Chapter 4

Advanced Molecular Dynamics

Joost de Graaf

4.1 Introduction

In this chapter, we cover intermediate to advanced methods for molecular dynamics (MD) simulations. In writing it, I tried to place an emphasis on two questions that become important as soon as you move beyond writing your first small test simulation; although these became somewhat conflated in the original version[1]. First, you should ask yourself: "How can my force calculation be organized so that systems containing many particles can be simulated efficiently?" Second, after studying MD yourself, you may also start to wonder: "What should be done when the interaction potential is not smooth and a conventional time-stepping algorithm is no longer the natural description of the dynamics?" at a conceptual level.

These questions are closely connected to hydrodynamics, or the study of flow, which is a strong interest of mine, hence the framing of the original writeup. MD simulations resolve the motion of individual particles, whereas hydrodynamics describes fields such as the local density and velocity. It is not particularly sensible to talk about flow when only two particles are present. Hydrodynamic behavior emerges only when sufficiently many particles interact for local averages to become meaningful on scales much larger than the molecular scale. The connection is provided by the conservation of mass, momentum, and energy, together with a statistical description of molecular motion. At intermediate scales, kinetic descriptions such as the Boltzmann transport equation (BTE) become useful. At still larger scales, continuum equations such as the Navier-Stokes equations emerge.

This hierarchy of descriptions is useful even when you are not interested in flow at all. It is important to ask when it makes sense to switch approaches in describing your system and thereby make better use of your time. Moreover, when asking any serious questions about a system using MD or other techniques, you will quickly find yourself having to study

[1]The published version of this chapter was written at the height of the COVID-19 pandemic. Prolonged isolation, long working hours spent preparing teaching, and the associated mental strain had left me in an unhappy place. It was made clear to me that I had no choice but to complete this task, even after I had indicated that doing so was not a good idea—not by the editors, I should add. All these factors taken together led me to write something that I did not, and still do not, feel proud of. After returning to my place of work in 2022, by then having received several vaccinations (thank you medical science), I started feeling better and made several attempts to give the chapter a quick polish. However, I had come to associate the work strongly with a period of misery, to the point that I was unable to make the improvements I should have made. I indicated that there were issues to editors, but I feel that this is not sufficient in terms of taking responsibility. Thus, here I have tried my best to remedy some of the most painful shortcomings of the original, and I hope that this version lands better than the published chapter. I appreciate your understanding.

increasingly large particle numbers in order to determine whether your choice of (periodic) system size influences the results. As we will see here, the computational problem is dominated by the evaluation of interparticle forces, which places relatively strong bounds on what can be studied. As you may already have learned, the naive implementation that typically serves as a starting simulator scales poorly with the number of particles. Early researchers recognized and addressed these issues through the development of neighbor lists, Verlet lists, and cell lists, which can—although, be careful, they do not necessarily always—reduce the computational cost dramatically. We will therefore first discuss the physical scales that motivate particle-based simulations, and then develop the algorithms required for efficient short-ranged force calculations. The associated exercises are intended for you to produce working programs on your own, and will also show you how such programs should be validated.

There is a second notion of efficiency that is worth keeping in mind, namely, the order of the time-integration algorithm. A higher-order integrator reduces the error more rapidly as the time step is decreased[2], but this does not automatically make it the better choice. Higher-order methods generally require additional force evaluations or more elaborate bookkeeping, and in MD the time step is often constrained by the fastest physical motion in the system anyway. This is one reason why relatively simple second-order schemes, such as the Verlet and velocity-Verlet algorithms, are so widely used. They provide a useful balance between accuracy, stability, computational cost, and the preservation of the underlying dynamics. As with the organization of the force calculation, choosing an integration algorithm therefore involves asking not only whether it is formally accurate, but whether that (potentially increased) accuracy is meaningful for the problem at hand.

The final part of the chapter concerns discontinuous interactions. Hard-sphere and square-well potentials provide simple examples of interactions commonly used in theoretical work. Here, however, the force is not a smooth function that can be integrated using a finite time step, making these systems unsuited to conventional MD. Their Newtonian dynamics can instead be followed from one collision event to the next using event-driven molecular dynamics (EDMD). This provides a useful contrast with conventional MD and introduces a second type of algorithmic bookkeeping problem.

All of the above is not to say that you should see the conclusion of this chapter as "Effective simulation is all about bookkeeping," but some bookkeeping is certainly an essential part of it. More advanced elements of algorithmic design leave considerable room for creativity and can be a great deal of fun to work on. That work does require a solid understanding of the basics and sufficient practice. At the time of writing, that is doubly true given the increasing presence of AI-generated code in the literature. Coding itself will undoubtedly change drastically, but identifying problems, judging whether code is reliable, and becoming genuinely proficient will still require a strong foundation.

4.2 Relevant Scales: Molecular Dynamics and Hydrodynamics

Imagine a system of water molecules. When a sufficiently large amount of water is simulated and an external driving is applied, flow can emerge on length and time scales that are large compared with the molecular scales. For water, the relevant molecular length scale is of order a few Ångströms, set by the size and separation of individual molecules, while characteristic microscopic time scales range from femtoseconds for molecular vibrations to picoseconds for rotational and collisional relaxation. Hydrodynamic behavior, by contrast, is associated with

[2]The scaling of an $\mathcal{O}\left(\Delta t^4\right)$ algorithm leads to greater accuracy at small time step Δt, than one with $\mathcal{O}\left(\Delta t^2\right)$. The subtlety here is that there is typically a prefactor, which is why we say "is decreased," because the error of the better-scaling algorithm may not be smaller for certain values of Δt. With sufficient care for the prefactor, one can use a higher-order algorithm to have a larger step size in the simulation.

variations over distances and times large enough that many molecules contribute to a local average—typically nanometers or more in length and nanoseconds or more in time, although the precise crossover depends strongly on the phenomenon being studied. In view of this, it is obviously preposterous to simulate a bucket of water emptying out using an atomistic description. The interesting problem is therefore to decide when molecular detail matters, when a coarser description becomes adequate, and, more subtly, which method captures the physics of interest. Developing a good instinct for this takes practice and experience.

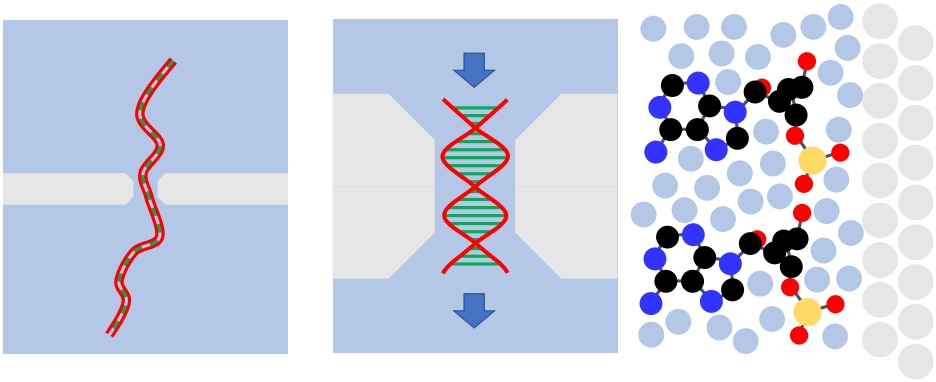

Figure 4.1: Impression of three levels of resolution for DNA translocation through a pore. From left to right, the microscopic resolution increases. In the left panel, both strand and solvent are treated at continuum level. In the central panel, features of the DNA and pore are resolved while the solvent is represented at a coarser level. In the right panel, the molecular structure of the strand, solvent, and pore wall is resolved explicitly. The appropriate description depends on the length and time scales of interest and on which microscopic details control the observable under study.

