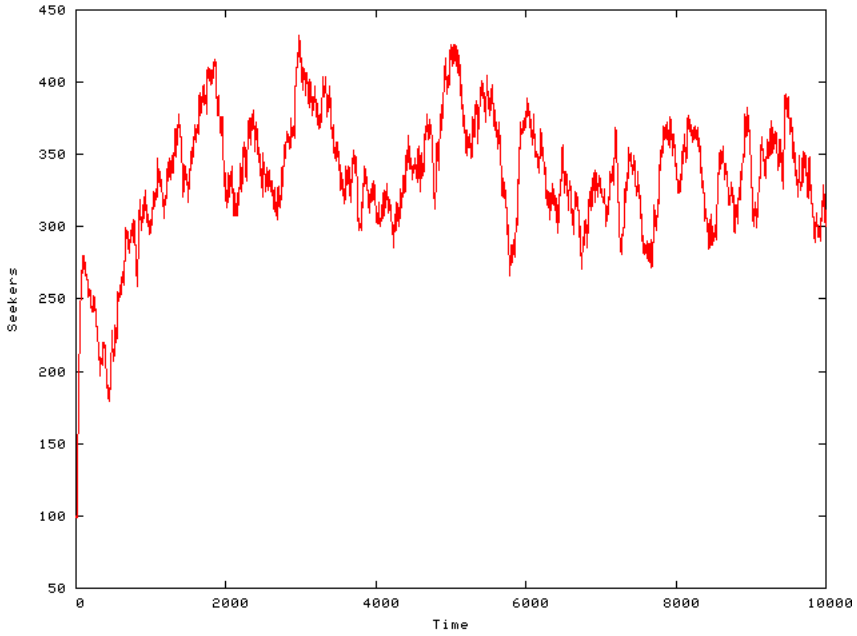

Figure 3: Population for 10,000 clock ticks with finite lifespan enabled, from an initial pool of 100 seekers.

4.3.2 Discussion

This model builds on the previous, to further support the Principles. Consider the following improvements:

1. This system is now *adaptive*, as it can deal with seeker death.
2. Localization is improved, since the leaders now emit an attractiveness chemical. Unbound seekers are able to use their local senses to travel.

5 Conclusion

The future of computing is in decentralized networking. Networks must be resilient to failures of individual parts, and we must learn to use these adaptive methods in our own designs.

Nature has shown us that there is much complexity to be found in simple designs. Systems that use decentralized methods tend to be more robust, more flexible, and more functional than systems that employ rigid centralized techniques. A tiny imperfection in a microprocessor can cause it to cease functioning entirely, while living things can lose entire body parts and still operate fine. The

Figure 4: Picture after 10,000 clock ticks with finite lifespan enabled, from an initial pool of 100 seekers.

most complex of human systems, the computer, appears to be little more than a toy compared to the sophistication of the simplest organism, and yet it is too complex.

References

[1] Ian Clarke. A distributed decentralised information storage and retrieval system. Division of Informatics, University of Edinburgh, 1999.

[2] Charles Darwin. *On the Origin of Species*. Atheneum, 1964. Originally published in 1859.

[3] Deborah M. Gordon. *Ants at Work: How an Insect Society is Organized*. The Free Press, 1999.

[4] John H. Holland. *Hidden Order: How Adaptation Builds Complexity*. Perseus Books, 1995.

[5] P. Mockapetris. RFC1034: Domain names - concepts and facilities. Technical report, Network Working Group, November 1987.

[6] Mitchel Resnick. StarLogo on the Web. `http://www.media.mit.edu/starlogo/`.

[7] Mitchel Resnick. *Turtles, Termites, and Traffic Jams*. The MIT Press, 1994.

[8] Sheldon Ross. *A First Course in Probability*. Prentice Hall, sixth edition, 2002.

[9] Herbert A. Simon. *The Sciences of the Artificial*. The MIT Press, third edition, 1996.

[10] Adam Smith. *An Inquiry into the Nature and Causes of the Wealth of Nations*. Encyclopedia Britannica, Inc., 1952. Originally published in 1776.

[11] U. Wilensky. NetLogo. `http://ccl.northwestern.edu/netlogo/`, 1999.

