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() {
```