Chapter 14

Complex Systems

©2005 by Harvey Gould, Jan Tobochnik, and Wolfgang Christian
27 June 2005

We introduce cellular automata, neural networks, genetic algorithms, and growing networks to explore the concepts of self-organization and complexity. Applications to sandpiles, fluids, earthquakes, and other areas are discussed.

14.1 Cellular Automata

Part of the fascination of physics is that it allows us to reduce natural phenomena to a few simple laws. It also is fascinating to think about how a few simple laws can produce the enormously rich behavior that we see in nature. In this chapter we will discuss several models that illustrate some of the new ideas that are emerging from the study of *complex systems*.

The first class of models that we will discuss are known as *cellular automata*. Cellular automata were introduced by von Neumann and Ulam in 1948 and are mathematical idealizations of dynamical systems in which space and time are discrete and the quantities of interest have a finite set of discrete values that are updated according to a local rule. A cellular automaton can be thought of as a lattice of sites or a checkerboard with colored squares (the cells). Each cell changes its state at the tick of an external clock according to a rule based on the present configuration of the cells in its neighborhood. Cellular automata are examples of discrete dynamical systems that can be simulated exactly on a digital computer.

Because the original motivation for studying cellular automata was their biological aspects, the discrete locations in space are frequently referred to as cells. More recently, cellular automata have been applied to a wide variety of physical systems ranging from fluids to galaxies. We will usually refer to sites rather then cells, except when we are explicitly discussing biological systems. The important characteristics of cellular automata include the following:

1. Space is discrete and consists of a regular array of sites. Each site has a finite set of values.

t:	111	110	101	100	011	010	001	000
t + 1:	0	1	0	1	1	0	1	0

Figure 14.1: Example of a local rule for the evolution of a one-dimensional cellular automaton. The variable at each site can have values 0 or 1. The top row shows the $2^3 = 8$ possible combinations of three sites. The bottom row gives the value of the central site at the next iteration. For example, if the value of a site is 0 and its left neighbor is 1 and its right neighbor is 0, the central site will have the value 1 in the next time step. This rule is termed 01011010 in binary notation (see the second row), the modulo-two rule, or rule 90. Note that 90 is the base ten (decimal) equivalent of the binary number 01011010, that is, $90 = 2^1 + 2^3 + 2^4 + 2^6$.

2. The rule for the new value of a site depends only on the values of a *local* neighborhood of sites near it.

3. Time is discrete. The variables at each site are updated *simultaneously* based on the values of the variables at the previous time step. Hence, the state of the entire lattice advances in discrete time steps.

We first consider one-dimensional cellular automata and assume that the neighborhood of a given site is the site itself and the sites immediately to the left and right of it. Each site is assumed to have two states (a Boolean automaton). An example of such a rule is illustrated in Figure 14.1, where we see that a rule can be labeled by the binary representation of the update rule for each of the eight possible neighborhoods and by the base ten equivalent of the binary representation. Because any eight digit binary number specifies a one-dimensional cellular automaton, there are $2^8 = 256$ possible rules.

Class `OneDimensionalAutomatonApp` takes the decimal representation of the rule as input and produces the rule array `update`, which is used to update each lattice site using periodic boundary conditions. The `OneDimensionalAutomatonApp` class manipulates numbers using their binary representation. Note the use of the bit manipulation operators `>>>` and `& (AND)` in method `setRule`. To understand how the right shift operator `>>>` works, consider the expression `13 >>> 1`. In this case the result of the shift operator is to shift the bits of the binary representation of the integer 13 to the right by one. Because the binary representation of 13 is 1101, the result of the shift operator is 0110. (The left-hand bits are filled with 0s as needed.) To understand the nature of the `&` operator, consider the expression `0110 & 1`, which we can write as `0110 & 0001`. In this case the result is `0000` because the `&` operator sets each of the the resulting bits to 1 if the corresponding bit in both operands is 1; otherwise the bit is zero.

We use the `LatticeFrame` class to represent the sites and their evolution. At a given time the sites are drawn in the horizontal direction; time increases in the vertical direction. In method `iterate` the `%` operator is used to determine the left and right neighbors of a site using periodic boundary conditions. Also note the use of the left shift operator `<<` in method `iterate`. A more complete discussion of bit manipulation is given in Section 14.6.

Listing 14.1: One-dimensional cellular automaton class.

```
package org.opensourcephysics.sip.ch14.ca;
import org.opensourcephysics.controls.*;
```

```
import org.opensourcephysics.frames.*;

public class OneDimensionalAutomatonApp extends AbstractCalculation {
   LatticeFrame automaton = new LatticeFrame("");
   int[] update = new int[8]; // update[] maps neighborhood configurations to 0 or 1

   public void calculate() {
      control.clearMessages();
      int L = control.getInt("Linear dimension");
      int tmax = control.getInt("Maximum time");
      automaton.resizeLattice(L, tmax); // default is lattice sites all zero
      // seed lattice by putting 1 in middle of first row
      automaton.setValue(L/2, 0, 1);
      // choose color of empty and occupied sites
      automaton.setIndexedColor(0, java.awt.Color.YELLOW); // empty
      automaton.setIndexedColor(1, java.awt.Color.BLUE);   // occupied
      setRule(control.getInt("Rule number"));
      for(int t = 1;t<tmax;t++) {
         iterate(t, L);
      }
   }

   public void iterate(int t, int L) {
      for(int i = 0;i<L;i++) {
         // read the neighborhood bits around index i, using periodic b.c's
         int left = automaton.getValue((i-1+L)%L, t-1);
         int center = automaton.getValue(i, t-1);
         int right = automaton.getValue((i+1)%L, t-1);
         // encode left, center, and right bits into one integer value
         // between 0 and 7
         int neighborhood = (left<<2)+(center<<1)+(right<<0);
         // update[neighborhood] gives the new site value for this neighborhood
         automaton.setValue(i, t, update[neighborhood]);
      }
   }

   public void setRule(int ruleNumber) {
      control.println("Rule = "+ruleNumber+"\n");
      control.println("111   110   101   100   011   010   001   000");
      for(int i = 7;i>=0;i--) {
         // (ruleNumber >>> i) shifts the contents of ruleNumber to the right by i
         // bits. In particular, the ith bit of ruleNumber resides in the rightmost
         // position of this expression. After "and"ing with the number 1, we are
         // left with either the number 0 or 1, depending on whether the ith
         // bit of ruleNumber was cleared or set.
         update[i] = ((ruleNumber>>>i)&1);
         control.print("  "+update[i]+"     ");
      }
      control.println();
   }
```

```
   public void reset() {
      control.setValue("Rule number", 90);
      control.setValue("Maximum time", 100);
      control.setValue("Linear dimension", 500);
   }

   public static void main(String args[]) {
      CalculationControl.createApp(new OneDimensionalAutomatonApp());
   }
}
```

The properties of all 256 one-dimensional cellular automata have been cataloged (see Wolfram, 1984). We explore some of the properties of one-dimensional cellular automata in Problems 14.1 and 14.3.

Problem 14.1. One-dimensional cellular automata

a. What is the result of `13 & 12`, `33 >> 1` (decimal representation), and 1101 & 0111 (binary representation)? Consider rule 90 and work out by hand the values of `update[]` according to method `setRule`.

b. Use `OneDimensionalAutomatonApp` and consider rule 90 shown in Figure 14.1. This rule also is known as the modulo-two rule, because the value of a site at step $t+1$ is the sum modulo 2 of its two neighbors at step t. Choose the initial configuration to be a single nonzero site (the seed) at the midpoint of the lattice. It is sufficient to consider the evolution for approximately twenty iterations. Is the resulting pattern of nonzero sites self-similar? If so, characterize the pattern by a fractal dimension.

c. Determine the properties of a rule for which the value of a site at step $t+1$ is the sum modulo 2 of the values of its neighbors plus its own value at step t. This rule is equivalent to 10010110 or rule $150 = 2^1 + 2^2 + 2^4 + 2^7$. Start with a single seed site.

d. Choose a random initial configuration for which the independent probability for each site to have the value 1 is $p = 1/2$; otherwise, the value of the site is 0. Determine the evolution of rule 90, rule 150, rule $18 = 2^1 + 2^4$ (00010010), rule $73 = 2^0 + 2^3 + 2^6$ (01001001), and rule 136 (10001000). How sensitive are the patterns that are formed to the initial conditions? Does the nature of the patterns depend on the use or nonuse of periodic boundary conditions?

Listing 14.2: A more efficient implementation of method iterate in `OneDimensionalAutomatonApp`.

```
public void iterate(int t, int L) {
   // encodes state(L-1) and state(0) in second and first bits of neighborhood variable
   int neighborhood = (automaton.getValue(L-1,t-1)<<1) + automaton.getValue(0,t-1);
   for (int i = 0; i < L; i++) {
     // clear third bit of neighborhood, but keep second and first bits
     neighborhood = neighborhood & 3;
     // shift second and first bits of neighborhood to third and second bits
     neighborhood = neighborhood << 1;
```

```
      // encode state(i+1) into first bit of neighborhood, using periodic b.c's
      neighborhood += automaton.getValue((i+1)%L, t-1);
      // neighborhood now encodes the three bits of state surrounding
      // index i at time t-1.  with neighborhood as an index, the
      // update[] table gives us the state at index i and time t.
      automaton.setValue(i,t,update[neighborhood]);
    }
}
```

Method `iterate` in class `OneDimensionalAutomatonApp` is not as efficient as possible because it does not use information about the neighborhood at site i to determine the neighborhood at site $i+1$. A more efficient implementation is given in Listing 14.2. To understand how this version of method `iterate` works, suppose that the lattice at $t = 0$ is 1011, and we want to determine the neighborhood of the site at $i = 0$. The answer is 6 in decimal, corresponding to 110 in binary. Because of periodic boundary conditions, the index to the left of $i = 0$, is $L - 1$. The expression `(automaton.getValue(L-1,t-1)<<1)` yields `001 << 1 = 010` because `<<` shifts all bits to the left. (Only 3 bits are needed to describe the neighborhood.) The statement

```
int neighborhood = (automaton.getValue(L-1,t-1)<<1) + automaton.getValue(0,t-1);
```

yields `010 + 001 = 011`. The effect of the statement

```
neighborhood = neighborhood & 3;
```

is to clear the third bit of the neighborhood, but to keep the second and first bits: 011 & 011 = 011. In this case, nothing is changed. We then shift the second and first bits of the neighborhood to the third and second bits:

```
neighborhood = neighborhood << 1;
```

and obtain `neighborhood = 110`. Finally the statement

```
neighborhood += automaton.getValue((i+1)%L, t-1);
```

gives `neighborhood = 011 + 000 = 011`, which is 2 in decimal.

*__Problem 14.2.__ Whose time is more important?

a. Work out another example to make sure that you understand the nature of the bit manipulations that are used in Listing 14.2 and in the more efficient version of method `iterate`.

b. Which version of method `iterate` would you use, the more efficient but more difficult to understand (and debug) version, or the less efficient but easier to understand version? What is more important, computer time or programmer time? In general, the answer depends on the context.

The dynamical behavior of many of the 256 one-dimensional Boolean cellular automata is uninteresting, and hence we also consider one-dimensional Boolean cellular automata with larger neighborhoods (including the site itself). Because a larger neighborhood implies that there are many more possible update rules, we place some reasonable restrictions on the rules. First, we

assume that the rules are symmetrical, for example, the neighborhood 100 produces the same value for the central site as 001. We also require that the zero neighborhood 000 yields 0 for the central site, and that the value of the central site depends only on the sum of the values of the sites in the neighborhood, for example, 011 produces the same value for the central site as 101 (see Wolfram, 1984).

A simple way of coding the rules that is consistent with these requirements is as follows. Each rule is labeled by a sequence of 0s and 1s such that the sequence indicates which sums set the central site equal to 1. If the lowest order digit is 1, then the central site is set to 1 if the sum is 0. If the next digit is 1, then the central site is set to 1 if the sum is 1, etc. For example, the rule 10110 indicates that the central site will be set to 1 if the number of neighbors equal to 1 is 1, 2, or 4.

Problem 14.3. More one-dimensional cellular automata

a. Modify class `OneDimensionalAutomatonApp` so that it incorporates the possible rules discussed in the text based on the number of sites equal to 1 in a neighborhood of $2z + 1$ sites. How many possible rules are there for $z = 1$? Choose $z = 1$ and a random initial configuration, and determine if the long time behavior for each rule belongs to one of the following categories:

   i. A homogeneous state where every site has the same value. An example is rule 1000.

   ii. A pattern consisting of separate stable or periodic regions. An example is rule 0100.

   iii. A chaotic, aperiodic pattern. An example is rule 1010.

   iv. A set of complex, localized structures that may not live forever. There are no examples for $z = 1$.

b. Modify your program so that $z = 2$. Wolfram (1984) claims that rules 010100 and 110100 are the only examples of complex behavior (category 4). Describe how the behavior of these two rules differs from the behavior of the other rules. Find at least one rule for each of the four categories.

The results of Problem 14.3 suggests that an important feature of cellular automata is their capability for self-organization. In particular, the class of complex localized structures is distinct from regular as well as aperiodic structures.

An important idea of complexity theory is that simple rules can lead to complex behavior. This complex behavior is not random, but has structure. Are there "coarse grained" descriptions that can predict the dynamical behavior of these systems, or do we have to implement the model on a computer using the dynamical rules at the lowest level of description? For example, our understanding of the flow of a fluid through a pipe would be very limited if the only way we could obtain information about the behavior of fluids was to solve the equations of motion for all the individual particles. In this case, there is a coarse grained description of fluids where the fundamental fluid variables are not the individual positions and velocities of the molecules, but rather a velocity field, which can be interpreted as a spatial average over the velocities of many particles. The resultant partial differential equation of fluid mechanics, known as the Navier-Stokes equation, provides the coarse grained description, which can be solved, in principle, to predict the motion of the fluid.

Is there an analogous coarse grained description of a cellular automaton? Israeli and Goldenfeld have found some examples for which a coarse grained description exists. We first simulate a cellular automaton that produces complex structures. Then we start with the same initial state and create a coarse grained lattice such that each of its cells is a coarse grained description of a group of cells on the original lattice. The idea is to determine a different update rule to evolve the coarse grained lattice such that the configurations of the coarse grained lattice are identical to the coarse grained configurations of the original lattice that were obtained using the original update rule. If it is possible to implement this procedure in general, we would be better able to develop theories of complex macroscopic systems without needing to know the details of the dynamics of the microscopic constituents that make up these systems. We explore two examples in Problem 14.4.

*__Problem 14.4.__ Coarse graining one-dimensional cellular automata

a. Add methods to `OneDimensionalAutomatonApp` that create a coarse grained lattice such that groups of three cells are coarse grained to 1 if all three cells are 1, and coarse grained to 0 otherwise. Allow the coarse grained lattice to evolve separately using a different update rule than the original lattice. The coarse grained lattice should be updated after every three updates of the original lattice. Draw the coarse grained lattice as a space-time diagram similar to what we have done for the original lattice, such that each cell in the coarse grained lattice is three times the size of a cell on the original lattice in both the space and time directions. Use rule 146 (10010010) for the original lattice and rule 128 (10000000) for the coarse grained lattice. Choose a lattice size L that is a multiple of 3 and run for a time that is a multiple of 3. You should see similar patterns in the two lattices, although the original lattice contains some details that are washed out by the coarse grained lattice. If you coarse grain the original lattice cells at each time step, you will obtain the same pattern as the coarse grained lattice.

b. Modify your program such that each pair of cells is coarse grained to 1 if two original cells are both 0 or both 1 and coarse grained to 0 otherwise. Use rule 105 (01101001) on the original cells with $L = 120$ for 60 iterations, and run the coarse grained system using rule 150 (10100110). You should obtain results similar to those found in part (a).

Traffic models. Physicists have been at the forefront of the development of a more systematic approach to the characterization and control of traffic. Much of this work was initiated at General Motors by Robert Herman in the late 1950s. The car-following theory of traffic flow that he and Elliott Montroll and others developed during this time is still used today. What has changed is the way we can implement these theories. The continuum approach used by Herman and Montroll is based on partial differential equations. An alternative that is more flexible and easier to understand is based on cellular automata.

We first consider a simple one lane highway where cars enter at one end and exit at the other end. To implement the Nagel-Schreckenberg cellular automaton model, we use integer arrays for the position, x_i and velocity v_i, where i indexes a car and not a lattice site. The important input parameters of the simulation are the maximum velocity, $v_{\max}$, the density of cars ρ, and the probability, p, of a car slowing down. This probability adds some randomization to the drivers. The algorithm implemented in class `Freeway` for the motion of each car at each iteration is as follows:

1. If $v_i < v_{\max}$, increase the velocity v_i of car i by one unit, that is, $v_i \to v_i + 1$. This change models the process of acceleration to the maximum velocity.
2. Compute the distance to the next car, d. If $v_i \geq d$, then reduce the velocity to $v_i = d - 1$ to prevent crashes.
3. With probability p, reduce the velocity of a moving car by one unit. Thus, $v_i \to v_i - 1$.
4. Update the position x_i of car i so that $x_i(t+1) = x_i(t) + v_i$.

This ordering of the steps ensures that cars do not overlap.

