

Chapter 14

Complex Systems

©2004 by Harvey Gould, Jan Tobochnik, and Wolfgang Christian
20 December 2004

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

14.1 Cellular Automata

Part of the fascination of physics is that it allows us in many cases 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 we will discuss are known as *cellular automata*. Cellular automata were originally introduced by von Neumann and Ulam in 1948 as an idealization of biological self-reproduction, and are examples of discrete dynamical systems that can be simulated exactly on a digital computer. A cellular automaton can be thought of as a lattice of sites or a checkerboard with colored squares (the cells). Each cell changes its color at the tick of an external clock according to a rule based on the present configuration of the cells in its neighborhood.

Because the original motivation for studying cellular automata was their biological aspects, the discrete sites 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 than cells, except when we are explicitly discussing biological systems.

Cellular automata 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. The important characteristics of cellular automata include the following:

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

2. Time is discrete, and the value of each site is updated in a sequence of discrete time steps.
3. The rule for the new value of a site depends only on the values of a *local* neighborhood of sites near it.
4. The variables at each site are updated *simultaneously* based on the values of the variables at the previous time step.

We first consider one-dimensional cellular automata and assume that the neighborhood of a given site to be the site itself and the sites immediately to the left and right of it. Each site is assumed to have two states (a Boolean automata). An example of such a rule is illustrated in Fig. 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 automata, there are $2^8 = 256$ possible rules.

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 time 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 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, $90 = 2^1 + 2^3 + 2^4 + 2^6$.

Class `OneDimensionalAutomatonApp` takes as input the decimal representation of the rule and produces the rule matrix (in array `update`). This array is used to update each site on the lattice using periodic boundary conditions. We use the `CellLattice` class, `automaton`, to represent the spatial and time development of the cellular automaton. The sites are represented by the array `row` and are plotted in the horizontal direction; time increases in the vertical direction.

Listing 14.1: One-dimensional cellular automata class.

```
/* One-Dimensional Cellular Automata */

package org.opensourcephysics.sip.ch14.ca;

import org.opensourcephysics.controls.*;

import org.opensourcephysics.display.*;

import org.opensourcephysics.display2d.CellLattice;

import java.awt.Color;
```

```

public class OneDimensionalAutomatonApp extends AbstractCalculation {

    DrawingPanel plottingPanel = new PlottingPanel("x","t","");

    DrawingFrame frame = new DrawingFrame(plottingPanel);

    CellLattice automaton;           // cellular automata lattice

    byte[] update = new byte[8];      // update[] maps neighborhood configurations to 0 or 1

    byte[] row;

    public OneDimensionalAutomatonApp() {

        frame.show();

    }

    public void calculate() {

        control.clearMessages();

        int L = control.getInt("Linear dimension");

        int tmax = control.getInt("Maximum time");

        automaton = new CellLattice(L,tmax);

        automaton.setIndexedColor(0, Color.blue);           // empty

        automaton.setIndexedColor(1, Color.lightGray);      // occupied

        plottingPanel.addDrawable(automaton);

        plottingPanel.setPreferredMinMaxX(0,L);

        plottingPanel.setPreferredMinMaxY(0,tmax);

        setRule(control.getInt("Rule number"));

        row = new byte[L];

        row[L/2] = (byte)1;
    }
}

```

```

    automaton.setRow(0,0,row);                // seed the first row

    for (int t = 1; t < tmax; t++) {

        iterate(t-1,L);

        automaton.setRow(t,0,row);

    }

    plottingPanel.repaint();
}

public void iterate(int tprev, int L) {

    int neighborhood = 2*automaton.getValue(L-1,tprev) + automaton.getValue(0,tprev);

    for (int i = 0; i < L; i++) {

        // first term picks out first 2 bits and moves result one bit to the left

        // lowest order bit taken from (i+1)st value

        neighborhood = 2*(neighborhood&3) + automaton.getValue((i+1)%L, tprev);

        row[i] = 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--) {

        update[i] = (byte)((ruleNumber >> i) % 2); // pick out ith bit

        control.print("  " + update[i] + "    ");

    }

}

```

```

        control.println ();
    }

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

    public static void main(String args[]) {
        OneDimensionalAutomatonApp app = new OneDimensionalAutomatonApp();
        Control control = new CalculationControl(app);
        app.setControl(control);
    }
}

```

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 14.2.

Problem 14.1. One-dimensional cellular automata

- a. Use `OneDimensionalAutomatonApp` and rule 90 shown in Fig. 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 (a seed) at the midpoint of the lattice. It is sufficient to consider the evolution for approximately twenty steps, although the default value is one hundred steps. Is the resulting pattern of nonzero sites self-similar? If so, characterize the pattern by a fractal dimension.
- b. Consider 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 termed rule 10010110 or rule $150 = 2^1 + 2^2 + 2^4 + 2^7$. Start with a single seed site.

- c. Choose a random initial configuration for which the independent probability for each site to have the value 1 is 50%; otherwise, the value of a site is 0. Consider the time 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 changes in the initial conditions? Does the nature of the patterns depend on the use or nonuse of periodic boundary conditions?

Because the dynamical behavior of many of the 256 one-dimensional Boolean cellular automata is uninteresting, we also consider one-dimensional Boolean cellular automata with larger, but local neighborhoods (the neighborhood of a site includes the site itself). Because a larger neighborhood implies that there are many more possible update rules, it is convenient to place some reasonable restrictions on the rules. First, we assume that the rules are symmetric, for example, the neighborhood 100 produces the same value for the central site as 001. We also assume 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 (Wolfram, 1984).

A simple way of coding the rules which is consistent with these requirements is as follows. Call the size of the neighborhood z if the neighborhood includes $2z + 1$ sites. Each rule is labeled by a sequence of 0s and 1s so that the sequence indicates which sums set the central site equal to unity. If the lowest order digit is 1, then the central site is set to unity if the sum is 0. If the next digit is 1, then the central site is set to unity if the sum is 1, etc. For example, the number 10011 indicates that the central site will be set to unity if the number of neighbors equal to unity is 0, 1, or 4.

Problem 14.2. More one-dimensional cellular automata

- a. Modify `OneDimensionalAutomatonApp` so that it incorporates the possible rules discussed in the text for 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 classes:
 - i. A homogeneous state where every site is either 0 or 1. An example is rule 8.
 - ii. A pattern consisting of separate stable or periodic regions. An example is rule 4.
 - iii. A chaotic, aperiodic pattern. An example is rule 10.
 - 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 10100 and 110100 are the only examples of complex behavior (class 4). Describe how the behavior of these two rules differs from the behavior of the other rules. Determine the fraction of the rules belonging to the four classes.
- c. Repeat part (b) for $z = 3$.
- d. Assume that sites can have three values, 0, 1, and 2. Classify the behavior of the possible rules for the case $z = 1$.

The results of Problem 14.2 suggest 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. These localized structures are the focus of *complexity theory* whose goal is to explain complex phenomena in nature.

The major idea of complexity theory is that simple rules can lead to complex behavior. There also is an implicit notion that most interesting macroscopic phenomena cannot be predicted starting from a microscopic description. By analogy, we cannot in general predict whether a non-linear dynamical system will exhibit chaotic behavior. In particular, we might expect that one-dimensional cellular automata that exhibit complex, localized structures may not be predictable by a rule that works by course graining of the cells. That is, is it possible to find a geometrical rule that groups cells into a coarse grained “supercell,” and then find an update rule for the coarse grained cells which gives the same qualitative behavior as the original dynamics? Recently, Israeli and Goldenfeld found a complex cellular automaton for which the combination of an update rule on the coarse grained cells is equivalent to running the original update rule on the original cells and then coarse graining. In other words, the update rule and coarse graining can be done in either order.

We have achieved this goal for many physical systems that are not considered complex. For example, we can describe the behavior of a baseball without knowing the detailed behavior of every molecule in the baseball. But can we do achieve such a description for biological systems? If we can for complex systems, then we would have a better chance of finding general laws of nature that work at macroscopic levels. The following problem explores some of these ideas.

***Problem 14.3.** Coarse graining one-dimensional cellular automata

- a. Consider rule 146 and determine its evolution starting from a random configuration of 1s and 0s. What class of behavior does it exhibit?
- b. In addition to drawing the state of each site, modify your program so that it draws the state of the coarse grained cells. Define a coarse grained cell by grouping three neighboring cells such that when all three of the original cells are 1, the coarse grained cell is 1; otherwise, the coarse grained cell is 0. Expand the window for the coarse grained system so that the original system and the coarse grained system are the same size. Begin with an initial random configuration of 150 cells and run for 45 iterations and draw the behavior of the original lattice and the coarse grained lattice.
- c. Coarse grain the same initial configuration as in part (a), and then run the coarse grained system using rule 128 for 15 iterations. How do the two simulations compare? What information is lost when the system is first coarse grained and then simulated compared to when the original system is simulated and then coarse grained at the end?
- d. Repeat part (b) beginning with rule 150 and replace every pair of cells with a coarse grained cell such that if two cells are both 0 or both 1, then the coarse grained cell is 1; otherwise, it is 0. Run the coarse grained system using rule 105. Use 120 original cells for 60 iterations. How do the two results compare?

Traffic models. Physicists have been at the forefront of research to develop a more systematic approach to the characterization and control of traffic. Much of this work was initiated by Robert Herman in the late 1950’s while he was at General Motors. 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 approach used by Herman and Montroll and others 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 freeway where cars enter at one end and exit at the other end. The Nagel-Schreckenberg model consists of a one-dimensional cellular automaton that represents a freeway. 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 p , the probability 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. Thus $v_i \rightarrow v_i + 1$. This change models the process of accelerating 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, $v_i \rightarrow v_i - 1$.
4. Update the position x_i so that $x_i(t + 1) = x_i(t) + v_i$.

The above ordering of the steps ensures that cars do not overlap.

Listing 14.2: One lane freeway class.

```
package org.opensourcephysics.sip.ch14.traffic ;
import org.opensourcephysics.display.*;
import org.opensourcephysics.display2d.*;
import org.opensourcephysics.numerics.*;
import java.awt.*;

/**
 * Freeway uses the Nagel–Schreckenberg model of single lane traffic
 *
 * @author Jan Tobochnik
 * @version 1.0
 */

public class Freeway implements Drawable {
    public int [] v, x, xTemp;
    public CellLattice 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;
private byte row[];

/**
 * Initializes arrays and starting configuration of cars.
 */
public void initialize () {
    x = new int[numberOfCars];
    xTemp = new int[numberOfCars]; // used to allow parallel updating
    v = new int[numberOfCars];
    spaceTime = new CellLattice(roadLength,100);
    row = new byte[roadLength];
    road = new CellLattice(roadLength,1);
    road.setIndexedColor(0, Color.red);
    road.setIndexedColor(1, Color.green);
    spaceTime.setIndexedColor(0, Color.red);
    spaceTime.setIndexedColor(1, 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;
}

/**
 * Does one time step
 */
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 correctly treats one car on road
            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 < 100) {
            spaceTime.setRow(t,0,new byte[roadLength]);
            for(int i = 0; i < numberOfCars; i++) spaceTime.setValue(x[i],t,(byte)1);
        }
        else {
            for(int y = 0; y < 99; y++) {
                for(int i = 0; i < roadLength; i++) row[i] = spaceTime.getValue(i,y+1);
                spaceTime.setRow(y,0,row);
            }
            spaceTime.setRow(99,0,new byte[roadLength]);
            for(int i = 0; i < numberOfCars; i++) spaceTime.setValue(x[i],99,(byte)1);
        }
    }

    /**
     * Draws freeway.
     */
    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,10,10);
        g.drawString("Flow = " + (double)flow/(roadLength*steps),10,30);
        g.drawString("Density = " + (double)numberOfCars/(roadLength),10,50);
    }
}

```

The target class, **FreewayApp**, shows the movement of the cars and a space time diagram which shows time on the vertical axis and space on the horizontal axis. When the number of iterations becomes equal to 100, the diagram scrolls down. The flow rate is the average of the car velocities divided by the length of the freeway. Thus, two cars moving at constant velocity will have twice the flow rate of one car moving at the same velocity.

