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

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

14.6 Lattice Gas Models of Fluid Flow

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

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

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

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

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

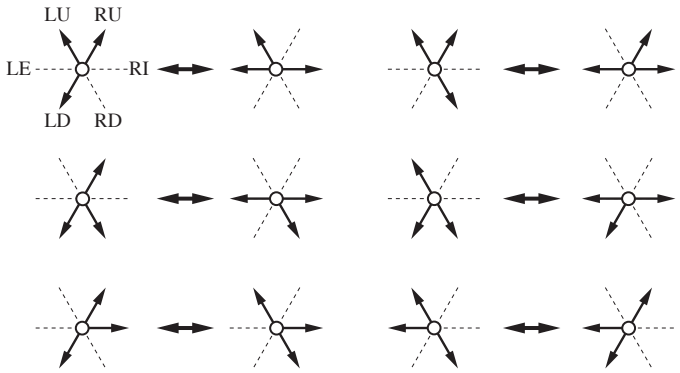

Figure 14.4: Examples of collision rules for three particles, with one particle unchanged and no stationary particles. Each direction or channel is represented by 32 bits, but we need only the first 8 bits. The various channels are summarized in Table 14.1.

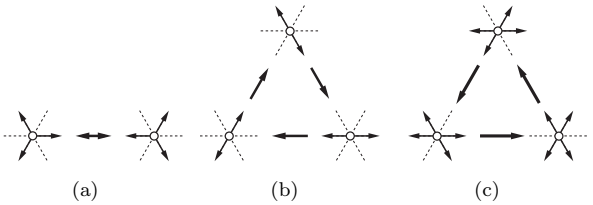

Figure 14.5: (a) Example of collision rule for three particles with zero net momentum. (b) Example of two particle collision rule. (c) Example of four particle collision rule. The rules for states that are not shown is that the velocities do not change after a collision. An open circle represents a lattice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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

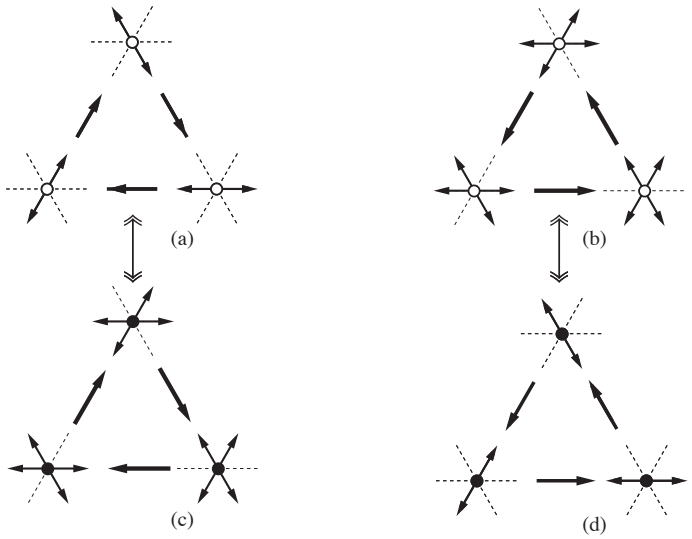

Figure 14.6: (a) and (c) and (b) and (d) are duals of each other. An open circle represents the absence of a stationary particle, and a filled circle represents the presence of a stationary particle. Note that the collision rule in (c) is similar to (b), and the collision rule in (d) is similar to (a),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

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

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

Listing 14.13: Listing of the `LatticeGas` class.

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

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

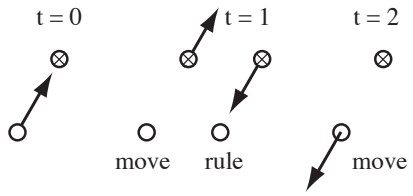

Figure 14.7: Example of a collision from a barrier. The symbol ⊗ denotes a barrier site.

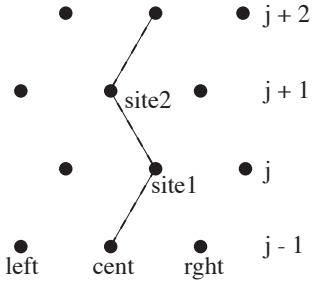

Figure 14.8: We update `site1` and `site2` at the same time. The rows are indexed by `j`. The dotted line connects sites in the same column.

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

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

static { // set rule table
```

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

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

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

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

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

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

Listing 14.14: Listing of the `LatticeGasApp` class.

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

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

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

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

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

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

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

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

Problem 14.20. Flow past a barrier

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

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

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

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

Problem 14.21. Approach to equilibrium

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

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

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

Problem 14.22. Fluid flow in porous media

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

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

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