Chapter 4

Molecular Dynamics Simulations

Molecular Dynamics simulation is a technique for computing the equilibrium and transport properties of a classical many-body system. In this context, the word *classical* means that the nuclear motion of the constituent particles obeys the laws of classical mechanics. This is an excellent approximation for a wide range of materials. Only when we consider the translational or rotational motion of light atoms or molecules (He, H_2, D_2) or vibrational motion with a frequency ν such that $h\nu > k_B T$ should we worry about quantum effects.

Of course, our discussion of this vast subject is necessarily incomplete. Other aspects of the Molecular Dynamics techniques can be found in [19, 39–41].

4.1 Molecular Dynamics: The Idea

Molecular Dynamics simulations are in many respects very similar to real experiments. When we perform a real experiment, we proceed as follows. We prepare a sample of the material that we wish to study. We connect this sample to a measuring instrument (e.g., a thermometer, manometer, or viscosimeter), and we measure the property of interest during a certain time interval. If our measurements are subject to statistical noise (as most measurements are), then the longer we average, the more accurate our measurement becomes. In a Molecular Dynamics simulation, we follow exactly the same approach. First, we prepare a sample: we select a model system consisting of N particles and we solve Newton's equations of motion for this system until the properties of the system no longer change with time (we

equilibrate the system). After equilibration, we perform the actual measurement. In fact, some of the most common mistakes that can be made when performing a computer experiment are very similar to the mistakes that can be made in real experiments (e.g., the sample is not prepared correctly, the measurement is too short, the system undergoes an irreversible change during the experiment, or we do not measure what we think).

To measure an observable quantity in a Molecular Dynamics simulation, we must first of all be able to express this observable as a function of the positions and momenta of the particles in the system. For instance, a convenient definition of the temperature in a (classical) many-body system makes use of the equipartition of energy over all degrees of freedom that enter quadratically in the Hamiltonian of the system. In particular for the average kinetic energy per degree of freedom, we have

$$\left\langle \frac{1}{2} m v_\alpha^2 \right\rangle = \frac{1}{2} k_B T. \tag{4.1.1}$$

In a simulation, we use this equation as an operational definition of the temperature. In practice, we would measure the total kinetic energy of the system and divide this by the number of degrees of freedom N_f ($= 3N - 3$ for a system of N particles with fixed total momentum[1]). As the total kinetic energy of a system fluctuates, so does the instantaneous temperature:

$$T(t) = \sum_{i=1}^{N} \frac{m_i v_i^2(t)}{k_B N_f}. \tag{4.1.2}$$

The relative fluctuations in the temperature will be of order $1/\sqrt{N_f}$. As N_f is typically on the order of 10^2–10^3, the statistical fluctuations in the temperature are on the order of 5–10%. To get an accurate estimate of the temperature, one should average over many fluctuations.

4.2 Molecular Dynamics: A Program

The best introduction to Molecular Dynamics simulations is to consider a simple program. The program we consider is kept as simple as possible to illustrate a number of important features of Molecular Dynamics simulations.

The program is constructed as follows:

1. We read in the parameters that specify the conditions of the run (e.g., initial temperature, number of particles, density, time step).

[1] Actually, if we define the temperature of a microcanonical ensemble through $(k_B T)^{-1} = (\partial \ln \Omega / \partial E)$, then we find that, for a d-dimensional system of N atoms with fixed total momentum, $k_B T$ is equal to $2E/[d(N-1)-2]$.

Algorithm 3 (A Simple Molecular Dynamics Program)

```
program md                          simple MD program

call init                           initialization
t=0
do while (t.lt.tmax)                MD loop
   call force(f,en)                 determine the forces
   call integrate(f,en)             integrate equations of motion
   t=t+delt
   call sample                      sample averages
enddo
stop
end
```

Comment to this algorithm:

1. *Subroutines* `init`, `force`, `integrate`, *and* `sample` *will be described in Algorithms 4, 5, and 6, respectively. Subroutine* `sample` *is used to calculate averages like pressure or temperature.*

2. We initialize the system (i.e., we select initial positions and velocities).
3. We compute the forces on all particles.
4. We integrate Newton's equations of motion. This step and the previous one make up the core of the simulation. They are repeated until we have computed the time evolution of the system for the desired length of time.
5. After completion of the central loop, we compute and print the averages of measured quantities, and stop.

Algorithm 3 is a short pseudo-algorithm that carries out a Molecular Dynamics simulation for a simple atomic system. We discuss the different operations in the program in more detail.

4.2.1 Initialization

To start the simulation, we should assign initial positions and velocities to all particles in the system. The particle positions should be chosen compatible with the structure that we are aiming to simulate. In any event, the particles should not be positioned at positions that result in an appreciable overlap of the atomic or molecular cores. Often this is achieved by initially placing

Algorithm 4 (Initialization of a Molecular Dynamics Program)

```
subroutine init                 initialization of MD program
sumv=0
sumv2=0
do i=1,npart
  x(i)=lattice_pos(i)           place the particles on a lattice
  v(i)=(ranf()-0.5)             give random velocities
  sumv=sumv+v(i)                velocity center of mass
  sumv2=sumv2+v(i)**2           kinetic energy
enddo
sumv=sumv/npart                 velocity center of mass
sumv2=sumv2/npart               mean-squared velocity
fs=sqrt(3*temp/sumv2)           scale factor of the velocities
do i=1,npart                    set desired kinetic energy and set
  v(i)=(v(i)-sumv)*fs           velocity center of mass to zero
  xm(i)=x(i)-v(i)*dt            position previous time step
enddo
return
end
```

Comments to this algorithm:

1. *Function* `lattice_pos` *gives the coordinates of lattice position* `i` *and* `ranf()` *gives a uniformly distributed random number. We do not use a Maxwell-Boltzmann distribution for the velocities; on equilibration it will become a Maxwell-Boltzmann distribution.*
2. *In computing the number of degrees of freedom, we assume a three-dimensional system (in fact, we approximate* N_f *by* $3N$*).*

the particles on a cubic lattice, as described in section 3.2.2 in the context of Monte Carlo simulations.

In the present case (Algorithm 4), we have chosen to start our run from a simple cubic lattice. Assume that the values of the density and initial temperature are chosen such that the simple cubic lattice is mechanically unstable and melts rapidly. First, we put each particle on its lattice site and then we attribute to each velocity component of every particle a value that is drawn from a uniform distribution in the interval $[-0.5, 0.5]$. This initial velocity distribution is Maxwellian neither in shape nor even in width. Subsequently, we shift all velocities, such that the total momentum is zero and we scale the resulting velocities to adjust the mean kinetic energy to the desired value. We know that, in thermal equilibrium, the following relation should hold:

$$\langle v_\alpha^2 \rangle = k_B T/m, \tag{4.2.1}$$

where v_α is the α component of the velocity of a given particle. We can use this relation to define an instantaneous temperature at time t $T(t)$:

$$k_B T(t) \equiv \sum_{i=1}^{N} \frac{m v_{\alpha,i}^2(t)}{N_f}. \tag{4.2.2}$$

Clearly, we can adjust the instantaneous temperature $T(t)$ to match the desired temperature T by scaling all velocities with a factor $(T/T(t))^{1/2}$. This initial setting of the temperature is not particularly critical, as the temperature will change anyway during equilibration.

As will appear later, we do not really use the velocities themselves in our algorithm to solve Newton's equations of motion. Rather, we use the positions of all particles at the present (`x`) and previous (`xm`) time steps, combined with our knowledge of the force (`f`) acting on the particles, to predict the positions at the next time step. When we start the simulation, we must bootstrap this procedure by generating approximate previous positions. Without much consideration for any law of mechanics but the conservation of linear momentum, we approximate `x` for a particle in a direction by `xm(i) = x(i) - v(i)*dt`. Of course, we could make a better estimate of the true previous position of each particle. But as we are only bootstrapping the simulation, we do not worry about such subtleties.

4.2.2 The Force Calculation

What comes next is the most time-consuming part of almost all Molecular Dynamics simulations: the calculation of the force acting on every particle. If we consider a model system with pairwise additive interactions (as we do in the present case), we have to consider the contribution to the force on particle `i` due to all its neighbors. If we consider only the interaction between a particle and the nearest image of another particle, this implies that, for a system of N particles, we must evaluate $N \times (N-1)/2$ pair distances.

This implies that, if we use no tricks, the time needed for the evaluation of the forces scales as N^2. There exist efficient techniques to speed up the evaluation of both short-range and long-range forces in such a way that the computing time scales as N, rather than N^2. In Appendix F, we describe some of the more common techniques to speed up the simulations. Although the examples in this Appendix apply to Monte Carlo simulations, the same techniques can also be used in a Molecular Dynamics simulation. However, in the present, simple example (see Algorithm 5) we will not attempt to make

Algorithm 5 (Calculation of the Forces)

```
subroutine force(f,en)                    determine the force
en=0                                      and energy
do i=1,npart
   f(i)=0                                 set forces to zero
enddo
do i=1,npart-1
  do j=i+1,npart                          loop over all pairs
     xr=x(i)-x(j)
     xr=xr-box*nint(xr/box)               periodic boundary conditions
     r2=xr**2
     if (r2.lt.rc2) then                  test cutoff
       r2i=1/r2
       r6i=r2i**3
       ff=48*r2i*r6i*(r6i-0.5)            Lennard-Jones potential
       f(i)=f(i)+ff*xr                    update force
       f(j)=f(j)-ff*xr
       en=en+4*r6i*(r6i-1)-ecut           update energy
     endif
  enddo
enddo
return
end
```

Comments to this algorithm:

1. *For efficiency reasons the factors 4 and 48 are usually taken out of the force loop and taken into account at the end of the calculation for the energy.*
2. *The term* ecut *is the value of the potential at* $r = r_c$*; for the Lennard-Jones potential, we have*

$$\texttt{ecut} = 4\left(\frac{1}{r_c^{12}} - \frac{1}{r_c^6}\right).$$

the program particularly efficient and we shall, in fact, consider all possible pairs of particles explicitly.

We first compute the current distance in the x, y, and z directions between each pair of particles i and j. These distances are indicated by xr. As in the Monte Carlo case, we use periodic boundary conditions (see section 3.2.2). In the present example, we use a cutoff at a distance r_c in the explicit calculation of intermolecular interactions, where r_c is chosen to be less than half the diameter of the periodic box. In that case we can always limit the evaluation

of intermolecular interactions between i and j to the interaction between i and the nearest periodic image of j.

In the present case, the diameter of the periodic box is denoted by box. If we use simple cubic periodic boundary conditions, the distance in any direction between i and the nearest image of j should always be less (in absolute value) than box/2. A compact way to compute the distance between i and the nearest periodic image of j uses the nearest integer function (nint(x) in FORTRAN). The nint function simply rounds a real number to the nearest integer.[2] Starting with the x-distance (say) between i and any periodic image of j, xr, we compute the x-distance between i and the nearest image of j as xr=xr-box*nint(xr/box). Having thus computed all Cartesian components of r_{ij}, the vector distance between i and the nearest image of j, we compute r_{ij}^2 (denoted by r2 in the program). Next we test if r_{ij}^2 is less than r_c^2, the square of the cutoff radius. If not, we immediately skip to the next value of j. It perhaps is worth emphasizing that we do not compute $|r_{ij}|$ itself, because this would be both unnecessary and expensive (as it would involve the evaluation of a square root).

If a given pair of particles is close enough to interact, we must compute the force between these particles, and the contribution to the potential energy. Suppose that we wish to compute the x-component of the force

$$\begin{aligned} f_x(r) &= -\frac{\partial u(r)}{\partial x} \\ &= -\left(\frac{x}{r}\right)\left(\frac{\partial u(r)}{\partial r}\right). \end{aligned}$$

For a Lennard-Jones system (in reduced units),

$$f_x(r) = \frac{48x}{r^2}\left(\frac{1}{r^{12}} - 0.5\frac{1}{r^6}\right).$$

4.2.3 Integrating the Equations of Motion

Now that we have computed all forces between the particles, we can integrate Newton's equations of motion. Algorithms have been designed to do this. Some of these will be discussed in a bit more detail. In the program (Algorithm 6), we have used the so-called Verlet algorithm. This algorithm is not only one of the simplest, but also usually the best.