Listing 14.3: FreewayApp class.

```

package org.opensourcephysics.sip.ch14.traffic ;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.display.*;
import javax.swing.*;
import java.awt.geom.*;
import java.awt.*;
import java.util.*;
import java.text.DecimalFormat;

```

```

/**
 * FreewayApp models traffic flow
 *
 * @author Jan Tobochnik
 */
public class FreewayApp extends AbstractAnimation {
    Freeway freeway;
    DrawingPanel drawingPanel;
    DrawingFrame drawingFrame;
    PlottingPanel spaceTimePanel;
    DrawingFrame spaceTimeFrame;

    /**
     * Constructs the FreewayApp.
     *
     */
    public FreewayApp() {
        freeway = new Freeway();
        drawingPanel = new DrawingPanel();
        drawingPanel.addDrawable(freeway);
        drawingFrame = new DrawingFrame(drawingPanel);
        spaceTimePanel = new PlottingPanel("space", "time", "Space Time Diagram");
        spaceTimeFrame = new DrawingFrame(spaceTimePanel);
    }

    /**
     * Initializes the animation using the values in the control.
     *
     */
    public void initializeAnimation() {
        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");
        drawingPanel.setPreferredMinMax(0, freeway.roadLength, -3, 4);
        freeway.initialize();
        spaceTimePanel.clear();
        spaceTimePanel.addDrawable(freeway.spaceTime);
        drawingPanel.render();
    }

    /**
     * Does one iteration.
     *
     */
    public void doStep() {
        int stepsBetweenPlots = control.getInt("Steps between plots");
        for(int i = 0; i < stepsBetweenPlots; i++) freeway.step();
        drawingPanel.render();
        spaceTimePanel.repaint();
    }

```

```

    }

    /**
     * Resets animation to a predefined state.
     */
    public void resetAnimation() {
        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);
    }

    /**
     * Resets data without changing configuration
     */
    public void resetAverages() {
        freeway.flow = 0;
        freeway.steps = 0;
    }

    /**
     * Starts Java application.
     * @param args command line parameters
     */
    public static void main (String[] args) {
        FreewayApp app = new FreewayApp();
        AnimationControl control = new AnimationControl(app);
        control.addButton("resetAverages", "resetAverages");
        app.setControl(control);
    }
}

```

Problem 14.4. 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 for awhile before recording the flow rate. Repeat the simulation with a new 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 in terms of what you would expect. 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 the data you have obtained with $v_{\max} = 2$. Also try $v_{\max} = 5$. Are there any qualitative differences in the behavior of the cars?
- e. Explore the effect of changing the slowing down probability by trying $p = 0.2$ and $p = 0.8$.
- f. Modify your program to add an on and off ramp that are separated by a fixed distance. One way to model the exit ramp is to assume that whenever a car passes the exit ramp, there is a certain probability of the car leaving the freeway. Then the next car to pass the on ramp will leave the freeway. Another possibility is to choose a car at random and have it slow down as it approaches the exit ramp and exit. To maintain a constant density allow a car to enter whenever a car leaves the freeway. What is the effect of adding on and off ramp?
- g. Modify your program to simulate a more realistic two lane highway. You will need to choose rules for moving from one lane to the other. Here are some possibilities to explore. A 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 is a free space to the left. A 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 (cars and trucks) with different values for $v_{\max}$. Describe the behavior for this situation. Compute separate values for the truck and car flows as well as the total flow. Also, compute the average speed of the trucks and compare it with that of cars.

One-dimensional models are too limited to study the complexity of nature, and we now consider several two-dimensional models. The philosophy is the same except that the neighborhood contains more sites. For the eight neighbor sites shown in Fig. 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 set up our rule matrix based on other considerations.

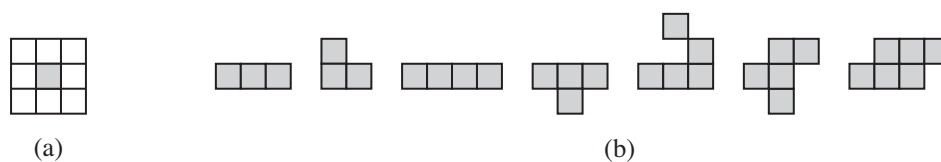


Figure 14.2: (a) The local neighborhood of a site in the Game of Life is given by the sum of its eight neighbors. (b) Examples of initial configurations for the Game of Life, some of which lead to interesting patterns. Live cells are shaded.

The Game of Life. The rule matrix incorporated in **LifeApp** implements a popular two-dimensional cellular automata model 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 Fig. 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 time step 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 14.4: Game of Life Class.

```

package org.opensourcephysics.sip.ch14.ca;

import org.opensourcephysics.controls.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.display2d.*;
import java.awt.*;
import java.awt.event.*;

public class LifeApp extends AbstractAnimation implements InteractiveMouseHandler {

    PlottingPanel    plottingPanel = new PlottingPanel("x","y"," ");
    DrawingFrame     frame = new DrawingFrame(plottingPanel);
    CellLattice      ca; // cellular automata lattice

    int [] update = new int[512];

    int L;

    int [][] newCA;

    public LifeApp() {
        frame.show();
        setRule();
    }

    public void initializeAnimation() {
        L = control.getInt("Lattice size");

```

```

plottingPanel.clear ();    // added by WC

ca = new CellLattice(L,L);

ca.setIndexedColor(0, Color.red);

ca.setIndexedColor(1, Color.green);

ca.setBlock(0,0,new int[L][L]);

plottingPanel.addDrawable(ca);

//plottingPanel.setPreferredMinMaxX(0,L); // not needed. Removed by WC

//plottingPanel.setPreferredMinMaxY(0,L);

plottingPanel.setInteractiveMouseHandler(this);

plottingPanel.repaint ();
}

public void handleMouseAction(InteractivePanel panel, MouseEvent evt) {

    panel.handleMouseAction(panel, evt);

    switch (panel.getMouseAction()) {

        case InteractivePanel.MOUSE_PRESSED:

            int ix= (int)panel.getMouseX();

            int iy= (int)panel.getMouseY();

            if (ix < 0 || iy < 0 || ix >= L || iy >= L)

                return; // outside the lattice

            if (ca.getValue(ix, iy) == (byte)0) // changed by WC

                ca.setValue(ix, iy, (byte)1);

            else

                ca.setValue(ix, iy, (byte)0);

            plottingPanel.repaint ();

```

```
        break;

    }

}

public void doStep() {
    newCA = new int[L][L];

    for(int i = 0; i < L; i++)
        for(int j = 0; j < L; j++) {
            int index = neighborhood(i,j);

            newCA[i][j] = update[index]; // changed by WC
        }

    ca.setBlock(0,0,newCA);

    plottingPanel.repaint();
}

public int neighborhood(int i, int j) {
    int ip = (i + 1) % L;
    int im = (L + i - 1) % L;
    int jp = (j + 1) % L;
    int jm = (L + j - 1) % L;

    return ca.getValue(i,jp) + 2*ca.getValue(i,jm) + 4*ca.getValue(im,j) +
           8*ca.getValue(ip,j) + 16*ca.getValue(ip,jp) + 32*ca.getValue(ip,jm) +
           64*ca.getValue(im,jp) + 128*ca.getValue(im,jm) + 256*ca.getValue(i,j);
}
```

```
public int powerOfTwo(int n) {  
    int x = 1;  
    for(int i = 0; i < n; i++)  
        x *= 2;  
    return x;  
}  
  
public void setRule() {  
    // three neighbors alive  
    for(int nn1 = 0; nn1 < 6; nn1++) {  
        int a1 = powerOfTwo(nn1);  
        for(int nn2 = nn1+1; nn2 < 7; nn2++) {  
            int a2 = powerOfTwo(nn2);  
            for(int nn3 = nn2+1; nn3 < 8; nn3++) {  
                int a3 = powerOfTwo(nn3);  
                int index = a1 + a2 + a3;  
                update[index] = 1;    // center dead  
                update[index+256] = 1; // center alive  
            }  
        }  
    }  
    // two neighbors and center alive  
    for(int nn1 = 0; nn1 < 7; nn1++) {  
        int a1 = powerOfTwo(nn1);  
        for(int nn2 = nn1+1; nn2 < 8; nn2++) {
```

```

        int a2 = powerOfTwo(nn2);

        update[a1 + a2 + 256] = 1;
    }
}

public void resetAnimation(){

    super.resetAnimation();

    control.setValue("Lattice size" , 20);
}

/**
 * Clears the cells .
 */
public void clear() {
    ca.setBlock (0, 0, new int[L][L]);

    plottingPanel.repaint ();
}

/* ----- application target ----- */

public static void main(String args[]) {
    LifeApp app = new LifeApp();

    AnimationControl control = new AnimationControl(app);

```

```

        control.addButton("clear", "Clear"); // optional custom action

        app.setControl(control);
    }
}

```

Note that `LifeApp` has not been optimized for the Game of Life and is written so that it can be easily modified for other cellular automata rules by changing method `setrule`.

Problem 14.5. The Game of Life

- a. `LifeApp` allows the user to determine the initial configuration interactively by clicking on sites to change their value before hitting the start button. Choose several initial configurations with a small number of live cells and investigate the different types of patterns that emerge. Some suggested initial configurations are shown in Fig. 14.2b. Does it matter whether you use fixed or periodic boundary conditions?
- b. Modify `LifeApp` so that each cell is initially alive with a 50% probability. 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.* As we found in part (b), the system will develop structure even if each cell is randomly populated at $t = 0$. One measure of the increasing order in the system was introduced by Schulman and Seiden and is analogous to the entropy S . The idea is to divide the system into boxes of linear dimension b and determine n_i , the number of live cells in the i th box. The quantity S is given by

$$S = \frac{1}{L^2} \log_2 \prod_{i=1}^{(L/b)^2} \binom{b^2}{n_i}. \quad (14.1)$$

The argument of the logarithm in (14.1) is the total number of microscopic states associated with a given sequence of n_i . Roughly speaking, S measures the extent to which the live cells are correlated. Determine S as a function of time starting from a 50% random configuration. First consider $L = 50$ and $b = 2, 3, 4$, and 5 . Average over at least ten runs. Describe how the behavior of S depends on the level of coarse graining determined by the value of b . Does S decrease monotonically with time? Does S reach an equilibrium value? Increase L and the number of runs.

- e.* In statistical mechanics, you probably learned that the entropy tends to increase as the system approaches equilibrium. Why does the entropy in part (d) decrease?

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.

14.2 Self-Organized Critical Phenomenon

In nature we rarely see 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. When such rare 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 very large events are part of a distribution of events and do not depend on special conditions or external forces.

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 $N(s)$ of size s follows a power law:

$$N(s) \sim s^{-\alpha}. \quad (\text{no characteristic scale}) \quad (14.2)$$

If $\alpha \approx 1$, the form (14.2) implies that there would be one large event of size 1000 for every 1000 events of size one. One implication of the form (14.2) is that there is no characteristic scale. Systems whose correlation or distribution functions decay as power laws are said to be *scale invariant*. This terminology reflects the fact that power laws look the same on all scales. For example, the replacement $s \rightarrow bs$ in the function $N(s) = As^{-\alpha}$ yields a function $\tilde{N}(s)$ that is indistinguishable from $N(s)$, except for a change in the amplitude A by the factor $b^{-\alpha}$.

