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 de-

sired 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 $\mathbf{r}_{ij}$, 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 $|\mathbf{r}_{ij}|$ 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:

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

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

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,

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

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