Chapter 8

The Dynamics of Many Particle Systems

We simulate the dynamical behavior of many particle systems such as dense gases, liquids, and solids and observe their qualitative features. Some of the basic ideas of equilibrium statistical mechanics and kinetic theory are introduced.

8.1 Introduction

Given our knowledge of the laws of physics at the microscopic level, how can we understand the behavior of gases, liquids, and solids and more complex systems such as polymers and proteins? For example, consider two cups of water prepared under similar conditions. Each cup contains approximately 10^{25} molecules which mutually interact and, to a good approximation, move according to the laws of classical physics. Although the intermolecular forces produce a complicated trajectory for each molecule, the observable properties of the water in each cup are indistinguishable and are easy to describe. For example, the temperature of the water in each cup is independent of time even though the positions and velocities of the individual molecules are changing continually.

One way to understand the behavior of a classical many particle system is to simulate the trajectory of each particle. This approach, known as *molecular dynamics*, has been applied to systems of up to 10^9 particles and has given us much insight into a variety of systems in which the particles obey the laws of classical dynamics.

A calculation of the trajectories of many particles would not be very useful unless we know the right questions to ask. Saving these trajectories would quickly fill up any storage medium, and we do not usually care about the trajectory of any particular particle. What are the useful quantities needed to describe these many particle systems? What are the essential characteristics and regularities they exhibit? Questions such as these are addressed by statistical mechanics and some of the ideas of statistical mechanics are discussed in this chapter. However, the only background needed for this chapter is a knowledge of Newton's laws of motion.

8.2 The Intermolecular Potential

The first step is to specify the model system we wish to simulate. We assume that the dynamics can be treated classically, the molecules are spherical and chemically inert and their internal structure can be ignored, and the interaction between any pair of particles depends only on the distance between them. In this case the total potential energy U is a sum of two-particle interactions:

$$U = u(r_{12}) + u(r_{13}) + \cdots + u(r_{23}) + \cdots = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} u(r_{ij}), \tag{8.1}$$

where $u(r_{ij})$ depends only on the magnitude of the distance $\mathbf{r}_{ij}$ between particles i and j. The pairwise interaction form (8.1) is appropriate for simple liquids such as liquid argon.

The form of $u(r)$ for electrically neutral molecules can be constructed by a first principles quantum mechanical calculation. Such a calculation is very difficult, and it usually is sufficient to choose a simple phenomenological form for $u(r)$. The most important features of $u(r)$ are a strong repulsion for small r and a weak attraction at large r. The repulsion for small r is a consequence of the Pauli exclusion principle. That is, the electron wave functions of two molecules must distort to avoid overlap, causing some of the electrons to be in different quantum states. The net effect is an increase in kinetic energy and an effective repulsive interaction between the electrons, known as *core repulsion*. The dominant weak attraction at larger r is due to the mutual polarization of each molecule; the resultant attractive potential is called the *van der Waals* potential.

One of the most common phenomenological forms of $u(r)$ is the Lennard-Jones potential:

$$u(r) = 4\epsilon\Big[\Big(\frac{\sigma}{r}\Big)^{12} - \Big(\frac{\sigma}{r}\Big)^{6}\Big]. \tag{8.2}$$

A plot of the Lennard-Jones potential is shown in Figure 8.1. The r^{-12} form of the repulsive part of the interaction was chosen for convenience only and has no fundamental significance. The attractive $1/r^6$ behavior at large r corresponds to the van der Waals interaction.

The Lennard-Jones potential is parameterized by a length σ and an energy ϵ. Note that $u(r) = 0$ at $r = \sigma$, and that $u(r)$ is close to zero for $r > 2.5\sigma$. The parameter ϵ is the depth of the potential at the minimum of $u(r)$; the minimum occurs at a separation $r = 2^{1/6}\,\sigma$.

Problem 8.1. Qualitative properties of the Lennard-Jones interaction

Write a short program to plot the Lennard-Jones potential (8.1) and the magnitude of the corresponding force:

$$\mathbf{f}(r) = -\nabla u(r) = \frac{24\epsilon}{r}\Big[2\Big(\frac{\sigma}{r}\Big)^{12} - \Big(\frac{\sigma}{r}\Big)^{6}\Big]\hat{\mathbf{r}}. \tag{8.3}$$

At what value of r is the force equal to zero? For what values of r is the force repulsive? What is the value of $u(r)$ for $r = 0.8\sigma$? How much does u increase if r is decreased to $r = 0.72\sigma$, a 10% change in r? What is the value of u at $r = 2.5\sigma$?

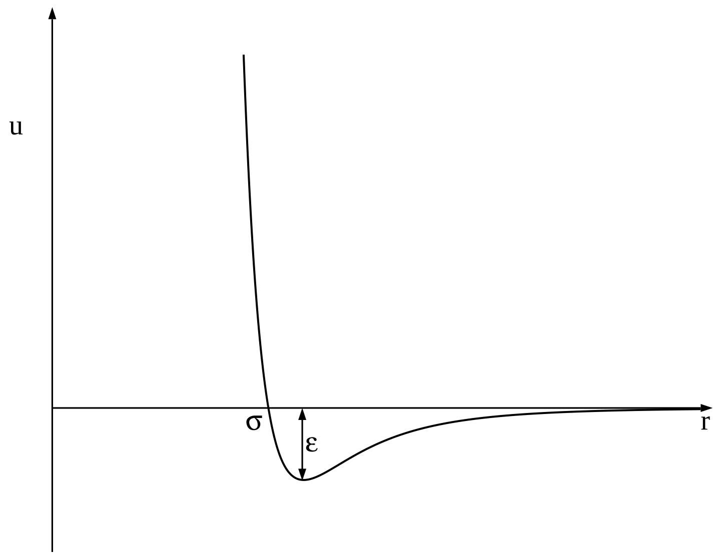

Figure 8.1: Plot of the Lennard-Jones potential $u(r)$. Note that the potential is characterized by a length σ and an energy ϵ.

8.3 Units

As usual, it is convenient to choose units so that the computed quantities are neither too small nor too large. Because the values of the distance and the energy associated with typical liquids are very small in SI units, we choose the Lennard-Jones parameters σ and ϵ as the units of distance and energy, respectively. We also choose the unit of mass to be the mass of one atom, m. We can express all other quantities in terms of σ, ϵ, and m. For example, we measure velocities in units of $(\epsilon/m)^{1/2}$, and the time in units of $\sigma(m/\epsilon)^{1/2}$. The values of σ, ϵ, and m for argon are given in Table 8.1. If we use these values, we find that the unit of time is 2.17×10^{-12} s. The units of some of the other physical quantities of interest also are shown in Table 8.1.

All program variables are in reduced units, for example, the time in our molecular dynamics program is expressed in units of $\sigma(m/\epsilon)^{1/2}$. Suppose that we run our molecular dynamics program for 2000 time steps with a time step $\Delta t = 0.01$. The total time of our run is $2000 \times 0.01 = 20$ in reduced units or 4.34×10^{-11} s for argon (see Table 8.1). The duration of a typical molecular dynamics simulation is in the range of 10–10^4 in reduced units, corresponding to a duration of approximately 10^{-11}–10^{-8} s. The longest practical runs are the order of 10^{-6} s.

8.4 The Numerical Algorithm

Now that we have specified the interaction between the particles, we need to introduce a numerical method for computing the trajectory of each particle. As we have learned, the criteria for a good numerical integration method include that it conserve the phase-space volume and be consistent with the known conservation laws, is time reversible, and is accurate for relatively large time steps

quantity	unit	value for argon
length	σ	3.4×10^{-10} m
energy	ϵ	1.65×10^{-21} J
mass	m	6.69×10^{-26} kg
time	$\sigma(m/\epsilon)^{1/2}$	2.17×10^{-12} s
velocity	$(\epsilon/m)^{1/2}$	1.57×10^{2} m/s
force	ϵ/σ	4.85×10^{-12} N
pressure	ϵ/σ^2	$1.43 \times 10^{-2}\,\mathrm{N \cdot m^{-1}}$
temperature	ϵ/k	120 K

Table 8.1: The system of units used in the molecular dynamics simulations of particles interacting via the Lennard-Jones potential. The numerical values of σ, ϵ, and m are for argon. The quantity k is Boltzmann's constant and has the value $k = 1.38 \times 10^{-23}$ J/K. The unit of pressure is for a two-dimensional system.

to reduce the CPU time needed for the total time of the simulation. These requirements mean that we should use a symplectic algorithm for the relatively long times of interest in molecular dynamics simulations. We adopt the commonly used second-order algorithm:

$$x_{n+1} = x_n + v_n \Delta t + \frac{1}{2} a_n (\Delta t)^2 \tag{8.4a}$$

$$v_{n+1} = v_n + \frac{1}{2}(a_{n+1} + a_n)\Delta t. \tag{8.4b}$$

To simplify the notation, we have written the algorithm for only one component of the particle's motion. The new position is used to find the new acceleration a_{n+1}, which is used together with a_n to obtain the new velocity v_{n+1}. The algorithm represented by (8.4) is known as the Verlet (or sometimes the velocity Verlet) algorithm (see Appendix 3A). We will use the `Verlet` implementation of the `ODESolver` interface to implement the algorithm. Thus, the x, v_x, y, and v_y values for the ith particle will be stored in the `state` array at `state[4*i]`, `state[4*i+1]`, `state[4*i+2]`, and `state[4*i+3]`, respectively.

8.5 Periodic Boundary Conditions

A useful simulation must incorporate as many of the relevant features of the physical system of interest as possible. Usually we want to simulate a gas, liquid, or a solid in the bulk, that is, systems of at least $N \sim 10^{23}$ particles. In such systems the fraction of particles near the walls of the container is negligibly small. The number of particles that can be studied in a molecular dynamics simulation is typically 10^3–10^5, although we can simulate the order of 10^9 particles using clusters of computers. For these relatively small systems, the fraction of particles near the walls of the container is significant, and hence the behavior of such a system would be dominated by surface effects.

The most common way of minimizing surface effects and to simulate more closely the properties of a bulk system is to use what are known as *periodic boundary conditions*, although the *minimum* *image approximation* would be a more accurate name. This boundary condition is familiar to anyone who has played the Pacman computer game. Consider N particles that are constrained to move on a line of length L. The application of periodic boundary conditions is equivalent to considering the line to be a circle, and hence the maximum separation between any two particles is $L/2$ (see Figure 8.2). The generalization of periodic boundary conditions to two dimensions is equivalent to imagining a box with opposite edges joined so that the box becomes the surface of a torus (the shape of a doughnut or a bagel). The three-dimensional version of periodic boundary conditions cannot be visualized easily, but the same methods can be used.

The implementation of periodic boundary conditions is straightforward. If a particle leaves the box by crossing a boundary in a particular direction, we add or subtract the length L of the box in that direction to the position. One simple way is to use an `if else` statement as shown:

Listing 8.1: Calculation of the position of particle in the central cell.

```java
private double pbcPosition(double s, double L) {
   if(s > L) {
      s -= L;
   } else if(s < 0) {
      s += L;
   }
   return s;
}
```

To compute the minimum distance `ds` in a particular direction between two particles, we can use the method `pbcSeparation` (see Figure 8.2):

Listing 8.2: Calculation of the minimum separation.

```java
private double pbcSeparation(double ds, double L) {
   if(ds > 0.5*L) {
      ds -= L;
   } else if(ds < -0.5*L) {
      ds += L;
   }
   return ds;
}
```

The equivalent static methods, `PBC.position` and `PBC.separation` in the Open Source Physics `numerics` package also can be used.

Exercise 8.2. Use of the % operator

a. Another way to compute the position of a particle in the central cell is to use the `%` (modulus) operator. For example, `17 % 5` equals 2 because 17 divided by 5 leaves a remainder of 2. The `%` operator also can be used with floating point numbers. For example, `10.2 % 5 = 0.2`. Write a little test program to see how the `%` function works and determine the result of `10.2 % 3.3`, `-10.2 % 3.3`, `10.2 % -3.3`, and `-10.2 % -3.3`. In what way does `%` act like a remainder operator?

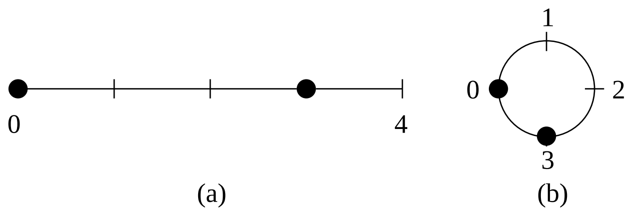

Figure 8.2: (a) Two particles at $x = 0$ and $x = 3$ on a line of length $L = 4$; the distance between the particles is 3. (b) The application of periodic boundary conditions for short range interactions is equivalent to thinking of the line as forming a circle of circumference L. In this case the minimum distance between the two particles is 1.

b. From the results of part (a) we might consider writing `x = x % L` as an alternative to Listing 8.1. What about negative values of x? In this case `-17 % 5 = -2`. Because we want the resultant position to be positive, we write

```
return x<0 ? x\%L+L: x\%L;
```

Explain this syntax and write a program to test if this statement works as claimed.

c. Write a simple program to determine if the `%` operator is faster than the if-else construction in Listing 8.1. Write another program that compares the speed of calling the `PCB.position` method to that of *inlining* the PBC code. In other words, replace the method call by the above statement.

We now discuss the nature of periodic boundary conditions. Imagine a set of N particles in a two-dimensional box or cell. The use of periodic boundary conditions implies that the central cell is duplicated an infinite number of times to fill the space. Figure 8.3 shows the first several image cells for $N = 2$. The shape of the central cell must be such that the cell fills space under successive translations. Each image cell contains the original particles in the same relative positions as the central cell. That is, periodic boundary conditions yield an infinite system, although the positions of the particles in the image cells are identical to the positions of the particles in the central cell. These boundary conditions also imply that every point in the cell is equivalent and that there is no surface.

As a particle moves in the original cell, its periodic images move in the image cells. Hence only the motion of the particles in the central cell needs to be followed. When a particle enters or leaves the central cell, the move is accompanied by an image of that particle leaving or entering a neighboring cell through the opposite face.

The total force on a given particle i is due to the force from every other particle j within the central cell and from the periodic images of particle j. That is, if particle i interacts with particle j in the central cell, then particle i interacts with *all* the periodic replicas of particle j. Hence in general, there are an infinite number of contributions to the force on any given particle. For long-range interactions such as the Coulomb potential, these contributions have to be included using special methods. For short-range interactions, we can reduce the number of contributions by adopting the minimum image approximation, which assumes that particle i in the central cell

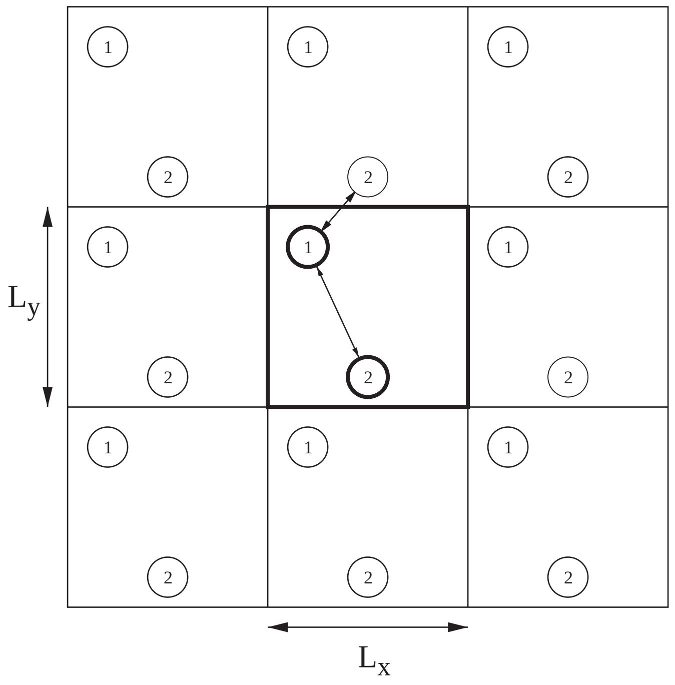

Figure 8.3: Example of the minimum image approximation in two dimensions. The minimum image distance convention implies that the separation between particles 1 and 2 is given by the lesser of the two distances shown.

interacts only with the nearest image of particle j; the interaction is set equal to zero if the distance of the image from particle i is greater than $L/2$. An example of the minimum image approximation is shown in Figure 8.3.

8.6 A Molecular Dynamics Program

In this section, we develop a molecular dynamics program to simulate a two-dimensional system of particles interacting via the Lennard-Jones potential. We choose two rather than three dimensions because it is easier to visualize the results and the calculations are not as time consuming.

In principle, we could define a class for a particle, and instantiate an object for each particle. However, this use would be very inefficient and would take up more memory and CPU time than using one class to represent all N particles. Instead we will store the x- and y-components of the positions and velocities in the `state` array and store the accelerations of the particles in a separate array. As usual, we will develop two classes, `LJParticles` and `LJParticlesApp`.

Because the system is deterministic, the nature of the motion is determined by the initial conditions. An appropriate choice of the initial conditions is more difficult than might first appear. For example, how can we choose the initial configuration (a set of positions and velocities) to correspond to a liquid at a desired temperature? According to the equipartition theorem, the mean kinetic energy of a particle per degree of freedom is $kT/2$, where k is Boltzmann's constant and T is the temperature. We can generalize this relation to define the temperature at time t by

$$kT(t) = \frac{2}{d}\frac{K(t)}{N} = \frac{1}{Nd}\sum_{i=1}^{N} m_i \mathbf{v}_i(t)\cdot\mathbf{v}_i(t), \tag{8.5}$$

where K is the total kinetic energy of the system, $\mathbf{v}_i$ is the velocity of particle i with mass m_i, and d is the spatial dimension of the system.

We can use (8.5) to choose an initial set of velocities. The following method gives the particles a random set of velocities, sets the total velocity (momentum) to zero, and then rescales the velocities so that the desired initial kinetic energy is achieved.

Listing 8.3: Method for choosing the initial velocities.

```java
public void setVelocities() {
  double vxSum = 0.0;
  double vySum = 0.0;
  for(int i = 0;i<N;++i) {    // assign random initial velocities
     state[4*i+1] = Math.random() - 0.5;    // vx
     state[4*i+3] = Math.random() - 0.5;    // vy
     vxSum += state[4*i+1];
     vySum += state[4*i+3];
   }
   // zero center of mass momentum
   double vxcm = vxSum/N;    // center of mass momentum (velocity)
   double vycm = vySum/N;
   for(int i = 0;i<N;++i) {
      state[4*i+1] -= vxcm;
      state[4*i+3] -= vycm;
   }
   double v2sum = 0;      // rescale velocities to get desired initial kinetic energy
   for(int i = 0;i<N;++i) {
      v2sum += state[4*i+1]*state[4*i+1] + state[4*i+3]*state[4*i+3];
   }
   double kineticEnergyPerParticle = 0.5*v2sum/N;
   double rescale = Math.sqrt(initialKineticEnergy/kineticEnergyPerParticle);
   for(int i = 0;i<N;++i) {
      state[4*i+1] *= rescale;
      state[4*i+3] *= rescale;
   }
}
```

We will find that setting the initial velocities so that the initial temperature is the desired value does not guarantee that the system will maintain this temperature when it reaches equilibrium. Determining an initial configuration that satisfies the desired conditions is an iterative process.