To derive it, we start with a Taylor expansion of the coordinate of a particle, around time t,

$$r(t+\Delta t) = r(t) + v(t)\Delta t + \frac{f(t)}{2m}\Delta t^2 + \frac{\Delta t^3}{3!}\dddot{r} + \mathcal{O}(\Delta t^4),$$

[2] Unfortunately, many FORTRAN compilers yield very slow nint functions. It is often cheaper to write your own code to replace the nint library routine.

Algorithm 6 (Integrating the Equations of Motion)

```
subroutine integrate(f,en)          integrate equations of motion
sumv=0
sumv2=0
do i=1,npart                        MD loop
   xx=2*x(i)-xm(i)+delt**2*f(i)     Verlet algorithm (4.2.3)
   vi=(xx-xm(i))/(2*delt)           velocity (4.2.4)
   sumv=sumv+vi                     velocity center of mass
   sumv2=sumv2+vi**2                total kinetic energy
   xm(i)=x(i)                       update positions previous time
   x(i)=xx                          update positions current time
enddo
temp=sumv2/(3*npart)                instantaneous temperature
etot=(en+0.5*sumv2)/npart           total energy per particle
return
end
```

Comments to this algorithm:

1. *The total energy* `etot` *should remain approximately constant during the simulation. A drift of this quantity may signal programming errors. It therefore is important to monitor this quantity. Similarly, the velocity of the center of mass* `sumv` *should remain zero.*
2. *In this subroutine we use the Verlet algorithm (4.2.3) to integrate the equations of motion. The velocities are calculated using equation (4.2.4).*

similarly,

$$-r(t-\Delta t) = r(t) - v(t)\Delta t + \frac{f(t)}{2m}\Delta t^2 - \frac{\Delta t^3}{3!}\dddot{r} + \mathcal{O}(\Delta t^4).$$

Summing these two equations, we obtain

$$r(t+\Delta t) + r(t-\Delta t) = 2r(t) + \frac{f(t)}{m}\Delta t^2 + \mathcal{O}(\Delta t^4)$$

or

$$r(t+\Delta t) \approx 2r(t) - r(t-\Delta t) + \frac{f(t)}{m}\Delta t^2. \tag{4.2.3}$$

The estimate of the new position contains an error that is of order Δt^4, where Δt is the time step in our Molecular Dynamics scheme. Note that the

Verlet algorithm does not use the velocity to compute the new position. One, however, can derive the velocity from knowledge of the trajectory, using

$$r(t+\Delta t) - r(t-\Delta t) = 2v(t)\Delta t + \mathcal{O}(\Delta t^3)$$

or

$$v(t) = \frac{r(t+\Delta t) - r(t-\Delta t)}{2\Delta t} + \mathcal{O}(\Delta t^2). \tag{4.2.4}$$

This expression for the velocity is only accurate to order Δt^2. However, it is possible to obtain more accurate estimates of the velocity (and thereby of the kinetic energy) using a Verlet-like algorithm (i.e., an algorithm that yields trajectories identical to that given by equation (4.2.3)). In our program, we use the velocities only to compute the kinetic energy and, thereby, the instantaneous temperature.

Now that we have computed the new positions, we may discard the positions at time $t-\Delta t$. The current positions become the old positions and the new positions become the current positions.

After each time step, we compute the current temperature (`temp`), the current potential energy (`en`) calculated in the force loop, and the total energy (`etot`). Note that the total energy should be conserved.

This completes the introduction to the Molecular Dynamics method. The reader should now be able to write a basic Molecular Dynamics program for liquids or solids consisting of spherical particles. In what follows, we shall do two things. First of all, we discuss, in a bit more detail, the methods available to integrate the equations of motion. Next, we discuss measurements in Molecular Dynamics simulations. Important extensions of the Molecular Dynamics technique are discussed in Chapter 6.

4.3 Equations of Motion

It is obvious that a good Molecular Dynamics program requires a good algorithm to integrate Newton's equations of motion. In this sense, the choice of algorithm is crucial. However, although it is easy to recognize a *bad* algorithm, it is not immediately obvious what criteria a *good* algorithm should satisfy. Let us look at the different points to consider.

Although, at first sight, speed seems important, it is usually not very relevant because the fraction of time spent on integrating the equations of motion (as opposed to computing the interactions) is small, at least for atomic and simple molecular systems.

Accuracy for large time steps is more important, because the longer the time step that we can use, the fewer evaluations of the forces are needed per unit of simulation time. Hence, this would suggest that it is advantageous to use a sophisticated algorithm that allows use of a long time step.

Algorithms that allow the use of a large time step achieve this by storing information on increasingly higher-order derivatives of the particle coordinates. As a consequence, they tend to require more memory storage. For a typical simulation, this usually is not a serious drawback because, unless one considers very large systems, the amount of memory needed to store these derivatives is small compared to the total amount available even on a normal workstation.

Energy conservation is an important criterion, but actually we should distinguish two kinds of energy conservation, namely, short time and long time. The sophisticated higher-order algorithms tend to have very good energy conservation for short times (i.e., during a few time steps). However, they often have the undesirable feature that the overall energy drifts for long times. In contrast, Verlet-style algorithms tend to have only moderate short-term energy conservation but little long-term drift.

It would seem to be most important to have an algorithm that accurately predicts the trajectory of all particles for both short and long times. In fact, no such algorithm exists. For essentially all systems that we study by MD simulations, we are in the regime where the trajectory of the system through phase space (i.e., the 6N-dimensional space spanned by all particle coordinates and momenta) depends sensitively on the initial conditions. This means that two trajectories that are initially very close will diverge exponentially as time progresses. We can consider the integration error caused by the algorithm as the source for the initial small difference between the "true" trajectory of the system and the trajectory generated in our simulation. We should expect that any integration error, no matter how small, will always cause our simulated trajectory to diverge exponentially from the true trajectory compatible with the same initial conditions. This so-called Lyapunov instability (see section 4.3.4) would seem to be a devastating blow to the whole idea of Molecular Dynamics simulations but we have good reasons to assume that even this problem need not be serious.

Clearly, this statement requires some clarification. First of all, one should realize that the aim of an MD simulation is *not* to predict precisely what will happen to a system that has been prepared in a precisely known initial condition: we are always interested in statistical predictions. We wish to predict the average behavior of a system that was prepared in an initial state about which we know something (e.g., the total energy) but by no means everything. In this respect, MD simulations differ fundamentally from numerical schemes for predicting the trajectory of satellites through space: in the latter case, we really wish to predict the true trajectory. We cannot afford to launch an ensemble of satellites and make statistical predictions about their destination. However, in MD simulations, statistical predictions are good enough. Still, this would not justify the use of inaccurate trajectories unless the trajectories obtained numerically, in some sense, are close to true trajectories.

This latter statement is generally believed to be true, although, to our knowledge, it has not been proven for any class of systems that is of interest for MD simulations. However, considerable numerical evidence (see, e.g., [66]) suggests that there exist so-called shadow orbits. A shadow orbit is a true trajectory of a many-body system that closely follows the numerical trajectory for a time that is long compared to the time it takes the Lyapunov instability to develop. Hence, the results of our simulation are representative of a true trajectory in phase space, even though we cannot tell *a priori* which. Surprisingly (and fortunately), it appears that shadow orbits are better behaved (i.e., track the numerical trajectories better) for systems in which small differences in the initial conditions lead to an exponential divergence of trajectories than for the, seemingly, simpler systems that show no such divergence [66]. Despite this reassuring evidence (see also section 4.3.5 and the article by Gillilan and Wilson [67]), it should be emphasized that it is just evidence and not proof. Hence, our trust in Molecular Dynamics simulation as a tool to study the time evolution of many-body systems is based largely on belief. To conclude this discussion, let us say that there is clearly still a corpse in the closet. We believe this corpse will not haunt us, and we quickly close the closet. For more details, the reader is referred to [27,67,68].

Newton's equations of motion are time reversible, and so should be our algorithms. In fact, many algorithms (for instance the predictor-corrector schemes, see Appendix E, and many of the schemes used to deal with constraints) are *not* time reversible. That is, future and past phase space coordinates do not play a symmetric role in such algorithms. As a consequence, if one were to reverse the momenta of all particles at a given instant, the system would not trace back its trajectory in phase space, even if the simulation would be carried out with infinite numerical precision. Only in the limit of an infinitely short time step will such algorithms become reversible. However, what is more important, many seemingly reasonable algorithms differ in another crucial respect from Hamilton's equation of motion: true Hamiltonian dynamics leaves the magnitude of any volume element in phase space unchanged, but many numerical schemes, in particular those that are not time reversible, do not reproduce this area-preserving property. This may sound like a very esoteric objection to an algorithm, but it is not. Again, without attempting to achieve a rigorous formulation of the problem, let us simply note that all trajectories that correspond to a particular energy E are contained in a (hyper) volume Ω in phase space. If we let Hamilton's equation of motion act on all points in this volume (i.e., we let the volume evolve in time), then we end up with exactly the same volume. However, a non-area-preserving algorithm will map the volume Ω on another (usually larger) volume Ω'. After sufficiently long times, we expect that the non-area-preserving algorithm will have greatly expanded the volume of our system in phase space. This is not compatible with energy conservation. Hence, it is plausible that nonreversible algorithms will have serious long-term energy drift problems. Reversible, area-preserving algo-

rithms will not change the magnitude of the volume in phase space. This property is not sufficient to guarantee the absence of long-term energy drift, but it is at least compatible with it. It is possible to check whether an algorithm is area preserving by computing the Jacobian associated with the transformation of old to new phase space coordinates.

Finally, it should be noted that even when we integrate a time-reversible algorithm, we shall find that the numerical implementation is hardly ever truly time reversible. This is so, because we work on a computer with finite machine precision using floating-point arithmetic that results in rounding errors (on the order of the machine precision).

In the remainder of this section, we shall discuss some of these points in more detail. Before we do so, let us first consider how the Verlet algorithm scores on these points. First of all, the Verlet algorithm is fast. But we had argued that this is relatively unimportant. Second, it is not particularly accurate for long time steps. Hence, we should expect to compute the forces on all particles rather frequently. Third, it requires about as little memory as is at all possible. This is useful when we simulate very large systems, but in general it is not a crucial advantage. Fourth, its short-term energy conservation is fair (in particular in the versions that use a more accurate expression for the velocities) but, more important, it exhibits little long-term energy drift. This is related to the fact that the Verlet algorithm is time reversible and area preserving. In fact, although the Verlet algorithm does not conserve the total energy of this system exactly, strong evidence indicates that it does conserve a pseudo-Hamiltonian approaching the true Hamiltonian in the limit of infinitely short time steps (see section 4.3.3). The accuracy of the trajectories generated with the Verlet algorithm is not impressive. But then, it would hardly help to use a better algorithm. Such an algorithm may postpone the unavoidable exponential growth of the error in the trajectory by a few hundred time steps (see section 4.3.4), but no algorithm is good enough that it will keep the trajectories close to the true trajectories for a time comparable to the duration of a typical Molecular Dynamics run.[3]

4.3.1 Other Algorithms

Let us now briefly look at some alternatives to the Verlet algorithm. The most naive algorithm is based simply on a truncated Taylor expansion of the particle coordinates:

$$r(t+\Delta t) = r(t) + v(t)\Delta t + \frac{f(t)}{2m}\Delta t^2 + \cdots .$$

[3] Error-free integration of the equations of motion is possible for certain discrete models, such as lattice-gas cellular automata. But these models do not follow Newton's equation of motion.

If we truncate this expansion beyond the term in Δt^2, we obtain the so-called Euler algorithm. Although it looks similar to the Verlet algorithm, it is much worse on virtually all counts. In particular, it is not reversible or area preserving and suffers from a (catastrophic) energy drift. The Euler algorithm therefore is not recommended.