In contrast to the power law dependence of $N(s)$ in (14.2), the result of combining a large number of independently acting random events is a Gaussian distribution (see Problem 7.15). In this case $N(s)$ has the form

$$N(s) \sim e^{-(s/s_0)^2}. \quad (\text{characteristic scale}) \quad (14.3)$$

Scale invariance does not hold for functions that decay exponentially, because the replacement $s \rightarrow bs$ in the function $e^{-(s/s_0)^2}$ changes s_0 (the characteristic scale or size of s) by the factor b . We note that for a power law distribution, there are events of all sizes, but for a Gaussian distribution, there are practically speaking no large events.

A common 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 stay more or less where they land, but after a while there will be small avalanches during which the grains move so that the local slope of the pile is not too large. 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 a configuration belonging to this state, 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 gain more insight into the nature of self-organized critical phenomena by considering some simple models. We begin with a one-dimensional model of a sandpile. Consider a lattice of L sites and let the height at each site be represented by the array element `height[i]`. One grain of sand is added to the left-most site, `height[0]++`, at each time step. During this time step all the sites are checked to see if `height[i] - height[i+1] > 1`. All sites that satisfy this condition are marked for “toppling.” Next each marked site is toppled. A simple rule is to let `height[i]--` and `height[i+1]++`, that is, the sand falls to the right. Any grains of sand that go beyond $i = L$ are lost forever. The class `Sandpile` implements this simple model. We will find in Problem 14.6 that slightly more complicated rules are necessary to find nontrivial behavior.

Listing 14.5: One Dimensional Sandpile Model.

```
package org.opensourcephysics.sip.ch14.sandpile;
import org.opensourcephysics.display.*;
import org.opensourcephysics.numerics.*;
import java.awt.*;

public class Sandpile implements Drawable {
    public int [] height, move;
    public double [] distribution;
    public int L;
    public int numberOfGrains = 0;

    public void initialize () {
        height = new int[L+1];
        move = new int[L+1];
        resetAverages();
    }

    public void step() {
        height[0]++; // add grain to first site
        numberOfGrains++;
        int topple = 0;
        boolean unstable = true;
        do {
            unstable = check();
            if(unstable) {
                topple += slide();
            }
        }
        while(unstable);
        distribution [topple]++;
    }
}
```

```

public boolean check() {
    boolean unstable = false;
    for(int i = 0; i < L; i++)
        if(height[i] - height[i+1] > 1) {
            move[i] = 1;
            unstable = true;
        }
        else
            move[i] = 0;
    return unstable;
}

public int slide() {
    int topple = 0;
    for(int i = 0; i < L; i++) {
        if(move[i] == 1) {
            height[i]--;
            topple++;
            if(i < L-1) height[i+1]++;
        }
    }
    return topple;
}

public void draw (DrawingPanel myWorld, Graphics g) {
    if (height == null) {
        return;
    }
    int dx = myWorld.xToPix(1.0) - myWorld.xToPix(0);
    int ypix = myWorld.yToPix(0);
    for(int i = 0; i < L; i++) {
        int dy = myWorld.xToPix(height[i]) - myWorld.xToPix(0);
        int xpix = myWorld.xToPix(i);
        g.fillRect (xpix, ypix-dy, dx, dy);
    }
}

public void resetAverages() {
    distribution = new double[L+1];
    numberOfGrains = 0;
}
}

```

Problem 14.6. One-dimensional sandpiles

- a. Use `Sandpile` and `SandpileApp` with $L = 20$. (The target class can be downloaded from the `ch14` directory.) Where is the sand added? What is the slope of the sand pile after a long time?
- b. The `Sandpile` class computes the number of sites s that topple during each time step and plots

the distribution of toppling sites, $N(s)$, versus s . Modify the program so that $N(s)$ is computed only after the sandpile reaches a steady state. Is the behavior of $N(s)$ interesting for this model?

- c. Modify the rules so that a site which topples loses *two* grains of sand, one to its nearest neighbor to the right and the other to its next nearest neighbor to the right. Plot the distribution $N(s)$ versus s after steady state behavior is reached. Is there a range of values of s for which $N(s)$ shows power law behavior? If so, make the appropriate plot and estimate the critical exponent α .
- d. Introduce the variable `slope[i] = height[i+1] - height[i]`, and convince yourself that the same results are obtained by replacing `height[i]` by `slope[i]` and using the rule `slope[i] > 1` for toppling. The variable `slope[i]` is called the local slope. Modify your program so that `slope[i]` is used instead of `height[i]` and adopt the toppling rule used in part (c). Do your results change if you add a grain of sand at each time step at random anywhere in the lattice?
- e. Use the same rule that you considered in parts (c) and (d) and compute $N(s)$ for different values of L and systematically investigate the importance of finite size effects. Average $N(s)$ over many updates and obtain your best estimate for the critical exponent α .

In Problem 14.6 we saw that the fundamental variable is the local slope. Most sandpile models are described using this variable. In the literature many authors refer to the height when they really mean the local slope. In Problem 14.7 we explore the behavior of a two-dimensional model of a sandpile.

Problem 14.7. Two-dimensional sandpile

- a. Write a program to simulate a two-dimensional sandpile using the rule that a site topples if `slope[i] > 3`. The pile is grown by choosing a site at random and adding one grain, that is, `slope[i] → slope[i] + 1`. If site `i` topples (exceeds its critical value), it distributes four grains of sand to its four nearest neighbors (on a square lattice), that is, `slope[i] → slope[i] - 4`. At the edges or corners, only three or two neighbors, respectively, are affected, and the sand that goes outside the boundary of the lattice is lost. Color code the sites according to their value of `slope[i]`. In class `Sandpile` we checked all sites for stability, but in two dimensions such a check would take too much time if the size of the lattice were sufficiently large. For this reason we suggest that you introduce two arrays that contain the x and y coordinates, respectively, of those sites that need to be checked for stability. Each time a site topples, add its neighbors' positions to these arrays. Then remove the last site from the arrays and check its stability. Continue removing sites and checking their stability, and adding neighbors to the array until there are no more sites to check.
- b. After the critical state has been reached, compute the number of sites s that topple in response to the addition of a single grain. Then compute the distribution of toppling sizes $N(s)$ and determine if $N(s)$ exhibits power law behavior. Begin with $L = 10$ and then consider larger values of L .

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

Earthquakes. Do model sandpiles have anything in common with earthquakes? The empirical Gutenberg-Richter law for $N(E)$, the number of earthquakes with energy release E , is consistent with power law behavior:

$$N(E) \sim E^{-b}, \quad (14.4)$$

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 (14.4) is that there is nothing special about large earthquakes. That is, if we could wait a couple of million years, we would likely observe earthquakes of size ten following the same Gutenberg-Richter law. In Problem 14.8 we explore the behavior of a model of tectonic plate motion which suggests that the Gutenberg-Richter law is a consequence of self-organized criticality.

Problem 14.8. Simple earthquake model

Given the long time scales between earthquakes and the complexity of the historical record, there is considerable interest in developing ways of studying earthquakes using simulations. The Burridge and Knopoff model of fault systems consists of a system of coupled masses in contact with a moving rough surface. The masses are subjected to static and dynamic frictional forces, and also are pulled by an external force corresponding to slow tectonic plate motion. The major difficulty with this model and similar ones that have been proposed is that the numerical solution of the corresponding Newton's equations of motion is computationally intensive. For this reason we first discuss several simple cellular automata models that retain some of the basic physics of the Burridge and Knopoff type models.

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:

1. Increase F everywhere by a small amount ΔF , for example, choose $\Delta F = 10^{-3}$, 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.
2. Check if $F(i, j)$ is greater than F_c , the critical 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.
3. The release of force due to the slippage of a block is represented by letting $F(i, j) = F(i, j) - F_c$. The transfer of force is represented by updating the force at the sites of the four neighbors at $(i, j \pm 1)$ and $(i \pm 1, j)$: $F \rightarrow F + 1$.

As an example, let the linear dimension of the lattice $L = 10$ so that the number of sites $N = L^2 = 100$. Do the simulation and show that the system eventually comes to a statistically stationary state, where the average value of the force 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 an instability. Increase L to $L = 30$ and repeat your simulations. Are your results for $N(s)$ consistent with the Gutenberg-Richter law?

Problem 14.9. Dissipative earthquake model

The simple earthquake model discussed in Problem 14.8 displays self-organized criticality due to the inherent conservation law of the dynamical variable, the stress. It is easy to modify the model so that the stress is not conserved. The Rundle-Jackson-Brown/Olami-Feder-Christensen (RJB/OFC) model of a earthquake fault is a simple example of such a nonconservative model.

- 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 . Then $\alpha(F(i, j) - F_R)$ is dissipated leaving the amount $(F(i, j) - F_R)(1 - \alpha)$ to be distributed equally to the four neighbors. If $\alpha = 0$, then the model would be equivalent to the Bak-Tang-Wiesenfeld model considered in Problem 14.8. Choose $\alpha = 0.2$ and determine if $N(s)$ exhibits power law scaling. For simplicity, take $F_c = 4$ and $F_R = 1$.
- 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 $\delta = 0.05$. (It would now be convenient to represent F_R by an array.) Also run the model in what is called the “zero-velocity limit” by finding the site that 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 qualitatively from what you found in part (a).
- c. The RJB/OFC model can be generalized and made more realistic by assuming that the interaction between the blocks is long range. That is, distribute the excess stress equally to all z neighbors that are within a circle 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 how the qualitative behavior of $N(s)$ changes as R becomes larger. In the literature lattices with $L \geq 256$ are typically considered with $R \simeq 30$.

The behavior of some other simple models is explored in the following four problems.

Problem 14.10. Forest fire model

- a. Consider the following simple 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. Those sites that have a tree are on fire with probability f . At each time step 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. Note that the changes in each site occur synchronously (simultaneously), and that 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. Use $L \geq 30$ 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. Choose a 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, it is possible that the behavior of a system 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 (14.10.c), but average your results over ten initial configurations. Is this forest fire model self-averaging?

Problem 14.11. Another forest fire model

- a. 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 time steps 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. Determine $N(s)$, the number of clusters of trees of size s that catch fire in each time step. Two trees are in the same cluster if they are nearest neighbors.

- b. Do the simulation and determine if the behavior of $N(s)$ is consistent with $N(s) \sim s^{-\alpha}$. If so, estimate the exponent α for several values of g and f .
- c.* 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 time step 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 $f\rho\langle s \rangle$, 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 $\langle s \rangle \rightarrow \infty$ in the limit $f/g \rightarrow 0$. 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 (b)?

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 (cf. 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 dynamical behavior expected of punctuated equilibrium. The model consists of a one-dimensional cellular automata of size L , where cell i represents the biological fitness of species i , normalized to unity. 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 show the fitness of each cell as a column of height f_i .
- b. Begin with $L = 64$ and describe what happens to the distribution of fitness values after a long time. 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 time steps)? Compute the distribution of fitness values, $N(f)$, averaged over all cells and over a long time. 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 an evolutionary point of view?
- c. Modify your program to compute the distance x between successive fitness changes and the distribution of these distances $C(x)$. Make a log-log plot of $C(x)$ 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 changed versus the position of the cell. Is the distribution of the plotted points approximately uniform? We might expect that the time of survival 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 which has been changed. Choose $c = 100$ for convenience 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 the meaning of 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 Neural Networks

Can computers think? This question has occupied philosophers, computer scientists, cognitive psychologists, and many others ever since the first computer was imagined. The assumption of workers in “strong” *artificial intelligence* is that it will be possible someday for computers to think. The reasoning is that thinking is based on symbolic manipulation of inputs from the external world. Of course, everyone agrees that we are far from having a computer think like a human. Some contend that computers will be able to only simulate the human brain and not be able to think

like it. Part of the argument is that the brain is not analogous to the hardware of a computer, and the mind is not analogous to its software.