Listing 14.3: One lane freeway class.

```
package org.opensourcephysics.sip.ch14.traffic;
import java.awt.Graphics;
import org.opensourcephysics.display.*;
import org.opensourcephysics.frames.*;
import org.opensourcephysics.display2d.*;
import org.opensourcephysics.controls.*;

public class Freeway implements Drawable {
   public int[] v, x, xtemp;
   public LatticeFrame spaceTime;
   public double[] distribution;
   public int roadLength;
   public int numberOfCars;
   public int maximumVelocity;
   public double p;              // probability of reducing velocity
   private CellLattice road;
   public double flow;
   public int steps, t;
   public int scrollTime = 100; // number of time steps before scrolling space-time diagram

   public void initialize(LatticeFrame spaceTime) {
      this.spaceTime = spaceTime;
      x = new int[numberOfCars];
      xtemp = new int[numberOfCars]; // used to allow parallel updating
      v = new int[numberOfCars];
      spaceTime.resizeLattice(roadLength, 100);
      road = new CellLattice(roadLength, 1);
      road.setIndexedColor(0, java.awt.Color.RED);
      road.setIndexedColor(1, java.awt.Color.GREEN);
      spaceTime.setIndexedColor(0, java.awt.Color.RED);
      spaceTime.setIndexedColor(1, java.awt.Color.GREEN);
      int d = roadLength/numberOfCars;
      x[0] = 0;
      v[0] = maximumVelocity;
      for(int i = 1;i<numberOfCars;i++) {
         x[i] = x[i-1]+d;
         if(Math.random()<0.5) {
```

```
            v[i] = 0;
         } else {
            v[i] = 1;
         }
      }
      flow = 0;
      steps = 0;
      t = 0;
   }

   public void step() {
      for(int i = 0;i<numberOfCars;i++) {
         xtemp[i] = x[i];
      }
      for(int i = 0;i<numberOfCars;i++) {
         if(v[i]<maximumVelocity) {
            v[i]++;                                    // acceleration
         }
         int d = xtemp[(i+1)%numberOfCars]-xtemp[i]; // distance between cars
         if(d<=0) {                                    // periodic boundary conditions, d = 0
            d += roadLength;
         }
         if(v[i]>=d) {
            v[i] = d-1;                                // slow down due to cars in front
         }
         if((v[i]>0)&&(Math.random()<p)) {
            v[i]--;                                    // randomization
         }
         x[i] = (xtemp[i]+v[i])%roadLength;
         flow += v[i];
      }
      steps++;
      computeSpaceTimeDiagram();
   }

   public void computeSpaceTimeDiagram() {
      t++;
      if(t<scrollTime) {
         for(int i = 0;i<numberOfCars;i++) {
            spaceTime.setValue(x[i], t, 1);
         }
      } else {                                          //scroll diagram
         for(int y = 0;y<scrollTime-1;y++) {
            for(int i = 0;i<roadLength;i++) {
               spaceTime.setValue(i, y, spaceTime.getValue(i, y+1));
            }
         }
         for(int i = 0;i<roadLength;i++) {
            spaceTime.setValue(i, scrollTime-1, 0);    // zero last row
         }
```

```
            for(int i = 0;i<numberOfCars;i++) {
               spaceTime.setValue(x[i], scrollTime-1, 1); // add new row
            }
         }
      }

      public void draw(DrawingPanel panel, Graphics g) {
         if(x==null) {
            return;
         }
         road.setBlock(0, 0, new byte[roadLength][1]);
         for(int i = 0;i<numberOfCars;i++) {
            road.setValue(x[i], 0, (byte) 1);
         }
         road.draw(panel, g);
         g.drawString("Number of Steps = "+steps, 10, 20);
         g.drawString("Flow = "+ControlUtils.f3((double) flow/(roadLength*steps)), 10, 40);
         g.drawString("Density = "+ControlUtils.f3((double) numberOfCars/(roadLength)), 10, 60)
      }
}
```

The target class, `FreewayApp`, shows the movement of the cars and a space-time diagram, with time on the vertical axis and space on the horizontal axis. When the number of iterations equals `scrollTime`, the diagram scrolls down. The flow rate is the average of the car velocities divided by the length of the highway. Thus, two cars moving at constant velocity will have twice the flow rate of one car moving at the same velocity.

Listing 14.4: FreewayApp Class.

```
package org.opensourcephysics.sip.ch14.traffic;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.*;

public class FreewayApp extends AbstractSimulation {
   Freeway freeway = new Freeway();
   DisplayFrame display = new DisplayFrame("Freeway");
   LatticeFrame spaceTime = new LatticeFrame("space", "time", "Space Time Diagram");

   public FreewayApp() {
      display.addDrawable(freeway);
   }

   public void initialize() {
      freeway.numberOfCars = control.getInt("Number of cars");
      freeway.roadLength = control.getInt("Road length");
      freeway.p = control.getDouble("Slow down probability");
      freeway.maximumVelocity = control.getInt("Maximum velocity");
      display.setPreferredMinMax(0, freeway.roadLength, -3, 4);
      freeway.initialize(spaceTime);
   }
```

```
    public void doStep() {
        freeway.step();
    }

    public void reset() {
        control.setValue("Number of cars", 10);
        control.setValue("Road length", 50);
        control.setValue("Slow down probability", 0.5);
        control.setValue("Maximum velocity", 2);
        control.setValue("Steps between plots", 1);
        enableStepsPerDisplay(true);
    }

    public void resetAverages() {
        freeway.flow = 0;
        freeway.steps = 0;
    }

    public static void main(String[] args) {
        SimulationControl control = SimulationControl.createApp(new FreewayApp());
        control.addButton("resetAverages", "resetAverages");
    }
}
```

Problem 14.5. Cellular automata traffic models

a. Run `FreewayApp` for 10 cars on a road of length 50, with $v_{\max} = 2$ and $p = 0.5$. Allow the system to evolve before recording the flow rate. Repeat the simulation with a different initial configuration at least several more times to estimate the uncertainty in the data. Repeat for 1, 2, 5, 20, 30, and 40 cars. Plot the flow rate versus the density. This plot is called the *fundamental diagram*. Explain its qualitative shape. At what density do traffic jams begin to occur?

b. Repeat part (a) with a road of length 500 and the same car densities. Use other road lengths to determine the minimum road length needed to obtain results that are independent of the length of the road.

c. Add methods to your classes to compute the velocity and gap distributions, where the gap is defined as the distance between two cars.

d. For a fixed road length compare your results for $v_{\max} = 1$ with your results for $v_{\max} = 2$. Also consider $v_{\max} = 5$. Are there any qualitative differences in the behavior of the cars?

e. Explore the effect the speed reduction probability by considering $p = 0.2$ and $p = 0.8$.

f. Add on- and off-ramps separated by a fixed distance. One way to do so is to choose a car at random and have it slow down as it approaches the off-ramp and exits. To maintain a constant density, allow a car to enter the on-ramp whenever a car leaves the highway. What is the effect of adding the on- and off-ramps?

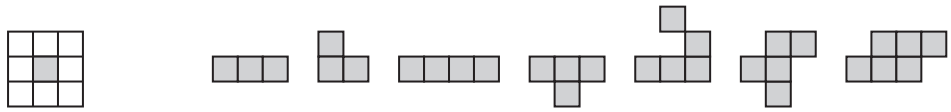

Figure 14.2: (a) The local neighborhood of a site in the Game of Life is given by the sum of its eight neighbors. (b) Examples of initial configurations for the Game of Life, some of which lead to interesting patterns. Live cells are shaded.

g. Modify your program to simulate a two lane highway. You will need to choose rules for moving from one lane to the other. Some possibilities to explore include the following. One reason for a car to move to the left lane is that the car is moving at less than the maximum speed and cannot increase its speed due to the car in front of it. Such a car could move to the left lane if there were a free space to the left. One reason for a car to move to the right lane is that there is a car immediately behind it. How does the behavior of the two lane highway differ from that of the one lane highway?

h. Modify your two lane simulation so that there are two kinds of vehicles (for example, cars and trucks) with different values of $v_{\max}$. How do the gap and velocity distributions change? Compute separate values for the truck and car flows as well as the total flow. Compute the average speed of the trucks and compare it with that of cars.

Because one-dimensional cellular automata models are limited, we consider several two-dimensional models. The philosophy is the same except that the neighborhood contains more sites. For the eight neighbor sites shown in Figure 14.2a, there are $2^9 = 512$ possible configurations for the eight neighbors and the center site, and 2^{512} possible rules. Clearly, we cannot go through all these rules in any systematic fashion as we did for one-dimensional cellular automata. For this reason, we will choose our rules based on other considerations.

The Game of Life. The rules used in `LifeApp` implement a popular two-dimensional cellular automaton known as the *Game of Life*. This model, invented in 1970 by the mathematician John Conway, produces many fascinating patterns. The rules of the game are simple. For each cell determine the sum of the values of its four nearest and four next-nearest neighbors (see Figure 14.2a). A "live" cell (value 1) remains alive only if this sum equals 2 or 3. If the sum is greater than 3, the cell will "die" (become 0) at the next iteration due to overcrowding. If the sum is less than 2, the cell will die due to isolation. A dead cell will come to life only if the sum equals 3.

Listing 14.5: Implementation of the Game of Life.

```
package org.opensourcephysics.sip.ch14.ca;
import org.opensourcephysics.frames.*;
import org.opensourcephysics.controls.*;
import java.awt.Color;

public class LifeApp extends AbstractSimulation {
   LatticeFrame latticeFrame = new LatticeFrame("Game of Life");
   byte[][] newCells;
```

```
int size = 16;

public LifeApp() {
   latticeFrame.setToggleOnClick(true, 0, 1);
   latticeFrame.setIndexedColor(0, Color.RED);
   latticeFrame.setIndexedColor(1, Color.BLUE);
}

public void initCells(int size) {
   this.size = size;
   newCells = new byte[size][size];
   latticeFrame.setAll(newCells, 0, size, 0, size);
   latticeFrame.setValue(size/2, size/2, 1);
   latticeFrame.setValue(size/2-1, size/2, 1);
   latticeFrame.setValue(size/2+1, size/2, 1);
   latticeFrame.setValue(size/2, size/2-1, 1);
   latticeFrame.setValue(size/2, size/2+1, 1);
}

public void clear() {
   latticeFrame.setAll(new byte[size][size]);
   latticeFrame.repaint();
}

public void reset() {
   control.println("Click in drawingPanel to toggle life.");
   control.setValue("grid size", 16);
   initCells(16);
}

public void initialize() {
   initCells(control.getInt("grid size"));
}

private int calcNeighborsPeriodic(int row, int col) {
   int neighbors = -latticeFrame.getValue(row, col); // do not count self
   row += size; // add the size so that the mod operator works for row=0 and col=0
   col += size;
   for(int i = -1;i<=1;i++) {
      for(int j = -1;j<=1;j++) {
         neighbors += latticeFrame.getValue((row+i)%size, (col+j)%size);
      }
   }
   return neighbors;
}

public void doStep() {
   for(int i = 0;i<size;i++) {
      for(int j = 0;j<size;j++) {
         newCells[i][j] = 0;
```

```
            }
         }
         for(int i = 0;i<size;i++) {
            for(int j = 0;j<size;j++) {
               switch(calcNeighborsPeriodic(i, j)) {
               case 0 :
               case 1 :
                  newCells[i][j] = 0;                                    //dies
                  break;
               case 2 :
                  newCells[i][j] = (byte) latticeFrame.getValue(i, j); //life goes on
                  break;
               case 3 :
                  newCells[i][j] = 1;                                    //condition for birth
                  break;
               default :
                  newCells[i][j] = 0;                                    // dies of overcrowding
               }
            }
         }
         latticeFrame.setAll(newCells);
      }

      /* ---------------- application target ---------------- */
      public static void main(String[] args) {
         OSPControl control = SimulationControl.createApp(new LifeApp());
         control.addButton("clear", "Clear"); // optional custom action
      }
   }
```

Problem 14.6. The Game of Life

a. `LifeApp` allows the user to determine the initial configuration interactively by clicking on a cell to change its value before hitting the Start button. Choose several initial configurations with a small number of live cells and determine the different types of patterns that emerge. Some suggested initial configurations are shown in Figure 14.2b. Does it matter whether you use fixed or periodic boundary conditions? Use a 16×16 lattice.

b. Modify `LifeApp` so that each cell is initially alive with a 50% probability. Use a 32×32 lattice. What types of patterns typically result after a long time? What happens for 20% live cells? What happens for 70% live cells?

c. Assume that each cell is initially alive with probability p. Given that the density of live cells at time t is $\rho(t)$, what is $\rho(t+1)$, the expected density at time $t+1$? Do the simulation and plot $\rho(t+1)$ versus $\rho(t)$. If $p = 0.5$, what is the steady-state density of live cells?

d.* `LifeApp` has not been optimized for the Game of Life and is written so that other rules can be implemented easily. Rewrite `LifeApp` so that it uses bit manipulation (see Section 14.6).

The Game of Life is an example of a universal computing machine. That is, we can choose an initial configuration of live cells to represent any possible program and any set of input data, run the Game of Life, and the output data will appear in some region of the lattice. The proof of this result (see Berlekamp et al.) involves showing how various configurations of cells represent the components of a computer, including wires, storage, and the fundamental components of a CPU – the digital logic gates that perform *and*, *or*, and other logical and arithmetic operations. Other cellular automata also can be shown to be universal computing machines.

14.2 Self-Organized Critical Phenomenon

Very large events such as a magnitude eight earthquake, an avalanche on a snow covered mountain, the sudden collapse of an empire (for example, the Soviet Union), or the crash of the stock market are rare. When such events occur, are they due to some special set of circumstances or are they part of a more general pattern of events that would occur without any specific external intervention? The idea of *self-organized criticality* is that in many cases the occurrence of very large events does not depend on special conditions or external forces and is due to the intrinsic dynamics of the system.

If s represents the magnitude of an event, such as the energy released in an earthquake or the amount of snow in an avalanche, then a system is said to be *critical* if the number of events, $N(s)$, follows a power law:

$$N(s) \sim s^{-\alpha}. \qquad \text{(no characteristic scale)} \tag{14.1}$$

If $\alpha \approx 1$, the form (14.1) implies that there would be one large event of size 1000 for every 1000 events of size one. One implication of the power law form (14.1) is that there is no characteristic scale, and the system is said to be *scale invariant*. This terminology reflects the fact that power laws look the same on all scales. For example, the replacement $s \to bs$ in the function $N(s) = As^{-\alpha}$ yields a function $\widetilde{N}(s)$ that is indistinguishable from $N(s)$, except for a change in the amplitude A by the factor $b^{-\alpha}$.

Contrast the nature of the power law dependence of $N(s)$ in (14.1) to the result of combining a large number of independently acting random events. In this case we know that the distribution of the sum is a Gaussian (see Problem 7.15), and $N(s)$ has the form

$$N(s) \sim e^{-(s/s_0)^2}. \qquad \text{(characteristic scale)} \tag{14.2}$$

Scale invariance does not hold for functions that decay as in (14.2), because the replacement $s \to bs$ in the function $e^{-(s/s_0)^2}$ changes s_0 (the characteristic scale or size of s) by the factor b. Note that for a power law distribution, there are events of all sizes, but for a Gaussian distribution, there are practically speaking no events much larger than the characteristic scale s_0. For example, if we take $s_0 = 100$, there would be one large event of size 1000 for every 2.7×10^{43} events of size one!

A simple example of self-organized critical phenomena is an idealized sandpile. Suppose that we construct a sandpile by randomly adding one grain at a time onto a flat surface with open edges. Initially, the grains will remain where they land, but after we add more grains there will be small avalanches during which the grains move so that the local slope of the pile is not too big. Eventually, the pile will reach a statistically stationary (time-independent) state, and the amount of sand added will balance the sand that falls off the edge (on the average). When a single grain of

sand is added to such a configuration, a rearrangement might occur that triggers an avalanche of any size (up to the size of the system), so that the mean slope again equals the critical value. We say that the statistically stationary state is critical because there are avalanches of all sizes. The stationary state is *self-organized* because no external parameter (such as the temperature) needs to be tuned to force the system to this state. In contrast, the concentration of fissionable material in a nuclear chain reaction has to be carefully controlled for the nuclear chain reaction to become critical.

We consider a two-dimensional model of a sandpile, and represent the height at site i by the array element `height[i]`. One grain of sand is added to a random site, j, `height[j]++`, at each iteration. If `height[j]` $= 4$, then we remove the four grains from site j and distribute them equally to its nearest neighbors. A site whose height is equal to four is said to *topple*. If any of the neighbors now have four grains of sand, they topple as well. This process continues until all sites have less than four grains of sand. Grains that fall outside the lattice are lost forever.

Class `Sandpile` implements this idealized model. The lattice is stored in a `LatticeFrame` and the arrays `toppleSiteX` and `toppleSiteY` store the coordinates of the sites with four grains of sand. The array `distribution` accumulates the data for the number of sites that topple at each addition of a grain of sand to the pile. It is possible, though rare, that a site will topple more than once in one step. Hence, the number of toppled sites may be greater than the number of sites in the lattice.

Physically, it is not the actual height that determines toppling, but the mean local slope between a site and its nearest neighbors. Thus, what we call the "height" really should be called the "slope." However, in the literature many authors use the term "height."