A useful example with which to illustrate the issues that can arise is transport through a nanopore, see Fig. 4.1. Solid-state nanopores can have geometrical dimensions of only a few to tens of molecular diameters [1]. Such small holes have been considered in the context of DNA sequencing. The promise of nanopore-based sequencing is to read DNA directly and rapidly, by pulling individual DNA molecules through nanometer-scale pores and infer each passing sequence from characteristic disruptions in ionic current. Such applications have been realized, but accuracy, throughput, sample preparation, and computational interpretation still impose practical limits. DNA strands are roughly 20 Ångströms wide and, for sequencing applications, may range in length from tens of base pairs to several million. A strand of 50 base pairs has a contour length of roughly 20 nanometers, whereas four million base pairs correspond to approximately 1.4 millimeters. Clearly, this is a setting in which large-scale features can control transport, while molecular details remain essential to the sequencing process itself. At one extreme, a DNA strand can be represented as a continuum object in a continuum solvent. At the other, one may wish to resolve the molecular structure of the strand, water, ions, and pore explicitly. Both descriptions can answer relevant questions, but each may also obscure important features of the problem. You might therefore start thinking about hybrid approaches capable of combining the advantages of both descriptions. Such approaches exist, although they are beyond the scope of this chapter.

Fluid flow through carbon nanotubes provides another instructive example. MD simulations and experiments have shown that the transport of water through narrow hydrophobic pores can differ strongly from the simplest macroscopic expectations [2, 3]. At the same time, continuum hydrodynamic descriptions can remain surprisingly successful down to nanometer

scales when suitable boundary conditions are used [4]. This coexistence of microscopic and continuum descriptions is precisely one of the reasons why particle-resolved simulations are useful. A continuum calculation requires constitutive relations and boundary conditions, whereas MD can help determine how such effective descriptions emerge and where their underlying assumptions begin to fail. Microscopic roughness or chemical functionalization, for example, may substantially alter the effective hydrodynamic slip at a solid surface [5].

Moving away from this flow-based example, the physics of the experiment and the observables of interest should guide your modeling choices. Though, as we have hopefully made clear, there is rarely a universally "best" level of description. A good model should contain enough microscopic detail to reproduce the mechanisms that control the quantity you wish to measure, while discarding degrees of freedom that merely increase the computational cost. The practical skill lies in choosing the simplest level that still retains the physics needed to answer the question at hand[3]. It is up to you to check whether the conclusions do not change when that choice is refined.

4.3 Neighbor, Verlet, Cell, and Linked Lists

When discussing MD, it is important to observe that a large part of the simulation revolves around computing forces between particles. These forces are required to advance the particle positions and velocities and thereby determine the evolution of the system. For simplicity, we will focus on pairwise interactions throughout this section. Angular, dihedral, electrostatic, and other many-body or long-ranged interactions require additional considerations.

A naive implementation of a pair force loops over all distinct pairs. Let us say that there are N particles in the system, then there are

$$N_{\text{pair}} = \frac{N(N-1)}{2} \tag{4.1}$$

pairs, so the force calculation has an algorithmic complexity $\mathcal{O}(N^2)$. This rapidly becomes problematic. If a three-dimensional system is enlarged at fixed number density by doubling each box dimension, the number of particles increases by a factor 8. A direct pair loop then requires roughly $8^2 = 64$ times as many pair evaluations.

This scaling is especially undesirable when finite-size effects have to be assessed. Large systems are often required because the physical correlation length can be appreciable, particularly in low-dimensional systems [8, 9]. In practice, finite-size effects are usually examined by simulating several system sizes and plotting the quantity of interest against an inverse measure of the system size, such as $1/L$ or $1/N$. The data can then be fitted to an appropriate finite-size scaling form[4] and extrapolated as $1/L \to 0$, or equivalently the thermodynamic limit $L, N \to \infty$. Increasing the system size also carries a second cost that is easy to overlook; equilibration and relaxation generally take longer, sometimes considerably so. You can compute a diffusion time based on the box length to get a hint of this equilibration and directly see the unfavorable effect. This means that the expense of a finite-size study grows by more than the cost of a single force evaluation would suggest. A good rule of thumb in

[3] For completeness, we should point out a few other methods before moving on. When it is too expensive to resolve all molecules in the fluid, mesoscale descriptions may be more appropriate. Lattice Boltzmann methods provide a discrete kinetic description based on the BTE and are widely used for simulating fluids when complex boundary conditions matter [6]. Multi-particle collision dynamics (MPCD) replaces a molecular description of the solvent with discrete phase-space carriers whose collective streaming and collisions generate the desired transport behavior [7]. At still larger scales one may solve the continuum equations directly using finite-volume, finite-element, spectral, or related numerical methods.

[4] You should not assume automatically that the leading correction is linear in $1/L$ or $1/N$. The appropriate scaling variable and functional form depend on the observable and the underlying physics.

project design is therefore to think carefully about both the physics and the algorithm before a single simulation becomes so expensive that gathering adequate statistics across several system sizes becomes computationally prohibitive.

Most molecular interactions used in simple MD models are short ranged, because the physical mechanisms from which they arise decay rapidly with inter-particle separation. The repulsive interaction associated with overlapping electron clouds (Pauli exclusion) becomes negligible once molecules are more than a few diameters apart. At the same time, dispersion interactions decay as r^{-6}. In practice, the force between two non-bonded particles separated by a sufficiently large distance can therefore be neglected to good approximation. Familiar examples of model pair potentials that are short ranged include: Lennard-Jones interactions, Weeks-Chandler-Andersen repulsions, square-well and square-shoulder potentials, hard-sphere interactions, and Morse potentials used to model excluded volume and short-ranged attraction and/or repulsion. Bonded interactions in molecular models, such as Finitely Extensible Nonlinear Elastic (FENE) springs, are even more local, since they act only between specifically connected particles that are purposefully kept close together.

This locality is precisely what makes neighbor-based force algorithms effective[5]. For a given particle, it is only necessary to take into account interactions with a small subset of potential neighbors in the full simulation box. Let the interaction be negligible beyond a cut-off radius r_c. At fixed number density, the average number $\bar{m}$ of particles within this radius is approximately independent of the total system size. If the force calculation can be restricted to those local neighbors, the computational work scales as $N\bar{m}$, and we therefore naively expect the algorithm to have $\mathcal{O}(N)$ scaling at fixed density. This is the basic reason why spatial decomposition is so useful and we will exploit this principle in the following.

4.3.1 The Reference Pair Loop

Before introducing more sophisticated data structures, let us make the properties of the naive approach explicit. That way, we have an idea of what our improvements are intended to replace. A direct pair loop is slow for large N, but it has two important advantages. It is super simple to implement and, once verified, it provides an excellent reference against which the accelerated routines can be tested. Optimization should therefore come only after a correct direct implementation. You can also leave your naive force loop intact when you improve your code, using it periodically to check if something has gone amiss.