If the system is a dilute gas, we can choose the initial positions of the particles by placing them at random, making sure that no two particles are too close to one another. If two particles were too close, they would exert a very large repulsive force F on one another and any simple finite difference integration method would break down because the condition $(F/m)(\Delta t)^2 \ll \sigma$ would not be satisfied. (In dimensionless units, this condition is $F(\Delta t)^2 \ll 1$.) If we assume that the separation between two particles is greater than $2^{1/6}\sigma$, this condition is satisfied. The following method places particles at random such that no two particles are closer than $2^{1/6}\sigma$. Note that we use a `do/while` statement to insure that the body of the loop is executed at least once.

Listing 8.4: Method for choosing the initial positions at random.

```
public void setRandomPositions() { // particles placed at random, but not closer than rMinim
   double rMinimumSquared = Math.pow(2.0, 1.0/3.0);
   boolean overlap;
   for(int i = 0;i<N;++i) {
      do {
         overlap = false;
         state[4*i] = Lx*Math.random();      // x
         state[4*i+2] = Ly*Math.random();    // y
         int j = 0;
         while(j<i&&!overlap) {
            double dx = state[4*i]-state[4*j];
            double dy = state[4*i+2]-state[4*j+2];
            if(dx*dx+dy*dy<rMinimumSquared) {
               overlap = true;
            }
            j++;
         }
      } while(overlap);
   }
}
```

What is the maximum density that you can reasonably obtain in this way?

Finding a random configuration of particles in which no two particles are closer than $2^{1/6}\sigma$ becomes much too inefficient if the system is dense. It is possible to choose the initial positions randomly without regard to their separations if we include a fictitious drag force proportional to the square of the velocity. The effect of such a force is to dampen the velocity of those particles whose velocities become too large due to the large repulsive forces exerted on them. We then would have to run for a while until all the velocities satisfy the condition $v\Delta t \ll 1$. As the velocities become smaller, we may gradually reduce the friction coefficient.

In general, the easiest way of obtaining an initial configuration with the desired density is to place the particles on a regular lattice. If the temperature is high or if the system is dilute, the system will "melt" and become a liquid or a gas; otherwise, it will remain a solid. If our goal is to equilibrate the system at fluid densities, it is not necessary to choose the correct equilibrium symmetry of the lattice. The method `setRectangularLattice` in Listing 8.5 places the particles on a rectangular lattice. To make the method simple, the user must specify the number of particles per row, `nx`, and the number per column, `ny`. The linear dimensions `Lx` and `Ly` are adjustable parameters and can be varied after initialization to be as close as possible to their desired values. In this way we can vary the density by varying the volume (area) without setting up a new initial configuration.

Listing 8.5: Placement of particles on a rectangular lattice.

```
  public void setRectangularLattice() {     // place particles on a rectangular lattice
     double dx = Lx/nx; // distance between columns
     double dy = Ly/ny; // distance between rows
     for(int ix = 0; ix<nx; ++ix) { // loop through particles in a row
        for(int iy = 0; iy<ny; ++iy) { // loop through rows
           int i = ix + iy*ny;
           state[4*i] = dx*(ix+0.5);
           state[4*i+2] = dy*(iy+0.5);
        }
     }
   }
```

The most time consuming part of a molecular dynamics simulation is the computation of the accelerations of the particles. The method `computeAcceleration` determines the total force on each particle due to the other $N-1$ particles and uses Newton's third law to reduce the number of calculations by a factor of two. Hence, for a system of N particles, there are a total of $N(N-1)/2$ possible interactions. Because of the short range nature of the Lennard-Jones potential, we could truncate the force at $r = r_c$ and ignore the forces from particles whose separation is greater than r_c. However, for $N \lesssim 400$, it is easier to include all possible interactions, no matter how small. The quantity `virial` accumulated in `computeAcceleration` is discussed in Section 8.7, where we will see that it is related to the pressure. It is convenient to also calculate the potential energy in `computeAcceleration`. Note that in reduced units, the mass of a particle is unity, and hence the acceleration and force are equivalent.

Listing 8.6: Calculation of the acceleration.

```
public void computeAcceleration() {
  for(int i = 0;i<N;i++) {
     ax[i] = 0;
     ay[i] = 0;
  }
  for(int i = 0;i<N-1;i++) {
     for(int j = i+1;j<N;j++) {
        double dx = pbcSeparation(state[4*i]-state[4*j], Lx);
        double dy = pbcSeparation(state[4*i+2]-state[4*j+2], Ly);
        double r2 = dx*dx+dy*dy;
        double oneOverR2 = 1.0/r2;
        double oneOverR6 = oneOverR2*oneOverR2*oneOverR2;
        double fOverR = 48.0*oneOverR6*(oneOverR6-0.5)*oneOverR2;
        double fx = fOverR*dx;
        double fy = fOverR*dy;
        ax[i] += fx;
        ay[i] += fy;
        ax[j] -= fx;
        ay[j] -= fy;
        totalPotentialEnergyAccumulator += 4.0*(oneOverR6*oneOverR6-oneOverR6);
        virialAccumulator += dx*fx+dy*fy;
     }
  }
}
```

The methods needed for the ODE interface are given in Listing 8.7. Note that the `getRate` method is invoked twice for every call to the `step` method because we are using the Verlet algorithm. The first rate call uses the current positions of the particles, and the second rate call uses the new positions. Because a particle's new position becomes its current position for the next step, we would compute the same accelerations twice. To avoid this inefficiency, we query the ODE solver using the `getRateCounter` method to determine if the position or the velocity is being computed. We store the accelerations in an array during the second computation so that we use these values the next time `getRate` is invoked. This trick is not general and should only be used if you understand exactly how the particular ODE solver behaves. Study the implementation of the `step` method in the `Verlet` class.

Listing 8.7: Methods needed for the `ODE` interface.

```
public void getRate(double[] state, double[] rate) {
   // getRate is called twice for each call to step.
   // accelerations computed for every other call to getRate because
   // new velocity is computed from previous and current acceleration.
   // Previous acceleration is saved in step method of Verlet.
   if(odeSolver.getRateCounter()==1) {
      computeAcceleration();
   }
   for(int i = 0; i<N; i++) {
      rate[4*i] = state[4*i+1];   // rates for positions are velocities
      rate[4*i+2] = state[4*i+3]; // vy
      rate[4*i+1] = ax[i];        // rate for velocity is acceleration
      rate[4*i+3] = ay[i];
   }
   rate[4*N] = 1; // dt/dt = 1
}

public double[] getState() {
   return state;
}

public void step(HistogramFrame xVelocityHistogram) {
   odeSolver.step();
   double totalKineticEnergy = 0;
   for(int i = 0; i<N; i++) {
      totalKineticEnergy += (state[4*i+1]*state[4*i+1]+state[4*i+3]*state[4*i+3]);
      xVelocityHistogram.append(state[4*i+1]);
      state[4*i] = pbcPosition(state[4*i], Lx);
      state[4*i+2] = pbcPosition(state[4*i+2], Ly);
   }
   totalKineticEnergy *= 0.5;
   steps++;
   totalKineticEnergyAccumulator += totalKineticEnergy;
   totalKineticEnergySquaredAccumulator += totalKineticEnergy*totalKineticEnergy;
   t += dt;
   }
```

Note that we accumulate data for the histogram of the x component of the velocity in the `step` method.

Alternatively, we can implement the Verlet algorithm without the ODE interface. In the following we show the code that would replace the call to the `step` method of the ODE solver. We have used different array names for clarity. This code uses about the same amount of CPU time as the code using the ODE solver.

```
for (int i = 0;i<N;i++) { // use old acceleration
   x[i] += vx[i]*dt + ax[i]*halfdt2;      // halfdt2 = 0.5*dt*dt
   y[i] += vy[i]*dt + ay[i]*halfdt2;
   vx[i] += ax[i]*halfdt;    // add old acceleration, halfdt = 0.5*dt
   vy[i] += ay[i]*halfdt;
   }
   // computes velocity in two steps using old and new acceleration
   computeAcceleration();
   for (int i = 0;i<N;i++) { // add new acceleration
      vx[i] += ax[i]*halfdt;
      vy[i] += ay[i]*halfdt;
   }
```

In Listing 8.8 we give some of the methods for computing the temperature, pressure (see (8.8)), and the heat capacity (see (8.12)). The mean total energy should remain constant, but we compute it to test how well the algorithm conserves the total energy.

Listing 8.8: Methods used to compute averages.

```
public double getMeanTemperature() {
   return totalKineticEnergyAccumulator/(N*steps);
}

public double getMeanEnergy() {
   return totalKineticEnergyAccumulator/steps+totalPotentialEnergyAccumulator/steps;
}

public double getMeanPressure() {
   double meanVirial;
   meanVirial = virialAccumulator/steps;
   return 1.0+0.5*meanVirial/(N*getMeanTemperature());    // quantity PA/NkT
}

public double getHeatCapacity() {
   double meanTemperature = getMeanTemperature();
   double meanTemperatureSquared = totalKineticEnergySquaredAccumulator/steps;
   // heat capacity related to fluctuations of kinetic energy
   double sigma2 = meanTemperatureSquared-meanTemperature*meanTemperature;
   double denom = sigma2/(N*meanTemperature*meanTemperature) - 1.0;
   return N/denom;
}
```

```
public void resetAverages() {
   steps = 0;
   virialAccumulator = 0;
   totalPotentialEnergyAccumulator = 0;
   totalKineticEnergyAccumulator = 0;
   totalKineticEnergySquaredAccumulator = 0;
}
```

The `resetAverages` method is used to set the accumulated averages to zero so that the initial transient behavior can be removed from the computed averages.

We use the `Drawable` interface to display the trajectories of the individual particles.

Listing 8.9: The `draw` method.

```
public void draw(DrawingPanel panel, Graphics g) {
   if(state==null) {
      return;
   }
   int pxRadius = Math.abs(panel.xToPix(radius)-panel.xToPix(0));
   int pyRadius = Math.abs(panel.yToPix(radius)-panel.yToPix(0));
   g.setColor(Color.red);
   for(int i = 0;i<N;i++) {
      int xpix = panel.xToPix(state[4*i])-pxRadius;
      int ypix = panel.yToPix(state[4*i+2])-pyRadius;
      g.fillOval(xpix, ypix, 2*pxRadius, 2*pyRadius);
   }  // draw central cell boundary
   g.setColor(Color.black);
   int xpix = panel.xToPix(0);
   int ypix = panel.yToPix(Ly);
   int lx = panel.xToPix(Lx)-panel.xToPix(0);
   int ly = panel.yToPix(0)-panel.yToPix(Ly);
   g.drawRect(xpix, ypix, lx, ly);
}
```

The beginning of the `LJParticles` class includes the usual `import` statements, instance variables, and the `initialize` method. If you include all of the code we have discussed in a single file, you will have a working `LJparticles` class. Alternatively, you can download the source code from the ch08 directory.

Listing 8.10: Beginning of class `LJParticles`.

```
package org.opensourcephysics.sip.ch08.md;
import java.awt.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.frames.HistogramFrame;
import org.opensourcephysics.numerics.*;

public class LJParticles implements Drawable, ODE {
   public double state[];
   public double ax[], ay[];
   public int N, nx, ny;   // number of particles, number per row, number per column
   public double Lx, Ly;
   public double rho = N/(Lx*Ly);
   public double initialKineticEnergy;
   public int steps = 0;
   public double dt = 0.01;
   public double t;
   public double totalPotentialEnergyAccumulator;
   public double totalKineticEnergyAccumulator, totalKineticEnergySquaredAccumulator;
   public double virialAccumulator;
   public String initialConfiguration;
   public double radius = 0.5;   // radius of particles on screen
   ODESolver ode_solver = new Verlet(this);

   public void initialize() {
      N = nx*ny;
      t = 0;
      rho = N/(Lx*Ly);
      resetAverages();
      state = new double[1+4*N];
      ax = new double[N];
      ay = new double[N];
      if(initialConfiguration.equals("triangular")) {
         setTriangularLattice();
      } else if(initialConfiguration.equals("rectangular")) {
         setRectangularLattice();
      } else {
         setRandomPositions();
      }
      setVelocities();
      computeAcceleration();
      ode_solver.setStepSize(dt);
   }
```

The target class is given in Listing 8.11. When the user presses the Stop button, various thermodynamic averages are displayed in the message area of the control window. As you will find in Problem 8.8, a time consuming part of a molecular dynamics simulation is equilibrating the system, especially at high densities. The quickest way to do so is to start with a configuration that is typical of the configurations at the desired energy and density. Hence, we will want to be able to read the positions and velocities of the particles from a previously saved file. The class `LJParticlesLoader` allows us to save a configuration. This class is used in the `getLoader` method in class `LJParticlesApp`. To save a given configuration, open the File menu in the control window and choose Save As... A dialog box will open so that you can choose a name for the file, which will have the extension `xml`. To read a previously saved configuration, choose Read in the File menu. Notice that in `LJParticlesLoader`, the `loadObject` makes a call to the `computeAcceleration` and `resetAverages` methods of `LJParticles` so that the simulation can be run starting from the newly loaded configuration by next clicking the Start button. The syntax for saving a model's configuration is described in more detail in Appendix 8A.

Listing 8.11: The `LJParticlesApp` target class.

```
package org.opensourcephysics.sip.ch08.md;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.*;
import org.opensourcephysics.display.GUIUtils;

public class LJParticlesApp extends AbstractSimulation {
   LJParticles md = new LJParticles();
   PlotFrame pressureData = new PlotFrame("time", "PA/NkT", "Mean pressure");
   PlotFrame temperatureData = new PlotFrame("time", "temperature", "Mean temperature");
   HistogramFrame xVelocityHistogram = new HistogramFrame("vx", "H(vx)", "Velocity histogram'
   DisplayFrame display = new DisplayFrame("x", "y", "Lennard-Jones system");

   public void initialize() {
      md.nx = control.getInt("nx"); // number of particles per row
      md.ny = control.getInt("ny"); // number of particles per column
      md.initialKineticEnergy = control.getDouble("initial kinetic energy per particle");
      md.Lx = control.getDouble("Lx");
      md.Ly = control.getDouble("Ly");
      md.initialConfiguration = control.getString("initial configuration");
      md.dt = control.getDouble("dt");
      md.initialize();
      display.addDrawable(md);
      display.setPreferredMinMax(0, md.Lx, 0, md.Ly); // assumes vmax = 2*initalTemp and bin
      xVelocityHistogram.setBinWidth(2*md.initialKineticEnergy/md.N);
   }

   public void doStep() {
      md.step(xVelocityHistogram);
      pressureData.append(0, md.t, md.getMeanPressure());
      temperatureData.append(0, md.t, md.getMeanTemperature());
   }

   public void stop() {
      control.println("Density = "+decimalFormat.format(md.rho));
      control.println("Number of time steps = "+md.steps);
      control.println("Time step dt = "+decimalFormat.format(md.dt));
      control.println("<T>= "+decimalFormat.format(md.getMeanTemperature()));
      control.println("<E> = "+decimalFormat.format(md.getMeanEnergy()));
      control.println("Heat capacity = "+decimalFormat.format(md.getHeatCapacity()));
      control.println("<PA/NkT> = "+decimalFormat.format(md.getMeanPressure()));
   }

   public void startRunning() {
      md.dt = control.getDouble("dt");
      double Lx = control.getDouble("Lx");
      double Ly = control.getDouble("Ly");
      if((Lx!=md.Lx)||(Ly!=md.Ly)) {
         md.Lx = Lx;
         md.Ly = Ly;
         md.computeAcceleration();
         display.setPreferredMinMax(0, Lx, 0, Ly);
         resetData();
      }
   }

   public void reset() {
      control.setValue("nx", 8);
      control.setValue("ny", 8);
      control.setAdjustableValue("Lx", 20.0);
      control.setAdjustableValue("Ly", 15.0);
      control.setValue("initial kinetic energy per particle", 1.0);
      control.setAdjustableValue("dt", 0.01);
      control.setValue("initial configuration", "rectangular");
      enableStepsPerDisplay(true);
      super.setStepsPerDisplay(10);  // draw configurations every 10 steps
      display.setSquareAspect(true); // so particles will appear as circular disks
   }

   public void resetData() {
      md.resetAverages();
      GUIUtils.clearDrawingFrameData(false); // clears old data from the plot frames
   }

   public static XML.ObjectLoader getLoader() {
      return new LJParticlesLoader();
   }

   public static void main(String[] args) {
      SimulationControl control = SimulationControl.createApp(new LJParticlesApp());
      control.addButton("resetData", "Reset Data");
   }
}
```

Listing 8.12: The `LJParticlesLoader` class for saving configurations.

```
package org.opensourcephysics.sip.ch08.md;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.display.GUIUtils;

public class LJParticlesLoader implements XML.ObjectLoader {
   public Object createObject(XMLControl element) {
      return new LJParticlesApp();
   }

   public void saveObject(XMLControl control, Object obj) {
      LJParticlesApp model = (LJParticlesApp) obj;
      control.setValue("initial_configuration", model.md.initialConfiguration);
      control.setValue("state", model.md.state);
```

```
    }

    public Object loadObject(XMLControl control, Object obj) {
       // GUI has been loaded with the saved values; now restore the LJ state
       LJParticlesApp model = (LJParticlesApp) obj;
       model.initialize(); // reads values from the GUI into the LJ model
       model.md.initialConfiguration = control.getString("initial_configuration");
       model.md.state = (double[]) control.getObject("state");
       int N = (model.md.state.length-1)/4;
       model.md.ax = new double[N];
       model.md.ay = new double[N];
       model.md.computeAcceleration();
       model.md.resetAverages();
       GUIUtils.clearDrawingFrameData(false); // clears old data from the plot frames
       return obj;
    }
}
```

Problem 8.3. Approach to equilibrium

a. Consider $N = 64$ particles interacting via the Lennard-Jones potential in a square central cell of linear dimension $L = 10$. Start the system on a square lattice with an initial temperature corresponding to $T = 1.0$. Let $\Delta t = 0.01$ and run the application to make sure that it is working properly. The total energy should be approximately conserved and the trajectories of all 64 particles should be seen on the screen.

b. The kinetic temperature of the system is given by (8.5). View the evolution of the temperature of the system starting from the initial temperature. Does the temperature reach an equilibrium value? That is, does it eventually fluctuate about some mean value? What is the mean value of the temperature for the given total energy of the system?

c. Modify method `setRectangularLattice` so that all the particles are initially on the left side of a box of linear dimensions $L_x = 20$ and $L_y = 10$. Does the system become more or less random as time increases?

d. Modify the program so it computes $n(t)$, the number of particles in the left half of the cell, and plot $n(t)$ as a function of t. What is the qualitative behavior of $n(t)$? What is the mean number of particles on the left half after the system has reached equilibrium? Compare your qualitative results with the results you found in Problem 7.2.

