```
   // 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;
   }
   // rescale velocities to get desired initial kinetic energy
   double v2sum = 0;
   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() {
   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 = pbcSeparation(state[4*i]-state[4*j],Lx);
            double dy = pbcSeparation(state[4*i+2]-state[4*j+2],Ly);
            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.

```
// place particles on a rectangular lattice
public void setRectangularLattice() {
   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.