For an orthorhombic (beam-like) periodic box with edge lengths L_α with $\alpha \in \{x, y, z\}$, the displacement from particle j to particle i is first formed as $\Delta r_\alpha = r_{i,\alpha} - r_{j,\alpha}$ and then mapped to the nearest periodic image using

$$\Delta r_\alpha \leftarrow \Delta r_\alpha - L_\alpha \, \mathcal{N}_{\text{int}}\left(\frac{\Delta r_\alpha}{L_\alpha}\right), \tag{4.2}$$

[5]Note that truly long-ranged interactions are an important exception. Roughly speaking, an interaction is considered long ranged when its decay with distance is sufficiently slow that the contribution from distant particles cannot simply be neglected. More formally, for an isotropic pair potential that decays as $u(r) \propto r^{-p}$ in d dimensions, the interaction is long ranged when $p \leq d$, since the corresponding spatial integral does not converge at large distances. The most familiar example is the Coulomb interaction between charges, $u(r) \propto 1/r$ in three dimensions. The two-dimensional case is considerably more subtle and forms a story of its own, since it suffers from a double divergence. This does not make for a mere mathematical curiosity, though. Effectively two-dimensional systems arise naturally in thin films, interfaces, membranes, adsorbed layers, electron gases, and atomically thin materials, making the peculiarities of two-dimensional physics of considerable practical importance. Dipole-dipole interactions, which decay as $1/r^3$ in three dimensions, form an important marginal case. Studying long-ranged interactions requires methods such as Ewald summation and its descendants [10], rather than a simple real-space cutoff. We will not cover these important methods here, but they are covered in detail in standard textbooks [11, 12].

where $\mathcal{N}_{\text{int}}$ returns the integer nearest to its argument. The squared separation $r_{ij}^2 = \boldsymbol{r}_{ij} \cdot \boldsymbol{r}_{ij}$ is compared to r_c^2 to decide proximity. For a radial pair force, the same pair calculation can be used to establish the force, potential energy, and virial contribution in one go. Newton's third law is then used to update both particles from a single pair evaluation.

The reference routine we have described above takes on the following form in pseudocode:

```
force[:] = 0
potential = 0
virial = 0

for i = 0,...,N-2:
    for j = i+1,...,N-1:

        dr = minimum_image(r[i] - r[j])
        r2 = dot(dr, dr)

        if r2 < rc2:
            fij, uij = pair_force_and_energy(dr, r2)
            force[i] += fij
            force[j] -= fij
            potential += uij
            virial += dot(dr, fij)
```

Here, we use pseudocode, because the syntax is not that important. For example, in **Python** you would typically not rely on **for** loops.

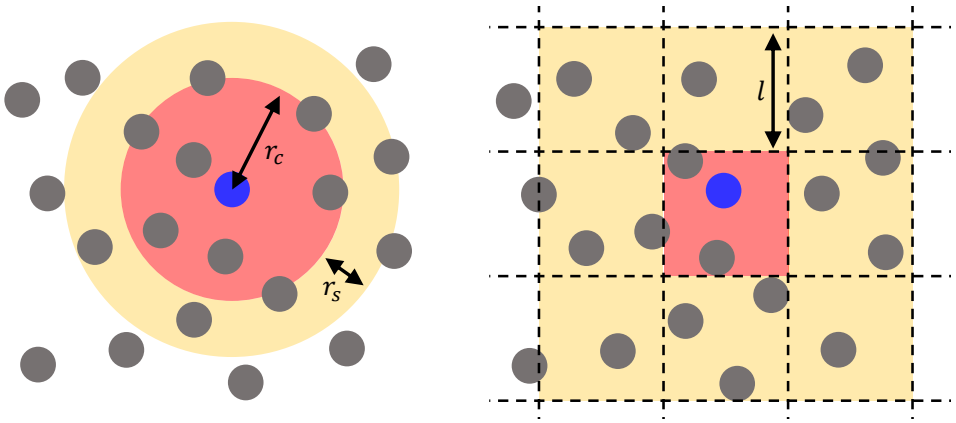

Figure 4.2: Two-dimensional illustration of a Verlet list (left) and a cell decomposition (right). In the Verlet construction, the red region indicates the force cut-off r_c and the yellow shell has thickness r_s, so the neighbor-list radius is $r_v = r_c + r_s$. In the cell decomposition, space is divided into cells of edge length l. If l is at least as large as the search radius, a particle need only be compared with particles in its own cell (red) and the surrounding cells. The particles in these cells are candidate neighbors; an actual distance test is still required before evaluating the force.

4.3.2 Verlet Lists

Given the above discussion of short-ranged interactions, a natural first attempt is to construct, for each particle, a list containing only those particles that lie within the interaction range r_c.

At first sight this appears to solve the problem. After all, why consider a pair if the particles are too far apart to interact? However, there is a catch: the particles move! Consequently, a pair that does not interact at the present time may do so a few time steps later. If the list contains only pairs with $r < r_c$, it must therefore be updated whenever a particle crosses the cut-off surface. In a dense fluid this happens sufficiently often that much of the computational gain can be lost to rebuilding the list[6].

The key observation made by Loup Verlet is that the list need not coincide with the range over which the force is evaluated [13]. Instead, we construct a somewhat larger list. Introducing a *skin* with length r_s, we define the Verlet radius as

$$r_v = r_c + r_s. \tag{4.3}$$

Pairs with $r < r_v$ are stored in the list, while the force between them is still evaluated only when $r < r_c$. The annular region between r_c and r_v in Fig. 4.2—a shell in three dimensions—therefore contains particles that do not presently interact, but which might do so before the list is reconstructed. The price to pay for this increased range is a few additional distance checks. However, this does not affect the scaling of the algorithm and the gain is that the same list can be reused for many MD steps.

Below we provide pseudocode for the construction and maintenance of the *Verlet list*, so that you can have a better feeling for the algorithm. We start by introducing a function that builds the neighbor list:

```
function build_neighbor_list(r, rv2):

    neighbor_list = empty_list()

    for i = 0,...,N-2:
        for j = i+1,...,N-1:

            dr = minimum_image(r[i] - r[j])
            r2 = dot(dr, dr)

            if r2 < rv2:
                neighbor_list.append((i,j))

    return neighbor_list
```

Next, we initialize the neighbor list at the start of the program:

```
rv = rc + rs
rv2 = rv * rv

reference_position[:] = r[:]
neighbor_list = build_neighbor_list(r, rv2)
```

While the simulation is running, we loop through the following sequence per time step:

```
advance_positions_and_velocities()

dmax2 = 0
```

[6]Note that we have made the implicit statement that building and rebuilding the list to begin with is somehow efficient. That is absolutely not obvious when you start thinking about neighbor lists.

```
for i = 0,...,N-1:
    dr = displacement(r[i], reference_position[i])
    d2 = dot(dr, dr)
    dmax2 = max(dmax2, d2)

if 4 * dmax2 >= rs * rs:
    reference_position[:] = r[:]
    neighbor_list = build_neighbor_list(r, rv2)

for (i,j) in neighbor_list:
    dr = minimum_image(r[i] - r[j])
    r2 = dot(dr, dr)
```

Because two particles can each move toward or away from one another, their relative displacement can be as large as twice the maximum displacement of any single particle, which is why the Verlet list is rebuilt once $2d_{\max} \geq r_s$, which explains the factor 4 above. Note that the last **`for`** statement can be merged with a force calculation as in the algorithm for the reference calculation in Section 4.3.1.

Inspecting the above pseudocode reveals that the initial construction still has an algorithmic complexity of $\mathcal{O}(N^2)$. Between reconstructions, determining whether the list remains valid requires only the displacement of each particle from its reference position to be checked, which is an $\mathcal{O}(N)$ operation. The stored list can then be reused for the force calculation until the displacement criterion is violated. That is, the expensive $\mathcal{O}(N^2)$ construction is performed only intermittently. Your intuition should tell you that we have gained something, but it is not so clear what that gain is, and the result is far from satisfying.