Several algorithms are equivalent to the Verlet scheme. The simplest among these is the so-called Leap Frog algorithm [24]. This algorithm evaluates the velocities at half-integer time steps and uses these velocities to compute the new positions. To derive the Leap Frog algorithm from the Verlet scheme, we start by defining the velocities at half-integer time steps as follows:

$$v(t-\Delta t/2) \equiv \frac{r(t) - r(t-\Delta t)}{\Delta t}$$

and

$$v(t+\Delta t/2) \equiv \frac{r(t+\Delta t) - r(t)}{\Delta t}.$$

From the latter equation we immediately obtain an expression for the new positions, based on the old positions and velocities:

$$r(t+\Delta t) = r(t) + \Delta t v(t+\Delta t/2). \tag{4.3.1}$$

From the Verlet algorithm, we get the following expression for the update of the velocities:

$$v(t+\Delta t/2) = v(t-\Delta t/2) + \Delta t \frac{f(t)}{m}. \tag{4.3.2}$$

As the Leap Frog algorithm is derived from the Verlet algorithm, it gives rise to identical trajectories. Note, however, that the velocities are not defined at the same time as the positions. As a consequence, kinetic and potential energy are also not defined at the same time, and hence we cannot directly compute the total energy in the Leap Frog scheme.

It is, however, possible to cast the Verlet algorithm in a form that uses positions and velocities computed at equal times. This velocity Verlet algorithm [69] looks like a Taylor expansion for the coordinates:

$$r(t+\Delta t) = r(t) + v(t)\Delta t + \frac{f(t)}{2m}\Delta t^2. \tag{4.3.3}$$

However, the update of the velocities is different from the Euler scheme:

$$v(t+\Delta t) = v(t) + \frac{f(t+\Delta t) + f(t)}{2m}\Delta t. \tag{4.3.4}$$

Note that, in this algorithm, we can compute the new velocities only after we have computed the new positions and, from these, the new forces. It is not

immediately obvious that this scheme, indeed, is equivalent to the original Verlet algorithm. To show this, we note that

$$r(t+2\Delta t) = r(t+\Delta t) + v(t+\Delta t)\Delta t + \frac{f(t+\Delta t)}{2m}\Delta t^2$$

and equation (4.3.3) can be written as

$$r(t) = r(t+\Delta t) - v(t)\Delta t - \frac{f(t)}{2m}\Delta t^2.$$

By addition we get

$$r(t+2\Delta t) + r(t) = 2r(t+\Delta t) + [v(t+\Delta t) - v(t)]\Delta t + \frac{f(t+\Delta t) - f(t)}{2m}\Delta t^2.$$

Substitution of equation (4.3.4) yields

$$r(t+2\Delta t) + r(t) = 2r(t+\Delta t) + \frac{f(t+\Delta t)}{m}\Delta t^2,$$

which, indeed, is the coordinate version of the Verlet algorithm.

Let us end the discussion of Verlet-like algorithms by mentioning two schemes that yield the same trajectories as the Verlet algorithm, but provide better estimates of the velocity. The first is the so-called Beeman algorithm. It looks quite different from the Verlet algorithm:

$$r(t+\Delta t) = r(t) + v(t)\Delta t + \frac{4f(t) - f(t-\Delta t)}{6m}\Delta t^2 \qquad (4.3.5)$$

$$v(t+\Delta t) = v(t) + \frac{2f(t+\Delta t) + 5f(t) - f(t-\Delta t)}{6m}\Delta t. \qquad (4.3.6)$$

However, by eliminating $v(t)$ from equation (4.3.5), using equation (4.3.6), it is easy to show that the positions satisfy the Verlet algorithm. However, the velocities are more accurate than in the original Verlet algorithm. As a consequence, the total energy conservation looks somewhat better. A disadvantage of the Beeman algorithm is that the expression for the velocities does not have time-reversal symmetry. A very simple solution to this problem is to use the so-called velocity-corrected Verlet algorithm for which the error both in the positions and in the velocities is of order $\mathcal{O}(\Delta t^4)$.

The velocity-corrected Verlet algorithm is derived as follows. First write down a Taylor expansion for $r(t+2\Delta t)$, $r(t+\Delta t)$, $r(t-\Delta t)$ and $r(t-2\Delta t)$:

$$\begin{aligned}
r(t+2\Delta t) &= r(t) + 2v(t)\Delta t + \dot{v}(t)(2\Delta t)^2/2! + \ddot{v}(2\Delta t)^3/3! + \cdots \\
r(t+\Delta t) &= r(t) + v(t)\Delta t + \dot{v}(t)\Delta t^2/2! + \ddot{v}\Delta t^3/3! + \cdots \\
r(t-\Delta t) &= r(t) - v(t)\Delta t + \dot{v}(t)\Delta t^2/2! - \ddot{v}\Delta t^3/3! + \cdots \\
r(t-2\Delta t) &= r(t) - 2v(t)\Delta t + \dot{v}(t)(2\Delta t)^2/2! - \ddot{v}(2\Delta t)^3/3! + \cdots .
\end{aligned}$$

By combining these equations, we can write

$$12v(t)\Delta t = 8[r(t+\Delta t) - r(t-\Delta t)] - [r(t+2\Delta t) - r(t-2\Delta t)] + \mathcal{O}(\Delta t^4)$$

or, equivalently,

$$v(t) = \frac{v(t+\Delta t/2) + v(t-\Delta t/2)}{2} + \frac{\Delta t}{12}[\dot{v}(t-\Delta t) - \dot{v}(t+\Delta t)] + \mathcal{O}(\Delta t^4). \qquad (4.3.7)$$

Note that this velocity can be computed only after the next time step (i.e., we must know the positions and forces at $t+\Delta t$ to compute $v(t)$).

4.3.2 Higher-Order Schemes

For most Molecular Dynamics applications, Verlet-like algorithms are perfectly adequate. However, sometimes it is convenient to employ a higher-order algorithm (i.e., an algorithm that employs information about higher-order derivatives of the particle coordinates). Such an algorithm makes it possible to use a longer time step without loss of (short-term) accuracy or, alternatively, to achieve higher accuracy for a given time step. But, as mentioned before, higher-order algorithms require more storage and are, more often than not, neither reversible nor area preserving. This is true in particular of the so-called predictor-corrector algorithms, the most popular class of higher-order algorithms used in Molecular Dynamics simulations. For the sake of completeness, the predictor-corrector scheme is described in Appendix E.1. We refer the reader who wishes to know more about the relative merits of algorithms for Molecular Dynamics simulations to the excellent review by Berendsen and van Gunsteren [70].

4.3.3 Liouville Formulation of Time-Reversible Algorithms

Thus far we have considered algorithms for integrating Newton's equations of motion from the point of view of applied mathematics. However, recently Tuckerman *et al.* [71] have shown how to systematically derive time-reversible, area-preserving MD algorithms from the Liouville formulation of classical mechanics. The same approach has been developed independently by Sexton and Weingarten [72] in the context of hybrid Monte Carlo simulations (see section 14.2). As the Liouville formulation provides considerable insight into what makes an algorithm a good algorithm, we briefly review the Liouville approach.

Let us consider an arbitrary function f that depends on all the coordinates and momenta of the N particles in a classical many-body system. The term $f(\mathbf{p}^N(t), \mathbf{r}^N(t))$ depends on the time t implicitly, that is, through the

dependence of $(\mathbf{p}^N, \mathbf{r}^N)$ on t. The time derivative of f is $\dot{f}$:

$$\begin{aligned} \dot{f} &= \dot{\mathbf{r}}\frac{\partial f}{\partial \mathbf{r}} + \dot{\mathbf{p}}\frac{\partial f}{\partial \mathbf{p}} \\ &\equiv iLf, \end{aligned} \tag{4.3.8}$$

where we have used the shorthand notation $\mathbf{r}$ for $\mathbf{r}^N$ and $\mathbf{p}$ for $\mathbf{p}^N$. The last line of equation (4.3.8) defines the *Liouville operator*

$$iL = \dot{\mathbf{r}}\frac{\partial}{\partial \mathbf{r}} + \dot{\mathbf{p}}\frac{\partial}{\partial \mathbf{p}}. \tag{4.3.9}$$

We can formally integrate equation (4.3.8) to obtain

$$f\left[\mathbf{p}^N(t), \mathbf{r}^N(t)\right] = \exp(iLt) f\left[\mathbf{p}^N(0), \mathbf{r}^N(0)\right]. \tag{4.3.10}$$

In all cases of practical interest, we cannot do much with this formal solution, because evaluating the right-hand side is still equivalent to the exact integration of the classical equations of motion. However, in a few simple cases the formal solution is known explicitly. In particular, suppose that our Liouville operator contained only the first term on the right-hand side of equation (4.3.9). We denote this part of iL by iL_r:

$$iL_r \equiv \dot{\mathbf{r}}(0)\frac{\partial}{\partial \mathbf{r}}, \tag{4.3.11}$$

where $\dot{\mathbf{r}}(0)$ is the value of $\dot{\mathbf{r}}$ at time $t = 0$. If we insert iL_r in equation (4.3.10) and use a Taylor expansion of the exponential on the right-hand side, we get

$$\begin{aligned} f(t) &= f(0) + iL_r t f(0) + \frac{(iL_r t)^2}{2!} f(0) + \cdots \\ &= \exp\left(\dot{\mathbf{r}}(0) t \frac{\partial}{\partial \mathbf{r}}\right) f(0) \\ &= \sum_{n=0}^{\infty} \frac{(\dot{\mathbf{r}}(0)t)^n}{n!} \frac{\partial^n}{\partial \mathbf{r}^n} f(0) \\ &= f\left[\mathbf{p}^N(0), (\mathbf{r} + \dot{\mathbf{r}}(0)t)^N\right]. \end{aligned} \tag{4.3.12}$$

Hence, the effect of $\exp(iL_r t)$ is a simple shift of coordinates. Similarly, the effect of $\exp(iL_p t)$, with iL_p defined as

$$iL_p \equiv \dot{\mathbf{p}}(0)\frac{\partial}{\partial \mathbf{p}}, \tag{4.3.13}$$

is a simple shift of momenta. The total Liouville operator, iL, is equal to $iL_r + iL_p$. Unfortunately, we cannot replace $\exp(iLt)$ by $\exp(iL_r t) \times \exp(iL_p t)$,

because iL_r and iL_p are noncommuting operators. For noncommuting operators A and B, we have

$$\exp(A + B) \neq \exp(A)\exp(B). \tag{4.3.14}$$

However, we do have the following *Trotter identity*:

$$e^{(A+B)} = \lim_{P\to\infty}\left(e^{A/2P} e^{B/P} e^{A/2P}\right)^P. \tag{4.3.15}$$

In the limit $P \to \infty$, this relation is formally correct, but of limited practical value. However, for large but finite P, we have

$$e^{(A+B)} = \left(e^{A/2P} e^{B/P} e^{A/2P}\right)^P e^{\mathcal{O}(1/P^2)}. \tag{4.3.16}$$

Now let us apply this expression to the formal solution of the Liouville equation. To this end, we make the identification

$$\frac{A}{P} \equiv \frac{iL_p t}{P} \equiv \Delta t \dot{\mathbf{p}}(0)\frac{\partial}{\partial \mathbf{p}}$$

and

$$\frac{B}{P} \equiv \frac{iL_r t}{P} \equiv \Delta t \dot{\mathbf{r}}(0)\frac{\partial}{\partial \mathbf{r}},$$

where $\Delta t = t/P$. The idea is now to replace the formal solution of the Liouville equation by the discretized version, equation (4.3.16). In this scheme, one time step corresponds to applying the operator

$$e^{iL_p \Delta t/2} e^{iL_r \Delta t} e^{iL_p \Delta t/2}$$

once. Let us see what the effect is of this operator on the coordinates and momenta of the particles. First, we apply $\exp(iL_p \Delta t/2)$ to f and obtain

$$e^{iL_p \Delta t/2} f\left[\mathbf{p}^N(0), \mathbf{r}^N(0)\right] = f\left\{\left[\mathbf{p}(0) + \frac{\Delta t}{2}\dot{\mathbf{p}}(0)\right]^N, \mathbf{r}^N(0)\right\}.$$

Next, we apply $\exp(iL_r \Delta t)$ to the result of the previous step

$$\begin{aligned} &e^{iL_r \Delta t} f\left\{\left[\mathbf{p}(0) + \frac{\Delta t}{2}\dot{\mathbf{p}}(0)\right]^N, \mathbf{r}^N(0)\right\} \\ &\quad = f\left\{\left[\mathbf{p}(0) + \frac{\Delta t}{2}\dot{\mathbf{p}}(0)\right]^N, [\mathbf{r}(0) + \Delta t \dot{\mathbf{r}}(\Delta t/2)]^N\right\}, \end{aligned}$$

and finally we apply $\exp(iL_p\Delta t/2)$ once more, to obtain

$$f\left\{\left[p(0)+\frac{\Delta t}{2}\dot{p}(0)+\frac{\Delta t}{2}\dot{p}(\Delta t)\right]^N,[r(0)+\Delta t\dot{r}(\Delta t/2)]^N\right\}.$$

Note that every step in the preceding sequence corresponds to a simple shift operation in either $\mathbf{r}^N$ or $\mathbf{p}^N$. It is of particular importance to note that the shift in $\mathbf{r}$ is a function of $\mathbf{p}$ only (because $\dot{\mathbf{r}} = \mathbf{p}/m$), while the shift in $\mathbf{p}$ is a function of $\mathbf{r}$ only (because $\dot{\mathbf{p}} = \mathbf{F}(\mathbf{r}^N)$). The Jacobian of the transformation from $\{\mathbf{p}^N(0),\mathbf{r}^N(0)\}$ to $\{\mathbf{p}^N(\Delta t),\mathbf{r}^N(\Delta t)\}$ is simply the product of the Jacobians of the three elementary transformations. But, as each of these Jacobians is equal to 1, the overall Jacobian is also equal to 1. In other words, the algorithm is area preserving.