Listing 14.6: Implementation of the two-dimensional sandpile model.

```
package org.opensourcephysics.sip.ch14.sandpile;
import java.awt.Graphics;
import org.opensourcephysics.frames.*;

public class Sandpile {
   int[] distribution; // distribution of number of sites toppling
   int[] toppleSiteX, toppleSiteY;
   LatticeFrame height;
   int L, numberToppledMax;
   int numberToppled, numberOfSitesToTopple, numberOfGrains;

   public void initialize(LatticeFrame height) {
      this.height = height;
      height.resizeLattice(L, L); // create new lattice
      numberToppledMax = 2*L*L+1;                // size of distribution array
      distribution = new int[numberToppledMax]; // should use histogramframe
      toppleSiteX = new int[L*L];
      toppleSiteY = new int[L*L];
      numberOfGrains = 0;
      resetAverages();
   }

   public void step() {
```

```
      numberOfGrains++;
      numberToppled = 0;
      int x = (int) (Math.random()*L);
      int y = (int) (Math.random()*L);
      int h = height.getValue(x, y)+1;
      height.setValue(x, y, h); // add grain to random site
      height.render();
      if(h==4) { // topple grain
         numberOfSitesToTopple = 1;
         boolean unstable = true;
         int[] siteToTopple = {x, y};
         while(unstable) {
            unstable = toppleSite(siteToTopple);
         }
      }
      distribution[numberToppled]++;
   }

   public boolean toppleSite(int siteToTopple[]) { // topple site
      numberToppled++;
      int x = siteToTopple[0];
      int y = siteToTopple[1];
      numberOfSitesToTopple--;
      height.setValue(x, y, height.getValue(x, y)-4); // remove grains from site
      height.render();
      // add grains to neighbors
      // if (x,y) is on the border of the lattice, then some grains will be lost.
      if(x+1<L) {
         addGrain(x+1, y);
      }
      if(x>0) {
         addGrain(x-1, y);
      }
      if(y+1<L) {
         addGrain(x, y+1);
      }
      if(y>0) {
         addGrain(x, y-1);
      }
      if(numberOfSitesToTopple>0) {
         siteToTopple[0] = toppleSiteX[numberOfSitesToTopple-1]; // next site to topple
         siteToTopple[1] = toppleSiteY[numberOfSitesToTopple-1];
         return true;
      } else {
         return false;
      }
   }

   public void addGrain(int x, int y) {
      int h = height.getValue(x, y)+1;
```

```
      height.setValue(x, y, h); // add grain to site
      height.render();
      if(h==4) { // new site to topple
         toppleSiteX[numberOfSitesToTopple] = x;
         toppleSiteY[numberOfSitesToTopple] = y;
         numberOfSitesToTopple++;
      }
   }

   public void resetAverages() {
      distribution = new int[numberToppledMax];
      numberOfGrains = 0;
   }
}
```

Listing 14.7: The target class for the two-dimensional sandpile model.

```
package org.opensourcephysics.sip.ch14.sandpile;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.*;

public class SandpileApp extends AbstractSimulation {
   Sandpile sandpile = new Sandpile();;
   LatticeFrame height = new LatticeFrame("x", "y", "Sandpile");
   PlotFrame plotFrame = new PlotFrame("ln s", "ln N", "Distribution of toppled sites");

   public SandpileApp() {
      height.setIndexedColor(0, java.awt.Color.WHITE);
      height.setIndexedColor(1, java.awt.Color.BLUE);
      height.setIndexedColor(2, java.awt.Color.GREEN);
      height.setIndexedColor(3, java.awt.Color.RED);
      height.setIndexedColor(4, java.awt.Color.BLACK);
   }

   public void initialize() {
      sandpile.L = control.getInt("L");
      height.setPreferredMinMax(0, sandpile.L, 0, sandpile.L);
      sandpile.initialize(height);
   }

   public void doStep() {
      sandpile.step();
   }

   public void stop() {
      super.stop();
      plotFrame.clearData();
      for(int s = 1;s<sandpile.distribution.length;s++) {
         if(sandpile.distribution[s]>0) {
            plotFrame.append(0, Math.log(s),
```

```
                                Math.log(sandpile.distribution[s]*1.0/sandpile.numberOfGrains));
            }
         }
         plotFrame.render();
      }

      public void reset() {
         control.setValue("L", 10);
         enableStepsPerDisplay(true);
      }

      public void resetAverages() {
         sandpile.resetAverages();
      }

      public static void main(String[] args) {
         SimulationControl control = SimulationControl.createApp(new SandpileApp());
         control.addButton("resetAverages", "resetAverages");
      }
}
```

Problem 14.7. A two-dimensional sandpile model

a. Use the classes `Sandpile` and `SandpileApp` to simulate a two-dimensional sandpile with linear dimension L. Run the simulation with $L = 10$, and stop it once toppling starts to occur. When this behavior occurs, black cells (with four grains) will momentarily appear. Use the Step button to watch individual toppling events, and obtain a qualitative sense of the dynamics of the sandpile model.

b. Comment out the `height.render()` statements in `Sandpile`, and add a statement to `SandPileApp` so that the number of grains added to the system is displayed. (The number of grains added is a measure of the number of configurations that are included in the various averages.) Now you will not be able to see individual toppling events, but you can more quickly collect data on the toppling distribution, the frequency of the number of sites that topple when a grain is added. The program outputs a log-log plot of the distribution. Estimate the slope of the log-log distribution from the part of the plot that is linear and thus determine the power law exponent α. Reset the averages and repeat your calculation to obtain another estimate of α. If your two estimates of α are within a few percent, you have added enough grains of sand. Compute α for $L = 10$, 20, 40, and 80. As you make the lattice size larger, the range over which the log-log plot is linear should increase. Explain why the plot is not linear for large values of the number of toppled sites.

Of course, the model of a sandpile in Problem 14.7 is over simplified. Laboratory experiments indicate that real sandpiles show power law behavior if the piles are small, but that larger sandpiles do not (see Jaeger et al.).

Earthquakes. The empirical Gutenberg-Richter law for $N(E)$, the number of earthquakes with energy release E, is consistent with power law behavior:

$$N(E) \sim E^{-b}, \tag{14.3}$$

with $b \approx 1$. The magnitude of earthquakes on the Richter scale is approximately the logarithm of the energy release. This power law behavior does not necessarily hold for individual fault systems, but holds reasonably accurately when all fault systems are considered. One implication of the power law dependence in (14.3) is that there is nothing special about large earthquakes. In Problems 14.8 and 14.9 and Project 14.26 we explore some models of earthquake models.

Given the long time scales between earthquakes, there is considerable interest in simulating models of earthquakes. The Burridge-Knopoff model considered in Project 14.26 consists of a system of coupled masses in contact with a rough surface. The masses are subjected to static and dynamic friction forces due to the surface, and also are pulled by an external force corresponding to slow tectonic plate motion. The major difficulty with this model is that the numerical solution of the corresponding equations of motion is computationally intensive. For this reason we consider several cellular automaton models that retain some of the basic physics of the Burridge-Knopoff model.

Problem 14.8. A simple earthquake model

Define the real variable $F(i,j)$ on a square lattice, where F represents the force or stress on the block at position (i,j). The initial state of the lattice at time $t = 0$ is found by assigning small random values to $F(i,j)$. The lattice is updated according to the following rules:

(i) Increase F at every site by a small amount ΔF, for example, $\Delta F = 10^{-3}$, and increase the time t by 1. This increase represents the effect of the driving force due to the slow motion of the tectonic plate.

(ii) Check if $F(i,j)$ is greater than F_c, the threshold value of the force. If not, the system is stable and step 1 is repeated. If the system is unstable, go to step 3. Choose $F_c = 4$ for convenience.

(iii) The release of stress due to the slippage of a block is represented by letting $F(i,j) = F(i,j) - F_c$. The transfer of stress is represented by updating the stress at the sites of the four neighbors at $(i, j \pm 1)$ and $(i \pm 1, j)$: $F \to F + 1$. Periodic boundary conditions are not used.

These rules are equivalent to the Bak-Tang-Wiesenfeld model. What is the relation of this model to the sandpile model considered in Problem 14.7?

As an example, choose $L = 10$. Do the simulation and show that the system eventually comes to a statistically stationary state, where the average value of the stress at each site stops growing. Monitor $N(s)$, the number of earthquakes of size s, where s is the total number of sites (blocks) that are affected by the instability. Then consider $L = 30$ and repeat your simulations. Are your results for $N(s)$ consistent with scaling?

Problem 14.9. A dissipative earthquake model

The Bak-Tang-Wiesenfeld earthquake model discussed in Problem 14.8 displays power law scaling due to the inherent conservation of the dynamical variable, the stress. It is easy to modify the model so that the stress is not conserved and the model is more realistic. The Rundle-Jackson-Brown/Olami-Feder-Christensen model of a earthquake fault is a simple example of such a nonconservative system.

a. Modify the toppling rule in Problem 14.8 so that when the stress on site (i, j) exceeds F_c, not all the excess stress is given to the neighbors. In particular, assume that when site (i, j) topples, $F(i, j)$ is reduced to the residual stress $F_r(i, j)$. The amount $\alpha(F_{ij} - F_r)$ is dissipated leaving $(F_{ij} - F_r)(1 - \alpha)$ to be distributed equally to the neighbors. If $\alpha = 0$, the model is equivalent to the model considered in Problem 14.8. Choose $\alpha = 0.2$ and determine if $N(s)$ exhibits power law scaling. For simplicity, choose $F_c = 4$ and $F_r = 1$ (see Grassberger).

b. Make the model more realistic by adding a small amount of noise to F_r so that F_r is uniformly distributed between $1 - \delta, 1 + \delta$ with $\delta = 0.05$. Also run the model in what is called the "zero-velocity limit" by finding the site with the maximum stress $F_{\max}$ and then increasing the stress on all sites by $F_c - F_{\max}$ so that only one site initially becomes unstable. Determine $N(s)$ and see if your results differ from what you found in part (a). Do you still observe power law scaling?

c. The model can be made more realistic still by assuming that the interaction between the blocks is long range due to the existence of elastic forces. Distribute the excess stress equally to all z neighbors that are within a distance of radius R of an unstable site. Each of the z neighbors receives a stress equal to $(F_{ij} - F_r)(1 - \alpha)/z$. First choose $R = 3$ and see if the qualitative behavior of $N(s)$ changes as R becomes larger. Lattices with $L \geq 256$ are typically considered with $R \simeq 30$ (see Rundle et al.).

The behavior of some other simple models of natural phenomena is explored in the following.

Problem 14.10. Forest fire model

a. Consider the following model of the spread of a forest fire. Suppose that at $t = 0$ the $L \times L$ sites of a square lattice either have a tree or are empty with probability p and $1 - p$ respectively. The sites that have a tree are on fire with probability f. At each iteration an empty site grows a tree with probability g, a tree that has a nearest neighbor site on fire catches fire, and a site that is already on fire dies and becomes empty. This model is an example of a probabilistic cellular automaton. Write a program to simulate this model and color code the three types of sites. Use periodic boundary conditions.

b. Choose $L \geq 30$ and determine the values of g for which the forest maintains fires indefinitely. Note that as long as $g > 0$, new trees will always grow.

c. Use the value of g that you found in part (b) and compute the distribution of the number of sites s_f on fire. If the distribution is critical, determine the exponent α that characterizes this distribution. Also compute the distribution for the number of trees, s_t. Is there any relation between these two distributions?

d.* To obtain reliable results it is frequently necessary to average over many initial configurations. However, the behavior of many systems is independent of the initial configuration and averaging over many initial configurations is unnecessary. This latter possibility is called *self-averaging*. Repeat parts (b) and (c), but average your results over ten initial configurations. Is this forest fire model self-averaging?

Problem 14.11. Another forest fire model

Consider a simple variation of the model discussed in Problem 14.10. At $t = 0$ each site is occupied by a tree with probability p; otherwise, it is empty. The system is updated in successive iterations as follows:

(i) Randomly grow new trees at time t with a small probability g from sites that are empty at time $t - 1$;

(ii) A tree that is not on fire at $t - 1$ catches fire due to lightning with probability f;

(iii) Trees on fire ignite neighboring trees, which in turn ignite their neighboring trees, etc. The spreading of the fire occurs instantaneously.

(iv) Trees on fire at time $t - 1$ die (become empty sites) and are removed at time t (after they have set their neighbors on fire).

As in Problem 14.10, the changes in each site occur synchronously.

a. Determine $N(s)$, the number of clusters of trees of size s that catch fire in each iteration. Two trees are in the same cluster if they are nearest neighbors. Is the behavior of $N(s)$ consistent with $N(s) \sim s^{-\alpha}$? If so, estimate the exponent α for several values of g and f.

b.* The balance between the mean rate of birth and burning of trees in the steady state suggests a value for the ratio f/g at which this model is likely to be scale invariant. If the average steady state density of trees is ρ, then at each iteration the mean number of new trees appearing is $gN(1 - \rho)$, where $N = L^2$ is the total number of sites. In the same spirit, we can say that for small f, the mean number of trees destroyed by lightning is $f\rho N\langle s\rangle$, where $\langle s\rangle$ is the mean number of trees in a cluster. Is this reasoning consistent with the results of your simulation? If we equate these two rates, we find that $\langle s\rangle \sim [(1 - \rho]/\rho)(g/f)$. Because $0 < \rho < 1$, it follows that $\langle s\rangle \to \infty$ in the limit $f/g \to 0$. Given the relation $\langle s\rangle = \sum_{s=1}^{\infty} sN(s)/\sum_s N(s)$ and the divergent behavior of $\langle s\rangle$, why does it follow that $N(s)$ must decay more slowly than exponentially with s? This reasoning suggests that $N(s) \sim s^{-\alpha}$ with $\alpha < 2$. Is this expectation consistent with the results that you obtained in part (a)?

In this model there are three well separated time scales, that is, the time for lightning to strike ($\propto f^{-1}$), the time for trees to grow ($\propto g^{-1}$), and the instantaneous spreading of fire through a connected cluster. This separation of time scales seems to be an essential ingredient for self-organized criticality (see Grinstein and Jayaprakash).

Problem 14.12. Model of punctuated equilibrium

a. The idea of *punctuated equilibrium* is that biological evolution occurs episodically rather than as a steady, gradual process. That is, most of the major changes in life forms occur in relatively short periods of time. Bak and Sneppen have proposed a simple model that exhibits some of the behavior of punctuated equilibrium. The model consists of a one-dimensional cellular automaton of linear dimension L, where cell i represents the biological fitness of species i. Initially, all cells receive a random fitness f_i between 0 and 1. Then the cell with the lowest

fitness and its two nearest neighbors are randomly given new fitness values. This update rule is repeated indefinitely. Write a program to simulate the behavior of this model. Use periodic boundary conditions, and display the fitness of each cell as a column of height f_i. Begin with $L = 64$ and describe what happens to the distribution of fitness values after a long time.

b. We can crudely think of the update process as replacing a species and its neighbors by three new species. In this sense the fitness represents a barrier to creating a new species. If the barrier is low, it is easier to create a new species. Do the low fitness species die out? What is the average value of fitness of the species after the model is run for a long time (10^4 or more iterations)? Compute the distribution of fitness values, $N(f)$, averaged over all cells and over many iterations. Allow the system to come to a fluctuating steady state before computing $N(f)$. Plot $N(f)$ versus f. Is there a critical value f_c below which $N(f)$ is much less than the values above f_c? Is the update rule reasonable from a evolutionary point of view?

c. Modify your program to compute the distance x between successive fitness changes and the distribution of these distances, $P(x)$. Make a log-log plot of $P(x)$ versus x. Is there any evidence of self-organized criticality (power law scaling)?

d. Another way to visualize the results is to make a plot of the time at which a cell is changed versus the position of the cell. Is the distribution of the plotted points approximately uniform? We might expect that the survival time of a species depends exponentially on its fitness, and hence each update corresponds to an elapsed time of e^{-cf_i}, where the constant c sets the time scale and f_i is the fitness of the cell that has been changed. Choose $c = 100$ and make a similar plot with the time axis replaced by the logarithm of the time, that is, the quantity $100f_i$. Is this plot more meaningful?

e. Another way of visualizing punctuated equilibrium is to plot the number of times groups of cells change as a function of time. Divide the time into units of 100 updates and compute the number of fitness changes for cells $i = 1$ to 10 as a function of time. Do you see any evidence of punctuated equilibrium?

14.3 The Hopfield Model and Neural Networks

Neural network models have been motivated in part by how neurons in the brain collectively store and recall memories. Usually, a neuron is in one of two states, a resting potential (not firing), or firing at the maximum rate. A neuron "fires" once it receives electrical inputs from other neurons whose strength reaches a certain threshold. An important characteristic of a neuron is that its output is a nonlinear function of the sum of its inputs. The assumption is that when memories are stored in the brain, the strengths of the connections between neurons change.

One of the uses of neural network models is pattern recognition. If we see someone more than once, the person's face provide input that helps us to recall the person's name. In the same spirit, a neural network can be given a pattern, for example, a string of ± 1s, that partially reflect a previously memorized pattern. The idea is to store memories so that a computer can recall them when the inputs are close to a particular memory.

We now consider an example of a neural network due to Hopfield. The network consists of N neurons and the state of the network is defined by the state of each neuron, S_i, which in the Hopfield model takes on the values -1 (not firing) and $+1$ (firing). The strength of the connection between neuron i and neuron j is denoted by w_{ij}, which is determined by the M stored memories:

$$w_{ij} = \sum_{s=1}^{M} S_i^\alpha S_j^\alpha, \tag{14.4}$$

where S_i^α represents the state of neuron i in stored memory α. Given the initial state of all the neurons, the dynamics of the network is simple. We choose a neuron i at random and change its state according to its input, which is $\sum_{i\neq j} w_{ij} S_j$, where S_j represents the current state of neuron j. Then we change the state of neuron i by setting

$$S_i = \begin{cases} +1, & \text{for } \sum_{i\neq j} w_{ij} S_j > 0 \\ -1, & \text{for } \sum_{i\neq j} w_{ij} S_j \leq 0. \end{cases} \tag{14.5}$$

The threshold value of the input has been set equal to zero, but other values could be used as well.

The `HopfieldApp` class in Listing 14.8 implements this model of a neural network and stores memories based on user input. The state of the network is stored in the array `S[i]` and the connections between the neurons are stored in the array `w[i][j]`. The user initially clicks on various cells to toggle their values between -1 and $+1$ and presses the Remember button to store a pattern. Then the user presses the Randomize button to initialize the S_i by setting S_i to ± 1 at random. After the memories are stored, press the Start button to update the neurons using the Hopfield algorithm to try to recall one of the stored memories.

Listing 14.8: `HopfieldApp` class.

```
package org.opensourcephysics.sip.ch14;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.*;

public class HopfieldApp extends AbstractSimulation { // Hopfield model of a neural network
   LatticeFrame lattice;
   int N;          // total number of neurons
   double[][] w; // connection array (N by N elements)
   int numberOfStoredMemories;

   public HopfieldApp() {
      lattice = new LatticeFrame("Hopfield state");
      lattice.setToggleOnClick(true, -1, 1);
      lattice.setIndexedColor(-1, java.awt.Color.blue);
      lattice.setIndexedColor(0, java.awt.Color.blue);
      lattice.setIndexedColor(1, java.awt.Color.green);
      lattice.setSize(600, 120);
   }

   public void doStep() {
      int[] S = lattice.getAll();
```

```
      for(int counter = 0;counter<N;counter++) {
         int i = (int) (N*Math.random()); // chooses random neuron index
         double sum = 0;
         for(int j = 0;j<N;j++) {
            sum += w[i][j]*S[j];
         }
         S[i] = (sum>0) ? 1 : -1;
      }
      lattice.setAll(S);
   }

   public void initialize() {
      N = control.getInt("Lattice size");
      w = new double[N][N];
      lattice.resizeLattice(N, 1);
      for(int i = 0;i<N;i++) {
         lattice.setAtIndex(i, -1);
      }
      lattice.setMessage("# memories = "+(numberOfStoredMemories = 0));
   }

   public void reset() {
      control.setValue("Lattice size", 8);
      lattice.setMessage("# memories = "+(numberOfStoredMemories = 0));
   }

   public void addMemory() {
      int[] S = lattice.getAll();
      for(int i = 0;i<N;i++) {
         for(int j = i+1;j<N;j++) {
            w[i][j] += S[i]*S[j];
            w[j][i] += S[i]*S[j];
         }
      }
      lattice.setMessage("# memories = "+(++numberOfStoredMemories));
   }

   public void randomizeState() {
      for(int i = 0;i<N;i++) {
         lattice.setAtIndex(i, Math.random()<0.5 ? -1 : 1);
      }
      lattice.repaint();
   }

   public static void main(String args[]) {
      SimulationControl control = SimulationControl.createApp(new HopfieldApp());
      control.addButton("addMemory", "Remember");
      control.addButton("randomizeState", "Randomize");
   }
}
```

Problem 14.13. Memory recall in the Hopfield model

a. Use the `HopfieldApp` class to explore the ability of the Hopfield neural network to store and recall memories. Begin with $N = 10$ neurons and click on the cells to choose a pattern to remember. Then click on the Randomize button to randomize the spins. Does the neural network find a pattern similar to the one you saved? Consider other values of N and various patterns to obtain a feel for how the algorithm works.

b. Store two memories of 20 bits, for example, $11111\bar{1}\bar{1}\bar{1}1\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}11111$ and $11\bar{1}\bar{1}11\bar{1}\bar{1}11\bar{1}\bar{1}11\bar{1}\bar{1}11\bar{1}\bar{1}$, where we have written -1 as $\bar{1}$. Try to recall a memory using the input $11111111\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}11111111$. This input is similar to the first memory. Record the Hamming distance between the final state and the closest memory, where the Hamming distance is the number of bits that differ between two strings. Repeat this procedure for several different values of the number of neurons and the memory length.

c. Estimate how many memories can be stored for a given number of neurons before recall becomes severely reduced. Make estimates for $N = 10$, 20, and 40. What is your criteria for the recall to be considered correct?

d. In the Hopfield model every neuron is linked to every other neuron. (The value of the links w_{ij} is determined by the stored memories.) Is the spatial dimension of the system relevant? Describe how the `HopfieldApp` class can store two-dimensional patterns.