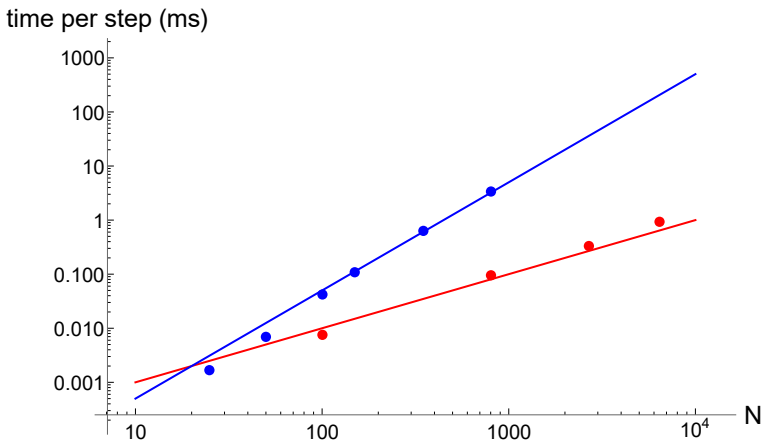

Figure 4.3: Comparison between a direct pair calculation (blue) and a cell-list only accelerated calculation (red). The data were obtained at fixed number density $\rho = 0.1$ using a `C++` variant of the exercise code. The expected asymptotic slopes on a log-log plot are approximately 2 for the direct pair loop and 1 for the cell-list implementation. The crossover at small N depends on implementation details and hardware, since constructing and traversing cells also carries a computational overhead.

4.3.3 Cell Lists

Let us turn to *cell lists* to solve the problem of Verlet-list construction by further exploiting spatial locality. The idea is as follows, see Fig. 4.2. We divide the simulation box into cells

with edge length l. This length is at least as large as the search radius. If forces are evaluated directly from the cell list, then we would use a distance that is at least (but close to) r_c. If the cell list is used to construct a Verlet list, then we instead use r_v. Each particle is assigned to one cell, which is clearly an $\mathcal{O}(N)$ operation. This point is illustrated in Fig. 4.3, which shows the favorable scaling comprared to a brute-force calculation. In d dimensions, a particle then needs to search only its own cell and the immediately surrounding cells; 3^d cells in total.

Periodic boundaries are handled by wrapping the cell indices as well as the particle displacement vectors. Care must also be taken not to compute each pair twice. One may either loop over a suitably chosen half-shell of neighboring cells (*e.g.*, see Ref. [12]) or retain a simple full neighborhood and enforce an index condition such as $j > i$. The latter is often easier to implement correctly when you start writing your own code, even if a production implementation can benefit from the former organization.

Before we move on, let us examine a pseudocode. The simplest way to maintain the cell list is to reconstruct the cell occupancies whenever an update is required. This may at first sound wasteful, but assigning N particles to cells is only an $\mathcal{O}(N)$ operation. There is therefore little reason, at this stage, to devise complicated bookkeeping to determine which individual particles have crossed a cell boundary. The procedure is particularly simple if each cell stores a list of the particle indices that presently belong to it:

```
for cell in cells:
    cell.clear()

for i = 0,...,N-1:
    ix = floor((r[i].x + Lx/2) / l)
    iy = floor((r[i].y + Ly/2) / l)
    iz = floor((r[i].z + Lz/2) / l)

    ix = ix mod nx
    iy = iy mod ny
    iz = iz mod nz

    cells[ix,iy,iz].append(i)
```

Different conventions can be used to determine the cell index, provided they are consistent with the chosen coordinate system. Here, we assumed that the simulation box is centered at the origin, so adding $L_x/2$ shifts the coordinate from $[-L_x/2, L_x/2)$ to $[0, L_x)$. Dividing by the cell width l and taking the floor then gives the corresponding cell index. There are n_x cells into which the simulation box is divided along the x-direction. Thus, when $l = L_x/n_x$, this is equivalent to multiplying the shifted position by n_x/L_x, while the modulo operation ensures periodic wrapping at the boundaries. Note that we did not *a priori* assumed that particles are always inside of the box. There are situations were maintaining unwrapped coordinates is beneficial, *e.g.*, in computing mean squared displacements.

Having assigned the particles to cells, we can now use the cell list to evaluate the forces. In the following pseudocode, we iterate over the occupied cells and inspect only those particles in the same and neighboring cells:

```
force[:] = 0
potential = 0
virial = 0

for cell in cells:
    for i in cell:
```

```
for neighbor_cell in neighborhood(cell):
    for j in neighbor_cell:

        if j > i:
            dr = minimum_image(r[i] - r[j])
            r2 = dot(dr, dr)

            if r2 < rc2:
                fij, uij = pair_force_and_energy(dr, r2)

                force[i] += fij
                force[j] -= fij

                potential += uij
                virial += dot(dr, fij)
```

Note that the full set of neighboring cells need not always be searched. Cells whose closest possible point lies beyond the search radius may be excluded, which can be advantageous when interaction ranges differ substantially between particle types [14].

4.3.4 Combining Cell and Verlet Lists

You may now be wondering: "What is the point of using a Verlet list to begin with, if the cell list algorithm already gives favorable scaling?" The answer is that asymptotic scaling and actual run time are not the same thing. Cell lists can be used to construct a short Verlet list efficiently, and the force loop can then use that compact list for many time steps before another cell-based rebuild is required.

Consider a three-dimensional Lennard-Jones system with cut-off radius $r_c = 2.5\sigma$, where σ is the diameter. Suppose that the cell edge is $l = r_c$ and that a Verlet skin $r_s = 0.2\sigma$ is used, so $r_v = 2.7\sigma$. A direct cell search over the 27 surrounding cells examines on average $n_{\text{cell}} = 27\rho l^3$ candidate particles per central particle, whereas an ideal spherical Verlet neighborhood contains approximately

$$n_{\text{Verlet}} = \frac{4\pi}{3}\rho r_v^3. \tag{4.4}$$

For these values, $n_{\text{cell}}/n_{\text{Verlet}} \approx 5.1$. The precise factor depends on density (choice of r_s), geometry, and implementation, but the example illustrates why combining the two ideas can reduce the prefactor even though both methods have favorable large-N scaling.

4.3.5 Linked-Cell Data Structures

Thus far, we have talked almost exclusively about spatial decomposition. This tells us which particles should be grouped together. It leaves unexplored how those groups should be stored in memory to make optimal use of your compute infrastructure. The traditional *linked-cell list* uses two integer arrays [12, 15].

The purpose of the linked-list representation is to store an arbitrary number of particles in each cell using only two fixed-size arrays. Each cell needs only one piece of information, the index of one particle that belongs to it. From that particle, we can then follow a chain to all other particles in the same cell. The array **`head`** marks the beginning of this chain, while **`next`** tells us which particle follows particle i. A value of -1 indicates that there is no next particle and therefore marks the end of the chain.

The construction is easiest to understand by imagining particles being placed into a cell one at a time. When a new particle i is assigned to cell c, whatever particle was previously at the front of the chain is made the successor of i, as follows:

```
next[i] = head[c]
```

and the new particle is then made the beginning of the chain:

```
head[c] = i.
```

Thus, particles are simply inserted at the front of the list. No existing particle has to be moved in memory, and only two integer assignments are required for each insertion.

This may be abstract, so let us consider an example. Suppose particles 4, 9, and 2 are assigned, in that order, to the same cell. The resulting links are:

```
head[c] = 2
next[2] = 9
next[9] = 4
next[4] = -1
```

Starting from **`head[c]`** and repeatedly following **`next`** therefore visits every particle in the cell, without requiring the cell itself to store a separate array of particle indices. Summarizing, the full form of the algorithm is:

```
head[:] = -1
next[:] = -1

for i = 0,...,N-1:
    c = cell_index(r[i])

    next[i] = head[c]
    head[c] = i
```

Since each particle is assigned once, constructing the entire linked-cell structure is an $\mathcal{O}(N)$ operation. The ordering of particles within a cell is irrelevant, *i.e.*, only membership matters for the subsequent neighbor search. Integer arrays are compact, easy to serialize, and usually have better memory behavior[7]. For ordinary time-stepped MD, it is often simpler to rebuild **`head`** and **`next`** from scratch whenever particle-to-cell assignments need to be refreshed, than to maintain the lists by deleting and inserting individual particles as they cross cell boundaries. Since rebuilding requires only an $\mathcal{O}(N)$ pass over the particles, its cost is typically modest compared with the subsequent force calculation.