If we now consider the overall effect of this sequence of operations on the positions and momenta, we find the following:

$$\mathbf{p}(0) \rightarrow \mathbf{p}(0)+\frac{\Delta t}{2}(\mathbf{F}(0)+\mathbf{F}(\Delta t)) \tag{4.3.17}$$

$$\begin{aligned}\mathbf{r}(0) &\rightarrow \mathbf{r}(0)+\Delta t\dot{\mathbf{r}}(\Delta t/2)\\ &= \mathbf{r}(0)+\Delta t\dot{\mathbf{r}}(0)+\frac{\Delta^2 t}{2m}\mathbf{F}(0).\end{aligned} \tag{4.3.18}$$

But these are precisely the equations of the Verlet algorithm (in the velocity form). Hence, we have shown that the Verlet algorithm is area preserving. That it is reversible follows directly from the fact that past and future coordinates enter symmetrically in the algorithm.

Finally, let us try to understand the absence of long-term energy drift in the Verlet algorithm. When we use the Verlet algorithm, we replace the true Liouville operator $\exp(iLt)$ by $\exp(iL_r\Delta t/2)\exp(iL_p\Delta t)\exp(iL_r\Delta t/2)$. In doing so, we make an error. If all (nth-order) commutators of L_p and L_r exist (i.e., if the Hamiltonian is an infinitely differentiable function of $\mathbf{p}^N$ and $\mathbf{r}^N$) then, at least in principle, we can evaluate the error that is involved in this replacement:

$$\exp(iL_r\Delta t/2)\exp(iL_p\Delta t)\exp(iL_r\Delta t/2)=\exp(iL\Delta t+\epsilon), \tag{4.3.19}$$

where ϵ is an operator that can be expressed in terms of the commutators of L_p and L_r:

$$\epsilon=\sum_{n=1}^{\infty}(\Delta t)^{2n+1}c_{2n+1}, \tag{4.3.20}$$

where c_m denotes a combination of mth-order commutators. For instance, the leading term is

$$-(\Delta t)^3\left(\frac{1}{24}[iL_r,[iL_r,iL_p]]+\frac{1}{12}[iL_p,[iL_r,iL_p]]\right).$$

Now the interesting thing to note is that, if the expansion in equation (4.3.20) converges, then we can define a pseudo-Liouville operator

$$iL_{pseudo}\equiv iL+\epsilon/\Delta t.$$

This pseudo-Liouville operator corresponds to a pseudo-Hamiltonian, and the remarkable thing is that this pseudo-Hamiltonian (H_{pseudo}) is rigorously conserved by Verlet style (or generalized multi-time-step) algorithms [73–75]. The difference between the conserved pseudo-Hamiltonian and the true Hamiltonian of the system is of order $(\Delta t)^{2n}$ (where n depends on the order of the algorithm). Clearly, by choosing Δt small (and, if necessary, n large), we can make the difference between the true and the pseudo-Hamiltonian as small as we like. As the true Hamiltonian is forced to remain close to a conserved quantity, we can now understand why there is no long-term drift in the energy with Verlet-style algorithms. In some cases, we can explicitly compute the commutators (for instance, for a harmonic system) and can verify that the pseudo-Hamiltonian is indeed conserved [68]. And, even if we cannot compute the complete series of commutators, the leading term will give us an improved estimate of the pseudo-Hamiltonian. Toxvaerd [68] has verified that even for a realistic many-body system, such an approximate pseudo-Hamiltonian is very nearly a constant of motion.

The Liouville formalism allows us to derive the Verlet algorithm as a special case of the Trotter expansion of the time-evolution operator. It should be realized that the decomposition of iL as a sum of iL_r and iL_p is arbitrary. Other decompositions are possible and may lead to algorithms that are more convenient.

4.3.4 Lyapunov Instability

To end this discussion of algorithms, we wish to illustrate the extreme sensitivity of the trajectories to small differences in initial conditions. Let us consider the position ($\mathbf{r}^N$) of one of the N particles at time t. This position is a function of the initial positions and momenta at $t = 0$:

$$\mathbf{r}(t)=f\left[\mathbf{r}^N(0),\mathbf{p}^N(0);t\right].$$

Let us now consider the position at time t that would result if we perturbed the initial conditions (say, some of the momenta) by a small amount ϵ. In that case, we would obtain a different value for r at time t:

$$\mathbf{r}'(t)=f\left[\mathbf{r}^N(0),\mathbf{p}^N(0)+\epsilon;t\right].$$

We denote the difference between $\mathbf{r}(t)$ and $\mathbf{r}'(t)$ by $\Delta\mathbf{r}(t)$. For sufficiently short times, $\Delta\mathbf{r}(t)$ is linear in ϵ. However, the coefficient of the linear dependence diverges exponentially; that is,

$$|\Delta\mathbf{r}(t)|\sim\epsilon\exp(\lambda t). \tag{4.3.21}$$

This so-called Lyapunov instability of the trajectories is responsible for our inability to accurately predict a trajectory for all but the shortest simulations. The exponent λ is called the Lyapunov exponent (more precisely, the largest Lyapunov exponent; there are more such exponents, 6N in fact, but the largest dominates the long-time exponential divergence of initially close trajectories). Suppose that we wish to maintain a certain bound Δ_{max} on $|\Delta \mathbf{r}(t)|$, in the interval $0 < t < t_{max}$. How large an initial error (ϵ) can we afford? From equation (4.3.21), we deduce

$$\epsilon \sim \Delta_{max} \exp(-\lambda t_{max}).$$

Hence, the acceptable error in our initial conditions decreases exponentially with t_{max}, the length of the run. To illustrate that this effect is real, we show the result of two almost identical simulations: the second differs from the first in that the x components of the velocities of 2 particles (out of 1000) have been changed by $+10^{-10}$ and -10^{-10} (in reduced units). We monitor the sum of the squares of the differences of the positions of all particles:

$$\sum_{i=1}^{N} |\mathbf{r}_i(t) - \mathbf{r}'_i(t)|^2 .$$

As can be seen in Figure 4.1, this measure of the distance does indeed grow exponentially with time.

After 1000 time steps, the two systems that were initially very close have become very nearly uncorrelated. It should be stressed that this run was performed using perfectly normal parameters (density, temperature, time step). The only unrealistic thing about this simulation is that it is extremely short. Most Molecular Dynamics simulations do require many thousands of time steps.

4.3.5 One More Way to Look at the Verlet Algorithm...

In Molecular Dynamics simulations, the Newtonian equations of motion are integrated approximately. An alternative route would be to *first* write down a time-discretized version of the action. (See Appendix A, and *then* find the set of coordinates (i.e., the discretized trajectory) that minimizes this action. This approach is discussed in some detail in a paper by Gillilan and Wilson [67].) Let us start with the continuous-time version of the action

$$S = \int_{t_b}^{t_e} dt \left[\frac{1}{2} m \left(\frac{dx(t)}{dt} \right)^2 - U(x) \right]$$

and discretize it as follows:

$$S_{discr} = \sum_{i=i_b}^{i_e-1} \Delta t \left[\frac{1}{2} m \left(\frac{x_{i+1} - x_i}{\Delta t} \right)^2 - U(x_i) \right],$$

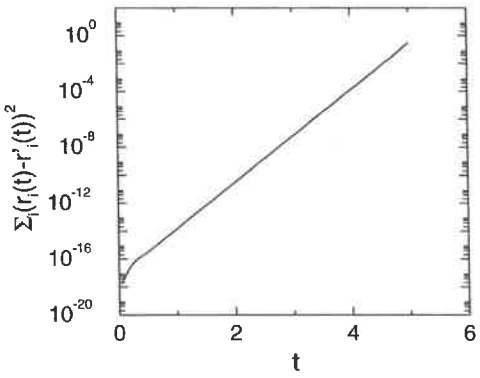

Figure 4.1: Illustration of the Lyapunov instability in a simulation of a Lennard-Jones system. The figure shows the time dependence of the sum of squared distances between two trajectories that were initially very close (see text). The total length of the run in reduced units was 5, which corresponds to 1000 time steps. Note that, within this relatively short time, the two trajectories become essentially uncorrelated.

where $t_b = i_b \Delta t$ and $t_e = i_e \Delta t$. As in the continuous case, we can determine the set of values of the coordinates x_i for which S_{discr} is stationary. At stationarity, the derivative of S_{discr} with respect to all x_i vanishes. It is easy to verify that this implies that

$$m \left(\frac{2x_i - x_{i+1} - x_{i-1}}{\Delta t} \right) - \Delta t \frac{\partial U(x_i)}{\partial x_i} = 0$$

or

$$x_{i+1} = 2x_i - x_{i-1} - \frac{\Delta t^2}{m} \left(\frac{\partial U(x_i)}{\partial x_i} \right),$$

which is, of course, the Verlet algorithm. This illustrates that the trajectories generated by the Verlet algorithms have an interesting "shadow" property (see ref. [67] and section 4.3): a "Verlet trajectory" that connects point x_{i_b} and x_{i_e} in a time interval $t_e - t_b$ will tend to lie close to the true trajectory that connects these two points. However, this true trajectory is not at all the one that has the same initial velocity as the Verlet trajectory. That is,

$$\left(\frac{dx(t_b)}{dt} \right)_{true} \neq \left(\frac{x_{i_b+1} - x_{i_b-1}}{2\Delta t} \right)_{Verlet}.$$

Nevertheless, as discussed in section 4.3, the Verlet algorithm is a good algorithm in the sense that it follows from a minimization principle that forces it

to approximate a true dynamical trajectory of the system under consideration.

This attractive feature of algorithms that can be derived from a discretized action has inspired Elber and co-workers to construct a novel class of MD algorithms that are designed to yield reasonable long-time dynamics with very large time steps [76,77]. In fact, Elber and co-workers do not base their approach on the discretization of the classical action but on the so-called Onsager-Machlup action [78]. The reason for selecting this more general action is that the Onsager-Machlup action is a *minimum* for the true trajectory, while the Lagrangian action is only an extremum. It would carry too far to discuss the practical implementation of the algorithm based on the Onsager-Machlup action. For details, we refer the reader to refs. [76,77].