Problem 8.4. Sensitivity to initial conditions

a. Modify your program to consider the following initial condition corresponding to $N = 11$ particles moving in the same direction with the same velocity (see Figure 8.4). Choose $L_x = L_y = 10$ and $\Delta t = 0.01$.

```
for (int i = 0; i < N; i++) {
   x[i] = Lx/2;
```

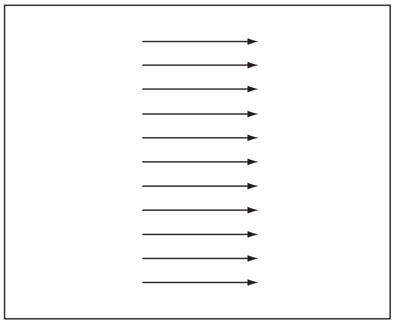

Figure 8.4: Example of a special initial condition; the arrows represent the magnitude and the direction of each particle's velocity.

```
    y[i] = (i - 0.5)*Ly/N;
    vx[i] = 1;
    vy[i] = 0;
}
```

Does the system eventually reach equilibrium? Why or why not?

b. Change the velocity of particle 6 so that $v_x(6) = 0.99999$ and $v_y(6) = 0.00001$. Is the behavior of the system qualitatively different than in part (a)? Does the system eventually reach equilibrium? Are the trajectories of the particles sensitive to the initial conditions? Explain why this behavior implies that almost all initial states lead to the same qualitative behavior (for a given total energy).

c. Modify `LJParticlesApp` so that the application runs for a predetermined time interval such as a 100 time steps, and then continues with the time reversed process, that is, the motion that would occur if the direction of time were reversed. This reversal is equivalent to letting $\mathbf{v} \to -\mathbf{v}$ for all particles or letting $\Delta t \to -\Delta t$. Do the particles return to their original positions? What happens if you reverse the velocities at a later time? What happens if you choose a smaller value of Δt?

d. Explain why you can conclude that the system is chaotic. Are the computed trajectories the same as the "true" trajectories?

From Problems 8.3 and 8.4 we see that from the *microscopic* point of view, the trajectories appear rather complex. In contrast, from the *macroscopic* point of view, the system can be described more simply. For example, in Problem 8.3 we described the approach of the system to equilibrium by specifying $n(t)$, the number of particles in the left half of the cell at time t. Your observations of the macroscopic variable $n(t)$ should be consistent with the following two general properties of systems of many particles:

1. After the removal of an internal constraint, an isolated system changes in time from a "less random" to a "more random" state.

2. A system whose macroscopic state is independent of time is said to be in *equilibrium*. The equilibrium macroscopic state is characterized by relatively small fluctuations about a mean that is independent of time. The relative fluctuations become smaller as the number of particles becomes larger.

In Problems 8.3b and 8.3c we found that the particles filled the box and did not return to their initial configuration. Hence, we were able to define a direction of time. This direction becomes better defined if we consider more particles. Note that Newton's laws of motion are time reversible and there is no a priori reason that gives the time a preferred direction.

Before we consider other macroscopic quantities, we need to monitor the total energy and verify our claim that the Verlet algorithm maintains conservation of energy with a reasonable choice of Δt. We also introduce a check for momentum conservation.

Problem 8.5. Tests of the Verlet algorithm

a. One essential check of a molecular dynamics program is that the total energy be conserved to the desired accuracy. Determine the value of Δt necessary for the total energy to be conserved to a given accuracy over a time interval of $t = 2$. One way is to compute $\Delta E_{\rm max}(t)$, the maximum value of the difference, $|E(t) - E(0)|$, over the time interval t, where $E(0)$ is the initial total energy, and $E(t)$ is the total energy at time t. Verify that $\Delta E_{\rm max}(t)$ decreases when Δt is made smaller for fixed t. If your application is working properly, $\Delta E_{\rm max}(t)$ should decrease as approximately $(\Delta t)^2$ because the Verlet algorithm is a second-order algorithm.

b. A simple way of monitoring how well the program is conserving the total energy is to use a least squares fit of the times series of $E(t)$ to a straight line. The slope of the line can be interpreted as the drift and the root mean square deviation from the straight line can be interpreted as the noise (σ_y in the notation of Section 7.6). How does the drift and the noise depend on Δt for a fixed time interval t? Most research applications conserve the energy to 1 part in 10^4 or better over the duration of the run.

c. Because of the use of periodic boundary conditions, all points in the central cell are equivalent and the system is translationally invariant. As you might have learned, translational invariance implies that the total linear momentum is conserved. However, floating point error and the truncation error associated with a finite difference algorithm can cause the total linear momentum to drift. Programming errors also might be detected by checking for conservation of momentum. Hence, it is a good idea to monitor the total linear momentum at regular intervals and reset the total momentum equal to zero if necessary. The method `setVelocities` in Listing 8.3 chooses the velocities so that the total momentum is initially zero. Add a method that resets the total momentum to zero and call it at regular intervals, for example, every 1000–10000 time steps. How well does class `LJParticles` conserve the total linear momentum for $\Delta t = 0.01$?

8.7 Thermodynamic Quantities

In the following we discuss how some of the macroscopic quantities of interest such as the temperature and the pressure can be related to time averages over the phase space trajectories of the particles.

We have already introduced the definition of the kinetic temperature in (8.5). The temperature that we measure in a laboratory experiment is the *mean* temperature, which corresponds to the time average of $T(t)$ over many configurations of the particles. For two dimensions ($d = 2$), we write the mean temperature T as

$$kT = \frac{1}{2N}\sum_{i=1}^{N} m_i\overline{\mathbf{v}_i(t)\cdot\mathbf{v}_i(t)}. \qquad \text{(two dimensions)} \tag{8.6}$$

where $\overline{X}$ denotes the time average of $X(t)$. The relation (8.6) is an example of the relation of a macroscopic quantity (the mean temperature) to a time average over the trajectories of the particles. (This definition of temperature is not adequate for particles moving relativistically or if quantum mechanics is important.)

The relation (8.5) holds only if the momentum of the center of mass of the system is zero – we do not want the motion of the center of mass to change the temperature. In a laboratory system the walls of the container ensure that the center of mass motion is zero (if the mean momentum of the walls is zero). In our simulation, we impose the constraint that the center of mass momentum (in each of the d directions) be zero. Consequently, the system has $dN - d$ independent velocity components rather than dN components, and we should replace (8.6) by

$$kT = \frac{1}{(N-1)d}\sum_{i=1}^{N} m_i\overline{\mathbf{v}_i(t)\cdot\mathbf{v}_i(t)}. \qquad \text{(correction for fixed center of mass)} \tag{8.7}$$

The presence of the factor $(N-1)d$ rather than Nd in (8.7) is an example of a *finite size* correction that becomes unimportant for large N. We shall ignore this correction in the following.

Another macroscopic quantity of interest is the mean pressure. The pressure is related to the force per unit area normal to an imaginary surface in the system. By Newton's second law, this force is related to the momentum that crosses the surface per unit time. We could use this relation to determine the pressure, but this relation uses information only from the fraction of particles that are crossing an arbitrary surface at a given time. Instead, we will use the relation of the pressure to the *virial*, which involves all the particles in the system.

In general, the momentum flux across a surface has two contributions. The contribution, NkT/V, where V is the volume (area) of the system, is due to the motion of the particles and is derived in many texts using simple kinetic theory arguments.

The other contribution to the momentum flux arises from the momentum transferred across the surface due to the forces between particles on different sides of the surface. It can be shown that the instantaneous pressure at time t including both contributions to the momentum flux is given by

$$P(t)V = NkT(t) + \frac{1}{d}\sum_{i<j}\mathbf{r}_{ij}(t)\cdot\mathbf{F}_{ij}(t), \tag{8.8}$$

where $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$ and $\mathbf{F}_{ij}$ is the force on particle i due to particle j. The second term in (8.8) is related to the virial and represents the correction to the ideal gas equation of state due to the interactions between the particles. (In two and one dimensions, we replace V by the area and length, respectively.)

The mean pressure, $P = \overline{P(t)}$, is found by computing the time average of the right-hand side of (8.8). The computed quantity is not P, but the ratio

$$\frac{PV}{NkT} - 1 = \frac{1}{dNkT} \sum_{i<j} \overline{\mathbf{r}_{ij} \cdot \mathbf{F}_{ij}}. \tag{8.9}$$

In class `LJParticles`, the sum on the right-hand side of (8.9) is computed in `computeAcceleration` and stored in the variable `virialAccumulator`.

The relation of information at the microscopic level to macroscopic quantities such as the temperature and pressure is one of the fundamental results of statistical mechanics. In brief, molecular dynamics allows us to compute various time averages of the trajectory in phase space over finite time intervals. One practical question is whether our time intervals are sufficiently long to allow the system to explore phase space and yield meaningful averages. Calculations in statistical mechanics are done by replacing time averages by *ensemble* averages over all possible configurations. The *quasi-ergodic* hypothesis asserts that these two types of averages give equivalent results if the same quantities are held fixed. In statistical mechanics, the ensemble of systems at fixed E, V, and N is called the microcanonical ensemble. Averages in this ensemble correspond to the time averages that we use in molecular dynamics which are at fixed E, V and N. (Molecular dynamics also imposes an additional, but unimportant, constraint on the center of mass motion.) Ensemble averages are explored using Monte Carlo methods in Chapter 16. A test for determining if a molecular dynamics simulation is exploring a reasonable amount of phase space is discussed in Project 8.23.

The goal of the following problems is to explore some of the qualitative features of gases, liquids, and solids. Because we will consider relatively small systems and relatively short runs, our results will only be qualitatively consistent with averages calculated in the thermodynamic limit where $N \to \infty$.

Problem 8.6. Distribution of speeds and velocities

a. In Section 7.2 we discussed how to use the `HistogramFrame` class from the Open Source Physics library. `LJParticlesApp` uses this class to compute the probability $P(v_x)\Delta v_x$ that a particle has a velocity in the x-direction between v_x and $v_x + \Delta v_x$. Add code to determine $P(v_y)$, the probability density for the y component of the velocity. What is the most probable value for the x and y velocity components? What are their average values? Plot the probability densities $P(v_x)$ versus v_x and $P(v_y)$ versus v_y. Better results can be found by plotting the average $\frac{1}{2}[P(v_x = u) + P(v_y = u)]$ versus u. What is the qualitative form of $P(\mathbf{v})$?

b. Write a method to compute the equilibrium probability $P(v)\Delta v$ that a particle has a speed between v and $v + \Delta v$. What is the qualitative form of the probability density $P(v)$? Does it have the same qualitative form as $P(\mathbf{v})$, the probability density for the velocity? What is the most probable value of v? What is the approximate width of $P(v)$? Compare your measured result to the theoretical form (in two dimensions)

$$P(v)dv = Ae^{-mv^2/2kT} v dv, \tag{8.10}$$

where A is a normalization constant. The form (8.10) of the distribution of speeds is known as the Maxwell-Boltzmann probability distribution.

c. Repeat part (b) for different densities and temperatures. Does the form of $P(v)$ depend on the density or temperature?

Problem 8.7. Qualitative properties of a liquid and a gas

a. Generate an initial configuration using `setRectangularLattice` with $N = 64$ and $L_x = L_y = 12$ and an initial temperature of 2.0. What is the density? Modify your program so that the values of the temperature and pressure are not stored until the system has reached equilibrium. One criterion for equilibrium is to compute the average values of T and P over finite time intervals and check that these averages do not drift with time.

b. Choose a value of the time step Δt so that the total energy is conserved to the desired accuracy and run the simulation for a sufficient time to estimate the equilibrium pressure and temperature. Compare your estimate for the ratio PV/NkT with its value for an ideal gas. (We have written V for the area of the system, so that the ideal gas equation of state has a familiar form.) Save the final configuration of your simulation in a file (see Appendix 8A).

c. One way of starting a simulation is to use the positions saved from an earlier run. The simplest way of obtaining an initial condition corresponding to a different density, but the same value of N, is to rescale the positions of the particles and the linear dimensions of the cell. The following code shows one way to do so.

```
for (int i = 0; i < N; i++) {
   x[i] *= rescale; // add rescale as a class variable
   y[i] *= rescale;
}
Lx = rescale*Lx
Ly = rescale*Ly
```

Incorporate this code into your program in a separate method, and add a button that lets the user call this method without initialization. This method must be used with care when increasing the density. If the density is increased too quickly, it is likely that two particles will become very close to each other so that the force will become too large and the numerical algorithm will break down. Allow the system to equilibrate after making a small density change, and then repeat until you reach the desired density. How do you expect P and T to change when the system is compressed? Gradually increase the density and determine how PV/NkT changes with increasing density. Can you distinguish the different phases? (The determination of the phase boundary between a gas, liquid, and a solid is nontrivial and is discussed in Problem 16.26.)

Another useful thermal quantity is the *heat capacity* at constant volume, which is defined by the relation $C_V = (\partial E/\partial T)_V$. (The subscript V denotes that the partial derivative is taken with the volume held fixed.) C_V is an example of a linear response function, that is, the response of the temperature to a change in the energy of the system. One way to obtain C_V is to determine $T(E)$, the temperature as a function of E. (Remember that a molecular dynamics simulation yields T as a function of E.) The heat capacity is given approximately by $\Delta E/\Delta T$ for two runs that have slightly different temperatures. This method is straightforward, but requires that simulations at different energies be done. An alternative way of determining C_V from the fluctuations of the kinetic energy is discussed in Problem 8.8c.

Problem 8.8. Energy dependence of the temperature and pressure

a. We found in Problem 8.7 that the total energy is determined by the initial conditions, and the temperature is a derived quantity found only after the system has reached thermal equilibrium. For this reason it is difficult to study the system at a particular temperature. The temperature can be changed to the desired value by rescaling the velocities of the system, but we have to be careful not to increase the velocities too quickly. Run your program to create an equilibrium configuration for $L_x = L_y = 12$ and $N = 64$ and determine $T(E)$, the energy dependence of mean temperature, in the range $T = 1.0$ to $T = 1.2$. Rescale the velocities by the desired amount over some time interval. For example, multiply all the velocities by a factor λ after each time step for a certain number of time steps. In general, the desired temperature is reached by a series of velocity rescalings over a sufficiently long time such that the system remains close to equilibrium during the rescaling.

b. Use your data for $T(E)$ found in part (a) to plot the total energy E as a function of T. Is T a monotonically increasing function of E? What percentage of the contribution to the heat capacity is due to the potential energy? Why is an accurate determination of C_V difficult to achieve?

c.* In Chapter 16 we will find that C_V is related to the fluctuations of the total energy in the canonical ensemble in which T, V, N are held fixed. In molecular dynamics simulations, the total energy is fixed, but the kinetic and potential energies can fluctuate. Another way of determining C_V is to relate it to the fluctuations of the kinetic energy. It can be shown that (cf. Ray and Graben)

$$\overline{T^2} - \overline{T}^2 = \frac{d}{2}N(k\overline{T})^2\Big[1 - \frac{dNk}{2C_V}\Big], \tag{8.11}$$

or

$$C_V = \frac{dNk}{2}\Big[1 - \frac{2}{dN}\frac{(\overline{T^2} - \overline{T}^2)}{(k\overline{T})^2}\Big]^{-1}. \tag{8.12}$$

The relation (8.12) reduces to the ideal gas result if $\overline{T^2} = \overline{T}^2$. Method `getHeatCapacity` determines C_V from (8.12). Compare your results obtained using (8.12) with the determination of C_V in part (b). What are the advantages and disadvantages of determining C_V from the fluctuations of the temperature compared to the method used in part (b)?

Problem 8.9. Ground state energy of two-dimensional lattices

To simulate a solid we need to choose the shape of the central cell to be consistent with the symmetry of the solid phase of the system. This choice is necessary even though we have used periodic boundary conditions to minimize surface effects. If the cell does not correspond to the correct crystal structure, the particles cannot form a perfect crystal, and some of the particles will wander around in an endless search for their "correct" positions. Consequently, a simulation of a small system at a high density and low temperature would lead to spurious results. In the following we compute the energy of a Lennard-Jones solid in two dimensions for the square and triangular lattices and determine which symmetry has lower energy.

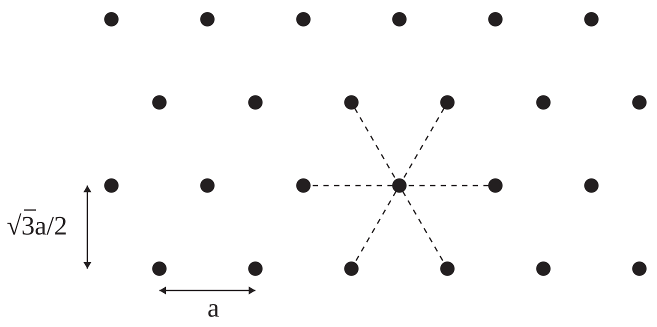

Figure 8.5: Each particle has six nearest neighbors in a triangular lattice.

a. The symmetry of the triangular lattice can be seen from Figure 8.5. Each particle has six nearest neighbors. Although it is possible to choose the central cell of the triangular lattice to be a rhombus, it is more convenient to choose the cell to be rectangular as in Figure 8.5. For a perfect crystal the linear dimensions of the cell are L_x and $L_y = \sqrt{3}L_x/2$, respectively. Use method `setTriangularLattice` in Listing 8.13 to generate the positions of the particles in a triangular lattice. Then compute the potential energy per particle of a system of $N = 64$ particles interacting via the Lennard-Jones potential. Determine the potential energy for $L_x = 8$ and $L_x = 9$.

b. Determine the potential energy for a square lattice with $L = \sqrt{L_x L_y}$, so that the triangular and square lattices have the same density. Which lattice symmetry has a lower potential energy for a given density? Explain your results in terms of the ability of the triangular lattice to pack the particles closer together.