Neural networks also can be used for difficult optimization problems. In Problem 14.14 we consider the problem of finding the minimum energy of a model *spin glass*. The latter is a magnetic analog of an ordinary glass in which the positions of the molecules are not ordered as in a crystal. In a spin glass the local magnetic moment is disordered because random magnetic interactions are "frozen in" and do not change. The simplest model of a spin glass is based on the simplest model of magnetism, the Ising model (see Section 16.5). In the Ising model the magnetic moment is represented by a spin s_i which can take on two values, ± 1. The spins are located on the sites of a lattice. Each spin is assumed to interact with all other spins, and the total energy of the system is given by

$$E = -\sum_{i,j \neq i} J_{ij} V_i V_j, \tag{14.6}$$

where the sum is over all pairs of spins. We have let $w \to J$ so that the notation is the same as the Ising model. If $J_{ij} > 0$, the spins i and j lower their energy by lining up in the same direction. If $J_{ij} < 0$, the spins lower their energy by lining up in opposite directions (see Figure 16.1).

We are interested in finding the ground state when the coupling constant J_{ij} randomly takes on the values $\pm J_0/N$, where N is the number of spins and J_0 is an arbitrary constant. To find the ground state we need to find the configurations of spins that give the lowest value of the energy. Finding the ground state of a spin glass is particularly difficult because there are many configurations that correspond to local minima of the energy. In fact the problem of finding the exact ground state is an example of a computationally difficult problem called *NP-complete*. (Another example of such a problem is considered in Problem 16.31.) In Problem 14.14 we explore if the Hopfield algorithm can find a good approximation to the global minimum.

Problem 14.14. Minimum energy of an Ising spin glass

a. Choose $J_0 = 4$ in (14.6) and modify the `HopfieldApp` class so that it applies to a model spin glass. Display the output string and the energy after every N attempts to change a spin. Begin with $N = 20$.

b. What happens to the energy after a long time? For different initial states, but the same set of the J_{ij}, is the value of the energy the same after the system has evolved for a long time? Explain your results in terms of the number of local energy minima.

c. What is the behavior of the system? Do you find periodic behavior, or random behavior, or does the system evolve to a state that does not change?

14.4 Growing Networks

A network is a collection of points called nodes that are connected by lines called links. Mathematicians refer to networks as graphs, and graph theory has been an active field of mathematics for many years. A mathematical network can represent an actual network by defining what a node represents, and the kind of relationship represented by a link. For example, in an airline network the nodes represent airports and the links represent flights between airports. In an acquaintance network, the nodes represent individuals, and the links represent the state of two people knowing each other. In a biochemical network the nodes represent various molecular types, and the links represent a reaction between molecules.

One reason for the recent interest in networks is that data on existing networks is now more readily available due to the widespread use of computers. Indeed, one of the networks of current interest is the network of websites. Another reason for the interest in networks is that some new models of networks have been developed.

We first discuss one of the original network models, the Erdös-Rényi model. In this model we start with N nodes and then form n links between pairs of nodes such that each pair has either one link or no links. The probability of a link between any pair of nodes is $p = n/(N(N-1)/2)$. One quantity of interest is the degree distribution, $D(\ell)$, which is the fraction of nodes that have ℓ links. An example of the determination of $D(\ell)$ is shown in Figure 14.3. In the Erdös-Rényi model this distribution is a Poisson distribution for large N. Thus, there is a peak in $D(\ell)$, and for large ℓ, $D(\ell)$ decreases exponentially.

In some network models there is a path between any pair of nodes. In other models, such as the Erdös-Rényi model, there are some nodes that cannot be reached from other nodes (see Figure 14.3). In these networks there are other quantities of interest that are analogous to those in percolation theory. The main difference is that in network models the position of the nodes is irrelevant, and only their connectivity is relevant. In particular, there is no spanning cluster as can exist in percolation models. Instead, there can be a cluster that is significantly larger than the other clusters. In the Erdös-Rényi model the transition at which such a "giant" cluster appears depends on the probability p that any pair of nodes is connected. In the large N limit this transition occurs at $p = 1/N$.

Problem 14.15. The Erdös-Rényi model

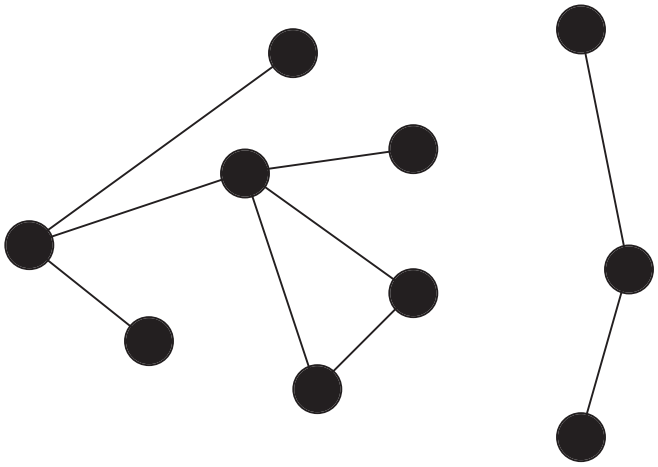

Figure 14.3: Example of a disconnected network with 10 nodes and 9 links. The degree distribution for this network is $D(1) = 5/10 = 0.5$, $D(2) = 3/10 = 0.3$, $D(3) = 1/10 = 0.1$, and $D(4) = 1/10 = 0.1$. The cluster coefficient or transitivity is defined as 3 times the number of triangles divided by the number of possible triples of connected nodes. In this case we have 1 triangle and 12 triples. Thus, the clustering coefficient equals $3 \times 1/12 = 0.25$. If a node has ℓ links, then the number of triples centered at that node is $\ell/(2!(\ell-2)!)$.

a. Write a program to create networks based on the Erdös-Rényi model. Choose $N = 100$ and $p \approx 0.01$, and compute $D(\ell)$; average over at least 10 networks. Show that $D(\ell)$ follows a Poisson distribution.

b. Define a giant cluster as one that has over three times as many nodes as any other cluster and at least 10% of the nodes. Find the value of p at which the giant cluster first appears for $N = 64$, 128, and 256. Average over 10 networks for each value of N. The cluster distribution should be updated after every link is added using the labeling procedure used in Chapter 12. In this case it is easier, because every time we add a link we either combine two clusters or we make no change in the cluster distribution.

Some of the networks that we will consider are by definition connected. In these cases one of the important quantities of interest is the mean path length between two nodes, where the path length between two nodes is the shortest number of links from one node to the other. If the mean path length weakly depends on the total number of nodes and is small, then this property of networks is known as the "small world" property. A well known example of the small world property is what is called "six degrees of separation," which refers to the fact that almost any person is connected through a sequence of six connections to almost any other person.

We wish to understand the structure of different networks. One structural property is the clustering coefficient or transitivity. If node A is linked to B and B to C, the clustering coefficient is the probability that A is linked to C (see Figure 14.3 for a precise definition). If this coefficient is large, then there will be many small loops of nodes in the network. If we think of the nodes as people and the links as friendship connections, then the clustering coefficient is a measure of

the tendency of people to form cliques. It also is of interest to see to what extent the network is hierarchically organized. Can we find groups of nodes that are linked together at different levels of organization? Can we produce an organizational chart for the network similar to what is used by many businesses? Algorithms for computing the hierarchical or community structure of a network are discussed in the references.

Two popular network models are the Watts-Strogatz small world model and the Barabasi-Albert preferential attachment model. In the Watts-Strogatz model a regular lattice of nodes connected by nearest neighbor links is "rewired" so that a link between two neighboring nodes is broken with probability p, and a link is randomly added between one of the nodes and any other node in the system. The small world property shows up as a logarithmic dependence of the mean path length on the system size N for large p. The degree distribution is similar to that of the Erdös-Rényi model.

In the preferential attachment model we begin with a few connected nodes and then add one node at a time. Each new node is then linked to m existing nodes, with preference given to those nodes that already have many links. The probability of a node with ℓ links being connected to a new node is proportional to ℓ. For example, if we have ten nodes in the network with 1, 1, 3, 2, 7, 3, 4, 7, 10, and 2 links, respectively, then there are a total of 40 links and the probability of getting the next link from a new node is 1/40, 1/40, 3/40, 2/40, 4/40, 3/40, 4/40, 7/40, 10/40, and 2/40, respectively. The result of this growth rule is that some nodes will accumulate many links. The key result is that the link distribution is a power law with $D(\ell) \sim \ell^{-\alpha}$. This *scale-free* behavior is very important because it says there in the limit of an infinite network, there is a non-negligible probability that a node exists with any particular number of links. Examples of real networks that have this behavior are actor networks where the links correspond to two actors appearing in the same movie, airport networks, the internet, and the links between various web sites. In addition to the scale free degree distribution, the preferential attachment model also has the small world property that the mean path length grows only logarithmically with the number of nodes.

The `PreferentialAttachmentModel` class implements the preferential attachment model. Method `setPosition` is not relevant to the actual growth model. It places the nodes in random positions so that the network can be drawn so that the nodes are too close to each other. This drawing method is useful only for networks less than about 100 nodes.

Listing 14.9: PreferentialAttachmentModel class: Preferential attachment network model.

```
package org.opensourcephysics.sip.ch14.networks;
import java.awt.Color;
import java.awt.Graphics;
import org.opensourcephysics.frames.*;
import org.opensourcephysics.display.Drawable;
import org.opensourcephysics.display.DrawingPanel;

public class PreferentialAttachment implements Drawable {
   int[] node, linkFrom, degree;
   double[] x, y;                  // positions of nodes, only meaningful for display purposes
   int N;                          // maximum number of nodes
   int m = 2;                      // number of attempted links per node
   int linkNumber = 0;             // twice current number of links
   int n = 0;                      // current number of nodes
```

```
boolean drawPositions = true; // only draw network if true
int numberOfCompletedNetworks = 0;

public void initialize() {
   degree = new int[N];           // degree distribution to be averaged over many network
   numberOfCompletedNetworks = 0; // will be drawing many networks
   startNetwork();
}

public void addLink(int i, int j, int s) {
   linkFrom[i*m+s] = j;
   node[i]++;
   node[j]++;
   linkNumber += 2; // twice current number of links
}

public void startNetwork() {
   n = 0;
   linkFrom = new int[m*N];
   node = new int[N];
   x = new double[N];
   y = new double[N];
   linkNumber = 0;
   for(int i = 0;i<=m;i++) {
      n++;
      setPosition(i);
   }
   for(int i = 1;i<m+1;i++) {
      for(int j = 0;j<i;j++) {
         addLink(i, j, j);
      }
   }
}

public void setPosition(int i) {
   double r2min = 1000./N;
   boolean ok = true; // used to insure two nodes are not drawn too close to each other
   do {
      ok = true;
      x[i] = Math.random()*100;
      y[i] = Math.random()*100;
      int j = 0;
      while(j<i&&ok) {
         double dx = x[i]-x[j];
         double dy = y[i]-y[j];
         double r2 = dx*dx+dy*dy;
         if(r2<r2min) {
            ok = false;
         }
         j++;
```

```
        }
    } while(!ok);
}

public int findNode(int i, int s) {
    boolean ok = true;
    int j = 0;
    do {
        ok = true;
        int k = (int) (1+Math.random()*linkNumber);
        j = -1;
        int sum = 0;
        do {
            j++;
            sum += node[j];
        } while(k>sum);
        for(int r = 0;r<s;r++) {
            if(linkFrom[i*m+r]==j) {
                ok = false;
            }
        }
    } while(!ok);
    return j;
}

public void addNode(int i) {
    n++;
    if(drawPositions) {
        setPosition(i);
    }
    for(int s = 0;s<m;s++) {
        addLink(i, findNode(i, s), s);
    }
}

public void step() {
    if(n<N) {
        addNode(n);
    } else {
        numberOfCompletedNetworks++;
        for(int i = 0;i<n;i++) { // accumulate data for degree distribution
            degree[node[i]]++;
        }
        startNetwork();           // start another network
    }
}

public void degreeDistribution(PlotFrame plot) {
    plot.clearData();
    for(int i = 1;i<N;i++) {
```

```
            if(degree[i]>0) {
               plot.append(0, Math.log(i), Math.log(degree[i]*1.0/(N*numberOfCompletedNetworks)
            }
         }
      }

      public void draw(DrawingPanel panel, Graphics g) {
         if(node!=null&&drawPositions) {
            int pxRadius = Math.abs(panel.xToPix(1.0)-panel.xToPix(0));
            int pyRadius = Math.abs(panel.yToPix(1.0)-panel.yToPix(0));
            g.setColor(Color.green);
            for(int i = 0;i<n;i++) {
               int xpix = panel.xToPix(x[i]);
               int ypix = panel.yToPix(y[i]);
               for(int s = 0;s<m;s++) {
                  int j = linkFrom[i*m+s];
                  int xpixj = panel.xToPix(x[j]);
                  int ypixj = panel.yToPix(y[j]);
                  g.drawLine(xpix, ypix, xpixj, ypixj);          // draw link
               }
            }
            g.setColor(Color.red);
            for(int i = 0;i<n;i++) {
               int xpix = panel.xToPix(x[i])-pxRadius;
               int ypix = panel.yToPix(y[i])-pyRadius;
               g.fillOval(xpix, ypix, 2*pxRadius, 2*pyRadius); // draw node
            }
         }
      }
   }
```

Problem 14.16. Preferential attachment model

a. Write a target class that uses the `PreferentialAttachmentModel` class and continuously creates new networks until stopped by the user (so we can compute averages over many networks). To speed up the computation, make it possible to optionally display the networks. The program should output the average degree distribution, $D(\ell)$.

b. Estimate the exponent α defined by $D(\ell) \sim \ell^{-\alpha}$ for $N = 100$ and $m = 2$. Repeat for $N = 500$. Does the exponent α change? If time permits, consider $N = 10000$. Does α depend on m?

c. Modify `PreferentialAttachmentModel` so that the ℓ links are made randomly so that the number of links a node already has is irrelevant to adding a link. What functional form does the link distribution have now? Is this model equivalent to the Erdös-Rényi model?

d.* Write a method to compute the clustering coefficient, which is defined in Figure 14.3. Plot $\ln C(N)$ versus $\ln N$ for both the preference attachment model and the Erdös-Rényi model. Compare and discuss your results in terms of the visual appearance of the networks.

Problem 14.17. Watts-Strogatz network

a. Write a class to create a Watts-Strogatz network. Begin with $N = 100$ nodes which you can visualize as equally spaced on a circle. (Their actual position is irrelevant.) Place links between the $2m$ nearest neighbors. Thus, if $m = 1$, then only the nearest neighbors are linked. If $m = 2$, then the nearest and next nearest neighbors are linked. Then write a method to go through each link and then with probability p, break the link connection at one end and reconnect it to another node at random.

b. Compute the degree distribution as a function of m for several values of p. Discuss your results.

c. As we increase p, the networks becomes more and more random. There is a transition from a network where the path length $\ell \sim N$ to one where $\ell \sim \ln N$. This transition occurs when $Np^{1/d} \sim 1$, where d is the dimension of the original lattice before rewiring ($d = 1$ for a circle). Draw a number of networks with different values of N and p and use this visualization to explain the dependence of ℓ on N.

d.* Write a method to compute the clustering coefficient, C. Plot C versus $\ln p$ for $N = 100$ and $m = 2$. Repeat for larger N.

Problem 14.18. A model of a social network

In many social situations we notice groups of people who interact closely with each other, but not necessarily with other groups. Usually, those in a group have some common interest or personal attribute. How can we model this situation? People do not usually become friends with other people just because they have many friends already (the preferential attachment mechanism). Instead, they choose someone to interact with and a friendship is established with some probability. A simple model is given by the following rule. As each node is added to a system, choose m existing nodes at random, and with probability p establish a link. This process will create a number of clusters of linked nodes. We can imagine that there is a possibility of a phase transition between the existence of a giant cluster that contains a large fraction of the nodes, and a situation where all the clusters are small. This model was analyzed by Zalányi et al.

a. Write a class to model this random attachment model and compute the degree distribution as well as the cluster distribution. Consider at least $N = 1000$ nodes and measure $D(\ell)$, the degree distribution, for $m = 2$, 3, and 5 and $p = 0.1$ and $p = 0.9$. Average over at least ten trials. You should not find power law behavior for $D(\ell)$. Explain why this behavior is expected.

b. Compute $D(\ell, t)$, the number of links connected to a node, as a function of t, the time when the node is added. We would expect nodes added in the beginning to have more links than those at the end. Describe and discuss the functional form of $D(\ell, t)$.

c. Consider $m = 5$ and generate many networks for different values of p. Determine the cluster distribution. A giant cluster exists when the largest cluster is at least three times larger than the next largest cluster. Estimate the value of p for which the giant cluster first appears. You should find an approximate power law cluster distribution only at the transition. What is the exponent of the power law?

d. How does the value of p at the transition change with m? Explain your results.

e. Consider $m = 1$ and generate networks for many values of p. Determine the cluster distribution. You should find an approximate power law distribution for all values of p. What are the exponents for the power law? Why do you think there is not a phase transition for $m = 1$? Consider the possibility of two clusters merging for different values of m.