4.4 Computer Experiments

Now that we have a working Molecular Dynamics program, we wish to use it to "measure" interesting properties of many-body systems. What properties are interesting? First of all, of course, those quantities that can be compared with real experiments. Simplest among these are the thermodynamic properties of the system under consideration, such as the temperature T, the pressure P, and the heat capacity C_V. As mentioned earlier, the temperature is measured by computing the average kinetic energy per degree of freedom. For a system with f degrees of freedom, the temperature T is given by

$$k_B T = \frac{\langle 2K \rangle}{f}. \tag{4.4.1}$$

There are several different (but equivalent) ways to measure the pressure of a classical N-body system. The most common among these is based on the virial equation for the pressure. For pairwise additive interactions, we can write (see, e.g., [79])

$$P = \rho k_B T + \frac{1}{dV} \left\langle \sum_{i<j} \mathbf{f}(r_{ij}) \cdot \mathbf{r}_{ij} \right\rangle, \tag{4.4.2}$$

where d is the dimensionality of the system, and $\mathbf{f}(\mathbf{r}_{ij})$ is the force between particles i and j at a distance r_{ij}. Note that this expression for the pressure has been derived for a system at constant N, V, and T, whereas our simulations are performed at constant N, V, and E. In fact, the expression for the pressure in the microcanonical ensemble (constant N, V, E) is not identical to the expression that applies to the canonical (constant N, V, T) ensemble. Lebowitz *et al.* [80] have derived a general procedure to convert averages from one ensemble to another. A more recent (and more accessible) description of these interensemble transformations has been given by Allen and Tildesley [41]. An example of a relation derived by such a transformation is the expression for the heat capacity at constant volume, as obtained from the fluctuations in the kinetic energy in the microcanonical ensemble:

$$\langle K^2 \rangle_{NVE} - \langle K \rangle^2_{NVE} = \frac{3k_B^2 T^2}{2N} \left(1 - \frac{3k_B}{2C_V} \right). \tag{4.4.3}$$

However, one class of thermodynamic functions cannot be measured directly in a simulation, in the sense that these properties cannot be expressed as a simple average of some function of the coordinates and momenta of all the particles in the system. Examples of such properties are the entropy S, the Helmholtz free energy F, and the Gibbs free energy G. Separate techniques are required to evaluate such thermal quantities in a computer simulation. Methods to calculate these properties are discussed in Chapter 7.

A second class of observable properties are the functions that characterize the local structure of a fluid. Most notable among these is the so-called radial distribution function $g(r)$. The radial distribution function is of interest for two reasons: first of all, neutron and X-ray scattering experiments on simple fluids, and light-scattering experiments on colloidal suspensions, yield information about $g(r)$. Second, $g(r)$ plays a central role in theories of the liquid state. Numerical results for $g(r)$ can be compared with theoretical predictions and thus serve as a criterion to test a particular theory. In a simulation, it is straightforward to measure $g(r)$: it is simply the ratio between the average number density $\rho(r)$ at a distance r from any given atom (for simplicity we assume that all atoms are identical) and the density at a distance r from an atom in an ideal gas at the same overall density. In Algorithm 7 an implementation to compute the radial distribution function is described. By construction, $g(r) = 1$ in an ideal gas. Any deviation of $g(r)$ from unity reflects correlations between the particles due to the intermolecular interactions.

Both the thermodynamic properties and the structural properties mentioned previously do not depend on the time evolution of the system: they are static equilibrium averages. Such averages can be obtained by Molecular Dynamics simulations or equally well by Monte Carlo simulations. However, in addition to the static equilibrium properties, we can also measure dynamic equilibrium properties in a Molecular Dynamics simulation (but not with a Monte Carlo simulation). At first sight, a dynamic equilibrium property appears to be a contradiction: in equilibrium all properties are independent of time; hence any time dependence in the macroscopic properties of a system seems to be related to nonequilibrium behavior. This is true, but it turns out that the time-dependent behavior of a system that is only weakly perturbed is completely described by the dynamic equilibrium properties of the system. Later, we shall provide a simple introduction to the quantities that play a central role in the theory of time-dependent

Algorithm 7 (The Radial Distribution Function)

```
subroutine gr(switch)                  radial distribution function
                                       switch = 0 initialization,
                                       = 1 sample, and = 2 results
if (switch.eq.0) then                  initialization
  ngr=0
  delg=box/(2*nhis)                    bin size
  do i=0,nhis                          nhis total number of bins
     g(i)=0
  enddo
else if (switch.eq.1) then             sample
  ngr=ngr+1
  do i=1,npart-1
    do j=i+1,npart                     loop over all pairs
      xr=x(i)-x(j)
      xr=xr-box*nint(xr/box)           periodic boundary conditions
      r=sqrt(xr**2)
      if (r.lt.box/2) then             only within half the box length
         ig=int(r/delg)
         g(ig)=g(ig)+2                 contribution for particle i and j
      endif
    enddo
  enddo
else if (switch.eq.2) then             determine g(r)
  do i=1,nhis
    r=delg*(i+0.5)                     distance r
    vb=((i+1)**3-i**3)*delg**3         volume between bin i+1 and i
    nid=(4/3)*pi*vb*rho                number of ideal gas part. in vb
    g(i)=g(i)/(ngr*npart*nid)          normalize g(r)
  enddo
endif
return
end
```

Comments to this algorithm:

1. *For efficiency reasons the sampling part of this algorithm is usually combined with the force calculation (for example, Algorithm 5).*
2. *The factor* `pi` *= 3.14159....*

processes near equilibrium, in particular the so-called time-correlation functions. However, we shall not start with a general description of nonequilibrium processes. Rather we start with a discussion of a simple specific example that allows us to introduce most of the necessary concepts.

4.4.1 Diffusion

Diffusion is the process whereby an initially nonuniform concentration profile (e.g., an ink drop in water) is smoothed in the absence of flow (no stirring). Diffusion is caused by the molecular motion of the particles in the fluid. The macroscopic law that describes diffusion is known as Fick's law, which states that the flux $\mathbf{j}$ of the diffusing species is proportional to the negative gradient in the concentration of that species:

$$\mathbf{j} = -D\nabla c, \tag{4.4.4}$$

where D, the constant of proportionality, is referred to as the *diffusion coefficient*. In what follows, we shall be discussing a particularly simple form of diffusion, namely, the case where the molecules of the diffusing species are identical to the other molecules but for a label that does not affect the interaction of the labeled molecules with the others. For instance, this label could be a particular polarization of the nuclear spin of the diffusing species or a modified isotopic composition. Diffusion of a labeled species among otherwise identical solvent molecules is called *self-diffusion*.

Let us now compute the concentration profile of the tagged species, under the assumption that, at time $t = 0$, the tagged species was concentrated at the origin of our coordinate frame. To compute the time evolution of the concentration profile, we must combine Fick's law with an equation that expresses conservation of the total amount of labeled material:

$$\frac{\partial c(r,t)}{\partial t} + \nabla \cdot \mathbf{j}(r,t) = 0. \tag{4.4.5}$$

Combining equation (4.4.5) with equation (4.4.4), we obtain

$$\frac{\partial c(r,t)}{\partial t} - D\nabla^2 c(r,t) = 0. \tag{4.4.6}$$

We can solve equation (4.4.6) with the boundary condition

$$c(r,0) = \delta(r)$$

($\delta(r)$ is the Dirac delta function) to yield

$$c(r,t) = \frac{1}{(4\pi Dt)^{d/2}} \exp\left(-\frac{r^2}{4Dt}\right).$$

As before, d denotes the dimensionality of the system. In fact, for what follows we do not need $c(r,t)$ itself, but just the time dependence of its second moment:

$$\langle r^2(t)\rangle \equiv \int \mathbf{dr}\, c(r,t) r^2,$$

where we have used the fact that we have imposed

$$\int \mathbf{dr}\, c(r,t) = 1.$$

We can directly obtain an equation for the time evolution of $\langle r^2(t)\rangle$ by multiplying equation (4.4.6) by r^2 and integrating over all space. This yields

$$\frac{\partial}{\partial t}\int \mathbf{dr}\, r^2 c(r,t) = D\int \mathbf{dr}\, r^2 \nabla^2 c(r,t). \tag{4.4.7}$$

The left-hand side of this equation is simply equal to

$$\frac{\partial \langle r^2(t)\rangle}{\partial t}.$$

Applying partial integration to the right-hand side, we obtain

$$\begin{aligned}
\frac{\partial \langle r^2(t)\rangle}{\partial t} &= D\int \mathbf{dr}\, r^2 \nabla^2 c(r,t) \\
&= D\int \mathbf{dr}\, \boldsymbol{\nabla}\cdot(r^2 \boldsymbol{\nabla} c(r,t)) - D\int \mathbf{dr}\, \boldsymbol{\nabla} r^2 \cdot \boldsymbol{\nabla} c(r,t) \\
&= D\int \mathbf{dS}\, (r^2 \boldsymbol{\nabla} c(r,t)) - 2D\int \mathbf{dr}\, \mathbf{r}\cdot\boldsymbol{\nabla} c(r,t) \\
&= 0 - 2D\int \mathbf{dr}\, (\boldsymbol{\nabla}\cdot \mathbf{r} c(r,t)) + 2D\int \mathbf{dr}\, (\boldsymbol{\nabla}\cdot\mathbf{r}) c(r,t) \\
&= 0 + 2dD\int \mathbf{dr}\, c(r,t) \\
&= 2dD.
\end{aligned} \tag{4.4.8}$$

Equation (4.4.8) relates the diffusion coefficient D to the width of the concentration profile. This relation was first derived by Einstein. It should be realized that, whereas D is a macroscopic transport coefficient, $\langle r^2(t)\rangle$ has a microscopic interpretation: it is the mean-squared distance over which the labeled molecules have moved in a time interval t. This immediately suggests how to measure D in a computer simulation. For every particle i, we measure the distance traveled in time t, $\Delta \mathbf{r}_i(t)$, and we plot the mean square of these distances as a function of the time t:

$$\langle \Delta r(t)^2\rangle = \frac{1}{N}\sum_{i=1}^{N} \Delta \mathbf{r}_i(t)^2.$$

This plot would look like the one that will be shown later in Figure 4.6. We should be more specific about what we mean by the displacement of a particle in a system with periodic boundary conditions. The displacement that we are interested in is simply the time integral of the velocity of the tagged particle:

$$\Delta \mathbf{r}(t) = \int_0^t dt'\, \mathbf{v}(t').$$

In fact, there is a relation that expresses the diffusion coefficient directly in terms of the particle velocities. We start with the relation

$$2D = \lim_{t\to\infty} \frac{\partial \langle x^2(t)\rangle}{\partial t}, \tag{4.4.9}$$

where, for convenience, we consider only one Cartesian component of the mean-squared displacement. If we write $x(t)$ as the time integral of the x component of the tagged-particle velocity, we get

$$\begin{aligned}
\langle x^2(t)\rangle &= \left\langle \left(\int_0^t dt'\, v_x(t')\right)^2 \right\rangle \\
&= \int_0^t\int_0^t dt'dt''\, \langle v_x(t')v_x(t'')\rangle \\
&= 2\int_0^t\int_0^{t'} dt'dt''\, \langle v_x(t')v_x(t'')\rangle .
\end{aligned} \tag{4.4.10}$$

The quantity $\langle v_x(t')v_x(t'')\rangle$ is called the velocity autocorrelation function. It measures the correlation between the velocity of a particle at times t′ and t″. The velocity autocorrelation function (VACF) is an equilibrium property of the system, because it describes correlations between velocities at different times along an equilibrium trajectory. As equilibrium properties are invariant under a change of the time origin, the VACF depends only on the difference of t′ and t″. Hence,

$$\langle v_x(t')v_x(t'')\rangle = \langle v_x(t'-t'')v_x(0)\rangle .$$

Inserting equation (4.4.10) in equation (4.4.9), we obtain

$$\begin{aligned}
2D &= \lim_{t\to\infty} 2\int_0^t dt''\, \langle v_x(t-t'')v_x(0)\rangle \\
D &= \int_0^\infty d\tau\, \langle v_x(\tau)v_x(0)\rangle .
\end{aligned} \tag{4.4.11}$$

In the last line of equation (4.4.11) we introduced the coordinate $\tau \equiv t - t''$. Hence, we see that we can relate the diffusion coefficient D to the integral

of the velocity autocorrelation function. Such a relation between a transport coefficient and an integral over a time-correlation function is called a *Green-Kubo relation* (see Appendix C for some details). Green-Kubo relations have been derived for many other transport coefficients, such as the shear viscosity η,

$$\eta = \frac{1}{Vk_BT}\int_0^\infty dt\ \langle \sigma^{xy}(0)\sigma^{xy}(t)\rangle \tag{4.4.12}$$

with

$$\sigma^{xy} = \sum_{i=1}^{N}\left(m_i v_i^x v_i^y + \frac{1}{2}\sum_{j\neq i} x_{ij} f_y(r_{ij})\right); \tag{4.4.13}$$

the thermal conductivity λ_T,

$$\lambda_T = \frac{1}{Vk_BT^2}\int_0^\infty dt\ \langle j_z^e(0)j_z^e(t)\rangle \tag{4.4.14}$$

with

$$j_z^e = \frac{d}{dt}\sum_{i=1}^{N} z_i \frac{1}{2}\left(m_i v_i^2 + \sum_{j\neq i} v(r_{ij})\right); \tag{4.4.15}$$

and electrical conductivity σ_e

$$\sigma_e = \frac{1}{Vk_BT}\int_0^\infty dt\ \langle j_x^{el}(0)j_x^{el}(t)\rangle \tag{4.4.16}$$

with

$$j_x^{el} = \sum_{i=1}^{N} q_i v_i^x. \tag{4.4.17}$$

For details, see, for example, [79]. Time-correlation functions can easily be measured in a Molecular Dynamics simulation. It should be emphasized that for classical systems, the Green-Kubo relation for D and the Einstein relation are strictly equivalent. There may be practical reasons to prefer one approach over the other, but the distinction is never fundamental. In Algorithm 8 an implementation of the calculation of the mean-squared displacement and velocity autocorrelation function is described.