Listing 8.13: Method for generating a triangular lattice.

```java
public void setTriangularLattice() { // place particles on triangular lattice
   double dx = Lx/nx;  // distance between particles on same row
   double dy = Ly/ny;  // distance between rows
   for(int ix = 0; ix<nx; ++ix) {
      for(int iy = 0; iy<ny; ++iy) {
         int i = ix + iy*ny;
         state[4*i+2] = dy*(iy+0.5);
         if(iy%2==0) {
            state[4*i] = dx*(ix+0.25);
         } else {
            state[4*i] = dx*(ix+0.75);
         }
      }
   }
}
```

Problem 8.10. Metastability

If we rapidly lower the temperature of a liquid below its freezing temperature, it is likely that the resulting state will not be an equilibrium crystal, but rather a supercooled liquid. If the properties of the supercooled state do not change with time for a time interval that is sufficiently long to obtain meaningful averages, we say that the system is in a *metastable* state. In general, we must carefully prepare our system so as to minimize the probability that the system becomes trapped in a metastable state. However, there is much interest in metastable states and how they eventually evolve to a more stable state (see Problem 16.20).

a. What happens if the initial positions of the particles are on the nodes of a square lattice. As we found in Problem 8.9, this symmetry is not consistent with the lowest energy state corresponding to a triangular lattice. If the initial velocities are set to zero, what happens when you run the program? Choose $N = 64$ and $L_x = L_y = 9$.

b. We can show that the system in part (a) is in a metastable state by giving the particles a small random initial velocity in the interval $[-0.5, +0.5]$. Does the symmetry of the lattice immediately change or is there a delay? When do you begin to see local structure that resembles a triangular lattice?

c. Repeat part (b) with random velocities in the interval $[-0.1, +0.1]$.

Problem 8.11. The solid state and melting

a. Choose $N = 64$, $L_x = 8$, and $L_y = \sqrt{3}L_x/2$, and place the particles on the nodes of a triangular lattice. Give each particle zero initial velocity. What is the total energy of the system? Do a simulation and measure the temperature and pressure as a function of time. Does the system remain a solid?

b. Give each particle a random velocity in the interval $[-0.5, +0.5]$. What is the total energy? Equilibrate the system and determine the mean temperature and pressure. Describe the trajectories of the particles. Are the particles localized? Is the system a solid? Save an equilibrium configuration for use in part (c).

c. Choose the initial configuration to be an equilibrium configuration from part (b), and gradually increase the kinetic energy by a factor of two. What is the new total energy? Describe the qualitative behavior of the motion of the particles. What is the equilibrium temperature and pressure of the system? After equilibrium is reached, increase the temperature again by rescaling the velocities in the same way. Repeat this rescaling and measure $P(T)$ and $E(T)$ for several different temperatures.

d. Use your results from part (c) to plot $E(T)-E(0)$ and $P(T)$ as a function of T. Is the difference $E(T)-E(0)$ proportional to T? What is the mean potential energy for a harmonic solid? What is its heat capacity?

e. Choose an equilibrium configuration from part (b) and decrease the density by rescaling L_x, L_y and the particle positions by a factor of 1.1. What is the nature of the trajectories? Decrease the density of the system until the system melts. What is your qualitative criterion for melting?

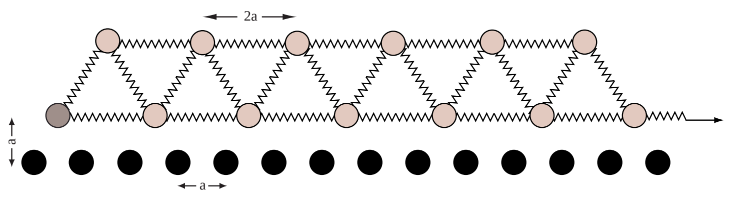

Figure 8.6: Initial configuration for the model of friction discussed in Problem 8.12. The atoms in the sliding object are placed in two rows of a triangular lattice with seven atoms in the bottom row and six atoms in the top row. There is a damping force on the left-most atom in the bottom row (the atom is shaded differently than the other atoms), and there is an external horizontal spring attached to the right-most atom in the bottom row.

Problem 8.12. Microscopic model of friction

In introductory physics texts sliding friction is usually described by the empirical law

$$f = \mu F_N, \tag{8.13}$$

where f is the magnitude of the friction force, F_N is the normal force acting on the sliding object, and μ is the coefficient of friction. If the object is not moving, then (8.13), with μ equal to the static coefficient of friction, represents the frictional force needed to start the motion. If the object is moving, then (8.13), with μ equal to the kinetic coefficient of friction, represents the kinetic frictional force, which is assumed to be independent of the speed of the sliding object.

In this problem we explore a simple model discussed by Ringlein and Robbins to investigate the microscopic origin of friction. The stationary surface is modeled by a line of fixed atoms spaced a distance of $a = 2^{1/6}$ apart as shown in Figure 8.6. The sliding object is modeled by two rows of atoms in a triangular lattice configuration initially spaced a distance $2a$ from each other. The bottom row of atoms in the sliding object is a vertical distance a from the line of fixed atoms. The interaction between all the atoms in the two objects interact via the Lennard-Jones potential. To keep the sliding object together, stiff springs with a spring constant of 500 (in reduced units) connect each atom to its nearest neighbors on the triangular lattice. The left-most atom on the bottom row has a damping force equal to $-10(v_x\hat{x} + v_y\hat{y}$ to help stabilize the motion. In addition, there is an external horizontal spring with spring constant equal to unity attached to the right-most atom on the bottom row. This spring is pulled at a constant rate causing this force on the atom to increase linearly. When this spring force is sufficiently large, the atoms start to move and the spring force suddenly drops. The point at which this decrease occurs defines the magnitude of the static frictional force.

a. Modify your molecular dynamics program to simulate this model. Choose the sliding object to consist of 13 atoms, 7 on the bottom row and 6 on the top row. Place this system of 13 atoms on the middle of a stationary surface of fixed atoms (100 such atoms should be more than enough). Your program should show the pulling force due to the spring on the right-most atom as a function of time, and a visual display of the atoms in the system. A reasonable rate for pulling the spring is 0.1, that is, the external horizontal spring force is $0.1t - u$, where u is the horizontal displacement of the right-most atom from its initial position.

b. As the system evolves, you should see the spring force suddenly drop when it reaches a value of about 14. Try different pulling rates and determine if the rate affects your results or the static friction force.

c. Add a load that is equivalent to increasing the normal force. To add a load W to the system, add a vertical force of $-W/N$ to each of the $N = 13$ atoms in the sliding object. Find the static friction force, f_s as a function of W for W between -20 and $+40$. To what does a negative load correspond? Determine the coefficient of static friction from the slope of f_s versus W.

d. Reduce the surface area by eliminating 4 atoms, 2 from each row. Rerun your simulations and discuss the results. Repeat for an increased size of 17 atoms, and fit your results to the form

$$f_s = \mu_s W + cA, \tag{8.14}$$

where A is the number of atoms in the bottom row, and c is a constant. The area dependence in (8.14) is different from what is usually assumed for sliding friction in introductory physics textbooks. The surfaces of macroscopic objects are typically rough at the microscopic level and thus the effective area of contact is much smaller than the surface area. The effective area can be proportional to the load and thus both terms in (8.14) can be proportional to the load, which is consistent with the usual assumption made in introductory physics texts.

8.8 Radial Distribution Function

We can gain more insight into the structure of a many body system by looking at how the positions of the particles are correlated with one another due to their interactions. The *radial distribution function* $g(r)$ is a measure of this correlation and has the following properties. Suppose that N particles are in a region of volume V with number density $\rho = N/V$. Choose one of the particles to be the origin. Then the mean number of other particles between $\mathbf{r}$ and $\mathbf{r} + d\mathbf{r}$ is defined to be $\rho g(\mathbf{r})\, d\mathbf{r}$. If the interparticle interaction is spherically symmetric and the system is a gas or a liquid, then $g(\mathbf{r})$ depends only on the separation $r = |\mathbf{r}|$. The normalization condition for $g(r)$ is

$$\rho \int g(r)\, d\mathbf{r} = N - 1 \approx N, \tag{8.15}$$

where the volume element $d\mathbf{r} = 4\pi r^2 dr\, (d = 3)$, $2\pi r dr\, (d = 2)$, and $2dr\, (d = 1)$. Equation (8.15) implies that if we choose one particle as the origin and count all the other particles in the system, we obtain $N - 1$ particles.

For an ideal gas, there are no correlations between the particles, and the normalization condition (8.15) implies that $g(r) = 1$ for all r. For the Lennard-Jones interaction, we expect that $g(r) \to 0$ as $r \to 0$, because the repulsive force between particles increases rapidly as $r \to \infty$. We also expect that $g(r) \to 1$ as $r \to \infty$, because the correlation of a given particle with the other particles decreases as their separation increases.

Several thermodynamic properties can be obtained from $g(r)$. Because $\rho g(r)$ can be interpreted as the local density of particles about a given particle, the potential energy of interaction between this particle and all other particles between r and $r + dr$ is $u(r)\rho g(r) d\mathbf{r}$, if we assume that only two-body interactions are present. The total potential energy is found by integrating over all r and multiplying by $N/2$. The factor of N is included because any of the N particles could be chosen as the particle at the origin, and the factor of 1/2 is included so that each pair interaction is counted only once. The result is that the mean potential energy per particle can be expressed as

$$\frac{U}{N} = \frac{\rho}{2}\int g(r)u(r)\,d\mathbf{r}. \tag{8.16}$$

It also can be shown that the relation (8.9) for the mean pressure can be rewritten in terms of $g(r)$ so that the equation of state can be expressed as

$$\frac{PV}{NkT} = 1 - \frac{\rho}{2kTd}\int g(r)\,r\frac{du(r)}{dr}\,d\mathbf{r}. \tag{8.17}$$

To determine $g(r)$ for a particular configuration of particles, we first compute $n(r, \Delta r)$, the number of particles in a spherical (circular) shell of radius r and a small, but nonzero width Δr, with the center of the shell centered about each particle. A method for computing $n(r)$ is given in Listing 8.14.

Listing 8.14: Method to compute $n(r)$.

```java
public void computeRDF() {
   // accumulate data for n(r)
   for (int i = 0; i < N-1; i++) {
      for (int j = i+1; i < N; j++) {
         double dx = PBC.separation(x[i] - x[j],Lx);
         double dy = PBC.separation(y[i] - y[j],Ly);
         double dy = (dy + Ly) % 0.5*Ly;
         double r2 = dx*dx + dy*dy;
         double r = Math.sqrt(r2);
         int bin = (int)(r/dr);  // dr = shell width
         RDFAccumulator[bin]++;
      }
   }
   numberRDFMeasurements++;
}
```

The use of periodic boundary conditions in `computeRDF` implies that the maximum separation between any two particles in the x and y direction is $L_x/2$ and $L_y/2$, respectively. Hence, we can determine $g(r)$ only for $r \leq \frac{1}{2}\min(L_x, L_y)$.

To obtain $g(r)$ from $n(r)$, we note that for a given particle i, we consider only those particles whose index j is greater than the index i (see `computeRDF`). Hence, there are a total of $\frac{1}{2}N(N-1)$ separations that are considered. In two dimensions we compute $n(r, \Delta r)$ for a circular shell whose area is $2\pi r\Delta r$. These considerations imply that $g(r)$ is related to $n(r)$ by

$$\rho g(r) = \frac{\overline{n(r,\Delta r)}}{\frac{1}{2}N\,2\pi r\Delta r}. \qquad \text{(two dimensions)} \tag{8.18}$$

Note the factor of $N/2$ in the denominator of (8.18). Method `normalizeRDF` normalizes the array `RDFAccumulator` and yields $g(r)$.

Listing 8.15: Method for obtaining $g(r)$ from $n(r)$.

```
public void normalizeRDF(PlotFrame dataRDF) {
   double density = N/(Lx*Ly);
   double L = Math.min(Lx,Ly);
   // maximum index is one less than binMax
   int binMax = (int)(L/(2*dr));
   double normalization = density*numberRDFMeasurements*N/2;
   for (int bin = 0; bin < binMax; bin++) {
      shellArea = Math.PI*(Math.pow(bin+dr,2) - Math.pow(bin,2));
      double RDF = RDFAccumulator[bin]/(normalization*shellArea);
      dataRDF.append(0,dr*(bin+0.5),g);   // adds results to be plotted
   }
}
```

The shell thickness Δr needs to be sufficiently small so that the important features of $g(r)$ are found, but large enough so that each bin has a reasonable number of contributions. The value of Δr should be a class variable. A reasonable choice for its magnitude is $\Delta r = 0.025$.

Problem 8.13. The structure of $g(r)$ for a dense liquid and a solid

a. Write a test program that incorporates `computeRDF` and `normalizeRDF` and compute $g(r)$ for a system of $N = 64$ particles that are fixed on a triangular lattice with $L_x = 8$ and $L_y = \sqrt{3}L_x/2$. What is the density of the system? What is the nearest neighbor distance between sites? At what value of r does the first maximum of $g(r)$ occur? What is the next nearest distance between sites? Does your calculated $g(r)$ have any other peaks? If so, relate these peaks to the structure of the triangular lattice.

b. Modify your molecular dynamics program and compute $g(r)$ for a dense fluid ($\rho > 0.6$, $T \approx 1.0$) with $N \geq 64$. How many peaks in $g(r)$ can you observe? In what ways do they change as the density is increased? How does the behavior of $g(r)$ for a dense liquid compare to that of a dilute gas and a solid?

8.9 Hard disks

How can we understand the temperature and density dependence of the equation of state and the structure of a dense liquid? One way to gain more insight into this dependence is to modify the interaction and see how the properties of the system change. In particular, we would like to understand the relative role of the repulsive and attractive parts of the interaction. For this reason, we consider an idealized system of hard disks for which the interaction $u(r)$ is purely repulsive:

$$u(r) = \begin{cases} +\infty, & r < \sigma \\ 0, & r \geq \sigma \end{cases} \tag{8.19}$$

The length σ is the diameter of the hard disks (see Figure 8.7). In three dimensions the interaction (8.19) describes the interaction of hard spheres (billiard balls); in one dimension (8.19) describes the interaction of hard rods.

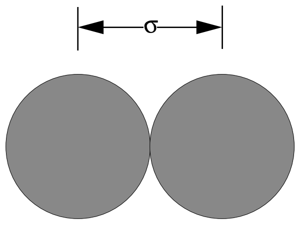

Figure 8.7: The closest distance between two hard disks is σ. The disks exert no force on one another unless they touch.

Because the interaction $u(r)$ between hard disks is a discontinuous function of r, the dynamics of hard disks is qualitatively different than it is for a continuous interaction such as the Lennard-Jones potential. For hard disks, the particles move in straight lines at constant speed between collisions and change their velocities instantaneously when a collision occurs. Hence the problem becomes finding the next collision and computing the change in the velocities of the colliding pair. The dynamics is *event driven* and can be computed exactly in principle; in practice, it is limited only by roundoff errors.

The dynamics of a system of hard disks can be treated as a sequence of two-body elastic collisions. The idea is to consider all pairs of particles i and j and to find the collision time t_{ij} for their next collision ignoring the presence of all other particles. In many cases, the particles will be going away from each other and the collision time is infinite. From the collection of collision times for all pairs of particles, we find the minimum collision time. We then move all the particles forward in time until the collision occurs and calculate the postcollision velocities of the colliding pair. The main problem is dealing with the large number of possible collision events.

We first determine the particle velocities of the colliding pair. Consider a collision between particles i and j. Let $\mathbf{v}_i$ and $\mathbf{v}_j$ be their velocities before the collision and $\mathbf{v}'_i$ and $\mathbf{v}'_j$ be their velocities after the collision. Because the particles have equal mass, it follows from conservation of energy and linear momentum that

$$v_i'^2 + v_j'^2 = v_i^2 + v_j^2, \tag{8.20}$$

$$\mathbf{v}'_i + \mathbf{v}'_j = \mathbf{v}_i + \mathbf{v}_j. \tag{8.21}$$

From (8.21) we have

$$\Delta\mathbf{v}_i = \mathbf{v}'_i - \mathbf{v}_i = -(\mathbf{v}'_j - \mathbf{v}_j) = -\Delta\mathbf{v}_j. \tag{8.22}$$

When two hard disks collide, the force is exerted along the line connecting their centers, $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$. Hence, the components of the velocities parallel to $\mathbf{r}_{ij}$ are exchanged, and the perpendicular components of the velocities are unchanged. It is convenient to write the velocity of particles i and j as a vector sum of their components parallel and perpendicular to the unit vector $\hat{\mathbf{r}}_{ij} = \mathbf{r}_{ij}/|\mathbf{r}_{ij}|$. We write the velocity of particle i as

$$\mathbf{v}_i = \mathbf{v}_{i,\parallel} + \mathbf{v}_{i,\perp}, \tag{8.23}$$

where $\mathbf{v}_{i,\parallel} = (\mathbf{v}_i \cdot \hat{\mathbf{r}}_{ij})\hat{\mathbf{r}}_{ij}$, and

$$\mathbf{v}'_{i,\parallel} = \mathbf{v}_{j,\parallel} \qquad \mathbf{v}'_{j,\parallel} = \mathbf{v}_{i,\parallel}, \tag{8.24a}$$
$$\mathbf{v}'_{i,\perp} = \mathbf{v}_{i,\perp} \qquad \mathbf{v}'_{j,\perp} = \mathbf{v}_{j,\perp}. \tag{8.24b}$$

Hence, we can write $\mathbf{v}'_i$ as

$$\begin{aligned}\mathbf{v}'_i &= \mathbf{v}'_{i,\parallel} + \mathbf{v}'_{i,\perp} = \mathbf{v}_{j,\parallel} + \mathbf{v}_{i,\perp} \\ &= \mathbf{v}_{j,\parallel} - \mathbf{v}_{i,\parallel} + \mathbf{v}_{i,\parallel} + \mathbf{v}_{i,\perp} \\ &= \big[(\mathbf{v}_j - \mathbf{v}_i)\cdot\hat{\mathbf{r}}_{ij}\big]\,\hat{\mathbf{r}}_{ij} + \mathbf{v}_i.\end{aligned} \tag{8.25}$$

The change in the velocity of particle i at a collision is given by

$$\Delta\mathbf{v}_i = \mathbf{v}'_i - \mathbf{v}_i = -\big[(\mathbf{v}_i - \mathbf{v}_j)\cdot\hat{\mathbf{r}}_{ij}\big]\hat{\mathbf{r}}_{ij}, \tag{8.26}$$

or

$$\Delta\mathbf{v}_i = -\Delta\mathbf{v}_j = \left(\frac{\mathbf{r}_{ij}\, b_{ij}}{\sigma^2}\right)_{\text{contact}}, \tag{8.27}$$

where $b_{ij} = \mathbf{v}_{ij}\cdot\mathbf{r}_{ij}$, $\mathbf{v}_{ij} = \mathbf{v}_i - \mathbf{v}_j$, and we have used the fact that $|\mathbf{r}_{ij}| = \sigma$ at contact.

Exercise 8.14. Velocity distribution of hard rods

Use (8.20) and (8.21) to show that $v'_i = v_j$ and $v'_j = v_i$ in one dimension, that is, two colliding hard rods of equal mass exchange velocities. If you start a system of hard rods with velocities chosen from a uniform random distribution, will the velocity distribution approach the equilibrium Maxwell-Boltzmann distribution?