A NetLogo Tutorial

This Appendix gives a short NetLogo tutorial, geared toward understanding the code examples given in this paper. For a more comprehensive guide, please refer to the NetLogo Web site.

A.1 The NetLogo World

The NetLogo world is composed of a grid of *patches*, upon which *turtles* are placed. More than one turtle can reside on the same patch. For directions, straight upward is considered 0 degrees, and the angles increase in the clockwise direction.

A.2 Commands

Here are some useful NetLogo commands:

```
fd 5                        ; Tells turtle to move 5 spaces forward.

set heading 0               ; Tells turtle to change heading to 0 degrees.

create-custom-turtles 10 [ ; Creates 10 turtles, and
    fd 5                    ; has them move 5 spaces forward.
]

random 5                    ; Returns a random number between 0 and 5.

setxy 10 20                 ; Tells turtle to move to (10, 20).

random-normal 0 17          ; Returns a number in the random normal
                            ; distribution with a mean of 0 and
                            ; a standard deviation of 17.

ask turtles [hunt]          ; Tells turtles to execute the hunt procedure.

turtles with [number = 0]   ; Returns the set of turtles that have their local
```

```
                          ; number variable set to 0.

count turtles             ; Returns the size of the set of turtles.

hatch 2 [fd 5]            ; Tells turtle to create 2 other turtles,
                          ; and have them move forward 5 spaces.

uphill achem              ; Returns the direction in which the patch
                          ; variable achem is strongest.
```

A.3 Variables

Variables can be global, or owned by turtles or patches:

```
globals [x]     ; Creates a global variable x.
turtles-own [y] ; Creates a turtle variable y.
patches-own [z] ; Creates a patch variable z.
```

To assign a variable, use the `set` command:

```
set x 1 ; Sets x equal to 1.
```

You can reference a variable of an agent:

```
set x-of one-of turtles 1 ; Sets the x of a random turtle equal to 1.
```

A.4 Procedures

Procedures are functions without return values. The following creates a procedure named `proc` that prints "Hello!"

```
to proc
    print "Hello!"
end
```

Procedures can have input parameters:

```
to add [x y]
    print x + y
end
```

A.5 Control Flow

There are two conditional constructs:

```
if x = 0 [print "x = 0"]      ; Prints ‘‘x = 0’’ if x = 0.

ifelse x = 0 [print "x = 0"]  ; Prints ‘‘x = 0’’ if x = 0,
             [print "x != 0"] ; otherwise prints ‘‘x != 0’’.
```

Procedures can be called by simply using their names. The following will print "Hello!" if p2 is run:

```
to p1
    print "Hello!"
end

to p2
    p1
end
```

B Centralized Model Code Listings

```
to arc [x y r s f]
    locals [a]
    set a (f - s)
    create-custom-turtles 1000 [
        setxy x y
        set heading random (s + a)
        fd r
    ]
end

to line [x y a l]
    create-custom-seekers 500 [
        setxy x y
        set color white
        set heading a
        fd random l
    ]
end

to draw_a [x y]
    line x y 20 22.3
    line (x + 10) (y + 20) 150 22.3
    line (x + 5) (y + 10) 90 10
end

to draw_i [x y]
    line x y 90 10
    line (x + 5) y 0 20
    line (x) (y + 20) 90 10
end

to draw_ai
```

```
    draw_a -20 -10
    draw_i 5 -10
end

to draw_p [x y]
    line x y 0 20
    arc x (y + 15) 3 0 180
end
```