The above singly linked list does not support deletion in constant time unless the predecessor is known. If particles must change cells one at a time, as occurs naturally in EDMD, a doubly linked representation is more convenient. In addition to **`next[i]`**, one stores **`prev[i]`** and **`cell_of[i]`**. Removal and insertion are then both $\mathcal{O}(1)$ operations; we return to this point in Section 4.6.

[7]Linked cells are only one possible representation of a cell decomposition. Particle indices may instead be stored contiguously by cell, typically using cell counts and prefix sums to determine the array offsets. Such layouts often improve cache efficiency and vectorization, while GPU implementations use further specialized variants [14]. At larger scales, the same principle is used in domain decomposition across processors [16].

4.3.6 Storing and Building a Verlet List

Now that we have spoken about linked lists as an effective organizational structure, let us return to Verlet lists. There is a practical complication associated with Verlet lists that is easy to overlook. The list has a variable length. The number of neighbors of particle i is not known beforehand and, moreover, changes whenever the list is reconstructed. Thus, rebuilding a Verlet list means not only finding a new set of neighbors, but also deciding where these neighbors are to be stored in memory.

The most straightforward implementation is a list of lists, where **`neighbors[i]`** contains the indices j associated with particle i. This representation is convenient because particles may have different numbers of neighbors and entries can simply be appended during construction. It is therefore a good choice for your first implementation. However, you will find that the individual lists may have to grow dynamically as neighbors are added, memory may be allocated in many separate blocks, and this storage has to be reused or reconstructed whenever the Verlet list is rebuilt. This issue becomes more noticeable in a large simulation. Suppose, for example, that particle i has 42 neighbors at one rebuild and 51 at the next. If exactly 42 entries had been allocated previously, additional storage is now required. Allocating considerably more space than necessary avoids frequent resizing, but wastes memory. Thus, even though the neighbor search itself can have favorable scaling, memory management can introduce a non-negligible prefactor.

A common solution is to store all neighbor indices in one contiguous integer array together with an offset array of length $N + 1$. The neighbors of particle i then occupy:

```
neighbors[offset[i] : offset[i+1]]
```

The difficulty is that the offsets cannot be known until the number of neighbors of every particle has been established. Construction is therefore naturally performed in two passes. In the first pass we count the neighbors but do not yet store them:

```
count[:] = 0

build_linked_cells(r)

for each particle i:
    c = cell_index(r[i])

    for each unique cell cn in stencil(c):
        for each particle j in cn:

            if j > i:
                dr = minimum_image(r[i] - r[j])

                if dot(dr, dr) < rv2:
                    count[i] += 1
```

The counts determine how much storage each particle requires. Taking a prefix sum[8] gives the starting position of each particle's neighbor list:

```
offset[0] = 0
```

[8] A prefix sum converts a list of cell occupancies into starting offsets. If cell c contains n_c particles, its offset is the total number of particles in all preceding cells, $s_c = \sum_{k<c} n_k$. The particles belonging to cell c can then be stored contiguously in positions $s_c, \ldots, s_c + n_c - 1$ of a single array.

```
for i = 0,...,N-1:
    offset[i+1] = offset[i] + count[i]

resize neighbors to offset[N] entries
```

And the second pass fills that storage

```
write[:] = offset[0:N]

for each particle i:
    c = cell_index(r[i])

    for each unique cell cn in stencil(c):
        for each particle j in cn:

            if j > i:
                dr = minimum_image(r[i] - r[j])

                if dot(dr, dr) < rv2:
                    neighbors[write[i]] = j
                    write[i] += 1

store the current unwrapped positions as r_ref
```

The price of the contiguous representation is clear. Unless additional bookkeeping is introduced, the geometric search is performed twice, once to determine the required storage and once to fill it. This may nevertheless be worthwhile because the result is compact and can subsequently be traversed efficiently during every force calculation until the next rebuild:

```
force[:] = 0

for i = 0,...,N-1:

    for k = offset[i],...,offset[i+1]-1:

        j = neighbors[k]

        dr = minimum_image(r[i] - r[j])
        r2 = dot(dr, dr)

        if r2 < rc2:
            fij = pair_force(dr, r2)

            force[i] += fij
            force[j] -= fij
```

Since the Verlet list contains pairs out to r_v, the distance test against r_c is still required before evaluating the force.

In practice, you want to avoid allocating new memory at every rebuild. Instead, the neighbor array is given a certain capacity. If the new list fits inside the previously allocated storage, that memory is simply reused. Only when the required number of entries exceeds the capacity is a larger block allocated. A separate choice concerns whether the stored neighbor

list is a *half list* or a *full list*. In a half list, a pair (i, j) appears only once, for example with $j > i$, and the force routine updates both particles using Newton's third law. In a full list, both $i \to j$ and $j \to i$ are stored. The full list roughly doubles the storage, but can simplify parallel force accumulation because each particle may process its own neighbors independently. For the exercises in this chapter, we will use a half list.

For further implementation details and examples of working simulation programs, the treatments by Allen and Tildesley [12] and Rapaport [15] are very useful. More specialized production codes introduce vectorization, thread-level parallelism, distributed spatial decomposition, and hardware-specific neighbor structures. Once you have familiarized yourself with the concepts laid out in this chapter, you can start thinking about these topics.

Exercise: Accelerating MD Simulations Using a Cell List

In this exercise, we will construct and verify a simple two-dimensional MD simulation of particles interacting through the Weeks-Chandler-Andersen (WCA) potential [17], and then accelerate the force calculation using a cell list. The WCA potential is given by

$$\Phi_{\mathrm{WCA}}(r) = \begin{cases} 4\epsilon\left[\left(\dfrac{\sigma}{r}\right)^{12} - \left(\dfrac{\sigma}{r}\right)^{6}\right] + \epsilon, & r < r_c \\ 0, & r \geq r_c \end{cases}, \tag{4.5}$$

with $r_c = 2^{1/6}\sigma$. We express all quantities in terms of the particle size σ, the strength of the potential ϵ, the particle mass m, and the Boltzmann constant k_{B}. We will use reduced units throughout, *i.e.*, $\sigma = \epsilon = m = k_{\mathrm{B}} = 1$. A convenient starting system is $N = 256$ particles placed on a square lattice in a square box of edge length $L = \sqrt{N/\rho}$, and time step $\Delta t = 0.002$. Unless stated otherwise, you can use number density $\rho = N/L^2 = 0.5$ and initial temperature $T_0 = 1$. To obtain this temperature, you can draw velocities from a Gaussian distribution, subtract the center-of-mass velocity, and rescale once to obtain T_0. The subsequent dynamics is microcanonical (NVE). You will not need to implement a thermostat for this exercise and should expect to find temperature fluctuations.

(a) Derive the pair force corresponding to Eq. (4.5). Implement it first in a direct pair loop. For later comparison, have the force routine return the forces, total potential energy, and virial contribution $\sum_{i<j} \boldsymbol{r}_{ij} \cdot \boldsymbol{f}_{ij}$.

(b) Implement periodic boundary conditions using the minimum-image convention. Write this in terms of the system dimension d rather than hard-coding a two-dimensional expression. Verify the distance routine separately using particles placed close to opposite sides of the box.

(c) Complete the initialization and Velocity-Verlet update. The instantaneous kinetic temperature after removal of the center-of-mass motion may be estimated[9] from

$$T = \frac{2K}{(d(N-1)k_{\mathrm{B}}}, \tag{4.6}$$

where K is the kinetic energy. Explain why the production trajectory belongs to the NVE rather than the NVT ensemble even though a target temperature was used to initialize the velocities.