We now consider the criteria for a collision to occur. Consider disks i and j at positions $\mathbf{r}_i$ and $\mathbf{r}_j$ at $t = 0$. If they collide at a time t_{ij} later, their centers will be separated by a distance σ:

$$|\mathbf{r}_i(t_{ij}) - \mathbf{r}_j(t_{ij})| = \sigma. \tag{8.28}$$

During the time t_{ij}, the disks move with constant velocities. Hence we have

$$\mathbf{r}_i(t_{ij}) = \mathbf{r}_i(0) + \mathbf{v}_i(0)\, t_{ij}, \tag{8.29}$$

and

$$\mathbf{r}_j(t_{ij}) = \mathbf{r}_2(0) + \mathbf{v}_2(0)\, t_{ij}. \tag{8.30}$$

If we substitute (8.29) and (8.30) into (8.28), we find

$$[\mathbf{r}_{ij} + \mathbf{v}_{ij}t_{ij}]^2 = \sigma^2, \tag{8.31}$$

where $\mathbf{r}_{ij} = \mathbf{r}_i(0) - \mathbf{r}_j(0)$, $\mathbf{v}_{ij} = \mathbf{v}_i(0) - \mathbf{v}_j(0)$, and

$$t_{ij} = \frac{-\mathbf{v}_{ij}\cdot\mathbf{r}_{ij} \pm \sqrt{(\mathbf{v}_{ij}\cdot\mathbf{r}_{ij})^2 - {v_{ij}}^2({r_{ij}}^2 - \sigma^2)}}{{v_{ij}}^2}. \tag{8.32}$$

Because $t_{ij} > 0$ for a collision to occur, we see from (8.32) that the condition,

$$\mathbf{v}_{ij} \cdot \mathbf{r}_{ij} < 0, \tag{8.33}$$

must be satisfied. That is if $\mathbf{v}_{ij} \cdot \mathbf{r}_{ij} > 0$, the particles are moving away from each other and there is no possibility of a collision.

If the condition (8.33) is satisfied, then the discriminant in (8.32) must satisfy the condition

$$(\mathbf{v}_{ij} \cdot \mathbf{r}_{ij})^2 - {v_{ij}}^2({r_{ij}}^2 - \sigma^2) \geq 0. \tag{8.34}$$

If the condition (8.34) is satisfied, then the quadratic in (8.32) has two roots. The smaller root corresponds to the physically significant collision because the disks are impenetrable. Hence, the physically significant solution for the time of a collision t_{ij} for particles i and j is given by

$$t_{ij} = \frac{-b_{ij} - \left[{b_{ij}}^2 - {v_{ij}}^2\left({r_{ij}}^2 - \sigma^2\right)\right]^{1/2}}{{v_{ij}}^2}. \tag{8.35}$$

Exercise 8.15. Calculation of collision times

Write a short program that determines the collision times (if any) of the following pairs of particles. It would be a good idea to draw the trajectories to confirm your results. Consider the cases: $\mathbf{r}_1 = (2, 1)$, $\mathbf{v}_1 = (-1, -2)$, $\mathbf{r}_2 = (1, 3)$, $\mathbf{v}_2 = (1, 1)$; $\mathbf{r}_1 = (4, 3)$, $\mathbf{v}_1 = (2, -3)$, $\mathbf{r}_2 = (3, 1)$, $\mathbf{v}_2 = (-1, -1)$; and $\mathbf{r}_1 = (4, 2)$, $\mathbf{v}_1 = (-2, \frac{1}{2})$, $\mathbf{r}_2 = (3, 1)$, $\mathbf{v}_2 = (-1, 1)$. As usual, choose units so that $\sigma = 1$.

Our hard disk program implements the following steps. We first find the collision times and the collision partners for all pairs of particles i and j. We then

1. locate the minimum collision time t_{min};
2. advance all particles using a straight line trajectory until the collision occurs, that is, displace particle i by $\mathbf{v}_i\, t_{\text{min}}$ and update its next collision time;
3. compute the postcollision velocities of the colliding pair `nextCollider` and `nextPartner`;
4. calculate the physical quantities of interest and accumulate data;
5. update the collision partners of the colliding pair, `nextCollider` and `nextPartner`, and all other particles that were to collide with either `nextCollider` or `nextPartner` if `nextCollider` and `nextPartner` had not collided first;
6. repeat steps 1–5 indefinitely.

Methods for carrying out these steps are listed in the following:

Listing 8.16: Methods for each step of the hard disk system.

```
public void step() {
   minimumCollisionTime(); // finds minimum collision time from list of collision times
   move();                 // moves particles for time equal to minimum collision time
   t += timeToCollision;
```

```
    contact();              // changes velocities of two colliding particles
    // sets collision times to bigTime for those particles set to collide with
    // two colliding particles.
    setDefaultCollisionTimes();
    newCollisionTimes();  // finds new collision times between all particles
    // and two colliding particles
    numberOfCollisions++;
}

public void minimumCollisionTime() {
    timeToCollision = bigTime;  // sets collision time very large
    // so that can find minimum collision time
    for(int k = 0; k<N; k++) {
        if(collisionTime[k]<timeToCollision) {
            timeToCollision = collisionTime[k];
            nextCollider = k;
        }
    }
    nextPartner = partner[nextCollider];
}

public void move() {
    for(int k = 0; k<N; k++) {
        collisionTime[k] -= timeToCollision;
        x[k] = PBC.position(x[k]+vx[k]*timeToCollision, Lx);
        y[k] = PBC.position(y[k]+vy[k]*timeToCollision, Ly);
    }
}

public void contact() {
    // computes collision dynamics between nextCollider and nextPartner at contact
    double dx = PBC.separation(x[nextCollider]-x[nextPartner], Lx);
    double dy = PBC.separation(y[nextCollider]-y[nextPartner], Ly);
    double dvx = vx[nextCollider]-vx[nextPartner];
    double dvy = vy[nextCollider]-vy[nextPartner];
    double factor = dx*dvx+dy*dvy;
    double delvx = -factor*dx;
    double delvy = -factor*dy;
    vx[nextCollider] += delvx;
    vy[nextCollider] += delvy;
    vx[nextPartner] -= delvx;
    vy[nextPartner] -= delvy;
    virialSum += delvx*dx+delvy*dy;
}

public void setDefaultCollisionTimes() {
    collisionTime[nextCollider] = bigTime;
    collisionTime[nextPartner] = bigTime;
   // sets collision times to bigTime for all particles set to collide with
   // the two colliding particles
   for(int k = 0; k<N; k++) {
      if(partner[k]==nextCollider) {
         collisionTime[k] = bigTime;
      } else if(partner[k]==nextPartner) {
         collisionTime[k] = bigTime;
      }
   }
}

public void newCollisionTimes() {
  // finds new collision times for all particles which were set to collide
  // with two colliding particles; also finds new collision
  // times for two colliding particles.
   for(int k = 0; k<N; k++) {
      if((k!=nextCollider)&&(k!=nextPartner)) {
         checkCollision(k, nextPartner);
         checkCollision(k, nextCollider);
      }
   }
}
```

The colliding pair and the next collision time are found in method `minimumCollisionTime`, and all the particles are moved forward in `move` until contact occurs. The collision dynamics of the colliding pair is computed in method `contact`, where the contribution to the virial also is found. In `setDefaultCollisionTimes` we set all the collisions times to an arbitrarily large value, `bigTime`, for all pairs of particles that need to be updated. Then in `newCollisionTimes` we update the collision times for those particles in step 5.

In method `initialize` we initialize various variables and most importantly compute the minimum collision time for each particle using method `checkCollision`. The ith element in the array, `collisionTime`, stores the minimum collision time for particle i with all the other particles. The array element `partner[i]` stores the particle label of the collision partner corresponding to this time. The collision time for each particle is initially set to an arbitrarily large value, `bigTime`, to take into account that at any given time, some particles have no collision partners. The methods for setting the initial positions and velocities are the same as those used for simulating Lennard-Jones particles.

Listing 8.17: Method for generating the initial configuration of hard disks.

```
public void initialize(String configuration) {
   resetAverages();
   x = new double[N];
   y = new double[N];
   vx = new double[N];
   vy = new double[N];
   collisionTime = new double[N];
   partner = new int[N];
   if(configuration.equals("regular")) {
      setRegularPositions();
   } else {
      setRandomPositions();
   }
   setVelocities();
   for(int i = 0; i<N; ++i) {
      collisionTime[i] = bigTime; // sets unknown collision times to a big number
   }
   // find initial collision times for all particles
   for(int i = 0; i<N-1; i++) {
      for(int j = i+1; j<N; j++) {
         checkCollision(i, j);
      }
   }
}

public void resetAverages() {
   t = 0;
   virialSum = 0;
}
```

Method `checkCollision` uses the relations (8.33) and (8.35) to determine whether particles `i` and `j` will collide and if so, the time `tij` until their collision. We check for collisions with particle `j` in the central cell as well as with particle `j` in the eight image cells surrounding the central cell as shown in Figure 8.8. For very dilute systems we might need to check further periodic images. For the densities we will consider, such a check should not be necessary.

Listing 8.18: Method for checking the collision time and collision partners of each particle.

```
public void checkCollision(int i, int j) {
   // consider collisions between i and j and periodic images of j
   double dvx = vx[i]-vx[j];
   double dvy = vy[i]-vy[j];
   double v2 = dvx*dvx+dvy*dvy;
   for(int xCell = -1; xCell<=1; xCell++) {
      for(int yCell = -1; yCell<=1; yCell++) {
         double dx = x[i]-x[j]+xCell*Lx;
         double dy = y[i]-y[j]+yCell*Ly;
         double bij = dx*dvx+dy*dvy;
         if(bij<0) {
            double r2 = dx*dx+dy*dy;
            double discriminant = bij*bij-v2*(r2-1);
            if(discriminant>0) {
               double tij = (-bij-Math.sqrt(discriminant))/v2;
               if(tij<collisionTime[i]) {
                  collisionTime[i] = tij;
                  partner[i] = j;
               }
               if(tij<collisionTime[j]) {
                  collisionTime[j] = tij;
                  partner[j] = i;
               }
            }
```

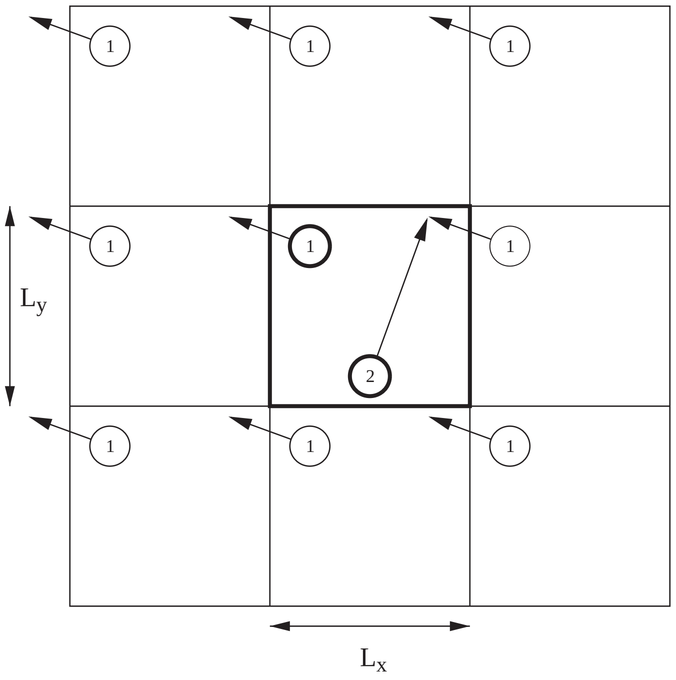

Figure 8.8: The positions and velocities of disks 1 and 2 in the figure are such that disk 1 collides with an image of disk 2 that is not the image closest to disk 1. The periodic images of disk 2 are not shown.

```
            }
         }
      }
   }
```

The main thermodynamic quantity of interest for hard disks is the mean pressure P. Because the forces act only when two disks are in contact, we have to modify the form of (8.9). We write $\mathbf{F}_{ij}(t) = \mathbf{I}_{ij}\,\delta(t - t_c)$, where t_c is the time at which the collision occurs. This form of $\mathbf{F}_{ij}$ implies that the force is nonzero only when there is a collision between i and j. The delta function $\delta(t)$ is infinite for $t = 0$ and is zero otherwise; $\delta(t)$ is defined by its use in an integral as shown in (8.36). This form of the force yields

$$\int_0^t \mathbf{I}_{ij}\,\delta(t' - t_c)\,dt' = \mathbf{I}_{ij} = m\Delta\mathbf{v}_{ij}, \tag{8.36}$$

where we have used Newton's second law and assumed that a single collision has occurred during the time interval t. The quantity $\Delta\mathbf{v}_{ij}$ is given by $\Delta\mathbf{v}_{ij} = \mathbf{v}'_i - \mathbf{v}_i - (\mathbf{v}'_j - \mathbf{v}_j)$. If we explicitly include the time average to account for all collisions during the time interval t, we can write (8.9) as

$$\begin{aligned}\frac{PV}{NkT} - 1 &= \frac{1}{dNkT}\frac{1}{t}\sum_{ij}\int_0^t \mathbf{r}_{ij}\cdot\mathbf{I}_{ij}\,\delta(t'-t_c)\,dt' \\ &= \frac{1}{dNkT}\frac{1}{t}\sum_{c_{ij}} m\Delta\mathbf{v}_{ij}\cdot\mathbf{r}_{ij}.\end{aligned} \tag{8.37}$$

The sum in (8.37) is over all collisions c_{ij} between disks i and j in the time interval t; $\mathbf{r}_{ij}$ is the vector between the centers of the disks at the time of a collision; the magnitude of $\mathbf{r}_{ij}$ in (8.37) is σ.

Listing 8.19: Method for calculating the pressure.

```java
public double pressure() {
   double area = Lx*Ly;
   return 1+virialSum/(2*t*area*N*temperature);
}
```

As discussed in Problem 8.16, an important check on the calculated trajectories of a hard disk system is that no two disks overlap. The following method tests for this condition.

```java
public void checkOverlap() {
   for (int i = 0; i < N-1; ++i) {
      for(int j = i+1; j < N; ++j) {
         double dx = PBC.separation(x[i] - x[j],Lx);
         double dy = PBC.separation(y[i] - y[j],Ly);
         if (dx*dx+dy*dy < 1.0) {
            System.out.println("Particles " + i + " and" + j + " overlap");
         }
      }
   }
}
```

To complete class `HardDisks`, we need to add the class declarations, which we show in Listing 8.20 and the `draw` method, which is the same as in class `LJParticles`. You can use a slightly modified version of class `LJParticlesApp` as the target class for this application, but note that you will need to modify the `LJParticlesLoader` class to store different arrays. The number of collisions, the time, and a plot of the pressure versus time should be displayed. We will leave the task of writing the target class as an exercise for the reader.

Listing 8.20: Class declarations for `HardDisks`.

```java
package org.opensourcephysics.sip.ch08.hd;
import java.awt.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.numerics.*;

public class HardDisks implements Drawable {
   public double x[], y[], vx[], vy[];
   public double collisionTime[];
   public int partner[];
   public int N;
   public double Lx;
   public double Ly;
   public double keSum = 0, virialSum = 0;
   public int nextCollider, nextPartner;
   public double timeToCollision;
   public double t = 0;
   public double bigTime = 1.0E10;
   public double temperature;
   public int numberOfCollisions = 0;
```

Problem 8.16. Initial test of class `HardDisks`

a. Because even a small error in computing the trajectories of the disks will eventually lead to their overlap and hence to a fatal error, it is necessary to test class `HardDisks` carefully. For simplicity, start from a lattice configuration. The most important test of the program is to monitor the computed positions of the hard disks for overlaps. If the distance between the centers of any two hard disks is less then unity (distances are measured in units of σ), there must be a serious error in the program. To check for the overlap of hard disks, include method `checkOverlap` in method `step` while you are testing the program.

b. The temperature for a system of hard disks is constant and can be defined as in (8.6). Why does the temperature not fluctuate as it does for a system of particles interacting with a continuous potential? The constancy of the temperature can be used as another check on your program. What is the effect of increasing all the velocities by a factor of two? What is the natural unit of time? Explain why the state of the system is determined only by the density and not by the temperature.

c. Generate equilibrium configurations of a system of $N = 64$ disks in a square cell of linear dimension $L = 12$. Suppose that at $t = 0$, the constraint that $0 \leq x \leq 12$ is removed, and the disks are allowed to move in a rectangular cell with $L_x = 24$ and $L_y = 12$. Does the system become more or less random? What is the qualitative nature of the time dependence of $n(t)$, the number of disks on the left half of the cell?

d. Modify your program so that averages are not computed until the system is in equilibrium. Compute the virial (8.37) and make a rough estimate of the error in your determination of the mean pressure due to statistical fluctuations.

e. Modify your program so that you can compute the velocity and speed distributions and verify that the computed distributions have the expected forms.

Problem 8.17. Static properties of hard disks

a. As we have seen in Section 8.7, a very time consuming part of the simulation is equilibrating a system from an arbitrary initial configuration. One way to obtain a set of initial positions is to add the hard disks sequentially with random positions and reject an additional hard disk if it overlaps any disks already present. Although this method is very inefficient at high densities, try it so that you will have a better idea of how difficult it is to obtain a high density configuration in this way. A much better method is to place the disks on the sites of a lattice.

b. The largest number of hard disks that can be placed into a fixed volume defines the maximum density. What is the maximum density if the disks are placed on a square lattice? What is the maximum density if the disks are placed on a triangular lattice? Suppose that the initial condition is chosen to be a square lattice with $N = 100$ and $L = 11$ so that each particle has four nearest neighbors. What is the qualitative nature of the system after several hundred collisions have occurred? Do most particles still have four nearest neighbors or are there regions where most particles have six neighbors?

c. The dependence of the mean pressure P on the density ρ is of interest, as it is for a system with a continuous potential. Is P a monotonically increasing function of ρ? Is a system of hard disks always a fluid or is there a fluid to solid transition at higher densities? You will not be able to find definitive answers to these questions for $N = 64$. Most simulations in the 1960s and 70s were done for systems of $N = 108$ hard disks and the largest simulations were for several hundred particles and much new understanding of the properties of liquids were found. Find the dependence of the pressure on the density, beginning at low densities and slowly increasing the density starting from a configuration from a lower density. At any given time the maximum density increase is given by the minimum distance between any two disks. To increase the density, rescale all the positions and the cell size so that the minimum distance is reduced by a factor of two. Repeat this procedure until you reach the desired density. You will need to equilibrate the system between rescalings.

d. Compute the radial distribution function $g(r)$ for the same densities you considered for the Lennard-Jones interaction. Computing $g(r)$ is more subtle for the hard disk system than it is for a system with a continuous potential. For the latter system, we can accumulate the sums needed to compute $g(r)$ at regular intervals and simply take the average of the computed quantities. However in an event driven dynamics, the time does not evolve uniformly. The simplest procedure is to keep track of the number of collisions and to compute the necessary sums after a certain number of collisions has occurred. If the number of collisions is sufficiently large, the time elapsed will be approximately the same. The relation of the pressure to $g(r)$ for hard disks is discussed on page 661.

e. Compare $g(r)$ for the hard disk and Lennard-Jones interactions at the same density. On the basis of your results, which part of the Lennard-Jones interaction plays the dominant role in determining the structure of a dense Lennard-Jones liquid?