4.4.2 Order-n Algorithm to Measure Correlations

The calculation of transport coefficients from the integral of a time-correlation function, or from a (generalized) Einstein relation, may require a lot of memory and CPU time, in particular if fluctuations decay slowly. As an example, we consider again the calculation of the velocity autocorrelation function and the measurement of the diffusion coefficient. In a dense

Algorithm 8 (Diffusion)

```
subroutine dif(switch,nsamp)          diffusion; switch = 0 init.
                                      = 1 sample, and = 2 results
if (switch.eq.0) then                 Initialization
   ntel=0                             time counter
   dtime=dt*nsamp                     time between two samples
   do i=1,tmax                        tmax total number of time step
      ntime(i)=0                      number of samples for time i
      vacf(i)=0
      r2t(i)=0
   enddo
else if (switch.eq.1) then            sample
   ntel=ntel+1
   if (mod(ntel,it0).eq.0) then       decide to take a new t = 0
      t0 = t0 + 1                     update number of t = 0
      tt0=mod(t0-1,t0max)+1           see note 1
      time0(tt0)=ntel                 store the time of t = 0
      do i=,npart
         x0(i,tt0)=x(i)               store position for given t = 0
         vx0(i,tt0)=vx(i)             store velocity for given t = 0
      enddo
   endif
   do t=1,min(t0,t0max)               update vacf and r2, for t = 0
     delt=ntel-time0(t)+1             actual time minus t = 0
     if (delt.lt.tmax) then
      ntime(delt)=ntime(delt)+1
      do i=1,npart
         vacf(delt)=vacf(delt)+       update velocity autocorr.
+          vx(i)*vx0(i,t)
         r2t(delt)=r2t(delt)+         update mean-squared displ.
+           (x(i)-x0(i,t))**2
      enddo
     endif
   enddo
 else if (switch.eq.2) then           determine results
   do i=1,tmax
     time=dtime*(i+0.5)               time
     vacf(i)=vacf(i)                  volume velocity autocorr.
+            /(npart*ntime(i))
     r2t(i)=r2t(i)                    mean-squared displacement
+          /(npart*ntime(i))
   enddo
 endif
 return
 end
```

Comments to this algorithm:

1. *We define a new* t = 0 *after each* `it0` *times this subroutine has been called. For each* t = 0*, we store the current positions and velocities. The term* `t0max` *is the maximum number of* t = 0 *we can store. If we sample more, the first* t = 0 *will be removed and replaced by a new one. This limits the maximum time we collect data to* `t0max*it0`*; this number should not be smaller than* `tmax`*, the total number of time steps we want to sample.*
2. *Because* `nsamp` *gives the frequency at which this subroutine is called, the time between two calls is* `nsamp*delt`*, where* `delt` *is the time step.*

medium, the velocity autocorrelation function changes rapidly on typically microscopic time scales. It therefore is important to have an even shorter time interval between successive samples of the velocity. Yet, when probing the long-time decay of the velocity autocorrelation function, it is not necessary to sample with the same frequency. The conventional schemes for measuring correlation functions do not allow for such adjustable sampling frequencies. Here, we describe an algorithm that allows us to measure fast and slow decay simultaneously at minimal numerical cost. This scheme can be used to measure the correlation function itself, but in the example that we discuss, we show how it can be used to compute the transport coefficient.

Let us denote by Δt the time interval between successive measurements of the velocity of the particles in the system. We can define block sums of the velocity of a given particle as follows:

$$\mathbf{v}^{(i)}(j) \equiv \sum_{l=(j-1)n+1}^{jn} \mathbf{v}^{(i-1)}(l) \tag{4.4.18}$$

with

$$\mathbf{v}^{(0)}(l) \equiv \mathbf{v}(l), \tag{4.4.19}$$

where $\mathbf{v}(l)$ is the velocity of a particle at time l. Equation (4.4.18) is a recursive relation between block sums of level i and $i-1$. The variable n determines the number of terms in the summation. For example, $\mathbf{v}^{(3)}(j)$ can

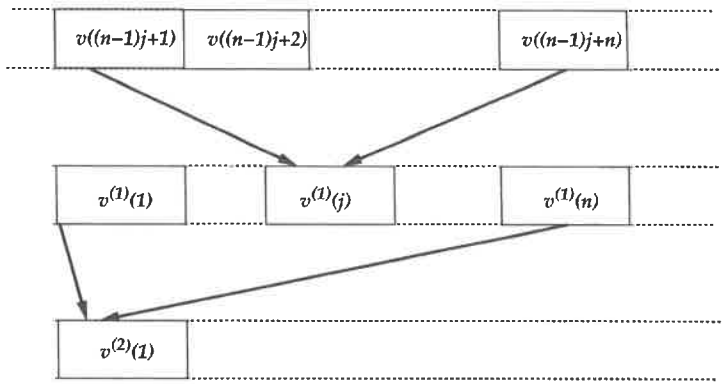

Figure 4.2: Coarse graining the velocities.

be written as

$$\begin{aligned}
\mathbf{v}^{(3)}(j) &= \sum_{l_1=(j-1)n+1}^{jn} \mathbf{v}^{(2)}(l_1) \\
&= \sum_{[l_1=(j-1)n+1]}^{jn} \sum_{[l_2=(l_1-1)n+1]}^{l_1 n} \sum_{[l_3=(l_2-1)n+1]}^{l_2 n} \mathbf{v}(l_3) \\
&= \sum_{l=(j-1)n^3+1}^{n^3 j} \mathbf{v}(l) \\
&\approx \frac{1}{\Delta t} \int_{l=(j-1)n^3+1}^{n^3 j} dt\, \mathbf{v}(t) = \frac{\mathbf{r}(n^3 j) - \mathbf{r}(n^3(j-1)+1)}{\Delta t}.
\end{aligned}$$

Clearly, the block sum of the velocity is related to the displacement of the particle in a time interval $n^i \Delta t$. In Figure 4.2 the blocking operation is illustrated. From the preceding block sums, it is straightforward to compute the velocity autocorrelation function with a resolution that decreases with increasing time. At each level of blocking, we need to store $n \times N$ block sums, where N is the number of particles (in practice, it will be more convenient to store the *block-averaged* velocities).

The total storage per particle for a simulation of length $t = n^i \Delta t$ is $i \times n$. This should be compared to the conventional approach where, to study correlations over the same time interval, the storage per particle would be n^i. In the conventional calculation of correlation functions, the number of floating-point operations scales at t^2 (or $t \ln t$, if the fast Fourier technique is used). In contrast, in the present scheme the number of operations scales as t.

At each time step we have to update $\mathbf{v}^{(0)}(t)$ and correlate it with all n entries in the $\mathbf{v}^{(0)}$-array. The next block sum has to be updated and correlated once every n time steps, the third every every n^2 steps, etc. This yields, for the total number of operations,

$$\frac{t}{\Delta t} \times n \left(1 + \frac{1}{n} + \frac{1}{n^2} + \cdots + \frac{1}{n^i}\right) < \frac{t}{\Delta t} n \frac{n}{n-1}.$$

Using this approach, we can quickly and efficiently compute a wide variety of correlation functions, both temporal and spatial. However, it should be stressed that each blocking operation leads to more coarse graining. Hence, any high-frequency modulation of long-time behavior of such correlation functions will be washed out.

Interestingly enough, even though the velocity autocorrelation function itself is approximate at long times, we can still compute the integral of the velocity autocorrelation function (i.e., the diffusion coefficient), with no loss in numerical accuracy. Next, we discuss in some detail this technique for computing the diffusion coefficient.

Let us define

$$\Delta\bar{\mathbf{r}}^{(i)}(j) \equiv \sum_{l=0}^{j} \mathbf{v}^{(i)}(l)\Delta t = \mathbf{r}(n^i) - \mathbf{r}(0). \tag{4.4.20}$$

The square of the displacement of the particle in a time interval $n^i\Delta t$ can be written as

$$(\Delta\bar{\mathbf{r}}^2)^{(i)}(j) = \left[\mathbf{r}(n^i) - \mathbf{r}(0)\right]^2 = \Delta\bar{\mathbf{r}}^{(i)}(j) \cdot \Delta\bar{\mathbf{r}}^{(i)}(j). \tag{4.4.21}$$

To compute the diffusion coefficient, we should follow the time dependence of the mean-squared displacement. As a first step, we must determine $\Delta\bar{\mathbf{r}}^{(i)}(j)$ for all i and all j. In fact, to improve the statistics, we wish to use every sample point as a new time origin. To achieve this, we again create arrays of length n. However, these arrays do not contain the same block sums as before, but partial block sums (see Algorithm 9).

1. At every time interval Δt, the lowest-order blocking operation is performed through the following steps:

   (a) We first consider the situation that all lowest-order accumulators have already been filled at least once (this is true if $t > n\Delta t$). The value of the current velocity $v(t)$ is added to

   $$\mathbf{v}_{\text{sum}}(1,j) = \mathbf{v}_{\text{sum}}(1,j+1) + \mathbf{v}(t)$$

   for `j = 1,n-1`, and

   $$\mathbf{v}_{\text{sum}}(1,j) = \mathbf{v}(t)$$

   for `j = n`.

Algorithm 9 (Diffusion: Order-n Algorithm)

```
subroutine dif(switch,nsamp)          diffusion
                                      switch = 0 initialization,
                                      = 1 sample, and = 2 results
if (switch.eq.0) then                 initialization
  ntel=0                              time counter for this subroutine
  dtime=dt*nsamp                      time between two samples
  do ib=1,ibmax                       ibmax max. number of blocks
   ibl(ib)=0                          length of current block
   do j=1,n                           n number of steps in a block
    tel(ib,j)=0                       counter number of averages
    delr2(ib,j)=0                     running average mean-sq. displ.
    do i=1,npart
      vxsum(ib,j,i)=0                 coarse-grained velocity particle i
    enddo
   enddo
  enddo
else if (switch.eq.2) then            print results
  do ib=1,max(ibmax,iblm)
    do j=2,min(ibl(ib),n)
     time=dtime*j*n**(ib-1)           time
     r2=delr2(ib,j)*dtime**2          mean-squared displacement
           /tel(ib,j)
    enddo
  enddo
   ...(continue)....
```

   (b) These operations yield

   $$\mathbf{v}_{\text{sum}}(1,l) = \sum_{j=t-n+l}^{j=t} \mathbf{v}(j).$$

   The equation allow us to update the accumulators for the mean-squared displacement (4.4.21) for $l = 1, 2, \ldots, n$:

   $$(\Delta\bar{\mathbf{r}}^2)^{(0)}(l) = (\Delta\bar{\mathbf{r}}^2)^{(0)}(l) + \mathbf{v}^2_{\text{sum}}(1,l)\Delta t^2.$$

2. If the current time step is a multiple of n, we perform the first blocking operation, if it is a multiple of n^2 the second, etc. Performing blocking operation i involves the following steps:

```
     ...(continue)....
  else if (switch.eq.1) then          sample
   ntel=ntel+1
   iblm=MaxBlock(ntel,n)              maximum number of possible
                                      blocking operations
   do ib=1,iblm
    if (mod(ntel,n**(ib-1))           test if ntel is a multiple of n^ib
+           .eq.0) then
     ibl(ib)=ibl(ib)+1                increase current block length
     inm=max(ibl(ib),n)               set maximum block length to n
     do i=1,npart
       if(ib.eq.1) then
         delx=vx(i)                   0th block: ordinary velocity
       else
         delx=vxsum(ib-1,1,i)         previous block velocity
       endif
       do in=1,inm
         if (inm.ne.n) then           test block length equal to n
           inp=in
         else
           inp=in+1
         endif
         if (in.lt.inm) then
          vxsum(ib,in,i)=
+          vxsum(ib,inp,i)+delx       eqns. (4.4.22) or (4.4.25)
         else
          vxsum(ib,in,i)=delx        eqns. (4.4.23) or (4.4.26)
         endif
       enddo
       do in=1,inm
        tel(ib,in)=tel(ib,in)+1      counter number of updates
        delr2(ib,in)=delr2(ib,in)    update equation (4.4.24)
+        +vxsum(ib,inm-in+1,i)**2
       enddo
     enddo
    endif
   enddo
  endif
  return
  end
```

Comment to this algorithm:

1. `MaxBlock(ntel,n)` *gives the maximum number of blocking operations that can be performed on the current time step* `ntel`.