Recent developments in two classes of models, known as neural networks and genetic algorithms, tend to weaken one argument against artificial intelligence. These models are based on the idea that the program can change itself based on the inputs, that is, the program statements and its data are the “mind” of the computer, and the program and its data can change depending on its own internal state and outside inputs. Indeed, we should consider the entire memory of the computer to constitute its mind. The result is that the state of the computer can change itself in ways that the programmer cannot anticipate. As we have learned, simple algorithms can lead to complex and unpredictable outcomes, and it probably comes as no surprise that the state of the computer can evolve through a sequence of states that can be very complex, that is, neither completely random nor completely ordered. Perhaps, the mind exists at the edge of chaos where the complex behavior just begins.

Neural networks model a piece of this ultimate computer mind. The idea is to store memories so that a computer can recall them when inputs are given that are close to a particular memory. As humans we have our own algorithms for doing so. For example, if we see someone more than once, the person’s face might provide input that helps us recall the person’s name. In the same spirit, a neural network can be given a pattern, for example, a string of 0s and 1s, that partially reflect a previously memorized pattern. The network then attempts to recall the memorized pattern. The significant difference between what the computer usually does to retrieve data from its memory and the memory recall of a neural network is that in the latter we consider *content addressable memory* in contrast to computer programs themselves which retrieve memory based on the address or location of the data, not on its content.

Neural network models have been motivated by how neurons in the brain might collectively store and recall memories. It is known that 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. Usually, a neuron is in one of two states, a resting potential (not firing) or firing at the maximum rate. The assumption is that when memories are stored in the brain, the strengths of the connections between neurons change. Neural network models attempt to maintain the key functions of biological neurons without the specific biological substrate.

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 potential at each neuron, V_i , which in the simple models we consider here takes on the values 0 or 1. The strength of the connection between the i th and j th neuron is denoted by T_{ij} and is determined by the M stored memories:

$$T_{ij} = \sum_{s=1}^M (2V_i^s - 1)(2V_j^s - 1). \quad (14.5)$$

The elements of the connection matrix are defined so that two neurons with the same value contribute a positive term to the sum and two neurons with different values contribute a negative term. V^s is the state of the s th stored memory. Given an initial state V_i^0 , the dynamics of the network is simple, that is, choose a neuron i at random and change its state according to its input.

Its input strength, S_i , is defined as

$$S_i = \sum_{i \neq j} T_{ij} V_j, \quad (14.6)$$

where V_j represents the current state of the j th neuron. We change the state of neuron i by setting

$$V_i = \begin{cases} 1, & \text{if } S_i > 0 \\ 0, & S_i \leq 0. \end{cases} \quad (14.7)$$

Note that the threshold value has been set equal to zero, but other values could be used as well.

The `Hopfield` and `HopfieldApp` classes, listed in the following, implement this model of a neural network and stores memories and recalls them based on user input. In `HopfieldApp` a button is used to add stored memories. The program uses the `BinaryLattice` class to store and recall memories. The user clicks on various cells to toggle their values back and forth between 0 and 1 and then clicks the `addMemory` button. At any given time the state, V_i , of the network is stored in the array `state[i]` and the connections, T_{ij} between neurons are stored in the array `connection[i][j]`. After the memories are stored, the user can click on the cells in the `plottingPanel`, and then press the `getInput` button to place the network into the input state. Then the user can step through the Hopfield algorithm to try to recall one of the stored memories.

Listing 14.6: Hopfield class: Model of a neural network.

```
/* Implements Hopfield model of a neural network */
package org.opensourcephysics.sip.ch14.hopfield;
import org.opensourcephysics.display.*;
import org.opensourcephysics.display2d.*;
import javax.swing.*;
import java.awt.*;
//import java.util.*;
import java.awt.event.*;

public class Hopfield implements InteractiveMouseHandler {
    PlottingPanel plottingPanel = new PlottingPanel("x", "y", " ");
    DrawingFrame frame = new DrawingFrame(plottingPanel);
    BinaryLattice memory; // cellular automata lattice
    int [][] connection;
    int [] state;
    int Lx, Ly, N;

    public void initialize () {
        N = Lx*Ly;
        connection = new int[N][N];
        state = new int[N];
        memory = new BinaryLattice(Lx, Ly);
        memory.setBlock(0, 0, new int[Lx][Ly]);
        plottingPanel.addDrawable(memory);
        plottingPanel.setPreferredMinMaxX(0, Lx);
        plottingPanel.setPreferredMinMaxY(0, Ly);
    }
}
```

```

        plottingPanel.setInteractiveMouseHandler(this);
        frame.show();
    }

    public void handleMouseAction(InteractivePanel panel, MouseEvent evt) {
        panel.handleMouseAction(panel, evt);
        switch (panel.getMouseAction()) {
            case InteractivePanel.MOUSE_PRESSED:
                int ix = (int)panel.getMouseX();
                int iy = (int)panel.getMouseY();
                if (ix < 0 || iy < 0 || ix >= Lx || iy >= Ly)
                    return; // outside the lattice
                if (memory.getValue(ix, iy) == 0)
                    memory.setValue(ix, iy, 1);
                else
                    memory.setValue(ix, iy, 0);
                plottingPanel.repaint();
                break;
        }
    }

    public void addMemory() {
        int n = 0;
        for(int j = 0; j < Ly; j++) {
            for(int i = 0; i < Lx; i++) {
                state[n] = memory.getValue(i,j);
                n++;
            }
        }
        for(int i = 0; i < N; i++) {
            for(int j = 0; j < N; j++) {
                if(i != j) {
                    connection[i][j] += (2*state[i] - 1)*(2*state[j] - 1);
                }
            }
        }
    }

    public void getInput() {
        int n = 0;
        for(int j = 0; j < Ly; j++) {
            for(int i = 0; i < Lx; i++) {
                state[n] = memory.getValue(i,j);
                n++;
            }
        }
    }

    public void step() {
        for(int k = 0; k < N; k++) {
            int i = (int)(Math.random()*N); // choose random neuron
            int sum = 0;
            for(int j = 0; j < N; j++) {
                if(i != j) {

```

```

        sum += connection[i][j]*state[j];
    }
    if(sum > 0) {
        state[i] = 1;
    }
    else {
        state[i] = 0;
    }
}
}
int n = 0;
for(int j = 0; j < Ly; j++) {
    for(int i = 0; i < Lx; i++) {
        memory.setValue (i,j,state[n]);
        n++;
    } }
    plottingPanel.repaint ();
}
}

```

Listing 14.7: HopfieldApp class.

```

/* Implements Hopfield model of a neural network */
package org.opensourcephysics.sip.ch14.hopfield;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.display2d.*;
import javax.swing.*;
import java.awt.geom.*;
import java.awt.*;
import java.util.*;
import java.awt.event.*;

public class HopfieldApp extends AbstractAnimation {
    Hopfield model;

    public HopfieldApp() {
        model = new Hopfield();
    }

    public void initializeAnimation() {
        model.Lx = control.getInt("Lattice width");
        model.Ly = control.getInt("Lattice height");
        model.initialize ();
    }

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

```

```

    public void resetAnimation(){
        control.setValue("Lattice width", 20);
        control.setValue("Lattice height", 1);
    }

    public void addMemory() {
        model.addMemory();
    }

    public void getInput() {
        model.getInput();
    }

    public static void main(String args[]) {
        HopfieldApp app = new HopfieldApp();
        AnimationControl control = new AnimationControl(app);
        control.addButton("addMemory", "addMemory" );
        control.addButton("getInput", "getInput");
        app.setControl(control);
    }
}

```

Problem 14.13. Memory recall in the Hopfield model

- a. Use the `Hopfield` and `HopfieldApp` classes to explore the ability of the Hopfield neural network to store and recall memories. Begin by storing two memories of 20 bits each lined up in a row. For example, store 11111000000000011111 and 11001100110011001100. Then try to recall a memory using the input string 111111000000111111. This input is similar to the first memory. Record the what is known as 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 the above procedure for a number of different values of the number of memories, N , and memory length.
- b. Estimate how many linear memories can be stored for a given sized string before recall becomes severely reduced. Make estimates for $N = 10, 20$, and 30 . What criteria did you adopt for correct recall?
- c. The `Hopfield` class can also store two-dimensional patterns. Describe how a two-dimensional pattern is stored in the one-dimensional array `state`. Consider a grid with $L \geq 10$ and store three patterns. The patterns could be simple geometric shapes or symbols. Then input a pattern similar to one of the stored memories and see how well the Hopfield algorithm is able to recall the correct pattern. Repeat for several different input patterns, and then increase the number of stored memories.

In addition to helping us understand biological memory recall, neural networks can be used to determine an optimal solution to a difficult optimization problem. In Problem 14.14 we consider the problem of finding the minimum energy of a model spin glass. A spin glass is a magnetic analog of an ordinary glass. In an ordinary glass the molecules are not organized in an ordered lattice as

in a crystal, but instead the molecular positions are disordered. 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. In this model the magnetic moment is represented by a spin s_i which can take on two values, ± 1 . The spins are located on lattice sites and the total energy of the system is given by

$$E = -J \sum_{\langle ij \rangle} s_i s_j, \quad (14.8)$$

where the notation $\sum_{\langle ij \rangle}$ means sum over all nearest neighbor pairs. For a ferromagnetic interaction $J > 0$ the spins lower their energy by lining up in the same direction. This model is known to have a phase transition such that below a certain critical temperature, the average value of the total spin or the magnetization will be nonzero. We will discuss this transition in detail in Chapter 15. In the present context we are interested in finding the lowest energy or ground state when the coupling constant J is not a constant, but takes on a random positive or negative value. This case is called a spin glass and the simplest version assigns a random value of $J = \pm 1$ to each link between two neighboring sites. To find the ground state we wish to find the configurations of spins that give the lowest value for the energy, E . This determination is very difficult because flipping individual spins might lower the energy locally, but might not lead to the lowest total energy of the system as a whole. Imagine listing all spin configurations on the horizontal axis of a graph such that any two neighboring configurations differ by only one spin value. If we plot the energy of the configurations on the vertical axis, we will see a plot with many hills and valleys, or local minima. What is needed is an algorithm that will find the global minimum, or the energy of the deepest valley. In Problem 14.14 we explore if the Hopfield model can help us find this global minimum.

Problem 14.14. Minimum energy of an Ising spin glass

- a. We can define an energy for the Hopfield model in analogy to the Ising model:

$$E = -\frac{1}{2} \sum_i \sum_{j \neq i} T_{ij} V_i V_j, \quad (14.9)$$

where we assume that $T_{ij} = T_{ji}$. If we give the T_{ij} random values, then the model is an example of a spin glass with long-range interactions (see Project 14.18). Modify your program so that the T_{ij} are given random values between -1 and 1 with $T_{ii} = 0$. Then the user provides an initial input of N bits. Have the program display the output string and the energy after every N attempts to change a neuron. Begin with $N = 20$.

- b. Describe what happens to the energy after a long time. For different initial states, but the same set of T_{ij} , is 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 states? Do you find periodic behavior or random behavior or do the states evolve to a state that does not change?

14.4 Growing Networks

Recently, there has been renewed interest in the general structure of networks and how they might apply to biological and social systems. A network is defined as a collection of points called nodes which are connected by lines called links. Mathematicians refer to networks as graphs, and graph theory has been an active field of mathematics for a long time. A mathematical network can represent an actual network by defining what each node represents, and what kind of relationship is 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 individual people, 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.