[9] This follows from equipartition, according to which each independent quadratic velocity degree of freedom contributes $k_{\mathrm{B}}T/2$ to the mean kinetic energy. For N particles in d dimensions there are initially dN translational degrees of freedom, but removing the center-of-mass velocity imposes d constraints, one for each spatial direction, which is why there is a term $N - 1$ in the denominator.

(d) Output particle coordinates occasionally and visualize the trajectory. Do not write away coordinates every time step, as this level of output can easily dominate the run time and fill up your hard drive. Check that particles cross periodic boundaries continuously and that no systematic drift of the center of mass develops.

(e) Verify the integrator before optimizing it. Record kinetic, potential, and total energy after a short equilibration over the same physical production time for each calculation (*e.g.*, $t_{\mathrm{prod}} = 40$ in reduced units). Repeat the calculation using $\Delta t = 0.004$, 0.002, and 0.001. Plot the relative energy deviation $[E(t) - E(0)]/|E(0)|$. In addition, compute a scalar measure of the energy error over the production interval, for example

$$\Delta_E^{\mathrm{rms}} = \left\langle \left(\frac{E(t) - E(0)}{E(0)} \right)^2 \right\rangle^{1/2} . \tag{4.7}$$

Show that reducing the time step improves energy conservation and verify that, in the converged regime, the characteristic energy error decreases approximately as Δt^2, as expected for velocity Verlet.

(f) Compute the mean pressure from the virial expression

$$\langle p \rangle = \frac{N k_{\mathrm{B}} \langle T \rangle}{V} + \frac{1}{V d} \left\langle \sum_{i<j} \boldsymbol{r}_{ij} \cdot \boldsymbol{f}_{ij} \right\rangle , \tag{4.8}$$

with $V = L^d$ and the angular brackets denoting a time average taken after equilibration. Report the mean temperature and pressure together with an estimate of their statistical variation. At this stage the direct pair implementation serves as the reference calculation. You will use the same observables below to verify the accelerated algorithm.

(g) Benchmark the direct pair calculation. Disable visualization and file output, time only the part of the code needed to advance the simulation, and repeat each timing several times. Use a sequence such as $N = 64$, 128, 256, 512, and 1,024 while keeping ρ fixed. Plot the time per step against N on logarithmic axes and fit the large-N slope of $\log t$ versus $\log N$, see Fig. 4.3.

(h) Implement the geometry of a cell list first. Choose the number of cells per direction as $n_c = \lfloor L/r_c \rfloor$ and define $l = L/n_c$, so that $l \geq r_c$. Write separate functions that map a wrapped coordinate to a cell index and that return the neighboring cells under periodic boundary conditions[10].

(i) Implement a linked-cell representation using integer arrays **`head`** and **`next`**. Rebuild the arrays from scratch from the current positions using the pseudocode in the text. Traverse the particle chains and rewrite the force calculation so that only the current and neighboring cells are searched. Make sure that each pair contributes exactly once. For debugging purposes, count both the number of candidate pairs examined and the number that actually satisfy $r < r_c$.

(j) Verify the linked-cell implementation on fixed configurations before comparing long trajectories. That is, compare the force on every particle, the potential energy, and the virial obtained from the direct and linked-cell routines. The differences should be at

[10]Warning: if you choose to have $n_c = 2$ you will run into issues of uniqueness in a periodic simulation box. This is kind of an edge case and you typically will want more cells to make optimal use of the acceleration provided by cell lists. However, we bring it up here, because you may become confused if you try to set up cell lists in a small system for testing purposes.

the level expected from numerical precision. Next, run both versions from the same initial state and verify agreement of thermodynamic observables over a short trajectory. Long trajectories are not required to remain bit-for-bit identical as small floating-point differences grow in a chaotic many-particle system.

(k) Repeat the scaling measurement from part (g) using the linked-cell force calculation. Report the fitted exponent and the approximate system size at which the accelerated method becomes faster than the direct pair loop on your computer. Time the cell rebuild and the force traversal separately. Explain why the crossover is hardware- and implementation-dependent even though the asymptotic scaling is not.

(l) Combine the linked-cell structure with a Verlet list. Choose a skin r_s, set $r_v = r_c + r_s$, and choose the cell stencil for the search radius r_v. Build a half neighbor list with $j > i$, store the unwrapped coordinates at rebuild, and rebuild when the maximum displacement exceeds $r_s/2$. Verify on stored configurations that the forces are identical to the direct pair calculation even immediately before a rebuild.

(m) For one representative state point, vary r_s over several values. Measure the mean neighbor-list length, the mean number of time steps between rebuilds, the fraction of wall time spent rebuilding, and the total time per MD step. Explain the existence of an optimal range of skin sizes rather than a universally optimal value.

(n) Optional implementation study: replace `head`/`next` by contiguous cell storage using particle counts and prefix sums. The physical algorithm is unchanged, so the forces must remain identical. Compare the run time of the two data layouts and comment on whether the difference is visible.

4.4 Nonequilibrium MD: Couette Flow

So far we have considered equilibrium MD, where the main purpose of the simulation was to reproduce the dynamics of a system in the absence of sustained external driving. We now turn to a simple nonequilibrium problem, planar Couette flow. Consider a fluid confined between two parallel walls. The upper wall moves with velocity $+v_w\hat{\boldsymbol{x}}$, while the lower wall moves with $-v_w\hat{\boldsymbol{x}}$. Momentum is transferred from the walls to the fluid and, once a stationary state has been reached, a velocity gradient develops across the channel.

In the continuum description of a Newtonian fluid, the velocity profile away from the molecular structure of the walls is linear,

$$u_x(z) = \dot{\gamma} z + u_0, \tag{4.9}$$

where $\dot{\gamma} = \partial u_x/\partial z$ is the shear rate. For two equivalent walls placed symmetrically about $z = 0$ and moving with equal and opposite speeds, symmetry implies[11] $u_0 = 0$. If the fluid obeyed a perfect no-slip boundary condition at walls separated by a distance H, the corresponding shear rate would be

$$\dot{\gamma}_0 = \frac{2v_w}{H}. \tag{4.10}$$

This serves as the reference, clearly an MD simulation is not expected to recover this profile exactly. For example, at molecular scales the fluid may slip relative to the wall, in which case the magnitude of the measured shear rate is smaller than the no-slip value.

[11] You may numerically find a small offset when both walls have the same kind of hydrodynamic boundary condition. When they are not hydrodynamically equivalent, there could be a more significant offset. Hence, we introduce this concept here and explore it further in the exercise, rather than ignore it on symmetry grounds.

There is an important microscopic detail hidden in the apparently simple statement that a wall "moves". Suppose that the wall is represented only by a perfectly smooth potential $\Phi_{\text{wall}}(z)$. The associated force

$$\boldsymbol{F}_{\text{wall}} = -\frac{\mathrm{d}\Phi_{\text{wall}}}{\mathrm{d}z}\hat{\boldsymbol{z}}, \tag{4.11}$$

is entirely normal to the surface. Translating such a potential in the x direction does not provide a mechanism by which x-momentum can be transferred to the fluid. A smooth wall can confine the particles, but it cannot by itself drive Couette flow. We therefore represent the walls microscopically by particles arranged on a regular lattice. The resulting corrugated interaction potential has both normal and tangential components, allowing momentum to be exchanged between the moving wall and the fluid. The wall particles are not treated as ordinary dynamical particles. Instead, their trajectories are prescribed. For a wall particle a we write $\boldsymbol{R}_a(t) = \boldsymbol{R}_a(0) + \boldsymbol{V}_a t$, with $\boldsymbol{V}_a$ appropriately chosen, *e.g.*, $+v_w\hat{\boldsymbol{x}}$ on the upper wall. The wall particles' x and y coordinates are wrapped periodically in the same way as those of the fluid particles. Their z coordinates remain fixed. The wall-fluid force is evaluated in the usual manner, but only the fluid particle is advanced by velocity Verlet.