(a) As before, we first consider the situation that all ith-order accumulators have already been filled at least once (i.e., $t > n^i \Delta t$). Using the $i-1$th block sum ($v_{sum}(i-1,1)$), we update

$$v_{sum}(i,j) = v_{sum}(i,j+1) + v_{sum}(i-1,1) \tag{4.4.22}$$

for `j = 1,n-1`, and

$$v_{sum}(i,j) = v_{sum}(i-1,1) \tag{4.4.23}$$

for `j = n`.

(b) These operations yield

$$v_{sum}(i,l) = \sum_{j=n-l+1}^{j=n} v_{sum}(i-1,j).$$

The equations allows us to update the accumulators for the mean-squared displacement, equation (4.4.21), for $l = 1,2,\ldots,n$:

$$(\Delta r^2)^{(i)}(l) = (\Delta r^2)^{(i)}(l) + v_{sum}^2(i,l)\Delta t^2. \tag{4.4.24}$$

3. Finally, we must consider how to handle arrays that have not yet been completely filled. Consider the situation that only `nmax` of the `n` locations of the array that contains the ith-level sums have been initialized. In that case, we should proceed as follows:

   (a) Update the current block length: `nmax = nmax+1` (`nmax` $< n$).

   (b) For `j = 1,nmax-1`

$$v_{sum}(i,j) = v_{sum}(i,j) + v_{sum}(i-1,1). \tag{4.4.25}$$

   (c) For `j = nmax`

$$v_{sum}(i,j) = v_{sum}(i-1,1). \tag{4.4.26}$$

   The update of equation (4.4.21) remains the same.

In Case Study 6, a detailed comparison is made between the present algorithm and the conventional algorithm for the diffusion of the Lennard-Jones fluid.

4.5 Some Applications

Let us illustrate the results of the previous sections with an example. Like in the section on Monte Carlo simulations we choose the Lennard-Jones fluid

as our model system. We use a truncated and shifted potential (see also section 3.2.2):

$$u^{tr-sh}(r) = \begin{cases} u^{lj}(r) - u^{lj}(r_c) & r \leq r_c \\ 0 & r > r_c \end{cases},$$

where $u^{lj}(r)$ is the Lennard-Jones potential and for these simulations $r_c = 2.5\sigma$ is used.

Case Study 4 (Static Properties of the Lennard-Jones Fluid)
Let us start a simulation with 108 particles on a simple cubic lattice. We give the system an initial temperature $T = 0.728$ and density $\rho = 0.8442$, which is close to the triple (gas-liquid-solid) point of the Lennard-Jones fluid [81–83].

In Figure 4.3 the evolution of the total energy, kinetic energy, and potential energy is shown. It is important to note that the total energy remains constant and does not show a (slow) drift during the entire simulation. The kinetic and potential energies do change initially (the equilibration period) but during the end of the simulation they oscillate around their equilibrium value. This figure shows that, for the calculation of the average potential energy or kinetic energy, we need approx. 1000 time steps to equilibrate the simulation. The figure also shows significant fluctuations in the potential energy, some of which may take several (100) time steps before they disappear.

Appendix D shows in detail how to calculate statistical error in the data of a simulation. In this example, we use the method of Flyvbjerg and Petersen [84]. The following operations on the set of data points are performed: we start by calculating the standard deviation of all the data points, then we group two consecutive data points and determine again the standard deviation of the new, blocked, data set. This new data set contains half the number of data points of the original set. The procedure is repeated until there are not enough data points to compute a standard deviation; the number of times we perform this operation is called M. What do we learn from this?

First of all, let us assume that the time between two samples is so large that the data points are uncorrelated. If the data are uncorrelated the standard deviation (as calculated according to the formula in Appendix D, i.e., correcting for the fact we have fewer data points) should be invariant to this blocking operation and we should get a standard deviation that is independent of M. In a simulation, however, the time between two data points is usually too short to obtain a statistically independent sample; as a consequence consecutive data points would be (highly) correlated. If we would calculate a standard deviation using these data, this standard deviation will be too optimistic. The effect of the block operation will be that after grouping two consecutive data points, the correlation between the two new data points will be less. This, however, will increase the standard deviation; the data will have more noise since consecutive data points no longer resemble

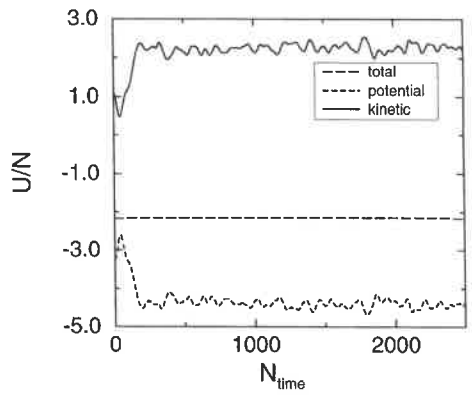

Figure 4.3: Total, potential, and kinetic energy per particle U/N as a function of the number of time steps N_{time}.

each other that closely. This decrease of accuracy as a function of the number of blocking operations will continue until we have grouped so many data points that two consecutive points are really uncorrelated. This is exactly the standard deviation we want to determine. It is important to note that we have to ensure that the standard deviations we are looking at are significant; therefore, we have to determine the standard deviation of the error at the same time.

The results of this error calculation for the potential energy are shown in Figure 4.4, as expected, for a low value of M; the error increases until a plateau is reached. For high values of M, we have only a few data points, which results in a large standard deviation in the error. The advantage of this method is that we have a means of finding out whether we have simulated enough; if we do not find such a plateau, the simulation must have been too short. In addition we find a reliable estimate of the standard deviation. The figure also shows the effect of increasing the total length of the simulation by a factor of 4; the statistical error in the potential energy has indeed decreased by a factor of 2.

In this way we obtained the following results. For the potential energy $U = -4.4190 \pm 0.0012$ and for the kinetic energy $K = 2.2564 \pm 0.0012$, the latter corresponds to an average temperature of $T = 1.5043 \pm 0.0008$. For the pressure, we have obtained 5.16 ± 0.02.

In Figure 4.5, the radial distribution function is shown. To determine this function we used Algorithm 7. This distribution function shows the characteristics of a dense liquid. We can use the radial distribution function to calculate the energy and pressure. The potential energy per particle can be

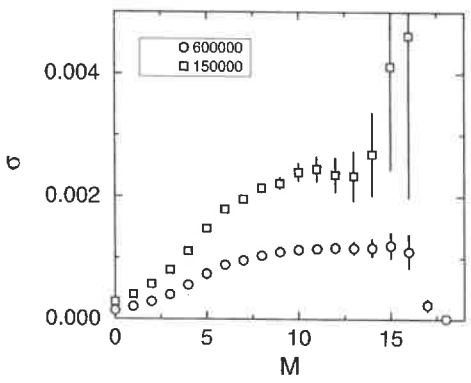

Figure 4.4: The standard deviation σ in the potential energy as a function of the number of block operations M for a simulation of 150,000 and 600,000 time steps. This variance is calculated using equation (D.3.4).

calculated from

$$\begin{aligned} U/N &= \frac{1}{2}\rho\int_0^\infty \mathbf{dr}\, u(r)g(r) \\ &= 2\pi\rho\int_0^\infty dr\, r^2 u(r)g(r) \end{aligned} \tag{4.5.1}$$

and for the pressure from

$$\begin{aligned} P &= \rho k_B T - \frac{1}{3}\frac{1}{2}\rho^2\int_0^\infty \mathbf{dr}\,\frac{du(r)}{dr} r g(r) \\ &= \rho k_B T - \frac{2}{3}\pi\rho^2\int_0^\infty dr\,\frac{du(r)}{dr} r^3 g(r), \end{aligned} \tag{4.5.2}$$

where $u(r)$ is the pair potential.

Equations (4.5.1) and (4.5.2) can be used to check the consistency of the energy and pressure calculations and the determination of the radial distribution function. In our example, we obtained from the radial distribution function for the potential energy $U/N = -4.419$ and for the pressure $P = 5.181$, which is in good agreement with the direct calculation.

Case Study 5 (Dynamic Properties of the Lennard-Jones Fluid)
As an example of a dynamic property we have determined the diffusion coefficient. As shown in the previous section, the diffusion coefficient can be determined from the mean-squared displacement or from the velocity autocorrelation function. We have determined these properties using Algorithm 8.

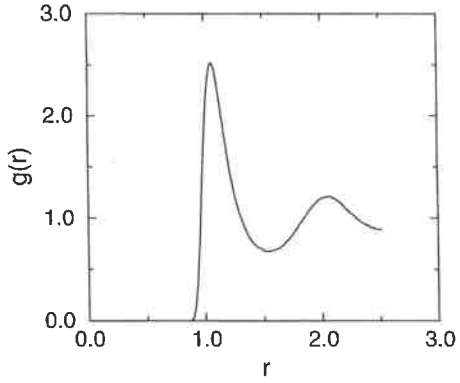

Figure 4.5: Radial distribution function of a Lennard-Jones fluid close to the triple point: $T = 1.5043 \pm 0.0008$ and $\rho = 0.8442$.

In Figure 4.6 the mean-squared displacement is shown as a function of the simulation time. From the mean-squared displacement we can determine the diffusion using equation (4.4.9). This equation, however, is valid only in the limit $t \to \infty$. In practice this means that we have to verify that we have simulated enough that the mean-squared displacement is really proportional to t and not to another power of t.

The velocity autocorrelation function can be used as an independent route to test the calculation of the diffusion coefficient. The diffusion coefficient follows from equation (4.4.11). In this equation we have to integrate to $t \to \infty$. Knowing whether we have simulated sufficiently to perform this integration reliably is equivalent to determining the slope in the mean-squared displacement. A simple trick is to determine the diffusion coefficient as a function of the truncation of the integration; if a plateau has been reached over a sufficient number of integration limits, the calculation is probably reliable.

Case Study 6 (Algorithms to Calculate the Mean-Squared Displacement)
In this case study, a comparison is made between the conventional (Algorithm 8) and the order-n methods (Algorithm 9) to determine the mean-squared displacement. For this comparison we determine the mean-squared displacement of the Lennard-Jones fluid.

In Figure 4.7 the mean-squared displacement as a function of time as computed with the conventional method is compared with that obtained from the order-n scheme. The calculation using the conventional scheme could not be extended to times longer than $t > 10$ without increasing the number of

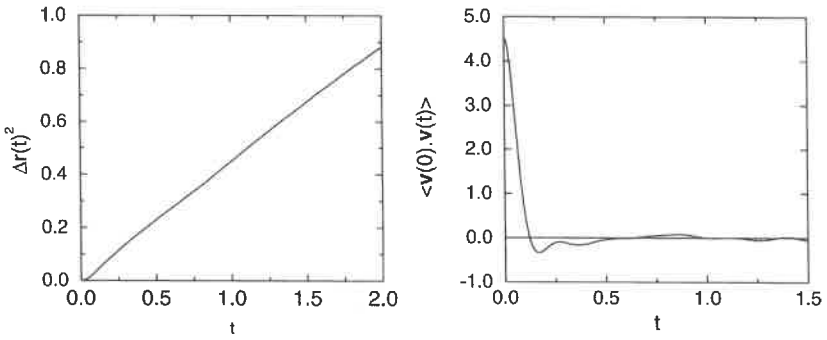

Figure 4.6: (left) Mean-squared displacement $\Delta r(t)^2$ as a function of the simulation time t. Note that for long times, $\Delta r(t)^2$ varies linearly with t. The slope is then given by 2dD, where d is the dimensionality of the system and D the self-diffusion coefficient. (right) Velocity autocorrelation function $\langle \mathbf{v}(0) \cdot \mathbf{v}(t) \rangle$ as a function of the simulation time t.