Why is there new interest in this topic? There are two reasons. First, some new models of network growth have been developed. Second, data on existing networks is more readily available now because of the exponential increase in computer use. Indeed, some of the networks being studied are networks of computers and their web sites.

Before we discuss some of the new models, we begin with an older model known as the Erdős-Rényi model. In this model a fixed set of N nodes are linked together by n random links with a maximum of one link between any two nodes. In the context of graph theory, the number of links of a node (a vertex) is called its degree. What are some of the important quantities of interest? One is the degree distribution of the network, $N(s)$, which is the number of nodes that have s links. (In the Erdős-Rényi model this distribution is a Poisson distribution for large N . Thus, there is a peak in this distribution, and for large s , $N(s)$ decreases exponentially. Some other quantities are analogous to the quantities measured in percolation theory. The main difference is that in network models there is no spatial location for the nodes, and only their connectivity is relevant. Thus, there is no spanning cluster. Instead there can be a “giant” cluster which is significantly larger than the other clusters. 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 $p = 1/N$. Some of the networks we will discuss are by definition connected so that there is only one cluster. In such cases there are a number of other important quantities. One is the mean distance between nodes, with the distance between two nodes defined as the shortest number of links from one node to the other. In many cases the mean distance weakly depends on the number of nodes and is surprisingly small. For this reason this characteristic of networks is known as the “small world” property.

Another property of networks is the clustering coefficient. That is, if node A is linked to B and B is linked to C, the clustering coefficient is the probability that A is linked to C. If this coefficient is large, then the network is highly connected. 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 groups. 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 like you might see for a business? An algorithm for computing the hierarchical or community structure of a network is described in the papers by Newman and Girvan listed in the references.

Two popular 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 made between one of the nodes and any other node in the system. The small world property shows up as a logarithmic dependence on system size N for large p . The shape of the degree distribution is similar to that of the Erdős-Rényi model.

The preferential attachment model is very simple. 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. Thus, the probability of a node with L links being connected to a new node is proportional to L . The result of this growth rule is that some nodes will accumulate many links. The key new result is that the degree distribution is a power law with $N(s) \sim s^{-\alpha}$. This *scale-free* behavior is very important because it says that there are nodes of all degrees. Examples of real networks that appear to have this behavior are actor networks where the links correspond to two actors appearing in the same movie, airport networks where the links correspond to a flight between two airports, and the internet where a link correspond to a web link on a web site.

The `Networks` 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 with the nodes not too close to each other. This drawing routine is useful only for networks less than about 100 nodes. There are sophisticated algorithms that you can find the web for drawing networks.

Listing 14.8: Networks class: Preferential attachment network model.

```
/* Preferential attachment model for networks */
package org.opensourcephysics.sip.ch14.networks;
import java.awt.*;
import org.opensourcephysics.display.*;

public class Networks implements Drawable {
    int [] node, linkFrom, edge;
    double [] x, y;
    int N; // maximum number of nodes
    int m = 2; // number of attempted links per node
    int linkNumber = 0;
    int n = 0; // current number of nodes
    int drawPositions = 1; // no-zero to draw nodes
    int nrun = 0;

    public void setEdgeArray() {
        edge = new int[N];
        nrun = 0;
    }

    public void addLink(int i, int j, int s) {
        linkFrom[i*m + s] = j;
        node[i]++;
        node[j]++;
        linkNumber += 2;
    }
}
```

```

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;
    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 == 1) setPosition(i);
    for (int s = 0; s < m; s++) {
        addLink(i,findNode(i,s),s);
    }
}

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

public void edgeDistribution(DatasetManager dm) {
    dm.clear();
    for(int i = 1; i < N;i++) {
        if(edge[i] > 0) dm.append(0,Math.log(i),Math.log(edge[i]*1.0/(N*nrn)));
    }
}

public void draw (DrawingPanel myWorld, Graphics g) {
    if(node == null || drawPositions == 0) {
        g.drawString(Integer.toString(nrun), 25, 25);
    }
    else {
        g.drawString(Integer.toString(nrun), 25, 25);
        int pxRadius = Math.abs(myWorld.xToPix(1.0) - myWorld.xToPix(0));
        int pyRadius = Math.abs(myWorld.yToPix(1.0) - myWorld.yToPix(0));
        g.setColor(Color.green);
        for(int i = 0; i < n; i++) {
            int xpix = myWorld.xToPix(x[i]); // - pxRadius;
            int ypix = myWorld.yToPix(y[i]); // - pyRadius;
            for(int s = 0; s < m; s++) {
                int j = linkFrom[i*m + s];
                int xpixj = myWorld.xToPix(x[j]) ; // - pxRadius;

```

```

        int ypixj = myWorld.yToPix(y[j]) ;// - pyRadius;
        g.drawLine(xpix,ypix,xpixj,ypixj);
    }
}
g.setColor(Color.red);
for(int i = 0; i < N; i++) {
    int xpix = myWorld.xToPix(x[i]) - pxRadius;
    int ypix = myWorld.yToPix(y[i]) - pyRadius;
    g.fillOval (xpix, ypix, 2*pxRadius, 2*pyRadius);
}
}
}
}

```

Problem 14.15. Preferential attachment model

- a. Write an application class that uses the **Networks** class and continuously creates new networks until stopped by the user. To speed up the computations, make it possible to optionally display the networks. The program should display the average degree distribution.
- b. Estimate the exponent α defined by $N(s) \sim s^{-\alpha}$ for $N = 100$ and $m = 2$. Repeat for $N = 500$. Does the exponent α change? If you have the computer resources, look at $N = 10000$.
- c. Determine if α depends on m .
- d. Modify the **Networks** class so that the m 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 degree distribution have now? Is this model the same as the Erdős-Rényi model?
- e.* Write a method to compute the clustering coefficient, $C(N)$ which is defined as three times the number of triangles (three nodes where A is connected to B, B is connected to C and C is connected to A) divided by the number of triples where one of the three links of a triangle is missing. Plot $\ln C(N)$ versus $\ln N$ for both the preference attachment model and the Erdős-Rényi model. Compare and discuss the results in terms of the visual appearance of the networks.

Problem 14.16. Small world network

- a. Write a class to produce a Watts-Strogatz network. Begin with N nodes which you can visualize as equally spaced on a circle. (Their actual position is irrelevant.) Then 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, etc. 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 a number of values of p . Discuss your results.

- c.* Compute the mean distance between nodes. If this distance is small and does not increase much with N , we say that the network has a small world character.

Problem 14.17. A possible model of a social network

In many social situations we notice groups of people 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 begin to model this situation. Usually, people don't become close friends with other people just because they have many close friends already (the preferential attachment mechanism). Instead they randomly choose someone to interact with and then with some probability a friendship will be established. perhaps the simplest model of this situation is the following growth rule. As each node is added to a system choose k existing nodes at random and with probability δ establish a link. This process will create a number of clusters of linked nodes, and we can imagine that there is a possibility for a phase transition between the existence of a giant cluster which contains a large fraction of the nodes, and a situation where all the clusters are small. This model was analyzed by Zalai et al.

- a. Write a class to model this random attachment model. Your class should be able to compute the degree distribution as well as the cluster distribution. Using at least $N = 1000$ nodes measure the degree distribution for $k = 1, 2$, and 5 and $\delta = 0.1$ and $\delta = 0.9$. Make sure you average over at least ten runs. You should not find a power law distribution. Explain why you would not expect to.
- b. Compute the degree distribution, $D(t)$, as a function of the age of the node. This function is the number of links connected to a node as a function of when the node was 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 $D(t)$.
- c. Consider the $k = 5$ case and run simulations for many values of δ . Determine the cluster distribution. In this case you should find that if δ is large enough a giant cluster appears. Define the existence of a giant cluster as when the largest cluster is at least three times larger the next largest cluster and contains at least 10% of the nodes. Estimate the value of δ where the giant cluster first appears. Roughly you should find a power law cluster distribution only at the transition. What do you find? What is the exponent for the power law if you can find it?
- d. How does the location of the transition change with k ? Explain your results.
- e. Consider the $k = 1$ case and run simulations for many values of δ . Determine the cluster distribution. You should find an approximate power law distribution for all values of δ . What are the exponents for the power law. Why do you think there would not be a phase transition for $k = 1$. Consider the possibility for two clusters merging for different values of k .

14.5 Genetic Algorithms

Many people find it difficult to accept that evolution is sufficiently powerful to generate the biological complexity seen 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.

One of the important examples is due to the biologist Tom Ray (see Lewin) who used a genetic algorithm in conjunction with low level machine coding. The memory of the computer was loaded with code segments that reproduce with small changes in their code. Because each code segment requires memory and the computer's processing time, the code segments compete with one another for memory and time. In what was a surprise to many, the memory of the computer, which initially contained a few simple code segments, evolved into a complicated "ecosystem" of code segments of many different sizes and structures.

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 0s and 1s. 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 least 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 second and much more powerful method is associated with sexual reproduction. We can 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 introduce the idea of genetic algorithms with a single population of strings, a simple phenotype, and a simple criteria for fitness.

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

$$E = - \sum_{i,j=\text{nn}(i)} T_{ij} s_i s_j, \quad (14.10)$$

where the sum is over all pairs of spins that are nearest neighbors. Note that the energy function in (14.10) assumes that only nearest neighbor spins interact, in contrast to the energy function

in (14.9) which assumes that every spin interacts with every other spin. The coupling constants T_{ij} can be either +1 (the ferromagnetic Ising model), -1 (the antiferromagnetic Ising model), randomly distributed (a spin glass), or have some other distribution. We adopt the energy as a measure of fitness. If we assume that $|T_{ij}| = 1$, 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? The genotype consists of a list or string of length N with 1s and 0s. Lattice site (i, j) corresponds to the n th 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 other applications and in biological systems the expression of the genotype into 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 parameters from the user, initializes the population of genotypes, and steps through the sequence of evolving the pool through the two modification processes, determining the fitness of each member of the population, and then selecting from the population those members to continue for the next generation. The **GenePool** class carries out the evolution. In method **recombine** two genotypes are picked 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 such 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 then determines the fitness of each member of the population by computing the energy of the spin glass lattice corresponding to each member of the population. Then members of this population are selected for the new generation using the method of generating discrete nonuniform probability distributions discussed in Section 11.6.