Simulations of systems of hard disks and hard spheres have shown that the structure of these systems does not differ significantly from the structure of systems with more complicated interactions. Given this insight, our present understanding of liquids are based on the use of the hard sphere (disk) system as a reference system; the differences between the hard sphere interaction and the more complicated interaction of interest are treated as a perturbation about this reference system. Thus even though the particles interact strongly in a dense gas and a liquid, we now have a perturbation theory of liquids thanks to the insight gained from simulations. Another important insight that was obtained from simulations is that the solid phase does not require an attractive part of the inter-molecular potential. That is, hard spheres and disks have a freezing or melting transition, although the nature of the latter is still a subject of current interest.

In Problem 8.18 we consider two physical quantities associated with the dynamics of a system of hard disks, namely the mean free time and the mean free path, quantities of interest in kinetic theory.

Problem 8.18. Mean free path and collision time

a. Class `HardDisks` provides the information needed to determine the mean free time t_c, that is, the average time a particle travels between collisions. For example, suppose we know that 40 collisions occurred in a time $t = 2.5$ for a system of $N = 16$ disks. Because two particles are involved in each collision, there was an average of 80/16 collisions per particle. Hence $t_c = 2.5/(80/16) = 0.5$. Write a method to compute t_c and determine t_c as a function of ρ.

b. The mean free path ℓ is the mean distance a particle travels between collisions. In introductory textbooks, the relation of ℓ to t_c is given by the simple relation $\ell = \overline{v}t_c$, where $\overline{v}$ is the root-mean square velocity, $\overline{v} = \sqrt{\overline{v^2}}$. Write a method to compute the mean free path of the particles. Note that the displacement of particle i during the time t is $v_i t$, where v_i is the speed of particle i. What relation do you find between ℓ and t_c?

c. Write a method to determine the distribution of times between collisions. What is the qualitative form of the distribution? How does the width of this distribution depend on ρ?

8.10 Dynamical Properties

The mean free time and the mean free path are well defined for hard disks for which the meaning of a collision is clear. From kinetic theory we know that both quantities are related to the transport properties of a dilute gas. However, the concept of a collision is not well-defined for systems with a continuous interaction such as the Lennard-Jones potential. In the following, we take a more general approach to the dynamics of a many body system and discuss how the transport of particles in a system near equilibrium is related to the *equilibrium* properties of the system.

Consider the trajectory of a particular particle, for example, particle 1, in a system that is in equilibrium. At some arbitrarily chosen time $t = 0$, its position is $\mathbf{r}_1(0)$. At a later time t, its displacement is $\mathbf{r}_1(t) - \mathbf{r}_1(0)$. If there were no net force on the particle during this time interval, then $\mathbf{r}_1(t) - \mathbf{r}_1(0)$ would increase linearly with t. However, a particle in a fluid undergoes many collisions, and on the average its net displacement would be zero. A more interesting quantity is the mean square displacement defined as

$$\overline{R(t)^2} = \overline{[\mathbf{r}_1(t) - \mathbf{r}_1(0)]^2}. \tag{8.38}$$

The average in (8.38) is over all possible choices of the time origin. Because the system is in equilibrium, the choice of $t = 0$ is arbitrary, and $\overline{R_1(t)^2}$ depends only on the time difference t.

If the collisions of particle 1 with the other particles are random, then we would suspect that particle 1 undergoes a random walk, and the t dependence of $\overline{R(t)^2}$ would be given by (see (7.75))

$$\overline{R(t)^2} = 2dDt, \qquad (t \to \infty) \tag{8.39}$$

where d is the spatial dimension. The coefficient D in (8.39) is known as the *self-diffusion* coefficient and is an example of a transport coefficient. Because the average behavior of all the particles should be the same, we would find better results if we average over all particles. The relation (8.39) relates the macroscopic transport coefficient D to a microscopic quantity, $\overline{R(t)^2}$, and gives us a way of computing D.

The easiest way of computing $\overline{R(t)^2}$ is to save the position of a particle at regular time intervals in a file. We later can use a separate program to read the data file and compute $\overline{R(t)^2}$. To understand the procedure for computing $\overline{R(t)^2}$, we consider a simple example. Suppose that the position of a particle in a one-dimensional system is given by $x(t=0)=1.65$, $x(t=1)=1.62$, $x(t=2)=1.84$, and $x(t=3)=2.22$. If we average over all possible time origins, we obtain

$$\begin{aligned}
\overline{R(t=1)^2} &= \frac{1}{3}\big[\big(x(1)-x(0)\big)^2 + \big(x(2)-x(1)\big)^2 + \big(x(3)-x(2)\big)^2\big] \\
&= \frac{1}{3}\big[0.0009 + 0.0484 + 0.1444\big]v = 0.0646 \\
\overline{R(t=2)^2} &= \frac{1}{2}\big[\big(x(2)-x(0)\big)^2 + \big(x(3)-x(1)\big)^2\big] = \frac{1}{2}\big[0.0361 + 0.36\big] = 0.1981 \\
\overline{R(t=3)^2} &= \big[x(3)-x(0)\big]^2 = 0.3249.
\end{aligned}$$

Note that there are fewer combinations of the positions as the time difference increases.

In Listing 8.21 we show a method that computes $R(t)^2$ assuming that the positions of all N particles has been collected for `n` times in the arrays `xSave[i][k]` and `ySave[i][k]`; the time is indexed by k. Because of periodic boundary conditions, we cannot find the distance moved by a particle by just keeping track of its position. Imagine that a particle moved in only one direction and returned to its original position. If we just subtracted the coordinates of the position, we would find that the particle's displacement was zero when in fact it really moved an amount equal to the length of the simulation cell. To keep track of the movement of each particle, we use two arrays, `xWrap` and `yWrap`. Every time particle i moves past the right boundary in the time interval `k*dk` to `(k+1)*dk`, `xWrap[i][k]` is incremented by `Lx`. Similarly, each time the particle moves past the left boundary `xWrap[i][k]` is decremented by `Lx`. A similar procedure is used for `yWrap`.

Listing 8.21: Listing of method `computeR2` for finding the mean square displacement.

```java
public void computeR2(PlotFrame data) {
   for(int dk = 1; dk < n-1; ++dk) { // loops over time intervals
      int norm = 0;
      double r2 = 0;
      for(int i = 0; i < N; i++) {  // loops over particles
         // time origin labeled by k0
         for(int k0 = 0; k0 < n-dk-1; ++k0)  { // loops over time origins
          double dx = (xSave[i][k0+dk]+xWrap[i][k0+dk])-(xSave[i][k0]+xWrap[i][k0+dk]);
          double dy = (ySave[i][k0+dk]+yWrap[i][k0+dk])-(ySave[i][k0]+yWrap[i][k0+dk]);
          r2 += dx*dx + dy*dy;
          norm++;
         }
      }
      data.append(0,dk*timeInterval,r2/norm);
   }
```

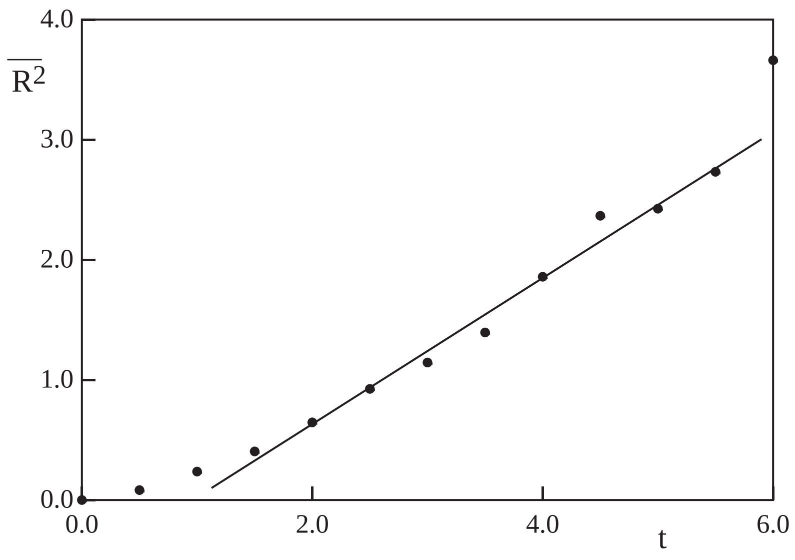

Figure 8.9: The time dependence of the mean square displacement $\overline{R(t)^2}$ for one particle in a two-dimensional Lennard-Jones system with $N = 16$, $L = 5$, and $E = 5.8115$. The position of a particle was saved at intervals of 0.5. Much better results can be obtained by averaging over all particles and over a longer run. The least squares fit was made between $t = 1.5$ and $t = 5.5$. As expected, this fit does not pass through the origin. The slope of the fit is 0.61.

```
}
```

We show our results for $\overline{R(t)^2}$ for a system of Lennard-Jones particles in Figure 8.9. Note that $\overline{R(t)^2}$ increases approximately linearly with t with a slope of roughly 0.61. From (8.39) the corresponding self-diffusion coefficient is $D = 0.61/4 \approx 0.15$. In Problem 8.19 we use method `computeR2` to compute the self-diffusion coefficient. An easier, but less direct way of computing D is discussed in Project 8.23.

Problem 8.19. The self-diffusion coefficient

a. Use either your hard disk or molecular dynamics program and visually follow the motion of a particular particle by "tagging" it, for example, by drawing its path with a different color. Describe its motion qualitatively.

b. Modify your program so that the coordinates of the particles are saved at regular intervals (see Appendix 8A). The optimum time interval needs to be determined empirically. If you save the coordinates too often, the data file will become very large, and you will waste time saving the coordinates. If you do not save the positions often enough, you will lose information. Because the time step Δt must be small compared to any interesting time scale, we know that the time interval for saving the positions must be at least an order of magnitude greater than Δt. A good first guess is to choose the time interval for saving the coordinates to be the order of $10\Delta t$.

The easiest procedure for hard disks is to save the coordinates at intervals measured in terms of the number of collisions. If you average over a sufficient number of collisions, you can find the relation between the elapsed time and the number of collisions.

c. For a finite system, the time difference t cannot be chosen to be too large because the displacement of a particle is bounded. What is the maximum value of $R^2(t)$?

d. Compute $\overline{R(t)^2}$ for conditions that correspond to a dense fluid. Does $\overline{R(t)^2}$ increase as t^2 as for a free particle or more slowly? Does $\overline{R(t)^2}$ increase linearly with t for longer times?

e. Use the relation (8.39) to estimate the magnitude of D from the slope of $\overline{R(t)^2}$ for the time interval for which $\overline{R(t)^2}$ is approximately linear. Obtain D for several different temperatures and densities. (A careful study of $\overline{R(t)^2}$ for much larger systems and much longer times would show that $\overline{R(t)^2}$ is not proportional to t in two dimensions. Instead, $\overline{R(t)^2}$ has a term proportional to $t \log t$, which dominates the linear t term if t is sufficiently large. We will not be able to observe the effects of this logarithmic term, and we can interpret our results for $\overline{R(t)^2}$ in terms of an "effective" diffusion coefficient. No such problem exists for three dimensions. See Problem 8.20d.)

f. Estimate the accuracy of your determination of D. How sensitive is it to the value of Δt? How does this accuracy compare to your estimates of other physical quantities such as the mean pressure?

g. Compute $\overline{R(t)^2}$ for an equilibrium configuration corresponding to a harmonic solid. What is the qualitative behavior of $\overline{R(t)^2}$?

h. Compute $\overline{R(t)^2}$ for an equilibrium configuration corresponding to a dilute gas. Is $\overline{R(t)^2}$ proportional to t for small times? Do the particles diffuse over short time intervals?

Another physically important property is the *velocity autocorrelation function* $C(t)$. Suppose that particle i has velocity $\mathbf{v}_i$ at time $t = 0$. If there were no net force on particle i, its velocity would remain constant. However, its interactions with other particles in the fluid will change the particle's velocity, and we expect that after several collisions, its velocity will not be strongly correlated with its velocity at an earlier time. We define $C(t)$ as

$$C(t) = \frac{1}{v_0^2} \overline{\mathbf{v}_i(t) \cdot \mathbf{v}_i(0)}, \tag{8.40}$$

where $v_0^2 = \overline{\mathbf{v}_i(0) \cdot \mathbf{v}_i(0)} = kTd/m$. We have defined $C(t)$ such that $C(t = 0) = 1$. As in our discussion of the mean square displacement, the average in (8.40) is over all possible time origins. Better results would be obtained by averaging over all particles. For large time differences t, we expect $\mathbf{v}_i(t)$ to be independent of $\mathbf{v}_i(0)$, and hence $C(t) \to 0$ for $t \to \infty$. (We have implicitly assumed that $\overline{\mathbf{v}}_i(t) = 0$.)

It can be shown that the self-diffusion coefficient defined by (8.39) can be related to the integral of $C(t)$:

$$D = v_0^2 \int_0^\infty C(t)\, dt. \tag{8.41}$$

Other transport coefficients such as the shear viscosity and the thermal conductivity also can be expressed as an integral over a corresponding autocorrelation function. The qualitative properties of the velocity autocorrelation function are explored in Problem 8.20.

Problem 8.20. The velocity autocorrelation function

a. Modify your hard disk or molecular dynamics program so that the velocity of a particular particle is saved at regular time intervals. Then modify method `computeR2` so that you can compute $C(t)$. The following code might be useful.

```
for(int timeDifference = 1; timeDifference < maxTimeDifference; timeDifference++)
   for(int timeInterval = 0; timeInterval < maxTimeInterval - timeDifference; timeInterval
      correl[timeDifference] += vxSave[timeInterval + timeDifference]*vx[timeInterval];
     correl[timeDifference] += vySave[timeInterval + timeDifference]*vy[timeInterval];
     normalization[timeDifference]++;
   }
}
```

First compute $C(t)$ for a relatively low density system. Plot $C(t)$ versus t and describe its qualitative behavior. Does it more or less decay exponentially?

b. Increase the density and compute $C(t)$ again. How does the qualitative behavior of $C(t)$ change? Why does $C(t)$ become negative after a relatively short time?

c.* To obtain quantitative results, modify your program so that $C(t)$ is averaged over all particles. Compute $C(t)$ for time differences in the range 10–40 mean collision times and densities that are about a factor of two less than maximum close packing. Also choose $N \geq 256$. If you are careful, you will be able to observe that $C(t)$ decays as t^{-1} for very long time differences. This long-time tail is due to hydrodynamic effects, that is, part of the velocity of a particle is stored in a microscopic vortex that dies off very slowly. The existence of this tail was first found by simulations and implies that that the self-diffusion constant is not defined in two dimensions because the integral (8.41) does not exist (the integral diverges for large t). In three dimensions, $C(t) \sim t^{-3/2}$ and the self-diffusion coefficient is well defined.

d. Compute $C(t)$ for an equilibrium solid. Plot $C(t)$ versus t and describe its qualitative behavior. Explain your results in terms of the oscillatory motion of the particles about their lattice sites.

e. Contrast the behavior of the mean square displacement, the velocity autocorrelation function, and the radial distribution function in the solid and fluid phases and explain how these quantities can be used to indicate the nature of the phase.

8.11 Extensions

The primary goals of this chapter have been to introduce the method of molecular dynamics, some of the concepts of statistical mechanics and kinetic theory, and the qualitative behavior of systems of many particles. Although we found that simulations of systems as small as 64 particles show some of the qualitative properties of macroscopic systems, we would need to simulate larger systems to obtain quantitative results. Most simulations of systems with simple interactions require only several hundred to several thousand particles to obtain reliable results for equilibrium quantities such as the equation of state. How do we know if the size of our system is sufficient to yield quantitative results? The simple answer is to repeat the simulation for a diferent value of N. In the same spirit, you can determine if your runs are long enough to give statistically meaningful averages.

In general, the most time consuming parts of a molecular dynamics simulation are generating an appropriate initial configuration and doing the bookkeeping necessary for the force and energy calculations. If the force is short range, there are several ways to reduce the equilibration time. For example, suppose we want to simulate a system of 864 particles in three dimensions. We first can simulate a system of 108 particles and allow the small system to come to equilibrium at the desired temperature. After equilibrium has been established, the small system can be replicated twice in each direction to generate the desired system of 864 particles. All of the velocities are reassigned at random using the Maxwell-Boltzmann distribution. Equilibration of the new system usually is established quickly.

The computer time required for our simple molecular dynamics program is order N^2 for each time step. The reason for this N^2 dependence is that the energy and force calculations require sums over all $\frac{1}{2}N(N-1)$ pairs of particles. If the interactions are short range, the time required for these sums can be reduced to approximately order N. The idea is to take advantage of the fact that many pairs of particles are separated by a distance much greater than the effective range r_c of the interparticle interaction. For example, if the distance between two particles interacting via the Lennard-Jones potential is sufficiently large, the magnitude of the potential is so small that it can be neglected. Popular choices for the cutoff r_c are 2.3σ and 2.5σ. The use of a cutoff is equivalent to assuming that $u(r)$ in (8.2) is given by the usual Lennard-Jones form for $r < r_c$ and is zero for $r > r_c$. However, this use of a cutoff implies that $u(r)$ has a discontinuity at $r = r_c$, which means that whenever a particle pair "crosses" the cutoff distance, the energy jumps, thus affecting the apparent energy conservation. To avoid this problem, it is a good idea to modify the potential so as to eliminate the discontinuity in both $u(r)$ and the force $-du/dr$. Hence, we write

$$\tilde{u}(r) = u(r) - u(r_c) - \frac{du(r)}{dr}\bigg|_{r=r_c} (r - r_c), \tag{8.42}$$

where $u(r)$ is the usual Lennard-Jones potential.

The use of the interparticle potential (8.42) to calculate the force and the energy requires considering only those pairs of particles whose separation is less than r_c. Because testing whether each pair satisfies this criterion is an order N^2 calculation, we have to limit the number of pairs tested. One way is to divide the box into small cells and to only compute the distance between particles that are in the same cell or in nearby cells. Another method is to maintain a list for each particle of its neighbors whose separation is less than a distance r_n, where r_n is chosen to be slightly greater than r_c. The idea is to use the same list of neighbors for several time steps (usually 10–20) so that the time consuming job of updating the list of neighbors does not have to be done too often. The cell method and the neighbor list method do not become efficient until N is approximately a few hundred particles.