4.4.1 Removing the Viscous Heat

The moving walls continuously perform work on the fluid. In a stationary shear flow this work is converted into heat by viscous dissipation. An unthermostatted simulation will therefore heat up rather than settle into a stationary state. This presents an annoyingly subtle problem. A thermostat that independently damps the laboratory-frame velocity of every particle may also damp the collective flow that we are trying to generate. For hydrodynamic simulations it is therefore desirable to employ a thermostat that preserves momentum and is Galilean invariant. Here, we will use the Lowe-Andersen thermostat [18] for convenience. Related momentum-conserving thermostats have been discussed by Stoyanov and Groot [19].

The Lowe-Andersen thermostat acts on *pairs* of particles rather than on individual velocities. Consider two equal-mass fluid particles i and j, and introduce $\boldsymbol{r}_{ij} = \boldsymbol{r}_i - \boldsymbol{r}_j$ and $\hat{\boldsymbol{r}}_{ij} = \boldsymbol{r}_{ij}/r_{ij}$, as well as $\boldsymbol{v}_{ij} = \boldsymbol{v}_i - \boldsymbol{v}_j$. Only the component of the relative velocity along the line joining the particles is thermostatted

$$v_{ij}^{\parallel} = \boldsymbol{v}_{ij} \cdot \hat{\boldsymbol{r}}_{ij}. \tag{4.12}$$

With a prescribed probability, this quantity is replaced by a new value $v_{ij}^{\parallel\prime}$ drawn from the equilibrium distribution. For two particles of equal mass m

$$v_{ij}^{\parallel\prime} \sim \mathcal{N}\left(0, \frac{2k_{\mathrm{B}}T}{m}\right), \tag{4.13}$$

where the second argument denotes the variance of the Gaussian distribution.

Defining $\Delta v_{ij}^{\parallel} = v_{ij}^{\parallel\prime} - v_{ij}^{\parallel}$, the two particle velocities are changed according to

$$\boldsymbol{v}_i' = \boldsymbol{v}_i + \frac{1}{2}\Delta v_{ij}^{\parallel}\hat{\boldsymbol{r}}_{ij}, \tag{4.14}$$

$$\boldsymbol{v}_j' = \boldsymbol{v}_j - \frac{1}{2}\Delta v_{ij}^{\parallel}\hat{\boldsymbol{r}}_{ij}. \tag{4.15}$$

You can directly see why this is useful by adding the two equations. The total momentum of the pair is unchanged

$$m\boldsymbol{v}_i' + m\boldsymbol{v}_j' = m\boldsymbol{v}_i + m\boldsymbol{v}_j. \tag{4.16}$$

At the same time, the thermostat depends only on the relative velocity $\boldsymbol{v}_{ij}$. Adding the same constant velocity to all particles therefore leaves the thermostat operation unchanged. This

makes the thermostat Galilean invariant, *i.e.*, it does not single out the laboratory frame or directly oppose a uniform streaming velocity.

In practice, the stochastic operation is applied only to pairs of fluid particles that are sufficiently close together. Let r_T denote the thermostat range and Γ the collision frequency. During a time step Δt, each eligible pair is thermostatted with probability[12] $1 - \exp(-\Gamma \Delta t)$. A simple implementation for equal masses is given in the following pseudocode:

```
for each fluid-fluid pair (i,j) with rij < rT:

    if random_uniform(0,1) < 1 - exp(-Gamma * dt):

        dr = minimum_image(r[i] - r[j])
        rhat = dr / sqrt(dot(dr,dr))

        vij_parallel = dot(v[i] - v[j], rhat)

        vij_new = random_normal( mean = 0,  std = sqrt(2*kB*T/m) )

        dv = vij_new - vij_parallel

        v[i] += 0.5 * dv * rhat
        v[j] -= 0.5 * dv * rhat
```

For convenience, r_T may be chosen equal to the interaction cut-off r_c, which also means that the same cell or neighbor list can be used to identify candidate thermostat pairs.

The parameter Γ is really not meant to be seen as a numerical knob that you can turn at your leisure. Increasing Γ changes the rate at which stochastic momentum is exchanged between neighboring particles and therefore also changes transport properties such as the viscosity. The thermostat is momentum conserving, but it impacts the dynamics. You can determine these properties seprately and you will want to check that your observables do not depend on these in an unexpected manner. Lastly, note that we apply the thermostat only to fluid-fluid pairs. Wall-fluid pairs should not be thermostatted! The moving walls are performing work on the fluid, thereby supplying momentum. The role of the thermostat is exclusively to remove the resulting heat from the fluid.

4.4.2 Temperature in a Flowing Fluid

There is one further point that deserves attention before turning to the exercise. In a flowing system, the total particle velocity contains both thermal motion and collective motion. The kinetic energy K_c associated with the approximate Couette profile that forms should not be interpreted as heat. That is, we need to determine the average fluid velocity $\bar{\boldsymbol{u}}$ and subtract that from the particle motion to determine the relevant contribution to the kinetic energy, say K_{kin}. We expect $\bar{\boldsymbol{u}}$ to have the form introduced in Eq. (4.9), though there will be variations due to ordering close to the wall. Once the average (local) velocity profile has been measured, the temperature of the fluid can be estimated using

$$T_{\text{kin}} = \frac{m}{3Nk_{\text{B}}} \sum_{i=1}^{N} |\boldsymbol{v}_i - u_x(z_i)\hat{\boldsymbol{x}}|^2 \,, \tag{4.17}$$

[12]The collision probability follows from assuming that thermostat events occur as a Poisson process with rate Γ. The probability that no event occurs during a time interval Δt is $e^{-\Gamma \Delta t}$, so the probability of at least one event is $1 - e^{-\Gamma \Delta t}$. For $\Gamma \Delta t \ll 1$, this becomes approximately $\Gamma \Delta t$, simply rate times time.

where we made use of the translational invariance of the system and equipartition. As we will see in the upcoming exercise, this is an approximation.

4.4.3 Measuring the Velocity Profile and Slip

The hydrodynamic velocity is a local statistical average rather than the velocity of any individual particle. To obtain it, we divide the channel into bins along z and accumulate both the number of fluid particles and the sum of their x velocities in each bin, over multiple points in time at which we sample. For bin k, we have the expression

$$\bar{u}_x(z_k) = \frac{\sum_{\text{samples}} \sum_{i \in k} v_{i,x}}{\sum_{\text{samples}} N_k}. \tag{4.18}$$

This approach is preferable to averaging a sequence of instantaneous bin averages, particularly when the number of particles occupying a bin fluctuates. Note, however, that this can only realistically be done when the fluid profile is sufficiently stationary.

Close to a molecular wall the density and velocity can oscillate strongly because the fluid develops distinct molecular layers. The continuum description is not expected to hold within this region. The shear rate is therefore best obtained by fitting the approximately linear portion of the velocity profile away from both walls to Eq. (4.9). If the fluid velocity at the wall differs from the wall velocity, the difference may be characterized by a slip length. Extrapolating the linear bulk profile toward the position of, for example, the top wall z_{wall}, we define the slip length as

$$b = \frac{|v_{\text{wall}} - u_x(z_{\text{wall}})|}{|\dot{\gamma}|}. \tag{4.19}$$

Geometrically, b is the distance beyond the imposed wall position at which the extrapolated fluid profile would attain the target wall velocity. For the present exercise we use the position of the wall lattice plane as setting the continuum boundary condition.

Exercise: Couette Flow

Take the verified cell-list MD code from the previous exercise and generalize it to three dimensions. Continue to use reduced units, and employ periodic boundaries in x and y but not in z. A suitable starting system has $L_x = L_y = 8$, $H = 16$, $\rho = 0.5$, $T = 1$, and $\Delta t = 0.002$. Be careful with the time step, though, as before, establishing numerical stability and convergence is part of the exercise.