C Decentralized Model Code Listings

```
turtles-own [charge state]
breeds [leaders seekers]
globals [clock]

to setup
    clear-all
    add-seekers
end

to add-seekers
    create-custom-seekers tnumber [
        set shape "circle"
        setxy random-normal 0 17 random-normal 0 17
        set state 0
        set color gray
        set charge 20
    ]
end

to setup-p [x y]
    place-leader x y -9
    place-leader x (y + 20) 9
    place-leader x (y + 15) 0
end

to setup-i [x y]
    place-leader x y -6
    place-leader (x + 10) y 6
    place-leader x (y + 20) -5
    place-leader (x + 10) (y + 20) 5
    place-leader (x + 5) (y + 20) -4
    place-leader (x + 5) y 4
end
```

```
to setup-a [x y]
    place-leader x y -3
    place-leader (x + 20) y -2
    place-leader (x + 11) (y + 21) 3
    place-leader (x + 10) (y + 20) 2
    place-leader (x + 5) (y + 10) -1
    place-leader (x + 15) (y + 10) 1
end

to go
    do-plot
    ask seekers [hunt]
    set clock clock + 1
end

to do-plot
    set-current-plot "Captured Seekers"
    plot count turtles with [charge != 20]
    set-current-plot "Seeker Population"
    plot count seekers
end

to place-leader [x y val]
    create-custom-leaders 1 [
        set color white
        set charge val
        setxy x y
    ]
end

to hunt
    if state = 0 [
        fd random 5
        if count leaders-here = 1 and counter-of one-of leaders-here <= 10 [
            set charge ((charge-of one-of leaders-here) * -1)
            ifelse charge = 0 [set color pink] [
                ifelse charge > 0 [set color red] [set color yellow]
            ]
            set state 1
        ]
    ]

    if state = 1 [
        set heading towards one-of leaders with [charge = charge-of myself]

        ifelse charge = 0 [
```

```
            if distance one-of leaders with [charge = charge-of myself] > 5 [
                fd random 2
            ]
        ] [fd random 2]

        if count leaders-here = 1 [
            if (charge-of one-of leaders-here) = charge [
                set charge (charge-of one-of leaders-here * -1)
                ask leaders-here [set counter 10]
                ifelse charge = 0 [
                    set heading random 180
                    jump 10
                ] [
                    ifelse charge > 0 [set color red] [set color yellow]
                ]

                set heading (heading + 180)
            ]
        ]

    ]
end
```

D Improved Decentralized Model Code Listings

Note that only the procedures that have changed from Appendix C are listed.

```
turtles-own [charge state counter life]
patches-own [achem]

to add-seekers
    create-custom-seekers tnumber [
        set shape "circle"
        setxy random-normal 0 17 random-normal 0 17
        set state 0
        set color gray
        set charge 20
        set life random tlife
    ]
end

to go
    diffuse achem 1
    do-plot
    ask seekers [hunt]
```

```
    ask leaders [simmer]
    set clock clock + 1
    ifelse see-chem [ask patches [set pcolor scale-color gray achem 0 100]]
                    [ask patches [set pcolor black]]
end

to place-leader [x y val]
    create-custom-leaders 1 [
        set color white
        set charge val
        setxy x y
    ]
end

to hunt
    if state = 0 [
        ifelse attract [
            set heading uphill achem
        ] [set heading random 360]

        fd random 5
        if count leaders-here = 1 and counter-of one-of leaders-here <= 10 [
            set charge ((charge-of one-of leaders-here) * -1)
            ifelse charge = 0 [set counter 5 set color pink] [
                ifelse charge > 0 [set color red] [set color yellow]
            ]
            set counter 10
            set state 1
        ]
    ]

    if state = 1 [
        set heading towards one-of leaders with [charge = charge-of myself]

        ifelse charge = 0 [
            if distance one-of leaders with [charge = charge-of myself] > 5 [
                fd random 2
            ]
        ] [fd random 2]

        if count leaders-here = 1 [
            if (charge-of one-of leaders-here) = charge [
                set charge (charge-of one-of leaders-here * -1)
                ask leaders-here [set counter 10]
                ifelse charge = 0 [
                    set heading random global-angle
```

```
                jump 10
            ] [
                ifelse charge > 0 [set color red] [set color yellow]
            ]

            set heading (heading + 180)
        ]
    ]

    if death and charge != 0 [
        if count seekers-here > 1 [set counter 10]
        if counter < 0 [
            set counter 0
            hatch 2 [
                set shape "circle"
                set state 0
                set color gray
                set charge 20
                set life random tlife
            ]
        ]
        set counter counter - 2
    ]
]

set life life - (random 5)
if life <= 0 and death [die]
end

to simmer
    set achem achem + 100
end
```