14.5 Genetic Algorithms

Many people find it difficult to accept that evolution is sufficiently powerful to generate the biological complexity observed in nature. Part of this difficulty arises from the inability of humans to intuitively grasp time scales that are much greater than their own lifetimes. Another reason is that it is very difficult to appreciate how random changes can lead to emergent complex structures. Genetic algorithms provide one way of understanding the nature of evolution. Their principal utility at present is in optimization problems, but they also are being used to model biological and social evolution.

Historically, developments in physics such as x-ray crystallography and quantum mechanics have lead to developments in biology. In recent years developments in biology as well as in computer science and other areas have had a direct impact on developments in physics. Genetic algorithms are an example of the influence of ideas in biology impacting ideas in physics.

The idea of genetic algorithms is to model the process of evolution by natural selection. This process involves two steps: random changes in the genetic code during reproduction and selection according to some fitness criteria. In biological organisms the genetic code is stored in the DNA. We will store the genetic code as a string of 1s and 0s. The genetic code constitutes the *genotype*. The conversion of this string to the organism or *phenotype* depends on the problem. The selection criteria is applied to the phenotype.

First we describe how change is introduced into the genotype. Typically, nature changes the genetic code in two ways. The most obvious, but less often used method, is mutation. Mutation corresponds to changing a character at random in the genetic code string from 0 to 1 or from 1 to 0. The other much more powerful method is associated with sexual reproduction. We take two strings, remove a piece from one string and exchange it with the same length piece from the other string. For example, if string $A = 0011001010$ and string $B = 0001110001$, then exchanging the piece from position 4 to position 7 leads to two new strings $A' = 0011110010$ and $B' = 0001001001$. This type of change is called recombination or crossover.

At each generation we produce changes using recombination and mutation. We then select from the enlarged population of strings (including strings from the previous generation), a new population for the next generation. Usually, a constant population size is maintained from one generation of strings to the next.

We next have to choose a selection criterion. If we want to model an actual ecosystem, we can include a physical environment and other sets of populations corresponding to different species. The fitness could depend on the interaction of the different species with one another, the interaction within each species, and the interaction with the physical environment. In addition, the behavior of the populations might change the environment from one generation to the next. For simplicity, we will consider only a single population of strings, a simple phenotype, and a simple criteria for fitness.

The phenotype we consider is a variant of the Ising model considered in Problem 14.14. We consider a square lattice of linear dimension L occupied by $N = L^2$ spins that have the values $s_i = \pm 1$. The energy of the system is given by

$$E = -\sum_{i,j=\text{nn(i)}} J_{ij} s_i s_j, \tag{14.7}$$

where the sum is over all pairs of spins that are nearest neighbors. The energy function in (14.7) assumes that only nearest neighbor spins interact, in contrast to the energy function in (14.6) which assumes that every spin interacts with every other spin. The coupling constants J_{ij} are either $+1$, -1, or distributed according to some probability distribution. If we assume that $|J_{ij}| = 1$, then the minimum energy equals $-2N$ and the maximum energy is $2N$. Because we want the fitness to be positive, we choose $2N - E$ as the measure of fitness, and take the probability of selecting a particular string with energy E for the next generation to be proportional to the fitness $2N - E$.

How does a genotype become "expressed" as a phenotype? A genotype consists of a string of length N with 1s and 0s. The lattice site (i, j) corresponds to the nth position in the string where $n = jL + i$. If the character in the string at position n is 0, then the spin at site (i, j) equals -1. If the character is 1, then the spin equals $+1$. Note that in this case the representation of the genotype is very similar to that of the phenotype. In particular, they have the same size, N, and each "piece" can have only two values. In general, the expression of the genotype in the phenotype is much more complicated. Usually, a sequence within the genotype corresponds to one value in the phenotype, which in biological systems is related to the coding for a specific protein. Such a sequence is what we call a gene.

We now have all the ingredients we need to apply the genetic algorithm. The `GeneticApp` class obtains the various parameters, initializes the population of genotypes, and calls the various methods needed to evolve the gene pool (see the `doStep` method). The `GenePool` class carries out the evolution. In method `recombine` two genotypes are chosen at random, and a random piece of one is exchanged for the equivalent piece of the other. In method `mutate` a random position in a randomly selected genotype is changed. We use a boolean array to represent the genotype, so that a change represents converting `true` to `false` or vice versa. In both methods we do not replace the original genotype, but instead add a new genotype to the population. The `Phenotype` class determines the fitness of each member of the population by computing the energy of the lattice of spins corresponding to each member of the population. Members of this population are selected for the new generation by generating a discrete nonuniform probability distributions as discussed in Section 11.5.

Listing 14.10: The `GeneticApp` class.

```
package org.opensourcephysics.sip.ch14.genetic;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.*;

public class GeneticApp extends AbstractSimulation {
   GenePool genePool = new GenePool();
   Phenotype phenotype = new Phenotype();
   DisplayFrame frame = new DisplayFrame("Gene pool");

   public void initialize() {
```

```
      phenotype.L = control.getInt("Lattice size");
      genePool.populationNumber = control.getInt("Population size");
      genePool.recombinationRate = control.getInt("Recombination rate");
      genePool.mutationRate = control.getInt("Mutation rate");
      genePool.genotypeSize = phenotype.L*phenotype.L;
      genePool.initialize(phenotype);
      phenotype.initialize();
      frame.addDrawable(genePool);
      frame.setPreferredMinMax(-1.0, genePool.genotypeSize+5, -1.0, genePool.populationNumber
      frame.setSize(phenotype.L*phenotype.L*10, genePool.populationNumber*20);
   }

   public void doStep() {
      genePool.evolve();
      phenotype.determineFitness(genePool);
      phenotype.select(genePool);
      control.clearMessages();
      control.println(genePool.generation+" generations, best fitness = "+phenotype.bestFitne
   }

   public void reset() {
      control.setValue("Lattice size", 8);
      control.setValue("Population size", 20);
      control.setValue("Recombination rate", 10);
      control.setValue("Mutation rate", 4);
   }

   public static void main(String args[]) {
      SimulationControl.createApp(new GeneticApp());
   }
}
```

Listing 14.11: The `GenePool` class.

```
package org.opensourcephysics.sip.ch14.genetic;
import java.awt.Color;
import java.awt.Graphics;
import org.opensourcephysics.display.*;

public class GenePool implements Drawable {
   int populationNumber;
   int numberOfGenotypes;
   int recombinationRate;
   int mutationRate;
   int genotypeSize;
   boolean[][] genotype;
   int generation = 0;
   Phenotype phenotype;

   public void initialize(Phenotype phenotype) {
```

```
        this.phenotype = phenotype;
        generation = 0;
        numberOfGenotypes = populationNumber+2*recombinationRate+mutationRate;
        genotype = new boolean[numberOfGenotypes][genotypeSize];
        for(int i = 0;i<populationNumber;i++) {
            for(int j = 0;j<genotypeSize;j++) {
                if(Math.random()>0.5) {
                    genotype[i][j] = true;                    //sets genes randomly
                }
            }
        }
    }

    public void copyGenotype(boolean a[], boolean b[]) { // copy a to b
        for(int i = 0;i<genotypeSize;i++) {
            b[i] = a[i];
        }
    }

    public void recombine() {
        for(int r = 0;r<recombinationRate;r += 2) {
            int i = (int) (Math.random()*populationNumber);          // chooses random genotype
            int j = 0;
            do {
                j = (int) (Math.random()*populationNumber);          // chooses second random g
            } while(i==j);
            int size = 1+(int) (0.5*genotypeSize*Math.random());     // random size to recombin
            int startPosition = (int) (genotypeSize*Math.random()); // random location
            int r1 = populationNumber+r;                             // index for new genotype
            int r2 = populationNumber+r+1;                           // index for second new ge
            copyGenotype(genotype[i], genotype[r1]);
            copyGenotype(genotype[j], genotype[r2]);
            for(int position = startPosition;position<startPosition+size;position++) {
                int pbcPosition = position%genotypeSize;
                genotype[r1][pbcPosition] = genotype[j][pbcPosition]; // make new genotypes
                genotype[r2][pbcPosition] = genotype[i][pbcPosition];
            }
        }
    }

    public void mutate() {
        int index = populationNumber+2*recombinationRate; // index for new genotype
        for(int m = 0;m<mutationRate;m++) {
            int n = (int) (Math.random()*populationNumber);       // choice random existing gen
            int position = (int) (genotypeSize*Math.random());    // random position to mutate
            copyGenotype(genotype[n], genotype[index+m]);         // copy genotype
            genotype[index+m][position] = !genotype[n][position]; // mutate
        }
    }
```

```
   public void evolve() {
      recombine();
      mutate();
      generation++;
   }

   public void draw(DrawingPanel panel, Graphics g) {
      // draws genotype as string of red or green squares and lists fitness for each genotype
      if(genotype==null) {
         return;
      }
      if(phenotype.selectedPopulationFitness==null) {
         return;
      }
      int sizeX = Math.abs(panel.xToPix(0.8)-panel.xToPix(0));
      int sizeY = Math.abs(panel.yToPix(0.6)-panel.yToPix(0));
      for(int n = 0;n<populationNumber;n++) {
         int ypix = panel.yToPix(1.5*n)-sizeY;
         for(int position = 0;position<genotypeSize;position++) {
            if(genotype[n][position]) {
               g.setColor(Color.red);
            } else {
               g.setColor(Color.green);
            }
            int xpix = panel.xToPix(position)-sizeX;
            g.fillRect(xpix, ypix, sizeX, sizeY);
         }
         g.setColor(Color.black);
         g.drawString(String.valueOf(phenotype.selectedPopulationFitness[n]),
                     panel.xToPix(genotypeSize+1), ypix+sizeY);
      }
   }
}
```

Listing 14.12: The `Phenotype` class.

```
/*  population of phenotypes (random bond Ising model)  */
package org.opensourcephysics.sip.ch14.genetic;
public class Phenotype {
   int L;
   int[][][] J; // random bonds
   int[] populationFitness, selectedPopulationFitness;
   int totalFitness;
   int highestEnergy;
   int bestFitness;

   public void initialize() {
      J = new int[L][L][2];
      highestEnergy = 2*L*L; // highest possible energy
      bestFitness = 0;
```

```
      for(int i = 0;i<L;i++) {
         for(int j = 0;j<L;j++) {
            for(int bond = 0;bond<2;bond++) {
               if(Math.random()>0.5) {
                  J[i][j][bond] = 1;
               } else {
                  J[i][j][bond] = -1;
               }
            }
         }
      }
   }

   public void determineFitness(GenePool genePool) {
      totalFitness = 0;
      int state[][] = new int[L][L];
      populationFitness = new int[genePool.numberOfGenotypes];
      for(int n = 0;n<genePool.numberOfGenotypes;n++) {
         for(int i = 0;i<L;i++) {
            for(int j = 0;j<L;j++) { // sets up lattice based on genotype
               int position = i+j*L;
               if(genePool.genotype[n][position]) {
                  state[i][j] = 1;
               } else {
                  state[i][j] = -1;
               }
            }
         }
         for(int i = 0;i<L;i++) {
            for(int j = 0;j<L;j++) { // compute energy of lattice configuration
               populationFitness[n] -= state[i][j]
                                       *(J[i][j][0]*state[(i+1)%L][j]+J[i][j][1]*state[i][(j-
            }
         }
         // define fitness to be positive and low energy -> high fitness
         populationFitness[n] = highestEnergy-populationFitness[n];
         totalFitness += populationFitness[n];
      }
   }

   public void select(GenePool genePool) {
      selectedPopulationFitness = new int[genePool.numberOfGenotypes];
      boolean savedGenotype[][] = new boolean[genePool.numberOfGenotypes][genePool.genotypeS
      for(int n = 0;n<genePool.numberOfGenotypes;n++) {
         genePool.copyGenotype(genePool.genotype[n], savedGenotype[n]);
      }
      for(int n = 0;n<genePool.populationNumber;n++) {
         int fitnessFraction = (int) (Math.random()*totalFitness);
         int choice = 0;
         int fitnessSum = populationFitness[0];
```

```
            while(fitnessSum<fitnessFraction) {
               choice++;
               fitnessSum += populationFitness[choice];
            }
            selectedPopulationFitness[n] = populationFitness[choice];
            if(selectedPopulationFitness[n]>bestFitness) {
               bestFitness = selectedPopulationFitness[n];
            }
            genePool.copyGenotype(savedGenotype[choice], genePool.genotype[n]);
         }
      }
   }
```

Problem 14.19. Ground state of Ising-like models

a. Use the genetic algorithm we have discussed to find the ground state of the ferromagnetic Ising model for which $J_{ij} = 1$. In this case the ground state energy is $E = -2L^2$ (all spins up or all spins down). It will be necessary to modify method `Initialize` in class `Phenotype`. Choose $L = 4$ and consider a population of 20 strings, with 10 recombinations and 4 mutations per generation. How long does it take to find the ground state energy? You might wish to modify the program so that each new generation is shown on the screen so that you can look at the new generations as they appear.

b. Find the mean number of generations needed to find the ground state for $L = 4$, 6, and 8. Repeat each run several times. Use a population of 100, a recombination rate of 50, and a mutation rate of 20. Are there any general trends as L is increased? How do your results change if you double the population size? What happens if you double the recombination rate or mutation rate? Use larger lattices if you have sufficient computer resources.

c. Repeat part (b) for the antiferromagnetic model for which $J_{ij} = -1$.

d. Repeat part (b) for a spin glass for which $J_{ij} = \pm 1$ at random. In this case we do not know the ground state energy in advance. What criterion can you use to terminate a run?

One of the important features of the genetic algorithm is that the change in the genetic code is selected not in the genotype directly, but in the phenotype. Note that the way we change the strings (particularly with recombination) is not closely related to the two-dimensional lattice of spins. We could have used some other prescription for converting a string of 0s and 1s to a configuration of spins on a two-dimensional lattice. If the phenotype is a three-dimensional lattice, we could use the same procedure for modifying the genotype, but a different prescription for converting the genetic sequence (the string of 0s and 1s) to the phenotype (the three-dimensional lattice of spins). The point is that it is not necessary for the genetic coding to mimic the phenotypic expression. This point becomes distorted in the popular press when a gene is tied to a particular trait, because specific pieces of DNA rarely correspond directly to any explicitly expressed trait in the phenotype.

velocity vector	direction	symbol	abbreviation	decimal	binary
$\mathbf{v}_0$	$(1, 0)$	RIGHT	RI	1	00000001
$\mathbf{v}_1$	$(1, -\sqrt{3})/2$	RIGHT_DOWN	RD	1	00000010
$\mathbf{v}_2$	$-(1, \sqrt{3})/2$	LEFT_DOWN	LD	4	00000100
$\mathbf{v}_3$	$(-1, 0)$	LEFT	LE	8	00001000
$\mathbf{v}_4$	$(-1, \sqrt{3})/2$	LEFT_UP	LU	16	00010000
$\mathbf{v}_5$	$(1, \sqrt{3})/2$	RIGHT_UP	RU	32	00100000
$\mathbf{v}_6$	$(0, 0)$	STATIONARY	S	64	01000000
		BARRIER		128	10000000

Table 14.1: Summary of the possible velocities and their representations.

14.6 Lattice Gas Models of Fluid Flow

We now return to cellular automaton models and discuss one of their more interesting applications – simulations of fluid flow. In general, fluid flow is very difficult to simulate because the partial differential equation describing the flow of incompressible fluids, the Navier-Stokes equation, is nonlinear, and this nonlinearity can lead to the failure of standard numerical algorithms. In addition, there are typically many length scales that must be considered simultaneously. These length scales include the microscopic motion of the fluid particles, the length scales associated with fluid structures such as vortices, and the length scales of macroscopic objects such as pipes or obstacles. Because of these considerations, simulations of fluid flow based on the direct numerical solutions of the Navier-Stokes equation typically require very sophisticated numerical methods (cf. Oran and Boris).

Cellular automaton models of fluids are known as *lattice gas* models. In a lattice gas model the positions of the particles are restricted to the sites of a lattice, and the velocities are restricted to a small number of vectors corresponding to neighbor sites. A time step is divided into two substeps. In the first substep the particles move freely to their corresponding nearest neighbor lattice sites. Then the velocities of the particles at each lattice site are changed according to a collision rule that conserves mass (particle number), momentum, and kinetic energy. The purpose of the collision rules is not to accurately model microscopic collisions, but rather to achieve the correct macroscopic behavior. The idea is that if we satisfy the conservation laws associated with microscopic collisions, then we can find the correct physics at the macroscopic level, including translational and rotational invariance, by averaging over many particles.

We assume a triangular lattice, because it can be shown that this symmetry is sufficient to yield the macroscopic Navier-Stokes equations for a continuum. In contrast, the more limited symmetry of a square lattice is not sufficient. Three-dimensional models are much more difficult to implement and justify theoretically.

All the moving particles are assumed to have the same speed and mass. The possible velocity vectors lie only in the direction of the nearest neighbor sites, and hence there are six possible velocities as summarized in Table 14.1. A rest particle also is allowed. The number of particles at each site moving in a particular direction (channel) is restricted to be zero or one.

In the first substep all particles move in the direction of their velocity to a neighboring site. In the second substep the velocity vectors at each lattice site are changed according to the appropriate

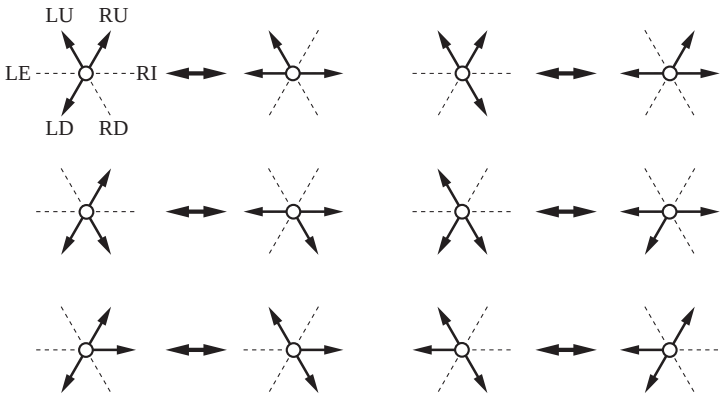

Figure 14.4: Examples of collision rules for three particles, with one particle unchanged and no stationary particles. Each direction or channel is represented by 32 bits, but we need only the first 8 bits. The various channels are summarized in Table 14.1.