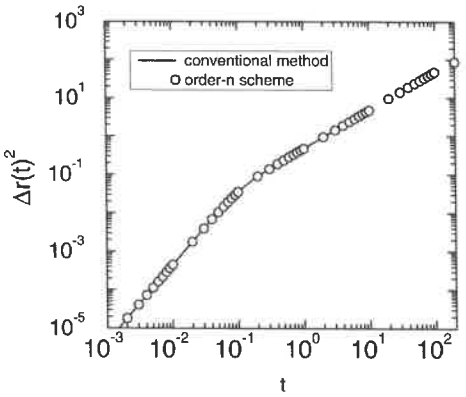

Figure 4.7: Mean-squared displacement as a function of time for the Lennard-Jones fluid ($\rho = 0.844$, $N = 108$, and $T = 1.50$); comparison of the conventional method with the order-n scheme .

time steps between two samples because of lack of memory. With the order-n scheme the calculation could be extended to much longer times with no difficulty. It is interesting to compare the accuracy of the two schemes. In the conventional scheme, the velocities of the particles at the current time step are used to update the mean-squared displacement of all time intervals. In

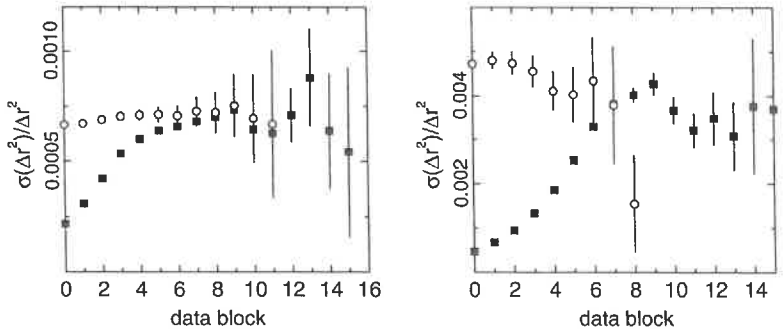

Figure 4.8: Relative error in the mean-squared displacement as a function of the number of data blocks as defined by Flyvbjerg and Petersen. The figures compare the conventional scheme (solid squares) with the order-n method (open circles) to determine the mean-squared displacement. The right figure is for $t = 0.1$ and the left figure for $t = 1.0$.

the order-n scheme the current time step is only used to update the lowest-order array of $\mathbf{v}_{\text{sum}}$ (see Algorithm 9). The block sums of level i are updated only once every n^i time step. Therefore, for a total simulation of M time steps, the number of samples is much less for the order-n scheme; for the conventional scheme, we have M samples for all time steps, whereas the order-n scheme has M/n^i samples for the ith block velocity. Naively, one would think that the conventional scheme therefore is more accurate. In the conventional scheme, however, the successive samples will have much more correlation and therefore are not independent. To investigate the effect of these correlations on the accuracy of the results, we have used the method of Flyvbjerg and Petersen [84] (see Appendix D.3 and Case Study 4). In this method, the standard deviation is calculated as a function of the number of data blocks. If the data are correlated, the standard deviation will increase as a function of the number of blocks until the number of blocks is sufficient that the data in a data block are uncorrelated. If the data are uncorrelated, the standard deviation will be independent of the number of blocks. This limiting value is the standard deviation of interest.

In these simulations the time step was $\Delta t = 0.001$ and the block length was set to $n = 10$. For both methods the total number of time steps was equal. To calculate the mean-squared displacement, we have used 100,000 samples for all times in the conventional scheme. For the order-n scheme, we have used 100,000 samples for $t \in [0, 0.01]$, 10,000 for $t \in [0.01, 0.1]$, 1,000 for for $t \in [0.1, 1]$, etc. This illustrates that the number of samples in the order-n scheme is considerably less than in the conventional scheme. The

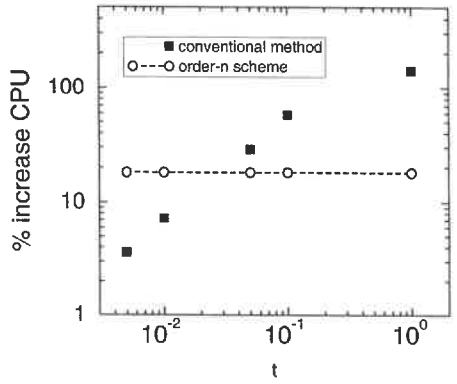

Figure 4.9: Percentage increase of the total CPU time as a function of the total time for which we determine the mean-squared displacement; comparison of the conventional scheme with the order-n scheme for the same system as is considered in Figure 4.7

accuracy of the results, however, turned out to be the same. This is shown in Figure 4.8 for $t = 0.1$ and $t = 1.0$. Since the total number of data blocking operations that can be performed on the data depends on the total number of samples, the number of blocking operations is less for the order-n method. Figure 4.8 shows that for $t = 0.1$ the order-n scheme yields a standard deviation that is effectively constant after three data blocking operations, indicating the samples are independent, whereas the standard deviation using the conventional method shows an increase for the first six to eight data blocking operations. For $t = 1.0$ the order-n method is independent of the number of data blocks, the conventional method only after 10 data blocks. This implies that one has to average over $2^{10} \approx 1000$ successive samples to have two independent data points. In addition, the figure shows that the plateau value of the standard deviation is essentially the same for the two methods, which implies that for this case the two methods are equally accurate.

In Figure 4.9 we compare the CPU requirements of the two algorithms for simulations with a fixed total number of time steps. This figure shows the increase of the total CPU time of the simulation as a function of the total time for which the mean-squared displacement has been calculated. With the order-n scheme the CPU time should be (nearly) independent of the total time for which we determine the mean-squared displacement, which is indeed what we observe. For the conventional scheme, however, the required CPU time increases significantly for longer times. At $t = 1.0$ the order-n scheme gives an increase of the total CPU time of 17%, whereas the conventional scheme shows an increase of 130%.

This example illustrates that the saving in memory as well as in CPU time of the order-n scheme can be significant, especially if we are interested in the mean-squared displacement at long times.

4.6 Questions and Exercises

Question 10 (Integrating the Equations of Motion)

1. *If you do an MD simulation of the Lennard-Jones potential with a time step that is much too large you will find an energy drift. This drift is towards a higher energy. Why?*
2. *Why don't we use Runga-Kutta methods to integrate the equations of motion of particles in MD?*
3. *Which of the following quantities are conserved in the MD simulation of Case Study 4: potential energy, total momentum, position of the center of mass of the system, or total angular momentum?*
4. *Show that the Verlet and velocity Verlet algorithms lead to identical trajectories.*
5. *Derive the Leap-Frog Algorithm by using Taylor expansions for* $v\left(t + \frac{\Delta t}{2}\right)$, $v\left(t - \frac{\Delta t}{2}\right)$, $x(t + \Delta t)$, *and* $x(t)$.

Question 11 (Correlation Functions)

1. *The value of the velocity autocorrelation function (vacf) at* $t = 0$ *is related to an observable quantity. Which one?*
2. *Calculate the limit of the vacf for* $t \to \infty$.
3. *What is the physical significance of vacf* < 0*?*
4. *When you calculate the mean-squared displacement for particles in a system in which periodic boundary conditions are used and in which particles are placed back in the box, you should be very careful in calculating the displacement. Why?*
5. *What is more difficult to calculate accurately: the self-diffusion coefficient or the viscosity? Explain.*

Exercise 10 (Molecular Dynamics of a Lennard-Jones System)
On the book's website you can find a Molecular Dynamics (MD) program for a Lennard-Jones fluid in the NVE ensemble. Unfortunately, the program does not conserve the total energy because it contains three errors.

1. Find the three errors in the code. Hint: there are two errors in *integrate.f* and one in *force.f*. See the file *system.inc* for documentation about some of the variables used in this code.

2. How is one able to control the temperature in this program? After all, the total energy of the system should be constant (not the temperature).
3. To test the energy drift ΔU of the numerical integration algorithm for a given time step Δt after N integration steps, one usually computes [85]

$$\Delta U(\Delta t) = \frac{1}{N}\sum_{i=1}^{i=N}\left|\frac{U(0) - U(i\Delta t)}{U(0)}\right|.$$

In this equation, $U(x)$ is the total energy (kinetic + potential) of the system at time x. Change the program (only in *mdloop.f*) in such a way that ΔU is computed and make a plot of ΔU as a function of the time step. How does the time step for a given energy drift change with the temperature and density?

4. One of the most time-consuming parts of the program is the calculation of the nearest image of two particles. In the present program, this calculation is performed using an *if-then-else-endif* construction. This works only when the distance between two particles is smaller than 1.5 and larger than -1.5 times the size of the periodic box. A way to overcome this problem is to use a function that calculates the nearest integer `nint`

$$x = x - box * \texttt{nint}(x * ibox),$$

in which $ibox = 1.0/box$. Which expression is faster? (Hint: You only have to make some modifications in *force.f*.) Which expression will be faster on a vector computer like a Cray C90? Because the `nint` function is usually slow, you can write your own `nint` function. For example, when $x < -998$, we can use

$$\texttt{nint}(x) = \texttt{int}(x + 999.5) - 999. \tag{4.6.1}$$

What happens with the speed of the program when you replace the standard `nint` function? Do you have an explanation for this? [4]

5. In equation (4.6.1), `ibox` is used instead of `1/box`. Why?
6. An important quantity of a liquid or gas is the so-called self-diffusivity D. There are two methods to calculate D:

   (a) by integrating the velocity autocorrelation function:

$$\begin{aligned} D &= \frac{1}{3}\int_0^\infty \left\langle \mathbf{v}(t)\cdot\mathbf{v}\left(t+t'\right)\right\rangle dt' \\ &= \frac{\int_0^\infty \sum_{i=1}^{i=N}\left\langle \mathbf{v}(i,t)\cdot\mathbf{v}\left(i,t+t'\right)\right\rangle dt'}{3N} \end{aligned} \tag{4.6.2}$$

[4] The result will strongly depend on the computer/compiler that is used.

in which N is the number of particles and $\mathbf{v}(i,t)$ is the velocity of particle i at time t. One should choose t in such a way that independent time origins are taken, i.e., $t = ia\Delta t$, $i = 1, 2, \cdots, \infty$ and $\langle \mathbf{v}(t)\cdot\mathbf{v}(t + a\Delta t)\rangle \approx \mathbf{0}$.

   (b) by calculating the mean-squared displacement:

$$D = \lim_{t'\to\infty}\frac{\left\langle \left|\mathbf{x}\left(t+t'\right) - \mathbf{x}(t)\right|^2\right\rangle}{6t'}. \tag{4.6.3}$$

One should be very careful with calculation of the mean-squared displacement when periodic boundary conditions are used. Why?

Modify the program in such a way that the self-diffusivity can be calculated using both methods. Only modifications in subroutine *sample_diff.f* are needed. Why is it sufficient to use only independent time origins for the calculation of the means-squared displacement and the velocity autocorrelation function? What is the unit of D in SI units? How can one transform D into dimensionless units?

7. For Lennard-Jones liquids, Naghizadeh and Rice [86] report the following equation for self-diffusivity (dimensionless units, $T^* < 1.0$ and $p^* < 3.0$):

$$^{10}\log(D^*) = 0.05 + 0.07p^* - \frac{1.04 + 0.1p^*}{T^*}. \tag{4.6.4}$$

Try to verify this equation with simulations. How can one translate D^* to a diffusivity in SI units?

8. Instead of calculating the average energy $\langle U\rangle$ directly, one can use the radial distribution function $g(r)$. Derive an expression for $\langle U\rangle$ using $g(r)$. Compare this calculation with a direct calculation of the average energy. A similar method can be used to compute the average pressure.
9. In the current version of the code, the equations of motion are integrated by the Verlet algorithm. Make a plot of the energy drift ΔU for the following integration algorithms:

   - Verlet
   - Velocity Verlet
   - Euler (never use this algorithm in real simulations).