Listing 14.9: GeneticApp class.

```
/* Implements Genetic Algorithm for random bond Ising model */
package org.opensourcephysics.sip.ch14.genetic;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.display2d.*;
import javax.swing.*;
import java.awt.geom.*;
import java.awt.*;
import java.util.*;
import java.awt.event.*;

public class GeneticApp extends AbstractAnimation {
    GenePool gp = new GenePool();
    Phenotype pt = new Phenotype();
    DrawingPanel drawingPanel = new DrawingPanel();
    DrawingFrame drawingFrame = new DrawingFrame(drawingPanel);
```

```

public void initializeAnimation() {
    drawingPanel.addDrawable(gp);
    pt.L = control.getInt("Lattice size");
    gp.populationNumber = control.getInt("Population size");
    gp.recombinationRate = control.getInt("Recombination Rate");
    gp.mutationRate = control.getInt("Mutation Rate");
    gp.genotypeSize = pt.L*pt.L;
    gp.initialize (pt);
    pt.initialize ();
    drawingPanel.setPreferredMinMax(-1.0,gp.genotypeSize + 5,-1.0,gp.populationNumber + 2 );
    gp.initialize (pt);
    pt.initialize ();
}

public void doStep() {
    gp.evolve ();
    pt.determineFitness(gp);
    pt.select (gp);
    drawingPanel.render();
    control.clearMessages();
    control.println(gp.generation + " generations, best fitness = " + pt.bestFitness);
}

public void resetAnimation() {
    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[]) {
    GeneticApp app = new GeneticApp();
    AnimationControl control = new AnimationControl(app);
    app.setControl(control);
}
}

```

Listing 14.10: GenePool class.

```

/* Implements genetic algorithm */
package org.opensourcephysics.sip.ch14.genetic;
import org.opensourcephysics.display.*;
import org.opensourcephysics.display2d.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class GenePool implements Drawable {
    int populationNumber;

```

```

int numberOfGenotypes;
int recombinationRate;
int mutationRate;
int genotypeSize;
boolean [][] genotype;
int generation = 0;
Phenotype pt;

public void initialize (Phenotype _pt) {
    pt = _pt;
    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; //set 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++) {
        int i = (int)(Math.random()*populationNumber);
        int j = 0;
        do {
            j = (int)(Math.random()*populationNumber);
        } while(i == j);
        int size = 1 + (int) (0.5*genotypeSize*Math.random());
        int startPosition = (int) (genotypeSize*Math.random());
        int r1 = populationNumber + 2*r;
        int r2 = populationNumber + 2*r + 1;
        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;
    for(int m = 0; m < mutationRate; m++) {
        int n = (int)(Math.random()*populationNumber);
    }
}

```

```

        int position = (int) (genotypeSize*Math.random());
        copyGenotype(genotype[n],genotype[index + m]); // copy genotype
        genotype[index + m][position] = !genotype[n][position];
    }
}

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

public void draw (DrawingPanel myWorld, Graphics g) {
    if(genotype == null) return;
    if(pt.selectedPopulationFitness == null) return;
    int sizeX = Math.abs(myWorld.xToPix(0.8) - myWorld.xToPix(0));
    int sizeY = Math.abs(myWorld.yToPix(0.6) - myWorld.yToPix(0));
    for(int n = 0; n < populationNumber; n++) {
        int ypix = myWorld.yToPix(n) - sizeY;
        for(int position = 0; position < genotypeSize; position++) {
            if(genotype[n][position]) {
                g.setColor(Color.red);
            } else {
                g.setColor(Color.green);
            }
            int xpix = myWorld.xToPix(position) - sizeX;
            g.fillRect (xpix,ypix,sizeX,sizeY);
        }
        g.setColor(Color.black);
        g.drawString(String.valueOf(pt.selectedPopulationFitness[n]), myWorld.xToPix(genotypeSize + 1),ypix+sizeY);
    }
}
}

```

Listing 14.11: Phenotype class.

```

/* population of phenotypes (random bond Ising model) */
package org.opensourcephysics.sip.ch14.genetic;

public class Phenotype {
    int L;
    int [][] J; //random bonds in network
    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 gp) {
    totalFitness = 0;
    int state[][] = new int[L][L];
    populationFitness = new int[gp.numberOfGenotypes];
    for(int n = 0; n < gp.numberOfGenotypes; n++) {
        for(int i = 0; i < L; i++) {
            for(int j = 0; j < L; j++) {
                int position = i + j*L;
                if(gp.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++)
                populationFitness[n] -= state[i][j]*(J[i][j][0]*state[(i+1)%L][j] + J[i][j][1]*state[i][(j+1)%L]);
        // define fitness to be positive and low energy -> high fitness
        populationFitness[n] = highestEnergy - populationFitness[n];
        totalFitness += populationFitness[n];
    }
}

public void select(GenePool gp) {
    selectedPopulationFitness = new int[gp.numberOfGenotypes];
    boolean savedGenotype[][] = new boolean[gp.numberOfGenotypes][gp.genotypeSize];
    for(int n = 0; n < gp.numberOfGenotypes; n++) {
        gp.copyGenotype(gp.genotype[n], savedGenotype[n]);
    }
    for(int n = 0; n < gp.populationNumber; n++) {
        int fitnessFraction = (int)(Math.random()*totalFitness); //select using non-uniform probability distribution
        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];
    }
}

```

```

        gp.copyGenotype(savedGenotype[choice],gp.genotype[n]);
    }
}
}

```

Problem 14.18. Ground state of Ising-like models

- Use the genetic algorithm classes to find the ground state of the ferromagnetic Ising model for which $T_{ij} = 1$ and show that the ground state energy is $E = -2L^2$. 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 slightly so that each new generation is shown on the screen and a pause statement is added so that you can look at the new generations as they appear.
- Find the mean number of generations needed to find the ground state for $L = 4, 6$, and 8 . Repeat each run at least twice. 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.
- Repeat part (b) for the antiferromagnetic model for which $T_{ij} = -1$.
- Repeat part (b) for the spin glass model where $T_{ij} = \pm 1$ randomly. In this case we do not know the ground state energy in advance. What criteria 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. Indeed, we could have used some other prescription for converting our 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.

14.6 Lattice Gas Models of Fluid Flow

We now return to cellular automata models and discuss one of their most promising applications – simulations of fluid flow. Fluid flow is very difficult to simulate because the partial differential equation describing fluid flow, the Navier-Stokes equation, is very nonlinear. As we have found, nonlinear equations can lead to the breakdown 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 all these considerations, simulations of fluid flow based on direct numerical solutions of the Navier-Stokes equation typically require very sophisticated numerical methods (cf. Oran and Boris).

The cellular automata models of fluids are known as *lattice gas* models. These models are based on the idea that if we maintain the conservation laws and symmetries associated with fluids, then we can produce the correct physics at the macroscopic level if we average over many particles. 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 velocity vectors. The model needs to include two processes, the free motion between collisions and the collisions. In the simplest models, the particles move freely to their nearest neighbor lattice site in one time step. Then the velocities of the particles at each lattice site are updated according to a collision rule that conserves mass, momentum, and kinetic energy. The free motion and the collisions for all sites are computed simultaneously.

To understand the nature of lattice gas models, it is easier to discuss a specific two-dimensional model. Three-dimensional models are being studied, but they are much more difficult to visualize and to understand theoretically. We assume a triangular lattice, because its symmetry is more closely related to that of a continuum than a square lattice. In addition, collision rules for square lattices do not typically mix the horizontal and vertical motions of the particles. All particles are assumed to have the same speed and mass. The possible velocity vectors lie only along the links connecting sites, and hence there are only six possible velocities. To efficiently simulate this lattice gas model, we introduce bit manipulation.

The smallest element of computer memory contains a bit, which is a 0 or a 1. A *byte* is the size of memory needed to hold a single character, for example, a letter or a digit. More precisely, a byte is eight bits. Because there are $2^8 = 256$ possible arrangements of 1s and 0s in a byte, a byte can represent the ASCII character set, including all upper and lower case letters, numerals, punctuation, and other control characters such as a line feed. A computer *word* is usually two, four, or eight bytes, and is the unit of storage that can be accessed simultaneously and moved back and forth from the central processing unit (CPU) to various storage devices. If we could manipulate bits directly, then we could represent each site in a Boolean cellular automaton by a bit and update a whole word of sites (32 sites on a 32 bit machine) simultaneously. This type of update is a simple example of parallel processing on a single processor machine. Java has intrinsic bit manipulation operations. We now return to how bit manipulation can be used in our lattice gas model.

We will use each bit to label one of the six possible velocity vectors (labeled 0 to 5 because the first bit in a computer byte is in the 0 position):

$$\begin{array}{lll} \mathbf{v}_0 = (1, 0) & \mathbf{v}_1 = (1, -\sqrt{3})/2 & \mathbf{v}_2 = -(1, \sqrt{3})/2 \\ \mathbf{v}_3 = (-1, 0) & \mathbf{v}_4 = (-1, \sqrt{3})/2 & \mathbf{v}_5 = (1, \sqrt{3})/2. \end{array}$$

In some models a rest particle also is allowed. Each site can have at most one particle moving in a particular direction. The update process proceeds by first moving all the particles in the direction of their velocity to a neighboring site. Then at each lattice site the velocity vectors are changed according to a collision rule. Particle number and kinetic energy are easily conserved because all particles have the same speed, and we need only insure momentum conservation. There is some flexibility in the choice of rules. One set of collision rules is illustrated in Fig. 14.3. These

rules are deterministic with only one possible set of velocities after a collision for each possible set of velocities before a collision. Momentum conservation is enforced by these rules.

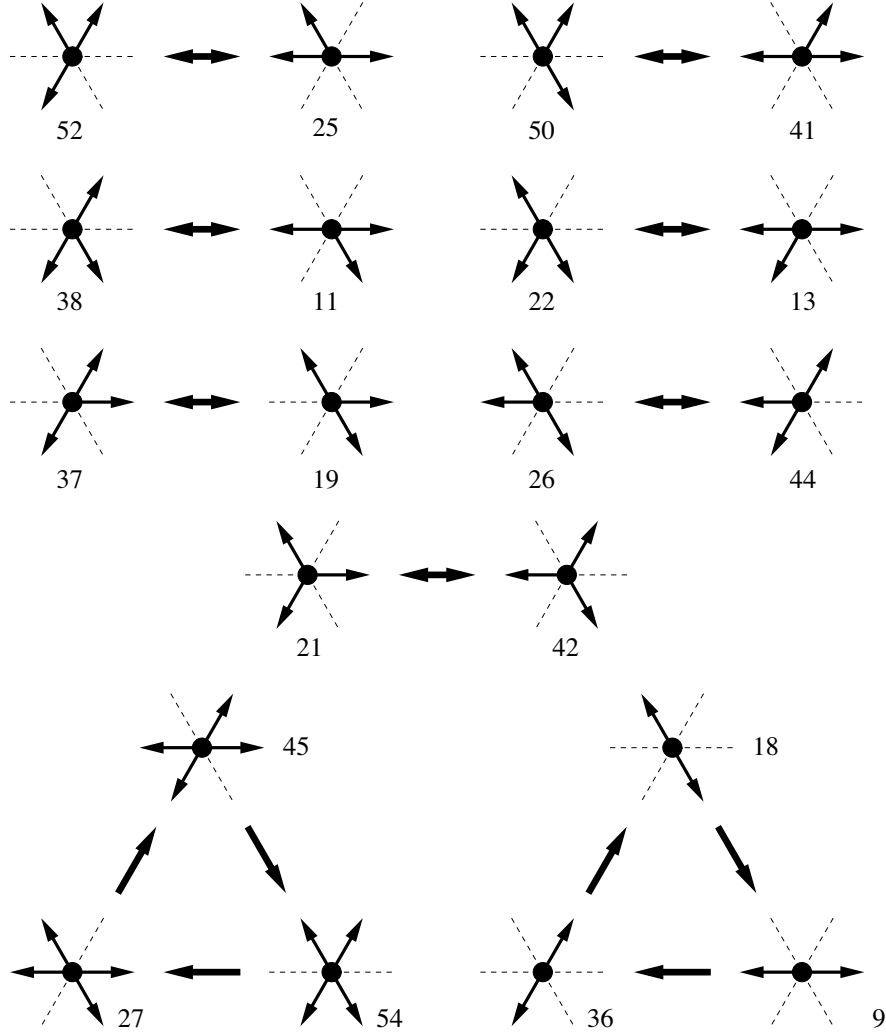


Figure 14.3: Examples of collision rules for a lattice gas on a triangular lattice. The rule for configurations that are not shown is that the velocities do not change after a collision. The numbers represent the way that the velocities at a lattice site are encoded.