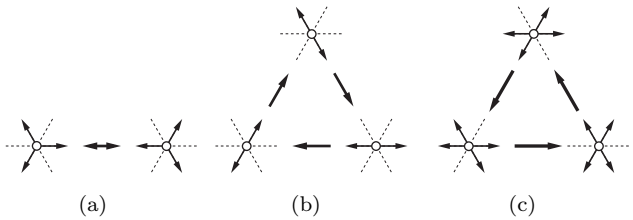

Figure 14.5: (a) Example of collision rule for three particles with zero net momentum. (b) Example of two particle collision rule. (c) Example of four particle collision rule. The rules for states that are not shown is that the velocities do not change after a collision. An open circle represents a lattice site and the absence of a stationary particle.

collision rule. Examples of the collision rules are illustrated in Figures 14.4–14.6. The rules are deterministic with only one possible set of velocities after a collision for each possible set of velocities before a collision. It is easy to check that momentum conservation for collisions between the particles is enforced by these rules.

As in Section 14.1, we use bit manipulation to efficiently represent a lattice site and the collision rules. Each lattice site is represented by one element of the integer array `lattice`. In Java each `int` stores 32 bits, but we will use only the first 8 bits. We use the first six bits from 0 to 5 to represent particles moving in the six possible directions with bit 0 corresponding to a particle moving with velocity $\mathbf{v}_0$ (see Table 14.1). If there are three particles with velocities $\mathbf{v}_0$, $\mathbf{v}_2$, and $\mathbf{v}_4$ at a site and no barrier, then the value of the lattice array element at this site is 00010101 in binary notation.

Bit 6 represents a possible rest (stationary) particle. If we want a site to act as a barrier that blocks incoming particles, we set bit 7. For example, a barrier site containing a particle with velocity $\mathbf{v}_1$ is represented by 10000010.

The rules for the collisions are given in the declaration of the class variables in class `LatticeGas`. Because `rule` is declared `static final`, we cannot normally overwrite its values. However, an exception is made for static initializers that are run when the class is first loaded. To construct the rules, we use the bitwise *or* operator, `|`, and use named constants for each of the possible states. As an example, the state corresponding to one particle moving to the right, one moving to the left and down, and one moving to the left and up is given by `LU + LD + RI`, which we write as `LU|LD|RI` or 00010101. The collision rule in Figure 14.5(a) is that this state transforms to one particle moving to the right and down, one moving left, and one moving to the right and up. Hence, this collision rule is given by `rule[LU|LD|RI] = RU|LE|RD`. The other rules are given in a similar way. Stationary particles also can be created or destroyed. For example, what are the states before and after the collision for `rule[LU|RI] = RU|S`?

To every rule corresponds a dual rule that flips the bits corresponding to the presence and absence of a particle. This duality means that we need to only specify half of the rules. The dual rules can be constructed by flipping all bits of the input and output. Our convention is to list the rules starting without a stationary particle. Then the corresponding dual rules are those that start with a stationary particle. The dual rules are implemented by the statement

```
rule[i^(RU|LU|LE|LD|RD|RI|S)] = rule[i]^(RU|LU|LE|LD|RD|RI|S);
```

where ^ is the bitwise exclusive or operator, which equals 1 if both bits are different, and is 0 otherwise. Two examples of dual rules are given in Figure 14.6.

The rules in Figures 14.5(b) and 14.5(c) cycle through the states in a particular direction. Although these rules are straightforward, they are not invariant under reflection. To help eliminate this bias, we cycle in the opposite direction when a stationary particle is present (see Figure 14.6).

We adopt the rule that when a particle moves onto a barrier site, we set the velocity $\mathbf{v}$ of this particle equal to $-\mathbf{v}$ (see Figure 14.7). Because of our ordering of the velocities, the rule for updating a barrier can be expressed compactly using bit manipulation. Reflection off a barrier is accomplished by shifting the higher order bits to the right by three bits (`>>>3`) and shifting the lower order bits to the left by three bits (`<<3`). Check the rules given in Listing 14.13. Other possibilities are to set the angle of incidence equal to the angle of reflection or to set the velocity to an arbitrary direction. The latter case would correspond to a collision off a rough surface.

The `step` method runs through the entire lattice and moves all the particles. The updated values of the sites are placed in the array `newLattice`. We then go through the `newLattice` array, implement the relevant collision rule at each site, and write the results into the array `Lattice`.

The movement of the particles is accomplished as follows. Because the even rows are horizontally displaced one half a lattice spacing from the odd rows, we need to treat odd and even rows separately. In the `step` method we loop through every other row and update `site1` and `site2` at the same time. An example will show how this update works. The statement

```
rght[j-1] |= site1 & RIGHT_DOWN;
```

means that if there is a particle moving to the right and down at `site1`, then the bit corresponding to `RIGHT_DOWN` is added to the site `rght` (see Figure 14.8). The statement

```
cent[j] |= site1 & (STATIONARY|BARRIER) | site2 & RIGHT_DOWN;
```

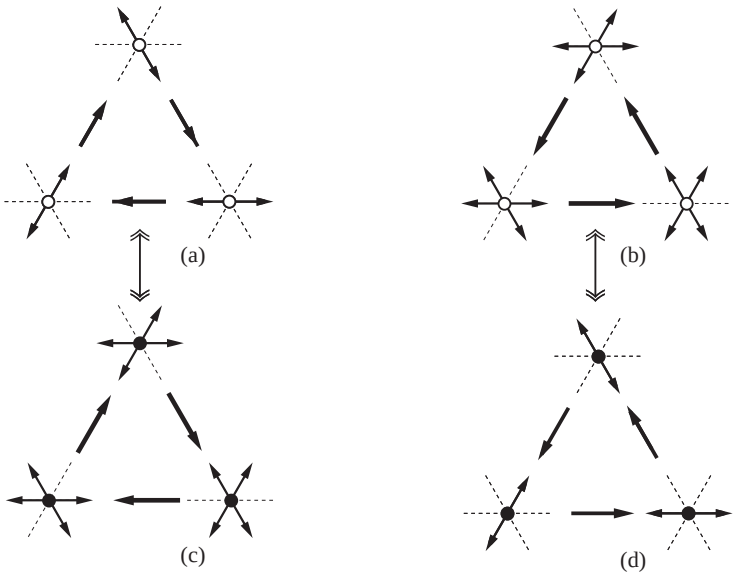

Figure 14.6: (a) and (c) and (b) and (d) are duals of each other. An open circle represents the absence of a stationary particle, and a filled circle represents the presence of a stationary particle. Note that the collision rule in (c) is similar to (b), and the collision rule in (d) is similar to (a), but in the opposite direction.

means that a stationary particle at `site1` remains there, and if `site1` is a barrier, it remains so. If `site2` has a particle moving in the direction `RD`, then `site1` will receive this particle.

To maintain a steady flow rate, we add the necessary horizonal momentum to the lattice uniformly after each time step. The procedure is to chose a site at random and determine if it is possible to change the sites's horizontal momentum. If so, we then remove the left bit and add the right bit or vice versa. This procedure is accomplished by the statements at the end of the `step` method.

Listing 14.13: Listing of the `LatticeGas` class.

```
package org.opensourcephysics.sip.ch14.latticegas;
import org.opensourcephysics.display.*;
import java.awt.*;
import java.awt.geom.AffineTransform;
import java.awt.geom.Line2D;

public class LatticeGas implements Drawable {
   // input parameters from user
   public double flowSpeed;            // controls pressure
   public double arrowSize;            // size of velocity arrows displayed
   public int spatialAveragingLength; // spatial averaging of velocity
   public int Lx, Ly;                  // linear dimensions of lattice
   public int[][] lattice, newLattice;
```

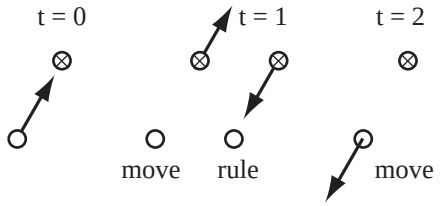

Figure 14.7: Example of a collision from a barrier. The symbol ⊗ denotes a barrier site.

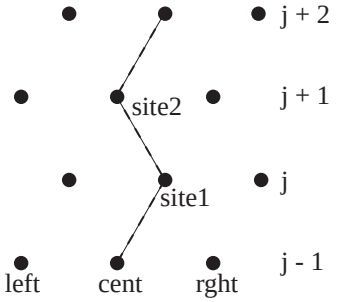

Figure 14.8: We update `site1` and `site2` at the same time. The rows are indexed by `j`. The dotted line connects sites in the same column.

```
private double numParticles;
static final double SQRT3_OVER2 = Math.sqrt(3)/2;
static final double SQRT2 = Math.sqrt(2);
static final int
   RIGHT = 1, RIGHT_DOWN = 2, LEFT_DOWN = 4;
static final int
   LEFT = 8, LEFT_UP = 16, RIGHT_UP = 32;
static final int
   STATIONARY = 64, BARRIER = 128;
static final int NUM_CHANNELS = 7; // maximum number of particles per site
static final int NUM_BITS = 8;     // 7 channel bits plus 1 barrier bit per site
static final int NUM_RULES = 1<<8; // total number of possible site configurations = 2^8
// 1 << 8 means move the zeroth bit over 8 places to the left to the eighth bit

static final double ux[] = {
   1.0, 0.5, -0.5, -1.0, -0.5, 0.5, 0
};
static final double uy[] = {
   0.0, -SQRT3_OVER2, -SQRT3_OVER2, 0.0, SQRT3_OVER2, SQRT3_OVER2, 0
};
static final double[] vx, vy;      // averaged velocities for every site configuration
static final int[] rule;

static { // set rule table
```

```
    // default rule is the identity rule
    rule = new int[NUM_RULES];
    for(int i = 0;i<BARRIER;i++) {
       rule[i] = i;
    }
    // abbreviations for channel bit indices
    int RI = RIGHT, RD = RIGHT_DOWN, LD = LEFT_DOWN;
    int LE = LEFT, LU = LEFT_UP, RU = RIGHT_UP;
    int S = STATIONARY;
    // three particle zero momentum rules
    rule[LU|LD|RI] = RU|LE|RD;
    rule[RU|LE|RD] = LU|LD|RI;
    // three particle rules with unperturbed particle
    rule[RU|LU|LD] = LU|LE|RI;
    rule[LU|LE|RI] = RU|LU|LD;
    rule[RU|LU|RD] = RU|LE|RI;
    rule[RU|LE|RI] = RU|LU|RD;
    rule[RU|LD|RD] = LE|RD|RI;
    rule[LE|RD|RI] = RU|LD|RD;
    rule[LU|LD|RD] = LE|LD|RI;
    rule[LE|LD|RI] = LU|LD|RD;
    rule[RU|LD|RI] = LU|RD|RI;
    rule[LU|RD|RI] = RU|LD|RI;
    rule[LU|LE|RD] = RU|LE|LD;
    rule[RU|LE|LD] = LU|LE|RD;
    // two particle cyclic rules
    rule[LE|RI] = RU|LD;
    rule[RU|LD] = LU|RD;
    rule[LU|RD] = LE|RI;
    // four particle cyclic rules
    rule[RU|LU|LD|RD] = RU|LE|LD|RI;
    rule[RU|LE|LD|RI] = LU|LE|RD|RI;
    rule[LU|LE|RD|RI] = RU|LU|LD|RD;
    // stationary particle creation rules
    rule[LU|RI] = RU|S;
    rule[RU|LE] = LU|S;
    rule[LU|LD] = LE|S;
    rule[LE|RD] = LD|S;
    rule[LD|RI] = RD|S;
    rule[RD|RU] = RI|S;
    rule[LU|LE|LD|RD|RI] = RU|LE|LD|RD|S;
    rule[RU|LE|LD|RD|RI] = LU|LD|RD|RI|S;
    rule[RU|LU|LD|RD|RI] = RU|LE|RD|RI|S;
    rule[RU|LU|LE|RD|RI] = RU|LU|LD|RI|S;
    rule[RU|LU|LE|LD|RI] = RU|LU|LE|RD|S;
    rule[RU|LU|LE|LD|RD] = LU|LE|LD|RI|S;
    // add all rules indexed with a stationary particle (dual rules)
    for(int i = 0;i<S;i++) {
       rule[i^(RU|LU|LE|LD|RD|RI|S)] = rule[i]^(RU|LU|LE|LD|RD|RI|S); // ^ is the exclusive
    }
```

```
      // add rules to bounce back at barriers
      for(int i = BARRIER;i<NUM_RULES;i++) {
         int highBits = i&(LE|LU|RU); // & is bitwise and operator
         int lowBits = i&(RI|RD|LD);
         rule[i] = BARRIER|(highBits>>3)|(lowBits<<3);
      }
   }
   static { // set average site velocities
      // for every particle site configuration i, calculate total net velocity
      // and place in vx[i], vy[i]
      vx = new double[NUM_RULES];
      vy = new double[NUM_RULES];
      for(int i = 0;i<NUM_RULES;i++) {
         for(int dir = 0;dir<NUM_CHANNELS;dir++) {
            if((i&(1<<dir))!=0) {
               vx[i] += ux[dir];
               vy[i] += uy[dir];
            }
         }
      }
   }
   public void initialize(int Lx, int Ly, double density) {
      this.Lx = Lx;
      this.Ly = Ly-Ly%2;                              // Ly must be even
      numParticles = Lx*Ly*NUM_CHANNELS*density; // approximate total number of particles
      // density is the number of particles divided by the maximum number possible
      lattice = new int[Lx][Ly];
      newLattice = new int[Lx][Ly];
      int sevenParticleSite = ((1<<NUM_CHANNELS)-1); // equals 127
      for(int i = 0;i<Lx;i++) {
         lattice[i][1] = lattice[i][Ly-2] = BARRIER; // wall at top and bottom
         for(int j = 2;j<Ly-2;j++) {
            // occupy site by 0 or 7 particles, average occupation will be about the density
            int siteValue = Math.random()<density ? sevenParticleSite : 0;
            lattice[i][j] = siteValue; // random particle configuration
         }
      }
      for(int j = 3*Ly/10;j<7*Ly/10;j++) {
         lattice[2*Lx/10][j] = BARRIER; // obstruction toward the left
      }
   }

   public void step() {
      // move all particles forward
      for(int i = 0;i<Lx;i++) {
         // define the columns of a 2-dim array
         int[] left = newLattice[(i-1+Lx)%Lx];
         int[] cent = newLattice[i]; // use abbreviations to align expressions
         int[] rght = newLattice[(i+1)%Lx];
         for(int j = 1;j<Ly-2;j += 2) {
```

```
            // loop j in increments of 2 in order to decrease reads and writes of neighbors
            int site1 = lattice[i][j];
            int site2 = lattice[i][j+1];
            // move all particles in site1 and site2 to their neighbors
            rght[j-1] |= site1&RIGHT_DOWN;
            cent[j-1] |= site1&LEFT_DOWN;
            rght[j] |= site1&RIGHT;
            cent[j] |= site1&(STATIONARY|BARRIER)|site2&RIGHT_DOWN;
            left[j] |= site1&LEFT|site2&LEFT_DOWN;
            rght[j+1] |= site1&RIGHT_UP|site2&RIGHT;
            cent[j+1] |= site1&LEFT_UP|site2&(STATIONARY|BARRIER);
            left[j+1] |= site2&LEFT;
            cent[j+2] |= site2&RIGHT_UP;
            left[j+2] |= site2&LEFT_UP;
         }
      } // handle collisions, find average x velocity
      double vxTotal = 0;
      for(int i = 0;i<Lx;i++) {
         for(int j = 0;j<Ly;j++) {
            int site = rule[newLattice[i][j]]; // use collision rule
            lattice[i][j] = site;
            newLattice[i][j] = 0;              // reset newLattice values to 0
            vxTotal += vx[site];
         }
      }
      int scale = 4;
      int injections = (int) ((flowSpeed*numParticles-vxTotal)/scale);
      for(int k = 0;k<Math.abs(injections);k++) {
         int i = (int) (Math.random()*Lx); // choose site at random
         int j = (int) (Math.random()*Ly);
         // flip direction of horizontally moving particle if possible
         if((lattice[i][j]&(RIGHT|LEFT))==((injections>0) ? LEFT : RIGHT)) {
            lattice[i][j] ^= RIGHT|LEFT;
         }
      }
   }

   public void draw(DrawingPanel panel, Graphics g) {
      if(lattice==null) {
         return;
      }
      // if s = 1 draw lattice and particle details explicitly
      // otherwise average velocity over an s by s square
      int s = spatialAveragingLength;
      Graphics2D g2 = (Graphics2D) g;
      AffineTransform toPixels = panel.getPixelTransform();
      Line2D.Double line = new Line2D.Double();
      for(int i = 0;i<Lx;i++) {
         for(int j = 2;j<Ly-2;j++) {
            double x = i+(j%2)*0.5;
```

```
            double y = j*SQRT3_OVER2;
            if(s==1) {
               g2.setPaint(Color.BLACK);
               for(int dir = 0;dir<NUM_CHANNELS;dir++) {
                  if((lattice[i][j]&(1<<dir))!=0) {
                     line.setLine(x, y, x+ux[dir]*0.4, y+uy[dir]*0.4);
                     g2.draw(toPixels.createTransformedShape(line));
                  }
               }
            }
            if((lattice[i][j]&BARRIER)==BARRIER||s==1) { // draw points at lattice sites
               Circle c = new Circle(x, y);
               c.pixRadius = ((lattice[i][j]&BARRIER)==BARRIER) ? 2 : 1;
               c.draw(panel, g);
            }
         }
      }
      if(s==1) {
         return;
      }
      for(int i = 0;i<Lx;i += s) {
         for(int j = 0;j<Ly;j += s) {
            double x = i+s/2.0;
            double y = (j+s/2.0)*SQRT3_OVER2;
            double
               wx = 0, wy = 0; // compute coarse grained average velocity
            for(int m = i;m!=(i+s)%Lx;m = (m+1)%Lx) {
               for(int n = j;n!=(j+s)%Ly;n = (n+1)%Ly) {
                  wx += vx[lattice[m][n]];
                  wy += vy[lattice[m][n]];
               }
            }
            Arrow a = new Arrow(x, y, arrowSize*wx/s, arrowSize*wy/s);
            a.setHeadSize(2);
            a.draw(panel, g);
         }
      }
   }
}
```

Listing 14.14: Listing of the `LatticeGasApp` class.

```
package org.opensourcephysics.sip.ch14.latticegas;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.*;

public class LatticeGasApp extends AbstractSimulation {
   LatticeGas model = new LatticeGas();
   DisplayFrame display = new DisplayFrame("Lattice gas");
```

```
   public LatticeGasApp() {
      display.addDrawable(model);
      display.setSize(800, (int) (400*Math.sqrt(3)/2));
   }

   public void initialize() {
      int lx = control.getInt("lx");
      int ly = control.getInt("ly");
      double density = control.getDouble("Particle density");
      model.initialize(lx, ly, density);
      model.flowSpeed = control.getDouble("Flow speed");
      model.spatialAveragingLength = control.getInt("Spatial averaging length");
      model.arrowSize = control.getInt("Arrow size");
      display.setPreferredMinMax(-1, lx, -Math.sqrt(3)/2, ly*Math.sqrt(3)/2);
   }

   public void doStep() {
      model.flowSpeed = control.getDouble("Flow speed");
      model.spatialAveragingLength = control.getInt("Spatial averaging length");
      model.arrowSize = control.getDouble("Arrow size");
      model.step();
   }

   public void reset() {
      control.setValue("lx", 1000);
      control.setValue("ly", 500);
      control.setValue("Particle density", 0.2);
      control.setAdjustableValue("Flow speed", 0.2);
      control.setAdjustableValue("Spatial averaging length", 20);
      control.setAdjustableValue("Arrow size", 2);
      enableStepsPerDisplay(true);
      control.setAdjustableValue("steps per display", 100);
   }

   public static void main(String[] args) {
      SimulationControl.createApp(new LatticeGasApp());
   }
}
```

An important application of lattice gas models is to simulate the flow in and around various geometries. In Problem 14.20 we will see that the fluid velocity field develops vortices, wakes, and other fluid structures near obstacles. Method `initialize` in class `LatticeGas` places an obstacle in the middle of the lattice and provides initial values for each site. Large lattices are required to obtain quantitative results, because it is necessary to average the velocity over many sites. The parameter `density` is the average number of particles divided by the maximum possible. The pressure can be varied by changing the `flowSpeed` parameter.