Usually, the neighbor list leads to the consideration of fewer particle pairs in the force calculation than the cell list. We provide a method to compute the neighbor list below. A more efficient approach is to use cells to construct the neighbor list.

```
public void computeNeighborList() {
   for(int i = 0; i < N-1; i++) {
      numberInList[i] = 0;
      for(int j = i+1; j < N; j++) {
         double dx = separation(x[i] - x[j],Lx);
         double dy = separation(y[i] - y[j],Ly);
         double r2 = dx*dx + dy*dy;
         if(r2 < r2ListCutoff) {
            list[i][numberInList[i]] = j;
            numberInList[i]++;
         }
      }
   }
}
```

To use this list in method `computeAcceleration`, we replace the for loops by

```
for (int i = 0; i < N-1; i++) {
   for (int k = 0; k < numberInList[i]; k++) {
      int j = list[i][k];
   }
}
```

The method `computeNeighborList` should be called before a particle may have moved a distance equal to the difference $r_n - r_c$. This time depends on the density and the temperature. For dense systems a reasonable value for r_n is 2.7σ. Simulations of small systems can be used to determine the time between calls of `computeNeighborList`.

Note that in method `computeNeighborList` only particles $j > i$ are included in `list[i]`. In Section 16.10 we will consider Monte Carlo simulations where a particle is chosen at random and its potential energy of interaction must be computed. In this case we cannot take advantage of Newton's third law and a neighbor list must be created for all particles that are within a distance r_n of particle i.

*__Problem 8.21.__ Neighbor lists

a. Simulate a system of $N = 64$ Lennard-Jones particles in a square cell with $L = 10$ at a temperature $T = 2.0$. After the system has reached equilibrium, determine the shortest time for any particle to move a distance equal to 0.2. Use half this time in the rest of the program as the time between updates of the neighbor list.

b. Run your simulation with and without the neighbor list starting from identical initial configurations. Choose $r_c = 2.3$ and use the modified potential given in (8.42). Calculate $g(r)$, the pressure, the heat capacity (see Problem 8.8), and the temperature. Make sure your results are identical. Compare the amount of CPU time with and without the use of the neighbor list.

c. Repeat part (b) with $N = 256$, but with the same density and total energy. You can adjust the total energy by scaling the initial velocities. Increase N until the CPU time for the neighbor list version is faster.

d. Continue increasing the number of particles by a factor of four, but only use the program with the neighbor list. Determine the CPU time required for one time step as a function of N.

So far we have discussed molecular dynamics simulations at fixed energy, volume, and number of particles. Molecular dynamics simulations at fixed temperature are discussed in Project 8.24. It also is possible to modify the dynamics so as to do molecular dynamics simulations at constant pressure and to do simulations in which the shape of the cell is determined by the dynamics rather than imposed by the program. Such a simulation is essential for the study of solid-to-solid transitions where the major change is the shape of the crystal.

In addition to these algorithmic advances, there is much more to learn about the properties of the system. For example, how are transport properties such as the viscosity and the thermal conductivity related to the trajectories? We also have not discussed one of the most fundamental properties of a many body system, namely, its entropy. In brief, not all macroscopic properties of a many body system, including the entropy, can be defined as a time average over some function of the phase space coordinates of the particles (but see Ma). However, changes in the entropy can be computed by using thermodynamic integration.

The fundamental limitation of molecular dynamics is the existence of *multiple time scales*. We must choose the time step Δt to be smaller than any physical time scale in the system. For a solid, the smallest time scale is the period of the oscillatory motion of individual particles about their equilibrium positions. If we want to know how the solid responds to the addition of an interstitial particle or a vacancy, we would have to run for millions of small time steps for the vacancy to move several interparticle distances. Although this particular problem can be overcome by using a faster computer, there are many problems for which no imaginable supercomputer would be sufficient. One of the biggest current challenges is the *protein folding problem*. The biological function of a protein is determined by its three-dimensional structure which is encoded by the sequence of amino acids in the protein. At present, we know little about how the protein forms its three-dimensional structure. Such formidable computational challenges remind us that we cannot simply put a problem on a computer and let the computer tell us the answer. In particular, for many problems molecular dynamics methods need to be complemented by other simulation methods, especially Monte Carlo methods (see Chapter 16).

The emphasis in current applications of molecular dynamics is shifting from studies of simple equilibrium fluids to studies of more complex fluids and nonequilibrium systems. For example, how does a solid form when the temperature of a liquid is lowered quickly? How does a crack propagate in a brittle solid? What is the nature of the glass transition? Molecular dynamics and related methods will play an important role in aiding our understanding of these and many other problems.

8.12 Projects

Many of the pioneering applications of molecular dynamics were done on relatively small systems. It is interesting to peruse the research literature of the past three decades and to see how much physical insight was obtained from these simulations. Many research-level problems can be generated by first reproducing previously published work and then extending the work to larger systems or longer run times to obtain better statistics. Many related projects are discussed in Chapter 16.

Project 8.22. The classical Heisenberg model of magnetism

Magnetism is intrinsically a quantum phenomenon. One common model of magnetism is the Heisenberg model which is defined by the Hamiltonian or energy function:

$$H = -J \sum_{<ij>} \mathbf{S}_i \cdot \mathbf{S}_j, \tag{8.43}$$

where $\mathbf{S}_i$ is the spin operator at the ith lattice site. The sum is over nearest neighbor sites of the lattice, and the (positive) coupling constant J is a measure of the strength of the interaction between spins. The The negative sign indicates that the lowest energy state is ferromagnetic. The magnetic moment of a particle on a site is proportional to the particle's spin, and the proportionality constant is absorbed into the constant J.

For many models of magnetism such as the Ising model (see Section 16.5), there is no obvious dynamics. However, for the Heisenberg model we can motivate a dynamics using the standard rule for the time evolution of an operator given in quantum mechanics texts. For simplicity, we will consider a one-dimensional lattice. The equation for the time development becomes (see Slanič et al.)

$$\frac{d\mathbf{S}_i}{dt} = J\mathbf{S}_i \times (\mathbf{S}_{i-1} + \mathbf{S}_{i+1}). \tag{8.44}$$

In general, $\mathbf{S}$ in (8.44) is an operator. However, if the magnitude of the spin is sufficiently large, the system can be treated classically and $\mathbf{S}$ can be interpreted as a three-dimensional unit vector. The dynamics in (8.44) conserves the total energy given in (8.43) and the total magnetization, $\mathbf{M} = \sum_i \mathbf{S}_i$.

We can simulate the classical Heiseneberg magnet using an ODE solver to solve the first-order differential equation (8.44).

a. Explain why there is no obvious way to determine the mean temperature of this system.

b. Write a program to simulate the Heisenberg model on a one-dimensional lattice using periodic boundary conditions. Choose $J = 1$ and $N \geq 100$. Use the RK4 ODE solver, and plot the energy and magnetization as a function of time. These two quantities should be constant within the accuracy of the ODE solver. Also, plot each component of the spin versus position or draw a three-dimensional representation of the spin at each site so that you can visualize the state of the system.

c. Begin with all spins in the positive z direction, except for one spin pointing in the negative z direction. Use $N \geq 1000$. Define the energy of spin i as $\epsilon_i = -\mathbf{S}_i \cdot (\mathbf{S}_{i-1} + \mathbf{S}_{i+1})/2$. Plot the local energy as a function of i. Describe how the local energy diffuses. What patterns do you observe? Do the locations of the peaks in the local energy move with a constant speed?

d. One of the interesting dynamical phenomena we can explore is that of *spin waves*. Begin with all $S_{z,i} = 1$ except for a group of 20 spins, where $S_{x,i} = A\cos ki$, $S_{y,i} = A\sin ki$, and $S_{z,i} = \sqrt{1 - S_{x,i}^2 + S_{y,i}^2}$. Choose $A = 0.2$ and $k = 1$. Describe the motion of the spins. Compute the mean position of this spin wave defined by $x = \sum_i i(1 - S_{z,i})$. Show that x changes linearly with time indicating a constant spin wave velocity. Vary k and A to determine what effect their values have on the speed of the spin wave.

e. Read about sympletic algorithms in the article by Tsai, Lee, and Landau and write your own ODE solver for one of them. Compare your results to the results you found for the RK4 algorithm. Is the total energy better conserved for the same value of the time step?

Project 8.23. Single particle metrics and ergodic behavior

As mentioned in Section 8.7, the quasi-ergodic hypothesis assumes that time averages and ensemble averages are identical for a system in thermodynamic equilibrium. The assumption is that if we run a molecular dynamics simulation for a sufficiently long time, then the dynamical trajectory will fill the accessible phase space.

One way to confirm the quasi-ergodic hypothesis is to compute an ensemble average by simulating many independent copies of the system of interest using different initial configurations. Another way is to simulate a very large system and compare the behavior of different parts. A more direct measure of ergodicity (see Thirumalai and Mountain) is based on a comparison of the time averaged quantity $\overline{f_i(t)}$ of f_i for particle i to its average for all other particles. If the system is ergodic, then all particles see the same average environment, and the time average $\overline{f_i(t)}$ for each particle will be the same if t is sufficiently long. Note that $\overline{f_i(t)}$ is the average of the quantity f_i over the time interval t and not the value of f_i at time t. The time average of f_i is defined as

$$\overline{f_i(t)} = \frac{1}{t}\int_0^t f(t')\,dt', \tag{8.45}$$

and the average of $\overline{f_i(t)}$ over all particles is written as

$$\langle f(t)\rangle = \frac{1}{N}\sum_{i=1}^{N}\overline{f_i(t)}. \tag{8.46}$$

One of the physical quantities of interest is the energy of a particle, e_i, defined as

$$e_i = \frac{p_i^2}{2m_i} + \frac{1}{2}\sum_{i\neq j} u(r_{ij}). \tag{8.47}$$

The factor of 1/2 is included in the potential energy term in (8.47) because the interaction energy is shared between pairs of particles. The above considerations lead us to define the energy *metric*, $\Omega_e(t)$, as

$$\Omega_e(t) = \frac{1}{N}\sum_{i=1}^{N}\Big[\overline{e_i(t)} - \langle e(t)\rangle\Big]^2. \tag{8.48}$$

a. Compute $\Omega_e(t)$ for a system of Lennard-Jones particles at a relatively high temperature. Determine $e_i(t)$ at time intervals of 0.5 or less and average Ω_e over as many time origins as possible. If the system is ergodic over the time interval t, then it can be shown that $\Omega_e(t)$ decreases as $1/t$. Plot $1/\Omega_e(t)$ versus t. Do you find that $1/\Omega_e(t)$ eventually behaves linearly with t? Nonergodic behavior might be found by rapidly reducing the kinetic energy (a temperature quench) and obtaining an amorphous solid or glass rather than a crystalline solid. However, it would be necessary to consider three-dimensional rather than two-dimensional systems because the latter system forms a crystalline solid very quickly.

b. Another quantity of interest is the velocity metric Ω_v:

$$\Omega_v(t) = \frac{1}{dN} \sum_{i=1}^{N} \big[\overline{\mathbf{v}_i(t)} - \langle \mathbf{v}(t) \rangle\big]^2. \tag{8.49}$$

The factor of $1/d$ in (8.49) is included because the velocity is a vector with d components. If we choose the total momentum of the system to be zero, then $\langle \mathbf{v}(t) \rangle = 0$, and we can write (8.49) as

$$\Omega_v(t) = \frac{1}{dN} \sum_{i=1}^{N} \overline{\mathbf{v}_i(t)} \cdot \overline{\mathbf{v}_i(t)}. \tag{8.50}$$

As we will see, the time dependence of $\Omega_v(t)$ is not a good indicator of ergodicity, but can be used to determine the diffusion coefficient D. We write

$$\overline{\mathbf{v}_i(t)} = \frac{1}{t} \int_0^t \mathbf{v}_i(t')\, dt' = \frac{1}{t} \big[\mathbf{r}_i(t) - \mathbf{r}_i(0)\big]. \tag{8.51}$$

If we substitute (8.51) into (8.50), we can express the velocity metric in terms of the mean square displacement:

$$\Omega_v(t) = \frac{1}{dNt^2} \sum_{i=1}^{N} \big[\mathbf{r}_i(t) - \mathbf{r}_i(0)\big]^2 = \frac{\overline{R^2(t)}}{d\, t^2}. \tag{8.52}$$

The average in (8.52) is over all particles. If the particles are diffusing during the time interval t, then $\overline{R^2(t)} = 2dDt$, and

$$\Omega_v(t) = 2D/t. \tag{8.53}$$

From (8.53) we see that $\Omega_v(t)$ goes to zero as $1/t$ as claimed in part (a). However, if the particles are localized (as in a crystalline solid and a glass), then $\overline{R^2}$ is bounded for all t, and $\Omega_v(t) \sim 1/t^2$. Because a crystalline solid is ergodic and a glass is not, the velocity metric is not a good measure of the lack of ergodicity. Use the t dependence of $\Omega_v(t)$ in (8.53) to determine D for the same configurations as in Problem 8.19.

Project 8.24. Constant temperature molecular dynamics

In the molecular dynamics simulations we have discussed so far the energy is constant up to truncation, discretization, and floating point errors, and the temperature fluctuates. However, sometimes it is more convenient to do simulations at constant temperature. In Chapter 16 we will see how to simulate systems at constant T, V, and N (the canonical ensemble) by using Monte Carlo methods. However, we can also do constant temperature simulations by modifying the dynamics.

A crude way of maintaining a constant temperature is to rescale the velocities after every time step to keep the mean kinetic energy per particle constant. This approach is equivalent to a constant temperature simulation when $N \to \infty$. However, the fluctuations of the kinetic energy can be non-negligible in small systems. For such systems keeping the total kinetic energy constant in this way is not equivalent to a constant temperature simulation.

A better way of maintaining a constant temperature is based on imagining that every particle in the system is connected to a much larger system called a *heat bath.* The heat bath is sufficiently large so that it has a constant temperature even if it loses or gains energy. The particles in the system of interest occasionally collide with particles in this heat bath. The effect of these collisions is to give the particles random velocities with the desired probability distribution (see Problem 8.6). We first list the algorithm and give its rationale later. Add the following statements to method `step` after all the particles have been moved.

Listing 8.22: Andersen thermostat.

```
for (int i = 0; i < N; i++) {
   if (Math.random() < collisionProbability) {
      double r1 = Math.random();
      double r2 = Math.random()*2.0*Math.PI;
      state[4*i+1] = Math.sqrt(-2.0*temperature*Math.log(r1))*Math.cos(r2);
// vx
      state[4*i+3] = Math.sqrt(-2.0*temperature*Math.log(r1))*Math.sin(r2);
// vy
   }
}
```

The parameter `collisionProbability` is much less than unity and determines how often there is a collision with the heat bath. This way of maintaining constant temperature is known as the *Anderson thermostat*.

a. Do a constant energy simulation as before, using an initial configuration for which the desired temperature is equal to 1.0. Make sure the total momentum is zero. Choose $N = 64$ and place the particles initially on a triangular lattice with $L_x = 10$ and $L_y = \sqrt{3}L_x/2$. Plot the instantaneous temperature defined as in (8.5) and compute the average temperature. Estimate the magnitude of the temperature fluctuations. Repeat your simulation for some other initial configurations.

b. Modify your program to use the Anderson thermostat at a constant temperature set equal to 1.0. Set `collisionProbability` = 0.0001. Repeat the calculations of part (a) and compare them. Discuss the differences. Do the results change significantly?

c. Modify your program to do a simple constant kinetic energy ensemble where the velocities are rescaled after every time step so that the total kinetic energy does not change. What is the final temperature now? How do your results compare with parts (a) and (b)? Are the differences in the computed thermodynamic averages statistically significant?

d. Compute the velocity probability distribution for each case. How do they compare? Consider `collisionProbability` = 0.001 and 0.00001.

e. A deterministic algorithm for constant temperature molecular dynamics is the Nosé-Hoover thermostat. The idea is to introduce an additional degree of freedom s that plays the role of the heat bath. The derivation of the appropriate equations of motion is an excellent example of the Lagrangian formulation of mechanics. The equations of motion of Nosé-Hoover dynamics are

$$\frac{d\mathbf{p}_i}{dt} = \mathbf{F}_i(t) - s\mathbf{p}_i \tag{8.54}$$

$$\frac{ds}{dt} = \frac{1}{M}\big[\sum_i \frac{p_i^2}{m_i} - dNkT\big], \tag{8.55}$$

where T is the desired temperature and M is a parameter that can be interpreted as the mass associated with the extra degree of freedom. Equation (8.54) is similar to Newton's equations of motion with an additional friction term. However, the coefficient s can be positive or negative. Equation (8.55) defines the way s is changed to control the temperature. Apply the Nosé-Hoover algorithm to simulate a simple harmonic oscillator at constant temperature. Plot the phase space trajectory. If the energy were constant, the trajectory would be an ellipse. How does the shape of the trajectory depend on M? Choose M so that the period of any oscillations due to the finite value of M is much longer than the period of the system.

Project 8.25. Simulations on the surface of a sphere

Because of the long-range nature of the Coulomb potential, we have to sum all the periodic images of the particles to compute the force on a given particle. Although there are special methods to do these sums so that they converge quickly (Ewald sums), the simulation of systems of charged particles is more difficult than systems with short-range potentials. An alternative approach that avoids periodic boundary conditions is to not have any boundaries at all. For example, if we wish to simulate a two-dimensional system, we can consider the motion of the particles on the surface of a sphere. If the radius of the sphere is sufficiently large, the curvature of the surface can be neglected. Of course, there is a price – the coordinate system is no longer Cartesian.

Although this approach also can be applied to systems with short-range interactions, it is more interesting to apply it to charged particles. The simplest system of interest is a model of charged particles moving in a uniform background of opposite charge to ensure overall charge neutrality, the one-component plasma (OCP). In two dimensions this system is a simplified model of electrons on the surface of liquid Helium. The properties of the OCP are determined by the dimensionless parameter Γ given by the ratio of the potential energy between nearest neighbor particles to the mean kinetic energy of a particle, $\Gamma = (e^2/a)/kT$, where $\rho\pi a^2 = 1$ and ρ is the number density. Systems with $\Gamma >> 1$ are called strongly coupled. For $\Gamma \sim 100$ in two dimensions, the system forms a solid. Strongly coupled one-component plasmas in three dimensions are models of dense astrophysical matter.