We now describe the procedure for storing information about the particles in an array of integers, `lattice`. Each site of the lattice is represented by one element of `lattice`. The eight bits in a byte are labeled from 0 to 7. In principle, we could use a four byte integer to update four sites simultaneously, but to avoid complications we will not do so. We will use the first six

bits from 0 to 5 of an integer to represent particles moving in the six possible directions with bit 0 corresponding to a particle moving with velocity v_0 . For example, if bit 0 equals unity, we know there is a particle at this site with velocity v_0 . If there are three particles with velocities v_0 , v_2 , and v_4 at a site, then this situation is represented by 00010101 in binary notation; the corresponding decimal equivalent is 21. From Fig. 14.3 we see that after the collision there are three particles with velocities v_1 , v_3 , and v_5 corresponding to 42. We can express this collision as `rule[21] = 42`. Similar decimal equivalents can be expressed for all the other possible collisions.

We reserve bit 6 for a possible rest particle. If we want to occupy a site with a fixed particle to represent a barrier, we use bit 7, that is, we set the value of barrier sites equal to $2^7 = 128$. What boundary condition should we use when a particle is adjacent to a barrier site and heading toward it? The simplest rule that insures that we do not lose any particles is to set the velocity $\mathbf{v}$ of such a particle equal to $-\mathbf{v}$. Other possibilities are to set the angle of incidence equal to the angle of reflection or to set the velocity to an arbitrary value. The latter case corresponds to a collision off a rough surface and is more difficult to implement because we need to insure that no site has more than one particle with the same velocity.

Class `LatticeGas` performs all of the operations that we have described. The list of rules for collisions is given in the `setRuleTable` method. The six nearest neighbors are found in the methods `mnx` and `mny`, which returns the horizontal and vertical array index, respectively, for the neighbors of a site. The `step` method runs through the entire lattice and either moves or reflects a particle. The updated values of the lattice are placed in the array `newLattice`. Notice the use of the array `antiparallel[dir]`. The reason for this array is that if a neighboring site in the direction `dir` from a site sends a particle to the site, that particle must be moving in the `-dir` direction. After the entire lattice is updated, the collisions are implemented for each site, and `newLattice` is reset to 0 to be ready for the next call to method `step`.

Listing 14.12: Listing of `LatticeGas` class.

```
package org.opensourcephysics.sip.ch14.latticegas;
import org.opensourcephysics.display.*;
import org.opensourcephysics.numerics.*;
import java.awt.*;

public class LatticeGas implements Drawable {
    public int [][] lattice, newLattice;
    public int lx,ly;
    public int [] rule;
    private int [] mask = {1,2,4,8,16,32,64,128};
    private int [] antiParallel = {3,4,5,0,1,2}; // 0 goes to 3, 1 to 4, etc.
    private double [] vx,vy;
    public int s;

    public LatticeGas() {
        setRuleTable();
        setVelocities ();
    }

    public void initialize () {
        lattice = new int[lx][ly];
```

```

newLattice = new int[lx][ly];
// fill block of sites in center of lattice with obstacles
for(int i = lx/2 - 2; i < lx/2 + 6; i++)
    for(int j = ly/2 - 2; j < ly/2 + 6; j++)
        lattice[i][j] = 128;
for(int i = 0; i < 20; i++)
    for(int j = 0; j < ly; j++)
        lattice[i][j] = 1;
}

public void setRuleTable() {
    rule = new int [1024];
    for(int i = 0; i < 1024; i++) rule[i] = i;
    rule[21] = 42;
    rule[42] = 21;
    rule[9] = 36;
    rule[36] = 18;
    rule[27] = 45;
    rule[45] = 54;
    rule[19] = 37;
    rule[37] = 19;
    rule[50] = 41;
    rule[41] = 50;
    rule[22] = 13;
    rule[13] = 22;
    rule[26] = 44;
    rule[44] = 26;
    rule[11] = 38;
    rule[38] = 11;
    rule[25] = 52;
    rule[52] = 25;
}

public void setVelocities() {
    // gives x and y component of velocity at each site for drawing
    vx = new double[1024];
    vy = new double[1024];
    double uym = Math.sqrt(0.75); // possible y component magnitude
    double ux[] = {-1.0,-0.5,0.5,1.0,0.5,-0.5};
    double uy[] = {0.0,uym,uym,0.0,-uym,-uym};
    for(int i = 1; i < 64; i++) {
        for(int dir = 0; dir < 6; dir++)
            if((i & mask[dir]) != 0) {
                vx[i] += ux[dir];
                vy[i] += uy[dir];
            }
    }
}

public int nnx(int i, int j, int dir) {

```

```

switch(dir) {
case 0:
    if(i > 0) return i-1; else return lx-1;
case 1:
    if(j % 2 == 0) return i;
    else if(i > 0) return i - 1;
    else return lx-1;
case 2:
    if(j % 2 != 0) return i;
    else if(i < lx-1) return i + 1;
    else return 0;
case 3:
    if(i < lx-1) return i+1; else return 0;
case 4:
    if(j % 2 != 0) return i;
    else if(i < lx-1) return i + 1;
    else return 0;
case 5:
    if(j % 2 == 0) return i;
    else if(i > 0) return i - 1;
    else return lx-1;
}
return i;
}

public int nny(int i, int j, int dir) {
switch(dir) {
case 0: case 3:
    return j;
case 1: case 2:
    if(j < ly-1) return j+1; else return 0;
case 4: case 5:
    if(j > 0) return j-1; else return ly-1;
}
return j;
}

public void step() {
// move or reflect particles
for(int i = 0; i < lx; i++)
for(int j = 0; j < ly; j++)
for(int dir = 0; dir < 6; dir++) {
    int iNN = nnx(i,j,dir);
    int jNN = nny(i,j,dir);
    if((lattice[iNN][jNN] & mask[antiParallel[dir]]) != 0)
// there is a neighboring particle moving toward site
    if((lattice[i][j] & mask[7]) != 0) // barrier site, thus reflect
        newLattice[iNN][jNN] |= mask[dir];
    else // particle moves from nearest neighbor
        newLattice[i][j] |= mask[antiParallel[dir]];
}
}

```

```

    }
    // on site collisions
    for(int i = 0; i < lx; i++)
        for(int j = 0; j < ly; j++)
            if((lattice[i][j] & mask[7]) == 0) { // not a barrier site
                lattice[i][j] = rule[newLattice[i][j]];
                newLattice[i][j] = 0;
            }
        // add new particles coming from the left
        for(int j = 0; j < ly; j++)
            lattice[0][j] = 1;

}

// draws average v field
public void draw (DrawingPanel myWorld, Graphics g) {
    if (lattice == null) {
        return;
    }
    int s2 = s/2;
    for(int x = s2; x < lx-s2; x += s+1)
        for(int y = s2; y < ly-s2; y += s+1){
            double wx = 0;
            double wy = 0;
            int count = 0;
            for (int i = (x-s2); i < (x+s2+1); i++)
                for (int j = (y-s2); j < (y+s2+1); j++)
                    if( lattice[i][j] != 128) {
                        wx += vx[lattice[i][j]];
                        wy += vy[lattice[i][j]];
                        count++;
                    }
            if(count > 0) {
                wx /= count;
                wy /= count;
                Arrow v = new Arrow(x-0.5*wx,y-0.5*wy,wx,wy);
                v.draw(myWorld,g);
            }
        }
    double ay = Math.sqrt(0.75);
    int pxRadius = Math.abs(myWorld.xToPix(0.5) - myWorld.xToPix(0));
    int pyRadius = Math.abs(myWorld.yToPix(0.5) - myWorld.yToPix(0));
    for (int i = 0; i < lx; i++)
        for (int j = 0; j < ly; j++) {
            double y = (j+0.5)*ay;
            double x;
            if((j % 2) == 0)
                x = 0.25 + i;
            else
                x = 0.75 + i;

```

```

        if(( lattice [i][j] & mask[7]) != 0){
            int xpix = myWorld.xToPix(x) - pxRadius;
            int ypix = myWorld.yToPix(y) - pyRadius;
            g. fillOval (xpix, ypix, 2*pxRadius, 2*pyRadius);
        }
    }
}
}

```

An important use of lattice gas models is to simulate flow in and around various geometries. The fluid velocity field can be seen to develop vortices, wakes, and other fluid structures near obstacles. Typically, particles are injected at one end and absorbed as they reach the other end. Method `initial` places an obstacle in the middle of the lattice and an initial flow. Periodic boundary conditions are used in the other direction. Large lattices are required to obtain quantitative results, because it is necessary to average the velocity over many sites. The density of the fluid is determined by the average number of particles per site, and the pressure can be varied by changing the flux of particles that are injected at one end. We discuss some typical applications in Problems 14.19–14.21.

Problem 14.19. Approach to equilibrium

- a. To maintain zero total velocity or momentum at all times, use an initial configuration for which all sites contain either no particles or six particles whose net momentum is zero. The number density $\rho = 6N/(L_x L_y)$, where N is the number of sites with six particles and the total number of sites is $L_x \times L_y$. Use periodic boundary conditions in both directions. The output of the program should be a pictorial representation of the average velocity at each site, for example, an arrow pointing in the direction of the average velocity with a size proportional to the magnitude of the average velocity. Begin with a dilute gas with $N = 10$ on a 10×10 triangular lattice. Plot the velocity vector field after each iteration of the lattice gas to check that your program is working correctly.
- b. Consider the approach to equilibrium. Use a 30×30 lattice and place six particles at every site in a 4×4 region. Describe qualitatively what happens to the particles as a function of time. Approximately how many iterations does it take for the particles to fill the box? Do the particles appear to be at random positions with random velocities? Describe your visual algorithm for determining when equilibrium has been reached.
- c. Repeat part (b) for an initial $b \times b$ region of particles with $b = 2, 6, 8$, 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 ?
- d. Repeat part (b) for an initial 4×4 region of particles, but vary the size of the box. Try $L_x = L_y = 10, 20$, and 40 . Estimate the equilibration time in each case. How does the equilibration time depend on ρ ?

Problem 14.20. Flow past a barrier

- a. Modify your program from Problem 14.19 so that at each iteration three particles are injected with velocities v_0 , v_1 , and v_5 at each site on the left-hand side. The particles are removed on the right-hand side. Include barrier sites in the middle of the cell representing a $b_x \times b_y$ rectangular barrier. Use the barrier boundary condition that the directions of the reflected particles are reversed, $\mathbf{v} \rightarrow -\mathbf{v}$. Use periodic boundary conditions in the vertical direction. Besides the left-hand column and the barrier sites, all other sites are initially empty. Represent the velocity field visually as described in Problem 14.19a.
- b. Choose $L_x = 50$ and $L_y = 20$ with a barrier of dimensions $b_x = 5$ and $b_y = 1$. Describe the flow once a steady state velocity field begins to appear. Can you see a wake appearing behind the obstacle? Are there vortices (circular fluid flow)?
- c. Repeat part (b) with different size obstacles. Are there any systematic trends?
- d. Reduce the pressure by injecting particles at the left every other time step. Are there any noticeable changes in behavior from parts (b) and (c)? Reduce the pressure still further and describe any changes in the fluid flow.
- f.* Increase the size of the lattice by a factor of 10 in each direction and average the velocity in each 5×5 region. Compare the flow patterns that you obtain with those obtained in parts (b)–(d).

Problem 14.21. Fluid flow in porous media

- a. Modify your lattice gas program so that instead of a rectangular barrier, the barrier sites are placed at random in the lattice. Add a method that sums the horizontal velocity of those particles that reach the right edge of the lattice. The current density is the average of this sum per unit height of the lattice. Compute the current density as a function of porosity, the fraction of sites not containing barriers. If time permits, average over at least ten pore configurations for each value of the porosity. Use $L_x = 50$ and $L_y = 20$.
- b.* Vary the size of the lattice and use the finite size scaling procedure discussed in Section 12.5 to estimate the critical exponent μ defined by the dependence of the current density J on the porosity ϕ . That is, $J \sim (\phi - \phi_c)^\mu$. 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 principle 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, which might be faster than floating point arithmetic. Their major limitation is that many sites must be averaged over 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).