Problem 14.20. Flow past a barrier

a. Convince yourself that you understand the collision rules and their implementation in class

`LatticeGas`. Then download the class `FastLatticeGas` from the ch14 directory. This latter class uses all 32 bits of an `int` variable and runs about twice as fast. The tradeoff is that the code is more difficult to debug and understand. Use the parameters in Listing 14.14. Describe the flow once a steady state velocity field begins to appear. Do you see a wake appearing behind the obstacle? Are there vortices?

b. Repeat part (a) with different size obstacles. Are there any systematic trends? (One limitation of the present program is that it naively redraws a circle to represent each barrier site. This redrawing requires a significant amount of computer resources and limits the size of the obstacles that we can consider.)

c. Reduce the pressure by reducing the flow speed. Are there any noticeable changes in behavior from parts (a) and (b)? Reduce the pressure still further and describe any changes in the fluid flow.

Problem 14.21. Approach to equilibrium

a. Consider the approach of a lattice gas to equilibrium. Modify `LatticeGas` so that the initial configuration has zero net momentum, the particles are localized in a $b \times b$ region, and there are no barrier sites. Choose $L = 30$ and $b = 4$ and place six particles at every site in the localized region. The other sites in the lattice are initially empty. Describe what happens to the particles as a function of time. Approximately how many time steps does it take for the system to come to equilibrium? Do the particles appear to be at random positions with random velocities? What is your visual algorithm for determining when equilibrium has been reached?

b. Repeat part (a) for $b = 2$, 6, 8, and 10. Estimate the equilibration time in each case. What is the qualitative dependence of the equilibration time on b? How does the equilibration time depend on the number density ρ?

c. Repeat part (a) with $b = 4$, but with $L = 10$, 20, and 40. Estimate the equilibration time in each case. How does the equilibration time depend on ρ?

Problem 14.22. Fluid flow in porous media

a. Modify class `LatticeGas` so that instead of a rectangular barrier, the barrier sites are placed at random in the system. We define the porosity, ϕ, as the fraction of sites without a barrier. The interesting quantity to measure is the permeability, k, which is a measure of the fluid conductivity. We can compute the permeability using the relation

$$k \propto \frac{\phi \sum_i \langle v_{i,x} \rangle}{\sum_j \langle \Delta p_{j,x} \rangle}, \tag{14.8}$$

where the sum in the numerator is over the horizontal velocity of all particles in the pore space (the sites at which there are no barriers), and the sum in the denominator is over the injected momentum at all sites used to maintain the flow. The brackets refer to averages over time. Compute the permeability as a function of the porosity ϕ and display your results on a log-log plot. You should average over at least 10 configurations of random barrier sites for each value of

the porosity. What value of ϕ corresponds to the percolation threshold, defined by $k = 0$? See Rothman and Zaleski for a discussion of the comparison of this type of simulation with results for real rocks.

b.* Vary the size of the lattice and use the finite size scaling procedure discussed in Section 12.4 to estimate the critical exponent μ defined by the dependence of the permeability on the porosity, that is, $k \sim (\phi - \phi_c)^\mu$. Assume that you know the value of the percolation exponent ν defined by the critical behavior of the connectedness length $\xi \sim |p - p_c|^{-\nu}$ (see Table 12.1).

The principal virtues of lattice gas models are their use of simultaneous updating, which makes them very fast on parallel computers, and their use of integer or boolean arithmetic and bit manipulation, which is faster than floating point arithmetic. Their major limitation is that it is necessary to average over many sites to obtain quantitative results. It is not yet clear whether lattice gas models are more efficient than standard simulations of the Navier-Stokes equation. The greatest promise for lattice gas models may not be with simple single component fluids, but with multicomponent fluids such as binary fluids and fluids containing bubbles (see the book by Rothman and Zaleski). A related technique that might hold greater promise is the lattice Boltzmann method (see the references).

14.7 Overview and Projects

The models we have discussed in this chapter have been presented as algorithms rather than in terms of differential equations and are a reflection of the way that technology affects the way we think. Can you discuss the models in this chapter without thinking about their implementation on a computer? Can you imagine understanding these models without the use of computer graphics?

We have given only a brief introduction to cellular automata and other models that are relevant to the rapidly developing study of complex systems. There are many more models and applications that we have not discussed, ranging from aging, the immune system, economic cycles, and pedestrian movements, to name just a few.

Models of opinion formation have become popular in recent years. The basic idea is that the opinions of others will influence the opinion of individuals. The following two projects explore some of the popular models.

Project 14.23. Models of opinion formation

a. *The voter model.* On a regular lattice assign each site the value ± 1. Choose a site (the voter) at random (the voter). The voter then adopts the same value as a randomly chosen neighbor. These two steps continue until all sites have the same value, that is, when they have reached consensus. Compute the probability of achieving a consensus of $+1$ given that the initial density of $+1$ sites is ρ_0. Use a 10×10 square lattice and make at least 20 runs at each density. Also compute the time to reach consensus as a function of the lattice size. In two dimensions this time scales as $N \ln N$, where N is the number of sites. How does the consensus time scale with N in $d = 1$ and $d = 3$ dimensions? How does it scale on a preferential attachment network (see the article by Sood and Redner)?

b. *The relative agreement interaction model.* N individuals are initially assigned an opinion that takes on a value between 0 and 1. Choose two individuals, i and j, at random. Assume that the ith opinion, O_i is greater than the jth opinion, O_j. If their opinions differ by less than the parameter ϵ, then increase O_j by $(m/2)(O_i - O_j)$ and decrease O_i by the same amount, where m is another parameter. This model implements the idea that two people will influence each other only if their opinions are sufficiently close. Write a program to simulate this model. Use a `LatticeFrame` for which each cell can take on one of 256 values. The approximation of the continuum by 256 values is for visualization purposes only, and the 256 values should be sufficiently large to approximate a continuum of values. Choose $\epsilon = 10$, 50 and 100, and $m = 0.3$ and 0.6. Include in your program the option to plot configurations only after a certain number of iterations to speed up the simulation (use `enableStepsPerDisplay(true)`). Choose $N \geq 2500$, begin with a random set of opinions, and discuss whether a single opinion emerges and the magnitude of the fluctuations.

c. *The Sznajd model.* Place individuals on a square lattice with linear dimension L and periodic boundary conditions. Each individual has one of two opinions. At each iteration, an individual and one of her neighbors is chosen at random. If the two individuals have the same opinion, the opinion of the six neighbors of the pair is changed to that of the pair. The idea is that people are more likely to change their opinion to those physically near them if more than one person shares the same opinion (peer pressure). Write a program to simulate this model and show that consensus is always reached for all sites if the simulation is run for a sufficiently long time. Discuss the visual appearance of the groups of like-minded individuals. Consider initial configurations where the individuals are randomly assigned the two opinions, and initial configurations where one opinion has a majority of 1%, 5%, and 10%. Choose $L \geq 50$.

d. Generalize the Sznajd model so that an individual may be assigned one of more than two opinions. Is consensus still always reached? What happens if the individuals are not on the sites of a square lattice, but rather are the nodes of a preferential attachment network of at least 5000 nodes?

Project 14.24. The minority game

In certain situations we wish to be in the minority. For example, we might wish to go to a popular restaurant on an off-night so that we do not have to wait in line. A business might want to sell goods and services that are not being sold by other businesses. The following algorithm, known as the *minority game*, is a model of adaptive competition where each player tries to maximize his gain. We will find that there is a phase transition between states where the players mainly act on their own, and states for which cooperative behavior emerges.

There are N players, where N is odd. At each iteration, each player can choose one of two actions which we call 1 or 0, but which we encode as the boolean `true` or `false`. A player's choice is determined by a strategy based on the previous m (memory) iterations. Each strategy is represented by a table of all the possible outcomes of the previous m iterations and a decision on what to do for each outcome. Each player has his own table of strategies. An outcome is defined as the action that was chosen *least* by all the players. For example, suppose $m = 2$. There are four possible pasts: (1,1), (1,0), (0,1), and (0,0). The past (1,1) means that in each of the last two iterations, action 1 was chosen by a minority of the players. A strategy would be encoded by a table

such as the following: (1,1,1), (1,0,0), (0,1,0), and (0,0,1). The first two entries in each triple are the possible outcomes of the last two iterations, and the third entry in the triple gives the action that the strategy suggests taking. Thus the triple (1,1,1) means that if (1,1) occurred in the past, the strategy is to choose action 1; (1,0,0) means that if (1,0) occurred in the past, the strategy is to use action 0. Each player is assigned at least two strategies that are chosen at random from all possible strategies at the beginning of the game. As the game is played, the performance of each strategy (whether or not it is used) is updated, such that if a strategy leads to the same action that was in the minority, then this strategy is successful and its performance is incremented by unity; otherwise it stays the same. At each iteration each player chooses the strategy with the best performance, and takes the action determined by his best performing strategy. Then the outcome (which action was in the minority) for that iteration is determined, and the past m outcomes and the performance for all the strategies are updated. Note that the strategies available to each player does not change, but which of each player's strategy is best changes as the game is iterated.

To simplify the code, represent the past outcomes by an integer where each bit represents an outcome. For example, the bit 110 means that the outcome was 0 in the last iteration and was 1 for each of the earlier two iterations. You will need the following arrays: `strategies[i][j][k]`, which gives the action for the ith player, using its jth strategy, when the kth past occurred; `performance[i][j]`, which gives the performance for the ith player's jth strategy, and `chosenStrategy[i]`, which gives the strategy chosen by the ith player in the current iteration.

Let N_1 equal the number of players who chose action 1 in one iteration. The outcome is best if at each iteration the value of N_1 is close to $N/2$ because in this way there would be as many players as possible in the minority. The quantity of interest is σ, where σ is defined as

$$\sigma^2 = \sum_k (N_1(k) - \langle N_1 \rangle)^2, \tag{14.9}$$

and the sum is over all the iterations of the game and $N_1(k)$ is the number of players choosing action 1 in the kth iteration. The quantity σ decreases as the efficiency increases. High efficiency means more players are in the minority on the average. We might think that the efficiency increases as the number of past outcomes increases, because then the players have more information to choose their strategy. However, you might be surprised!

a. Write a program to simulate the minority game. For simplicity, give each player only two strategies chosen at random. Run your program for a memory m varying from 2 to about 12. A reasonable choice for N is 101, but for testing purposes choose $N = 11$. Each game should be run for at least 1000 iterations, and your results should be averaged over at least 10 independent runs for the same m, with different strategies for the players. Plot the average of σ versus m, and describe the behavior for different values of N. Explain why there is a minimum in these plots.

b. The results of the minority game scale unambiguously. Plot the average of σ^2/N versus $2^m/N$ for different values of N. You should find that your data fall on the same curve. What does 2^m represent? Discuss this scaling behavior and describe the behavior of the efficiency on either side of the minimum. Can you describe your results as a phase transition? Where is the ordered phase and where the disordered phase?

c. Plot the spread in the values of σ versus m. The spread can be taken to be the standard deviation of each game's value of σ over many games. Discuss the significance of your results.

Project 14.25. A cellular automaton for Burger's equation

In Section 14.6 we mentioned that the partial differential equation describing the flow of incompressible fluids, the Navier-Stokes equation, is very difficult to solve numerically. A one-dimensional approximation of the Navier-Stokes equation was given by Burgers, and is given by

$$\frac{\partial n}{\partial t} + c\frac{\partial}{\partial x}\big(n - \frac{n^2}{2}\big) = D\frac{\partial^2 n}{\partial x^2}, \tag{14.10}$$

where $n(x,t)$ corresponds to the velocity field at position x at time t, c is the linear advection (drift) coefficient, and D is a diffusion coefficient. Equation (14.10) is of general interest because it can be solved analytically and its solutions exhibit discontinuities (shock waves) depending on the values of the parameters and the initial conditions.

Boghosian and Levermore have proposed a cellular automaton that is equivalent to (14.10). The study of this cellular automaton raises many of the same issues as the lattice gas models of the incompressible Navier-Stokes equation considered in Section 14.6. Its study also illustrates the idea that many partial differential equations can be formulated as cellular automata.

We know that if all particles on the lattice move one lattice site to either the right or the left in one time step, then the density of the particles obeys the diffusion equation (see Appendix 7A)

$$\frac{\partial n}{\partial t} = D\frac{\partial^2 n}{\partial x^2}, \tag{14.11}$$

where $D = (\Delta x)^2/2\Delta t$, Δx is the lattice spacing, and Δt is the time between successive steps of the random walk. If add a bias so that the probability of a step to the right is $(1+\alpha)/2$ and the probability of a step to the left is $(1-\alpha)/2$, the density of the walkers satisfies

$$\frac{\partial n}{\partial t} + c\frac{\partial n}{\partial x} = D\frac{\partial^2 n}{\partial x^2}, \tag{14.12}$$

where $c = \alpha\Delta x/\Delta t$. To incorporate the quadratic term term, we add the rule that no two particles occupying the same site may be moving in the same direction. In this way the state of each site is specified by two bits. The right bit is 1 if a particle moving to the right is present and is 0 otherwise. Similarly, the left bit stores information about the presence of a particle moving to the left. Thus each site has four possible states labeled by the binary numbers 00, 01, 10, and 11.

$b_1(i,t)$	$b_0(i,t)$	$\tilde{b}_1(i,t)$	$\tilde{b}_0(i,t)$
0	0	0	0
0	1	$(1-\alpha(i,t)/2$	$(1+\alpha(i,t)/2$
1	0	$(1-\alpha(i,t)/2$	$(1+\alpha(i,t)/2$
1	1	1	1

Table 14.2: Rules for the collision substep.

In the first part of the step, the collision substep, the particles change their direction at random at their present lattice sites subject to the exclusion rule. In the second substep, the particles move

to the neighboring lattice site in their new direction. We follow Boghosian and Levermore and denote the right (left) bit at lattice site i and time step t by $b_0(x,t)$ ($b_1(x,t)$). After the collision substep, the new states are $\tilde{b}_{0,1}(x,t)$ and are given in Table 14.2, where $\alpha(x,t) = \pm 1$ with mean α. The rules in Table 14.2 may be written in the form

$$\tilde{b}_0(x,t) = \frac{1+\alpha(x,t)}{2} b_0(x,t) | b_1(x,t) + \frac{1-\alpha(x,t)}{2} b_0(x,t) \,\&\, b_1(x,t) \tag{14.13a}$$

$$\tilde{b}_1(x,t) = \frac{1-\alpha(x,t)}{2} b_0(x,t) | b_1(x,t) + \frac{1+\alpha(x,t)}{2} b_0(x,t) \,\&\, b_1(x,t), \tag{14.13b}$$

where | is the inclusive or operation and & denotes the and operation on a pair of bits. In the advection substep, the particles move to the neighboring lattice site in their new direction. The rules for these moves are

$$\tilde{b}_0(x+1,t+1) = \tilde{b}_0(x,t) \tag{14.14a}$$

$$\tilde{b}_1(x-1,t+1) = \tilde{b}_0(x,t). \tag{14.14b}$$

We can combine these two substeps to arrive at the rule for one full time step of the cellular automaton:

$$b_0(x+1,t+1) = \frac{1+\alpha(x,t)}{2} b_0(x,t) | b_1(x,t) + \frac{1-\alpha(x,t)}{2} b_0(x,t) \,\&\, b_1(x,t) \tag{14.15a}$$

$$b_1(x-1,t+1) = \frac{1-\alpha(x,t)}{2} b_0(x,t) | b_1(x,t) + \frac{1+\alpha(x,t)}{2} b_0(x,t) \,\&\, b_1(x,t). \tag{14.15b}$$

Write a program to implement (14.15) using periodic boundary conditions. Choose $c = 1$, $D = 2^{-15}$ and the initial condition

$$n(x, t=0) = 1.0 + 0.4\cos(2\pi x), \tag{14.16}$$

where x denotes the position of a lattice site. Boghosian and Levermore used $2^{16} = 65536$ lattice sites so that $\Delta x = 2^{-16}$. The bias is given by $\alpha = c\Delta x/2D = 0.25$ and the time step is $\Delta t = (\Delta x)^2/2D = 2^{-18}$. Average your results for 128 lattice sites and plot the average density as a function of x for different values of t up to $t = 1$. Do you see any evidence of a shock wave (a sharp discontinuity in $n(x)$)?