Assume that the origin of the coordinate system is at the center of the sphere and that $\mathbf{u}_i$ is a unit vector from the origin to the position of particle i on the sphere. Then $R\theta_{ij}$ is the length of the chord joining particle i and j, where $\cos\theta_{ij} = \mathbf{u}_i \cdot \mathbf{u}_j$. Newton's equation of motion for the ith electron has the form

$$m\ddot{\mathbf{u}}_i = -\frac{e^2}{R^2}\sum_{j\neq i}\frac{1}{\theta_{ij}^2 \sin\theta_{ij}}[\mathbf{u}_j - (\cos\theta_{ij}\mathbf{u}_i]. \tag{8.56}$$

Note that the unit vector $\mathbf{w}_{ij} = [\mathbf{u}_j - (\cos\psi_{ij}\mathbf{u}_i]/\sin\theta_{ij}$ is orthogonal to $\mathbf{u}_i$. In addition, we must take into account that the particles must stay on the surface of the sphere, so there is an additional force on particle i toward the center of magnitude $m|\dot{\mathbf{u}}_i|^2/R$.

a. What are the appropriate units for length, time, and the self-diffusion constant?

b. Write a program to compute the velocity correlation function given by

$$C(t) = \frac{1}{v_0^2}\overline{\dot{\mathbf{u}}(t)\cdot\dot{\mathbf{u}}(0)}, \tag{8.57}$$

where $v_0^2 = \overline{\mathbf{u}(0)\cdot\mathbf{u}(0)}$. To compute the self-diffusion constant D, we let $\cos\theta(t) = \mathbf{u}(t)\cdot\mathbf{u}(0)$, so that $R\theta$ is the circular arc from the initial position of a particle to its position on the sphere at time t. We then define

$$D(t) = \frac{1}{a^2}\frac{\overline{\theta^2(t)}}{4t}, \tag{8.58}$$

where D and t are dimensionless variables. The self-diffusion constant D corresponds to the limit $t \to \infty$. Choose $N = 104$ and a radius R corresponding to $\Gamma \approx 36$ as in the original simulations by Hansen et al., and then consider bigger systems. Can you conclude that the self-diffusion exists for the two-dimensional OCP?

c.* Use a similar procedure to compute the velocity autocorrelation function and the self-difusion constant D for a two-dimensional system of Lennard-Jones particles. Can you conclude that the self-diffusion exists for this two-dimensional system?

Project 8.26. Granular matter

Recently, physicists have become very interested in granular matter such as sand. The key difference between molecular systems and granular systems is that the inter-particle interactions in the latter are inelastic. The lost energy goes into the internal degrees of freedom of a grain, and ultimately is dissipated. From the point of view of the motion of the granular particles, the energy is lost. Experimentalists have studied models of granular material composed of small steel balls or glass beads using sophisticated imaging techniques that can track the motion of individual particles. There also have been many complementary computer simulation studies.

What are some of the interesting properties of granular matter? Because the interactions are inelastic, granular particles will ultimately come to rest unless there is an external source of energy, usually a vibrating wall or gravity (for example, the fall of particles through a funnel). When granular particles come to rest, they can form a granular solid that is different than molecular solids. One difference is that there frequently exists a complex network of force lines within the solid. In addition, unlike ordinary liquids, the pressure does not increase with depth because the walls of the container help support the grains. As a consequence, sand flowing out of an aperture flows at a constant rate independent of the height of the sand above the aperture. For this reason sand is used in hour glasses. Another interesting property is that under some conditions the large grains in a mixture of large and small grains can move to the top while the container is being vibrated – the "Brazil nut" effect. Under other conditions the large grains might move to the bottom. What happens depends on the size and density of the large grains compared to the small grains (see Sanders et al.).

It also is known that there is a critical angle for the slope of a sand pile, above which the sand pile is unstable. This slope is called the angle of repose. These and many other effects have been studied using theoretical, computational, and experimental techniques.

The first step in simulating granular matter is to determine the effective force law between particles. For granular gases the details of the force do not influence the qualitative results, as long as the force is purely repulsive and short range, and there is some mechanism for dissipating energy. Common examples of force laws are spring-like forces with stiff spring constants and hard disks with inelastic collisions. For simplicity, we will consider the Lennard-Jones potential with a cut off at $r_c = 2^{1/6}$ so that the force is always repulsive. To remove energy during a collision, we will introduce a viscous damping force given by

$$f_{ij} = -\gamma(\mathbf{v}_{ij} \cdot \mathbf{r}_{ij})\frac{\mathbf{r}_{ij}}{r_{ii}^2}, \tag{8.59}$$

where the viscous damping coefficient $\gamma = 100$ in reduced units. A more realistic force model necessary for granular flow problems is given in Hirchfeld et al.

a. Modify class `LJParticles` so that the cutoff is at $2^{1/6}$. Is the total energy conserved. If not, why? Include a viscous damping force as in (8.59) and plot the kinetic energy per particle versus time. We will define the kinetic temperature to be the mean kinetic energy per particle. Why does this definition of temperature not have the same significance as the temperature in molecular systems in which the energy is conserved? Choose $N = 64$, $L = 20$, and $\Delta t = 0.001$. Begin with a random configuration and initial kinetic temperature equal to 10. How long does it take for the kinetic temperature to decrease to 10% of its initial value? Describe the spatial distribution of the particles at this time.

b. Compute the mean kinetic temperature versus time averaged over three runs. What functional form describes your results for the mean kinetic temperature at long times?

c. To prevent "granular collapse" where the particles ultimately come to rest, we need to add energy to the system. The simplest way of doing so is to give random kicks to randomly selected particles. You can use the same algorithm we used to set the initial velocities in `LJParticles`:

```
int i = (int)(N*math.random())   // selects random particle
double r = Math.random();        // use to generate Gaussian distribution
double a = -Math.log(r);
double theta = 2.0*Math.PI*Math.random();
// assign velocities according to Maxwell-Boltzmann distribution using Box-Muller method
state[4*i+1] = Math.sqrt(2.0*desiredKE*a)*Math.cos(theta);    // vx
state[4*i+3] = Math.sqrt(2.0*desiredKE*a)*Math.sin(theta);    // vy
```

(The Box-Mueller method is described in Section 12.5.) Assume that at each time step one particle is chosen at random and receives a random kick. Adjust `desiredKE` so that the mean kinetic energy per particle remains roughly constant at about 5.0. Compute the velocity distribution function for each component of the velocity. Compare this distribution on the same plot to the Gaussian distribution

$$p(v_x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(v_x - \langle v_x \rangle)^2/2\sigma^2}, \tag{8.60}$$

where $\sigma^2 = \langle v_x^2 \rangle - \langle v_x \rangle^2$. Is the velocity distribution function of the system a Gaussian? If not, give a physical explanation for the difference.

Appendix: Reading and Saving Configurations

For most of the problems in this chapter qualitative results can be obtained fairly quickly. However, in research applications the time for running a simulation is likely to be much longer than a few minutes and runs that require days or even months are not uncommon. In such cases it is important to be able to save the intermediate configurations to prevent the potential loss of data in the case of a computer crash or power failure.

Also, in many cases it is easier to save the configurations periodically and then use a separate program to analyze the configurations and compute the quantities of interest. In addition, if we wish to compute averages as a function of a parameter such as the temperature, it is convenient to make small changes in the temperature and use the last configuration from the previous run as the initial configuration for the simulation at the new temperature.

The standard Java API has methods for reading and writing files. The usual way of saving a configuration is to use these methods to write all the positions and velocities simply as numbers into a file. Additional simulation parameters and information about the configuration would be saved using a custom format. Although this approach is the traditional one for data storage, the use of a custom format means that you might not remember the format later and sharing data between programs and other users becomes more difficult.

An alternative is to use a more structured and widely shared format for storing data. The Open Source Physics library has support for the Extensive Markup Language (XML). The XML format offers a number of advantages for computational physics: clear markup of input data and results, standardized data formats, and easier exchange and archival stability of data. In simple terms the main advantage of XML is that it is a human readable format; just by looking at an XML file you can get an idea of the nature of the data.

The XML classes in the Open Source Physics library can be understood by reading the `XML-ExampleApp` example. The XML API is very similar to the control API. For example, we use `setValue` to add data to an XML control, and we use `getInt`, `getDouble`, and `getString` to read data. We start by importing the necessary definitions from the controls package and defining the `main` method for the `ExampleXMLApp` class. Note that `XMLControl` defines an interface and `XMLControlElement` defines an implementation of this interface.

```
import org.opensourcephysics.controls.XMLControl;
import org.opensourcephysics.controls.XMLControlElement;

public class ExampleXMLApp {
   public static void main(String[] args) {
   ...
}
```

The following Java statements are placed in the body of the `main` method. An empty XML document is created using an `XMLControl` object by calling the `XMLControlElement` constructor without any parameters.

```
XMLControl xmlOut = new XMLControlElement();
```

Invoking the control's `setValue` method creates an XML element consisting of a tag and and a data value. The tag is the first parameter and the data to be stored is the second. Data that can be stored includes numbers, number arrays, and strings. Because the tag is unique, the data can later be retrieved from the control using the appropriate get method.

```
xmlOut.setValue("comment", "An XML description of an array.");
xmlOut.setValue("x positions", new double[]{1,3,4});
xmlOut.setValue("x velocities", new double[]{0,-1,1});
```

Once the data has been stored in an `XMLControl` object, it can be exported to a file by calling the `write` method. In this example, the name of the file is `MDconfiguration.xml`.

```
xmlOut.write("MDconfiguration.xml");
```

An `XMLControl` also can be used to read XML data from a file. In the next example, we will read from the file that we just saved. We start by instantiating a new `XMLControl` named `xmlIn`.

```
XMLControl xmlIn = new XMLControlElement("particle_configuration.xml");
```

The new `XMLControl` object, `xmlIn`, contains the same data as the object we saved, `xmlOut`. Its data can be accessed using a tag name. Note that the `getObject` method returns a generic `Object` and must be cast to the appropriate data type.

```
System.out.println(xmlIn.getString("comment"));
double[] xPos = (double[])xmlIn.getObject("x positions");
double[] xVel = (double[])xmlIn.getObject("x velocities");
for(int i = 0; i < xPos.length; i++) {
   System.out.println("x[i] = " + xPos[i] +" vx[i] = " + xVel[i]);
}
```

Exercise 8.27. Saving XML data

a. Combine the above statements to create a working `XMLControlApp` class. Examine the saved data using a text editor. Describe how the parameters are stored.

b. Run `HardDisksApp` and save the control's configuration using the `Save As` item under the file menu in the toolbar. Examine the saved file using a text editor and describe how this file is different from the file you generated in part (a).

c. What is the minimum amount of information that must be stored in a configuration file to specify the current `HardDisks` state?

d. Add custom buttons to `HardDisksApp` to store and load the current `HardDisks` state. Test your code by showing that quantities, such as the temperature, remain the same if a configuration is stored and reloaded.

Open Source Physics user interfaces, such as a `SimulationControl`, store a program's configuration in two steps. During the first step, parameters from the graphical user interface are stored. During the second step, the model is given the opportunity to store runtime data using an `ObjectLoader`. Study the `LJParticlesLoader` class and note how storing and loading are done in the `saveObject` and `loadObject` methods, respectively. You will adapt this `ObjectLoader` to store `HardDisks` data in Problem 8.28. Additional information about how Open Source Physics applications store XML-based configurations is provided in the *Open Source Physics Users Guide.*

Problem 8.28. Hard disk configuration

a. Create a `HardDisksLoader` class that stores the `HardDisks` runtime data.

b. Add the `getLoader` method to `HardDisksApp` and test the loader.

```
public static XML.ObjectLoader getLoader() {
   return new HardDisksLoader();
}
```

Method `XML.ObjectLoader getLoader` allows the `SimulationControl` to obtain the `HardDisksLoader`, which will be used to store the runtime data. Data written by the loader's `saveObject` method will be included in the output file when the user saves a program configuration. Describe how the initialization parameters and the runtime data are separated in the XML file.

Because XML allows for the creation of custom tags, various companies and professional organizations have defined other XML grammars such as *MathML*. See `<http://xml.comp-phys.org/>` for another example of the use of XML in computational physics.

References and Suggestions for Further Reading

One of the best ways of testing your programs is by comparing your results with known results. The Web site, `<http://www.cstl.nist.gov/lj>`, maintained by the National Institute of Standards and Technology (U.S.) provides some useful benchmark results for the Lennard-Jones fluid.

Farid F. Abraham, "Computational statistical mechanics: Methodology, applications and supercomputing," Adv. Phys. **35**, 1–111 (1986). The author discusses both molecular dynamics and Monte Carlo techniques.

B. J. Alder and T. E. Wainwright, "Phase transition for a hard sphere system," J. Chem. Phys. **27**, 1208–1209 (1957).

M. P. Allen and D. J. Tildesley, *Computer Simulation of Liquids*, Clarendon Press (1987). A classic text on molecular dynamics and Monte Carlo methods.

Jean-Louis Barrat and Jean-Pierre Hansen, *Basic Concepts for Simple and Complex Liquids*, Cambridge University Press (2003). Also see Jean-Pierre Hansen and Ian R. McDonald, *Theory of Simple Liquids*, second edition, Academic Press (1986). Excellent graduate level texts that derive most of the theoretical results mentioned in this chapter.

Kurt Binder, Jürgen Horbach, Walter Kob, Wolfgang Paul, and Fathollah Varnik, "Molecular dynamics simulations," J. Phys.: Condens. Matter **16**, S429–S453 (2004).

R. P. Bonomo and F. Riggi, "The evolution of the speed distribution for a two-dimensional ideal gas: A computer simulation," Am. J. Phys. **52**, 54–55 (1984). The authors consider a system of hard disks and show that the system evolves to the Maxwell-Boltzmann distribution.

J. P. Boon and S. Yip, *Molecular Hydrodynamics*, Dover (1991). Their discussion of transport properties is an excellent supplement to our brief discussion.

Giovanni Ciccotti and William G. Hoover, editors, *Molecular-Dynamics Simulation of Statistical-Mechanics Systems*, North-Holland (1986).

Giovanni Ciccotti, Daan Frenkel, and Ian R. McDonald, editors, *Simulation of Liquids and Solids*, North-Holland (1987). A collection of reprints on the simulation of many body systems. Of particular interest are B. J. Alder and T. E. Wainwright, "Phase transition in elastic disks," Phys. Rev. **127**, 359–361 (1962) and earlier papers by the same authors; A. Rahman, "Correlations in the motion of atoms in liquid argon," Phys. Rev. **136**, A405–A411 (1964), the first application of molecular dynamics to systems with continuous potentials; and Loup Verlet, "Computer 'experiments' on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules," Phys. Rev. **159**, 98–103 (1967).

Daan Frenkel and Berend Smit, *Understanding Molecular Simulation: From Algorithms to Applications*, second edition, Academic Press (2002). This monograph is one of the best on molecular dynamics and Monte Carlo simulations. It is particularly strong on simulations in various ensembles and on methods for computing free energies.

J. M. Haile, *Molecular Dynamics Simulation*, John Wiley & Sons (1992). A derivation of the mean pressure using periodic boundary conditions is given in Appendix B.

J. P. Hansen, D. Levesque, and J. J. Weis, "Self-diffusion in the two-dimensional, classical electron gas," Phys. Rev. Lett. **43**, 979–982 (1979).

D. Hirchfeld, Y. Radzyner, and D. C. Rapaport, "Molecular dynamics studies of granular flow through an aperture," Phys. Rev. E **56**, 4404–4415 (1997).

W. G. Hoover, *Molecular Dynamics*, Springer-Verlag (1986) and W. G. Hoover, *Computational Statistical Mechanics*, Elsevier (1991).

K. Kadau, T. C. Germann and P. S. Lomdahl, "Large-scale molecular-dynamics simulation of 19 billion particles," Int. J. Mod. Phys. C **15**, 193–201 (2004).

J. Krim, "Friction at macroscopic and microscopic length scales," Am. J. Phys. **70**, 890–897 (2002).

J. Kushick and B. J. Berne, "Molecular dynamics methods: Continuous potentials" in *Statistical Mechanics Part B: Time-Dependent Processes*, Bruce J. Berne, editor, Plenum Press (1977). Also see the article by Jerome J. Erpenbeck and William Wood on "Molecular dynamics techniques for hard-core systems" in the same volume.

Shang-keng Ma, "Calculation of entropy from data of motion," J. Stat. Phys. **26**, 221 (1981). Also see Chapter 25 of Ma's graduate level text, *Statistical Mechanics*, World Scientific (1985). Ma discusses a novel approach for computing the entropy directly from the trajectories. Note that the coincidence rate in Ma's approach is related to the recurrence time for a finite system to return to an arbitrarily small neighborhood of almost any given initial state. The approach is intriguing, but is practical only for small systems.

A. McDonough, S. P. Russo, and I. K. Snook, "Long-time behavior of the velocity autocorrelation function for moderately dense, soft-repulsive, and Lennard-Jones fluids," Phys. Rev. E **63**, 026109-1–9 (2001).

S. Ranganathan, G. S. Dubey, and K. N. Pathak, "Molecular-dynamics study of two-dimensional Lennard-Jones fluids," Phys. Rev. A **45**, 5793–5797 (1992).

Dennis Rapaport, *The Art of Molecular Dynamics Simulation*, second edition, Cambridge University Press (2004). The most complete text on molecular dynamics written by one of its leading practitioners.

John R. Ray and H. W. Graben, "Direct calculation of fluctuation formulae in the microcanonical ensemble," Mol. Phys. **43**, 1293 (1981).

F. Reif, *Fundamentals of Statistical and Thermal Physics*, McGraw-Hill (1965.) An intermediate level text on statistical physics with a more thorough discussion of kinetic theory than found in most undergraduate texts. *Statistical Physics*, Vol. 5 of the Berkeley Physics Course, McGraw-Hill (1965), by Reif was one of the first texts to use computer simulations to illustrate the approach of macroscopic systems to equilibrium.

Marco Ronchetti and Gianni Jacucci, editors, *Simulation Approach to Solids*, Kluwer Academic Publishers (1990). Another excellent collection of classic reprints.

James Ringlein and Mark O. Robbins, "Understanding and illustrating the atomic origins of friction," Am. J. Phys. **72** (7), 884–891 (2004). A very readable paper on the microscopic origins of sliding friction.

Duncan A. Sanders, Michael R. Swift, R. M. Bowley, and P. J. King, "Are Brazil nuts attractive?," Phys. Rev. Lett. **93**, 208002 (2004). An example of a simulation of granular matter.

Tamar Schlick, *Molecular Modeling and Simulation*, Springer-Verlag (2002). Although the book is at the graduate level, it is an accessible introduction to computational molecular biology.

Leonardo E. Silbert, Deniz Ertas, Gary S. Grest, Thomas C. Halsey, Dov Levine, and Steven J. Plimpton, "Granular flow down an inclined plane: Bagnold scaling and rheology," Phys. Rev. E **64**, 051302-1–14 (2001). This paper discusses the contact force model, which captures the major features of granular interactions.

R. M. Sperandeo Mineo and R. Madonia, "The equation of state of a hard-particle system: A model experiment on a microcomputer," Eur. J. Phys. **7**, 124–129 (1986).

D. Thirumalai and Raymond D. Mountain, "Ergodic convergence properties of supercooled liquids and glasses," Phys. Rev. A **42**, 4574–4587 (1990).

Shan-Ho Tsai, H. K. Lee, and D. P. Landau, "Molecular and spin dynamics simulations using modern integration methods," Am. J. Phys., to be published.

James H. Williams and Glenn Joyce, "Equilibrium properties of a one-dimensional kinetic system," J. Chem. Phys. **59**, 741–750 (1973). Simulations in one dimension are even easier than in two.

Zoran Slanič, Harvey Gould, and Jan Tobochnik, "Dynamics of the classical Hesisenberg chain," Computers in Physics **5** (6), 630–635 (1991).