14.7 Overview and Projects

All of the models we have discussed in this chapter have been presented in the form of a computer algorithm rather than in terms of a differential equation. These models are an example of the development of a “computer culture” and are a reflection of the way that technology affects the way we think (cf. Vichniac). Can you discuss the models in this chapter without thinking about their computer implementation? 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 newly developing study of complexity, and there are many models and applications that we have not discussed. These models range from biological evolution, fluid flow, the immune system, economic cycles, and pedestrian movement, just to name a few. In addition, one of the major motivations for the study of cellular automata is their relation to theories of computation and the development of new computer architectures (cf. Hillis). Some researchers believe that ultimately all models of nature can be reduced to cellular automata. One of the attractive features of these models is that the complexity of nature can be ultimately understood as the result of simple and local rules of evolution (cf. Wolfram 2002).

Project 14.22. Modeling of opinion formation

There are a number of models of opinion formation. We will discuss a few that are popular and easy to program and that give well defined results. The basic idea of these models is that the opinions of others will influence the opinion of individuals (peer pressure).

- a. In a model by Deffuant et al, N individuals each have an opinion which takes on a value between 0 and 1. Then choose two individuals, i and j at random. Assume the i th opinion, O_i is greater than the j th opinion, O_j . If their opinions differ by less than a parameter ϵ , then we 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 simulates the idea that an individual’s opinion will only be influenced by another individual’s opinion if they are close enough. Write a class to simulate this model. Use an $N = L \times L$ `CellLattice`, where each cell can take on one of 256 values to visualize the model. The 256 values should be a large enough number to approximate a continuum of values. Consider $\epsilon = 10, 50$ and 100 , and $m = 0.3$ and $m = 0.6$. You should include in your program the option to plot configurations only after a certain number of steps to speed up the simulation. Use at least $L = 50$, begin with a random set of opinions, and discuss whether a single opinion emerges and the level of fluctuations that you find.
- b. Sznajd suggested the following model. N individuals are placed on a square lattice with periodic boundary conditions, and each individual has one of two opinions. At each step, an individual and one of its neighbors is each 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 of this model is that individuals are more likely to change their opinion to those physically near them if more than one person shares the same opinion. Write a class to simulate this model and show that consensus is always reached for all sites if the simulation is run long enough. Discuss the visual appearance of the clusters of like-minded individuals. Consider initial configurations where the individuals are randomly assigned the two opinions, and also initial configurations where one opinion has a majority of 1%, 5%, and 10%.

- c. Consider the following variations of the Sznajd model. What happens if there are $q > 2$ opinions? Is consensus still always reached? Modify the model so that the individuals do not sit on the sites of a square lattice, but rather they are the nodes of a preferential attachment network of at least 5000 nodes.

Project 14.23. The minority game

In certain situations we wish to be in the minority. Thus, we might want to go to a popular restaurant on an off night so that we don't need to wait in line. A business will want to sell goods and services that are not being sold by other businesses. The following algorithm, known as *The Minority Game*, is a simple model of adaptive competition where the players of the game try to maximize their own benefit. We will find that there is a phase transition between when the players mainly act on their own, and when cooperative behavior emerges.

The model is as follows. There are N players, where N is an odd number. At each time step, each player can choose one of two actions which we encode as 0 or 1. The player's choice is determined by a strategy based on the previous m steps. Each strategy is a table of all possible outcomes of the previous m steps and a decision on what to do for each outcome. An outcome is simply which action 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 steps, action 1 was chosen by a minority of the players. A strategy would be a table like the following: (1,1,1), (1,0,0), (0,1,0), and (0,0,1), which says that if (1,1) occurred in the past, the player chooses action 1, if (1,0) occurred the player chooses 0, etc. Each player is given at least two strategies that are assigned at random at the beginning of the game. As the game is played the performance of each strategy is computed. Thus, if a strategy led to action 1 and action 1 was the minority, then the performance of the strategy is incremented by unity, otherwise it stays the same. At each step each player looks at the past, looks at their strategies, chooses the one with the best performance, and then the outcome for all players is determined (which action was in the minority), and finally the performance for each player's strategy and the past is updated.

To simplify the code we represent the past outcomes by an integer where each bit represents one outcome. Thus, the integer 6 which has a binary representation of 110 means that the outcome in the last step was 0 and for each of the two steps before that it was 1. You will need the following arrays: `strategies[i][j][k]` which gives the action for the i th player, using its j th strategy, when the k th past occurred; `performance[i][j]` which gives the performance for the i th player's j th strategy; and `chosenStrategy[i]` which lists which strategy was chosen by the i th player in the current step.

Let N_1 equal the number of players who chose action 1 in a step. The mean of N_1 over many steps will be close to half the players. The players will cooperate best if at each step the value of N_1 is close to half of the players because in that way there are as many players as possible in the minority. Then the important quantity to measure is the standard deviation, σ of N_1 . If σ is large, a large number of players were not in the minority, and we say the game is inefficient. If σ is small, we say it is efficient. Naively, one might think that the efficiency increases as the number of past outcomes taken into account increases, because then the players have more information to choose their strategy. However, this is not the case!

- a. Write a class to simulate the minority game and an application class to run it. For simplicity give each player just two strategies chosen at random. You will wish to run your program for

- m varying from 2 to about 12. A reasonable number for N is 101, but for testing purposes you can choose $N = 11$. Each game should be run for at least 1000 steps and you should average your results over at least 10 independent runs at the same m but different strategies for the players. Plot σ versus m initially, and describe the behavior for different values of N . Is there a minimum in these plots? Explain why this minimum occurs.
- b. The results of the minority game scale in a very clear way. Plot σ^2/N versus $2^m/N$ for different values of N . You should find that all your data fall on the same curve. What does 2^m represent? Discuss this scaling behavior. Describe the behavior of the efficiency on either side of the minimum. 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 as the standard deviation of each game's σ over many games. Discuss the significance of your results.

References and Suggestions for Further Reading

- Réka Albert and Albert-László Barabási, "Statistical mechanics of complex networks," *Rev. Mod. Phys.* **74**, 47 (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 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 (1988).
- Per Bak and Michael Creutz, "Fractals and self-organized criticality," in *Fractals in Science*, Armin Bunde and Shlomo Havlin, editors, Springer-Verlag (1994).
- Dale P. Bentz, Peter V. Coveny, Edward J. Garboczi, Michael F. Kleyn, and Paul E. Stutzman, "Cellular automaton simulations of cement hydration and microstructural development," *Modelling Simul. Mater. Sci. Eng.* **2**, 783 (1994).
- E. R. Berlekamp, J. H. Conway, and R. K. Guy, *Winning Ways for your Mathematical Plays*, Vol. 2, Academic Press (1982). A discussion is given of how the Game of Life simulates a universal computer.
- J. M. Carlson and J. S. Langer, "Mechanical model of an earthquake fault," *Phys. Rev. A* **40**, 6470 (1989).

- D. Challet and Y.-C. Zhang, “Emergence of cooperation and organization in an evolutionary game,” *Physica A* **246**, 407 (1997). The first description of the minority game.
- John W. Clark, Johann Rafelski, and Jeffrey V. Winston, “Brain without mind: Computer simulation of neural networks with modifiable neuronal interactions,” *Phys. Repts.* **123**, 215 (1985).
- Michael Creutz, “Deterministic Ising dynamics,” *Ann. Phys.* **167**, 62 (1986). A deterministic cellular automaton rule for the Ising model is introduced.
- Guillaume Deffuant, Frédéric 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>](mailto: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).
- 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 (1992).
- B. Hayes, “Computer recreations,” *Sci. Amer.* **250** (3), 12 (1984). An introduction to cellular automata.
- 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 (1982).
- W. Daniel Hillis, *The Connection Machine*, MIT Press (1985). A discussion of a new massively parallel computer architecture influenced in part by physical models of the type discussed in this chapter.
- 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). The authors discuss their experiments on real sandpiles.

- Stuart A. Kauffman, *At Home in the Universe: The Search for the Laws of Self-Organization and Complexity*, Oxford University Press (1995); *The Origins of Order: Self-Organization and Selection in Evolution*, Oxford University Press (1993); “Cambrian explosion and Permian quiescence: Implications of rugged fitness landscapes,” *Evol. Ecol.* **3**, 274 (1989). These references discuss the NK model of genetics and evolution which have some interesting properties.
- 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).
- 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 (1986). See also Christopher G. Langton, editor, *Artificial Life*, Addison-Wesley (1989); Christopher G. Langton, Charles Taylor, J. Doynne 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 on complexity theory.
- 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.
- Sergei Maslov, Maya Paczuski, and Per Bak, “Avalanches and $1/f$ noise in evolution and growth models,” *Phys. Rev. Lett.* **73**, 2162 (1994).
- K. Nagel and M. Schreckenberg, “A cellular automaton model for freeway traffic,” *J. Phys. I* **2**, 2221 (1992);
- 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 (1998).
- M. E. J. Newman, “The structure and function of complex networks,” *SIAM Rev.* **45**, 167 (2003).
- M. E. J. Newman, “Detecting community structure in networks,” *Eur. Phys. J. B* **38**, 321(2004); M. E. J. Newman and M. Girvan, “Finding and evaluating community structure in networks,” *Phys. Rev. E* **69**, 026113 (2004). These papers describe an algorithm for detecting the hierarchical structure of networks.
- Z. Olami, H. J. S. Feder, and K. Christensen, “Self-organized criticality in a continuous, nonconservative cellular automaton modeling earthquakes,” *Phys. Rev. Lett.* **68**, 1244–1247 (1992).
- 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.

- Daniel H. Rothman and Stéphane Zaleski, *Lattice Gas Cellular Automata*, Cambridge University Press (1997).
- Daniel H. Rothman and Stéphane Zaleski, “Lattice-gas models of phase separation: interfaces, phase transitions, and multiphase flow,” *Rev. Mod. Phys.* **66**, 1417 (1994). A comprehensive review paper.
- 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.
- L. Schulman and P. Seiden, “Statistical mechanics of a dynamical system based on Conway’s Game of Life,” *J. Stat. Phys.* **19**, 293 (1978).
- Dietrich Stauffer, “Cellular automata,” Chapter 9 in *Fractals and Disordered Systems*, Armin Bunde and Shlomo Havlin, editors, Springer-Verlag (1991). Also see “Programming cellular automata,” *Computers in Physics* **5** (1), 62 (1991).
- Dietrich Stauffer, “Monte Carlo simulations of Sznajd models,” *J. Artificial Societies and Social Simulation* **5**(1) paper #4 (2002). This and other relevant papers can be found at jasss.soc.surrey.ac.uk.
- 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 (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 (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. (see above).
- 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.
- Gérard Y. Vichniac, “Cellular automata models of disorder and organization,” in *Disordered Systems and Biological Organization*, E. Bienenstock, F. Fogelman Soulie, and G. Weisbuch, eds. Springer-Verlag (1986). See also Gérard Y. Vichniac, “Taking the computer seriously in teaching science (an introduction to cellular automata),” in *Microscience*, Proceedings of

the UNESCO Workshop on Microcomputers in Science Education, G. Marx and P. Szucs, editors, Balaton, Hungary (1985).

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 (1983), and S. Wolfram, “Universality and complexity in cellular automata,” *Physica B* **10**, 1 (1984).

Stephen Wolfram, *A New Kind of Science*, Wolfram Media (2002).

László Zalaiyi, Gábor Csárdi, Tamás Kiss, Máté Lengyel, Rebecca Warner, Jan Tobochnik, and Péter Erdi, “Properties of a random attachment growing network,” *Phys. Rev. E.* **68**, 066104 (2003), cond-mat/0305299.