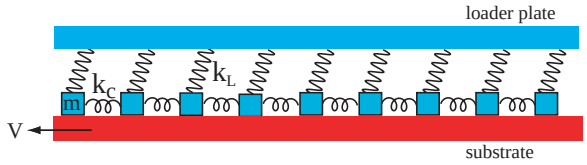

Figure 14.9: Schematic of the Burridge-Knopoff model. Blocks of mass m are connected by springs with spring constant k_c and move on a substrate with a velocity-dependent friction force. Each spring is connected by a spring with spring constant k_L to a loader plate that moves with velocity v to the left.

Project 14.26. Spring-block model of earthquakes

The first simulations of earthquakes were done by Burridge and Knopoff in 1967. Their model represents the motion of one side of a lateral fault that is driven by a slow shear deformation and subject to a velocity-weakening friction force (see Figure 14.9). The model consists of a one-dimensional array of blocks, each connected to its two neighbors by springs with spring constant k_c, and constrained to move on the surface of a substrate. The coupling represents the linear elastic response of the system to compressional deformations. Each block also is connected by a spring with spring constant k_L to a fixed loader plate. The system is loaded slowly by moving the substrate at a velocity v to the left until the force on one of the blocks exceeds the static friction threshold and the block slips. As this block moves relative to the substrate, the springs connecting it to its neighbors change length, thus changing the forces acting on them. If the change in the force is sufficient, the neighboring blocks begin to move. The blocks move until the friction force opposing the slip is large enough to stop their motion. The slip of the blocks represents the slip of the two surfaces of the fault past one another during an earthquake. The stick-slip behavior of this model is similar to that of a real earthquake fault.

The equation of motion of the Burridge-Knopoff model can be written as

$$m\ddot{x}_j = k_c(x_{j+1} - 2x_j + x_{j-1}) - k_L x_j - F(v + \dot{x}_j), \tag{14.17}$$

where x_j is the displacement of block j from its equilibrium position and F represents the velocity-dependent friction force. The friction force has the functional form

$$F(\dot{x}) = F_0\phi(\dot{x}/\tilde{v}), \tag{14.18}$$

which is characterized by the speed $\tilde{v}$ and the static friction F_0. In order for the system to exhibit a dynamical instability corresponding to an earthquake, the friction force must become weaker as the block slides. The form of $\phi(y)$ is taken to be

$$\phi(y) = \begin{cases} (-\infty, 1], & y \le 0 \\ \frac{1-\sigma}{1+\frac{y}{1-\sigma}}, & y > 0. \end{cases} \tag{14.19}$$

The parameter σ represents the drop of the friction force at the onset of the slip. Note that back-slip has been inhibited by imposing an infinite;ly large friction force for $\dot{u} < 0$.

As usual, it is convenient to introduce dimensionless variables, which we take to be $u_j = (k_L/F_0)x_j$, $\omega_L^2 = k_L/m$, and $\tau = \omega_L t$. We rewrite (14.17) as

$$m\ddot{u}_j = \ell^2(x_{j+1} - 2x_j + x_{j-1}) - u_j - \phi(2\alpha\nu + 2\alpha\dot{u}_j), \tag{14.20}$$

where the stiffness parameter $\ell = \sqrt{k_c/k_L}$, $\nu = vk_L/\omega_L F_0$, and $2\alpha = \omega_L F_0/k_L\tilde{v}$, and the dots now denote differentiation with respect to τ. Typical values of the parameters are $\ell = 10$, $\sigma = 0.01$, and $\alpha = 2.5$.

The equation of motion (14.20) can be solved using the Euler-Richardson algorithm with $\Delta t = 10^{-3}$. We define a block as stuck if its velocity is smaller than a parameter v_0, its velocity is decreasing, and the total force on it (not counting the friction force) is smaller than the maximum static friction force F_0, which we choose to be unity. If a block is stuck, its velocity is set equal

to zero. An earthquake begins with the slip of the first block and ends when all blocks are stuck. Blocks can become stuck and begin to slip again during an event. We take $v_0 = 10^{-5}$.

Initially we set $\dot{u}_j = 0$ for all j and assign small random displacements to all the blocks. We then calculate the force on all the blocks and move each block according to (14.20). We continue this iteration until all the blocks become stuck. We then move the substrate until the force on one block exceeds unity so that one block initiates the next event. This way of moving the substrate is called the zero velocity limit, which means that we have let $\nu = 0$.

The main quantities of interest are $P(s)$, the distribution of the number of blocks in an event, and $P(M)$, the distribution of the moment of the event, where the latter is defined as

$$M = \sum_i du_i, \tag{14.21}$$

where the sum over i in (14.21) is over the blocks involved in an event and du_i is the net displacement of the blocks during the event. Do $P(s)$ and $P(M)$ exhibit scaling consistent with Gutenberg-Richter? Other interesting questions are posed in the references (see Klein et al., Ferguson et al., and Mori and Kawamura).

References and Suggestions for Further Reading

Réka Albert and Albert-László Barabási, "Statistical mechanics of complex networks," Rev. Mod. Phys. **74**, 47–97 (2002).

Per Bak, *How Nature Works*, Copernicus Books (1999). A good read about self-organized critical phenomena from earthquakes to stock markets. Nature is not as simple as Bak believed, but his interest in complex systems spurred many others to become interested.

P. Bak, "Catastrophes and self-organized criticality," Computers in Physics **5** (4), 430 (1991). A good introduction to self-organized critical phenomena.

Per Bak and Michael Creutz, "Fractals and self-organized criticality," in *Fractals in Science*, Armin Bunde and Shlomo Havlin, editors, Springer-Verlag (1994).

Per Bak and Kim Sneppen, "Punctuated equilibrium and criticality in a simple model of evolution," Phys. Rev. Lett. **71**, 4083 (1993); Henrik Flyvbjerg, Kim Sneppen, and Per Bak, "Mean field theory for a simple model of evolution," Phys. Rev. Lett. **71**, 4087 (1993);

P. Bak, C. Tang, and K. Wiesenfeld, "Self-organized criticality," Phys. Rev. A **38**, 364–374 (1988).

E. R. Berlekamp, J. H. Conway, and R. K. Guy, *Winning Ways for your Mathematical Plays*, Vol. 2, Academic Press (1982). A discussion of how the Game of Life simulates a universal computer.

Bruce M. Boghosian and C. David Levermore, "A cellular automaton for Burger's equation," Complex Systems **1**, 17–30 (1987). Reprinted in Doolen et al. (see below).

D. Challet and Y.-C. Zhang, "Emergence of cooperation and organization in an evolutionary game," Physica A **246**, 407–418 (1997), or adap-org/9708006. The authors give the first description of the minority game. A web site devoted to the minority game and other topics is at <http://www.unifr.ch/econophysics/>.

Debashish Chowdhury, Ludger Santen, and Andreas Schadschneider, "Simulation of vehicular traffic: A statistical physics perspective," Computing in Science and Engineering **2** (5), 80–87 (2000).

John W. Clark, Johann Rafelski, and Jeffrey V. Winston, "Brain without mind: Computer simulation of neural networks with modifiable neuronal interactions," Physics Reports **123**, 215–273 (1985).

Aaron Clauset, M. E. J. Newman, and Cristopher Moore, "Finding community structure in very large networks," Phys. Rev. E **70**, 066111-1–6 (2004). This paper describes a faster algorithm than that discussed in Newman and Girvan.

J. P. Crutchfield and M. Mitchell, "The evolution of emergent computation," Proc. Natl. Acad. Sci. **92**, 10742–10746 (1995). The authors use genetic algorithms to evolve a cellular automata model.

Guillaume Deffuant, Fréd'eric Amblard, Gérard Weisbuch, and Thierry Faure, "How can extremism prevail? A study based on the relative agreement interaction model," J. Artificial Societies and Social Simulation **5**(4) paper #1 (2002). This paper and others can be found at <jasss.soc.surrey.ac.uk>.

Gary D. Doolen, Uriel Frisch, Brosl Hasslacher, Steven Orszag, and Stephen Wolfram, editors, *Lattice Gas Methods for Partial Differential Equations*, Addison-Wesley (1990). A collection of reprints and original articles by many of the leading workers in lattice gas methods.

Stephanie Forrest, editor, *Emergent Computation: Self-Organizing, Collective, and Cooperative Phenomena in Natural and Artificial Computing Networks*, MIT Press (1991).

Stephen I. Gallant, *Neural Network Learning and Expert Systems*, MIT Press (1993).

M. Gardner, *Wheels, Life and Other Mathematical Amusements*, W. H. Freeman (1983).

Peter Grassberger, "Efficient large-scale simulations of a uniformly driven system," Phys. Rev. E **49**, 2436–2444 (1994). Grassberger considers substantially larger lattices and longer simulation times than that used by Olami et al. and finds that the Olami, Feder, Christensen model does not exhibit power law scaling.

G. Grinstein and C. Jayaprakash, "Simple models of self-organized criticality," Computers in Physics **9**, 164 (1995).

G. Grinstein, Terence Hwa, and Henrik Jeldtoft Jensen, "$1/f^{\alpha}$ noise in dissipative transport," Phys. Rev. A **45**, R559–R562 (1992).

B. Hayes, "Computer recreations," Sci. Am. **250** (3), 12–21 (1984). An introduction to cellular automata.

J. E. Hanson and J. P. Crutchfield, "Computational mechanics of cellular automata: An example," Physica D **103**, 169–189 (1997). The authors discuss energence in cellular automata.

Robert Herman, editor, *The Theory of Traffic Flow*, Elsevier (1961).

John Hertz, Anders Krogh, and Richard G. Palmer, *Introduction to the Theory of Neural Computation*, Addison-Wesley (1991).

J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," Proc. Natl. Acad. Sci. USA **79**, 2554–2558 (1982).

Navot Israeli and Nigel Goldenfeld, "Computational irreducibility and the predictability of complex physical systems," Phys. Rev. Lett. **92**, 074105 (2004).

H. M. Jaeger, Chu-heng Liu, and Sidney R. Nagel, "Relaxation at the angle of repose," Phys. Rev. Lett. **62**, 40 (1989). These authors discuss experiments on real sandpiles.

W. Klein, C. Ferguson, and J. B. Rundle, "Spinodals and scaling in slider block models," in *Reduction and Predictability of Natural Disasters*, J. B. Rundle, D. L. Turcotte, and W. Klein, editors, Addison-Wesley (1995). Also see C. D. Ferguson, W. Klein, and John B. Rundle, "Spinodals, scaling, and ergodicity in a threshold model with long-range stress transfer," Phys. Rev. E **60**, 1359–1373 (1999).

J. A. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press (1992).

Chris Langton, "Studying artificial life with cellular automata," Physica D **22**, 120–149 (1986). See also Christopher G. Langton, editor, *Artificial Life*, Addison-Wesley (1989); Christopher G. Langton, Charles Taylor, J. Doyne Farmer, and Steen Rasmussen, editors, *Artificial Life II*, Addison-Wesley (1989); Christopher G. Langton, editor, *Artificial Life III*, Addison-Wesley (1994).

Roger Lewin, *Complexity: Life at the Edge of Chaos*, University of Chicago Press (2000). A popular exposition of complexity theory.

Sergei Maslov, Maya Paczuski, and Per Bak, "Avalanches and 1/f noise in evolution and growth models," Phys. Rev. Lett. **73**, 2162 (1994).

Stephan Mertens, "Computational complexity for physicists," Computing in Science and Engineering **4** (3), 31–47 (2002).

Takahiro Mori and Hikaru Kawamura, "Simulation study of the one-dimensional Burridge-Knopoff model of earthquakes," physics/0504218.

K. Nagel and M. Schreckenberg, "A cellular automaton model for freeway traffic," J. Phys. I France **2**, 2221–2229 (1992). Also see `<http://www.traffic.uni-duisburg.de/>`.

Kai Nagel, Dietrich E. Wolf, Peter Wagner, and Patrice Simon, "Two-lane traffic rules for cellular automata: A systematic approach," Phys. Rev. E **58**, 1425–1437 (1998).

M. E. J. Newman, "The structure and function of complex networks," SIAM Rev. **45**, 167–256 (2003).

M. E. J. Newman, "Detecting community structure in networks," Eur. Phys. J. B **38**, 321–330 (2004); M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," Phys. Rev. E **69**, 026113-1–15 (2004). These papers describe an algorithm for detecting the heirarchical structure of networks.

J. A. Niesse, R. P. White, and H. R. Mayne, "Genetic algorithm approaches to minimum energy geometry of aromatic hydrocarbon clusters," J. Chem. Phys. **108**, 2208–2218 (1998).

Z. Olami, H. J. S. Feder, and K. Christensen, "Self-organized criticality in a continuous, nonconservative cellular automaton modeling earthquakes," Phys. Rev. Lett. **68**, 1244 (1992).

Suzana Moss de Oliveira, Jorge S. Sá Martins, Paulo Murilo C. de Oliveira, Karen Luz-Burgoa, Armando Ticona, and Thadeu J. P. Penna, "The Penna model for biological aging and speciation," Computing in Science and Engineering **6** (3), 74–81 (2004). Also see Dietrich Stauffer, "The complexity of biological ageing," cond-mat/0310038.

Elaine S. Oran and Jay P. Boris, *Numerical Simulation of Reactive Flow*, second edition, Cambridge University Press (2002). Although much of this book assumes an understanding of fluid dynamics, the discussion of simulation methods and the numerical solution of the differential equations of fluid flow does not require much background.

Michel Peyrard, "Nonlinear dynamics and statistical physics of DNA," Nonlinearity **17**, R1–R40 (2004). The author describes a simple mechanical model of DNA (see Fig. 10 and Eq. (1)) that is in the same spirit as the Burridge-Knopoff model of earthquakes.

William Poundstone, *The Recursive Universe*, Contemporary Books (1985). A book on the Game of Life that attempts to draw analogies between the patterns of Life and ideas of information theory and cosmology.

Drek de Solla Price, "Networks of scientific papers," Science **149**, 510–515 (1965); "A genral theory of bibliometric and other cummulative advantage processes," J. Amer. Soc. Inform. Sci. **27**, 292–306 (1976). Possibly the first description of a scale free network and the explanation for power law distributions.

Daniel H. Rothman and Stéphane Zalesk, *Lattice-Gas Cellular Automata*, Cambridge University Press (1997). This text includes a discussion of fluid flow through porous media as well as the lattice-Boltzmann method for simulating fluids. Also see Daniel H. Rothman and Stéphane Zaleski, "Lattice-gas models of phase separation: interfaces, phase transitions, and multiphase flow," Rev. Mod. Phys. **66**, 1417–1479 (1994).

David E. Rumelhart and James L. McClelland, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. 1: *Foundations*, MIT Press (1986). See also Vol. 2 on applications.

Robert Savit, Radu Manuca, and Rick Riolo, "Adaptive competition, market efficiency, and phase transitions," Phys. Rev. Lett. **82**, 2203 (1999). Analysis of the scaling behavior of the minority game.

Herbert A Simon, "On a class of skew distribution functions," Biometrika, **42**, 425–440 (1955). An early paper that shows power laws coming from preferential attachment.

V. Sood and S. Redner, "Voter model on heterogeneous graphs," Phys. Rev. Lett. **94**, 178701 (2005).

Dietrich Stauffer, "Monte Carlo simulations of Sznajd models," J. Artificial Societies and Social Simulation **5**(1) paper #4 (2002). This paper and other relevant papers can be found at <jasss.soc.surrey.ac.uk>.

Dietrich Stauffer, "Cellular automata," Chapter 9 in *Fractals and Disordered Systems*, Armin Bunde and Shlomo Havlin, editors, Springer-Verlag (1991). Also see Dietrich Stauffer, "Programming cellular automata," Computers in Physics **5** (1), 62 (1991).

Daniel L. Stein, editor, *Lectures in the Sciences of Complexity*, Vol. 1, Addison-Wesley (1989); Erica Jen, editor, *Lectures in Complex Systems*, Vol. 2, Addison-Wesley (1990); Daniel L. Stein and Lynn Nadel, editors, *Lectures in Complex Systems*, Vol. 3, Addison-Wesley (1991).

Patrick Sutton and Sheri Boyden, "Genetic algorithms: A general search procedure," Am. J. Phys. **62**, 549–552 (1994). This readable paper discusses the application of genetic algorithms to Ising models and function optimization.

K. Sznajd-Weron and J. Sznajd, "Opinion evolution in closed community," Int. J. Mod. Phys. C **11** (6), 1157–1165 (2000).

Tommaso Toffoli and Norman Margolus, *Cellular Automata Machines – A New Environment for Modeling*, MIT Press (1987). See also Norman Margolus and Tommaso Toffoli, "Cellular automata machines," in the volume edited by Doolen et al.

D. J. Tritton, *Physical Fluid Dynamics*, second edition, Oxford Science Publications (1988). An excellent introductory text that integrates theory and experiment. Although there is only a brief discussion of numerical work, the text provides the background useful for simulating fluids.

M. Mitchell Waldrop, *Complexity: The Emerging Science at the Edge of Order and Chaos*, Simon and Schuster (1992). A popular exposition of complexity theory.

Stephen Wolfram, editor, *Theory and Applications of Cellular Automata,* World Scientific (1986). A collection of research papers on cellular automata that range in difficulty from straightforward to specialists only. An extensive annotated bibliography also is given. Two papers in this collection that discuss the classification of one-dimensional cellular automata are S. Wolfram, "Statistical mechanics of cellular automata," Rev. Mod. Phys. **55**, 601–644 (1983), and S. Wolfram, "Universality and complexity in cellular automata," Physica B **10**, 1–35 (1984).

Stephen Wolfram, *A New Kind of Science*, Wolfram Media (2002). This book discusses many important ideas and computer experiments on cellular automata. More information about can be found at <http://www.stephenwolfram.com/>. An interesting review of this book is given by L. Kadanoff, "Wolfram on cellular automata," Phys. Today **55** (7), 55–56 (2002).

László Zalányi, Gábor Csárdi, Tamás Kiss, Máté Lengyel, Rebecca Warner, Jan Tobochnik, and Péter Érdi, "Properties of a random attachment growing network," Phys. Rev. E. **68**, 066104-1–9 (2003).