(a) Construct two stationary atomistic walls. Use one or more layers of wall particles arranged on a regular lattice in the xy plane, with a lattice spacing of order σ. Place the innermost wall planes a distance H apart. Let wall and fluid particles interact through the WCA potential. The wall coordinates are prescribed and should not be advanced by velocity Verlet. Equilibrate the confined fluid with the walls initially at rest. Verify that fluid particles cannot cross either wall and inspect the number density $\rho(z)$. You should expect to find oscillatory layering close to the solid surfaces.

(b) Set the upper and lower wall velocities with $v_w = 0.4$ as an initial value. At each MD step, update the wall coordinates according to their prescribed velocity and wrap them periodically in x and y. The fluid feels the wall-fluid forces, but the reaction force is not used to integrate the wall motion. Explain why the momentum and total energy of the fluid alone are therefore no longer conserved.

(c) Introduce a Lowe-Andersen thermostat acting only between fluid particles. Take $r_T = r_c$ initially and choose, for example, $\Gamma = 1$ in reduced units. Apply the thermostat after the conservative velocity Verlet update. For each eligible pair, use the collision probability $1-\exp(-\Gamma\Delta t)$ and implement the pairwise procedure provided above. Verify numerically that every thermostat event conserves the momentum of the participating pair. Do not thermostat wall-fluid pairs.

(d) Start the wall motion and allow the system to approach a nonequilibrium steady state before collecting production data. Divide the channel into $N_b = 20$ to 40 bins in the z direction and monitor the evolving streaming profile $u_x(z)$. Give the system time to establish momentum transport across the entire channel. Decide from the measured velocity profiles when the initial transient has decayed sufficiently.

(e) Use the above procedure to average the fluid profile and plot the resulting $u_x(z)$. Do not include wall particles. Away from the molecularly layered regions next to the walls, the profile should be approximately linear. Adjust H if you cannot see this.

(f) Fit the central part of the profile to $u_x(z) = \dot{\gamma}z + u_0$. For equivalent upper and lower boundaries (in terms of their respective hydrodynamic boundary conditions), verify that u_0 is small compared with v_w. Compare the measured $\dot{\gamma}$ with the no-slip reference and explain physically why $\dot{\gamma}$ can differ from $\dot{\gamma}_0$.

(g) Use the measured streaming profile to estimate the kinetic temperature according to Eq. (4.17). Verify that it fluctuates around the target value $T = 1$.

(h) Estimate the slip length at the upper and lower walls using Eq. (4.19). For equivalent walls, the two estimates of b should agree within statistical uncertainty. Discuss possible reasons if they do not.

(i) Change one microscopic property of the wall. Suitable choices include the wall lattice spacing, the areal density of wall particles, or the wall-fluid interaction strength. Repeat the measurements of $u_x(z)$, $\dot{\gamma}$, and b. Explain how changing the microscopic structure or interaction of the wall changes the effective hydrodynamic boundary condition. Compare your observations with the molecular-scale discussion of slip in Ref. [5].

(j) Establish that the measured flow profile is not a numerical artifact. At minimum, (i) repeat part of the calculation using a smaller time step, (ii) increase the production time, (iii) repeat the analysis using a different number of spatial bins, and (iv) divide the production trajectory into several time blocks and use the variation between the corresponding profiles to estimate the statistical uncertainty. Note that increasing the number of spatial bins improves spatial resolution but simultaneously reduces the number of particles contributing to each bin. Thus, finer spatial resolution generally comes at the cost of larger statistical noise.

(k) Finally, repeat the calculation for a smaller wall speed. Compare $u_x(z)/v_w$ and the measured slip length with the results obtained at $v_w = 0.4$. In the linear-response regime, reducing the driving should not strongly change the normalized velocity profile or the slip length. A pronounced dependence on v_w may indicate nonlinear rheological behavior, significant viscous heating, or inadequate equilibration. As an additional check, repeat one calculation with a different value of Γ. The thermostat should control the temperature, but transport properties need not be exactly independent of Γ, since the Lowe-Andersen collisions themselves contribute to momentum transport.

4.5 Accuracy, Stability, and the Cost of Time Integration

We briefly return to the question of time integration before considering EDMD. In principle, it would seem natural to use an integration algorithm with as high an order as possible. After all, if one method has an error that decreases as Δt^2 and another's error scales as Δt^4, then the latter would appear to be superior. However, in considering MD, drawing this conclusion is too simple. You should instead ask yourself: "How accurately is a trajectory generated for a given computational cost and over the time scales of interest?"

A toy model for understanding this point is the one-dimensional harmonic oscillator, with position coordinate $x(t)$. The evolution equation is given by $m\ddot{x} = -kx$, with the double dot denoting the time derivative and k the spring constant. The associated angular frequency is $\omega = \sqrt{k/m}$. For the initial conditions $x(0) = x_0$ and $v(0) = v_0$, the exact trajectory is known

$$x(t) = x_0 \cos(\omega t) + \frac{v_0}{\omega} \sin(\omega t). \tag{4.20}$$

This is what makes this system particularly nice to test time updates. Any difference between the numerical and exact trajectories can be attributed directly to the integration algorithm.

For the usual velocity Verlet algorithm[13], you have learned about earlier in this book, one time step can be written as:

```
v_half = v + 0.5 * dt * force(x) / m
x = x + dt * v_half
f_new = force(x)
v = v_half + 0.5 * dt * f_new / m
```

After the initial force has been established, only one new force evaluation is required per time step. The algorithm is second order, so at a fixed physical time the trajectory error decreases as $E_{\mathrm{VV}} \propto \Delta t^2$ for sufficiently small Δt.

For comparison, consider another well-known algorithm, the fourth-order Runge-Kutta method (RK4). We write the state of the oscillator using a vector that combines position and velocity $\boldsymbol{y} = (x, v)$. The associated equation of motion is given by

$$\dot{\boldsymbol{y}} = \boldsymbol{g}(\boldsymbol{y}) = \left(v, -\omega^2 x\right). \tag{4.21}$$

A single RK4 step is can then be written as follows using pseudocode:

```
k1 = g(y)
k2 = g(y + 0.5 * dt * k1)
k3 = g(y + 0.5 * dt * k2)
k4 = g(y + dt * k3)

y = y + dt * (k1 + 2*k2 + 2*k3 + k4) / 6
```

and the corresponding global error decreases as $E_{\mathrm{RK4}} \propto \Delta t^4$. At the same value of Δt, RK4 can therefore be much more accurate than velocity Verlet.

However, this naive comparison is not entirely fair. Evaluating $\boldsymbol{g}$ requires evaluating the force, and RK4 performs four such evaluations per step. In a realistic MD simulation it is the force calculation, rather than the few arithmetic operations involved in updating the positions, that typically dominates the computational cost. One should therefore also ask how the algorithms compare for the same number of force evaluations.

[13] Note how this form of velocity Verlet closely resembles the leapfrog algorithm. The velocity is advanced by half a time step, the position by a full step, and the velocity by another half step. The difference is mainly one of bookkeeping, since leapfrog usually stores velocities at half-integer times whereas velocity Verlet also provides velocities at the same integer times as the positions.

4.5.1 Stability and Long-Time Dynamics

There is a second issue that formal order does not capture. A numerical method must remain stable for the fastest motion present in the system. For velocity Verlet applied to the harmonic oscillator, stability requires $\omega\Delta t < 2$. This bound is easy to derive, consider $\ddot{x} = -\omega^2 x$. Then, discrete positions obey $x_{n+1} = 2(1 - \omega^2\Delta t^2/2)x_n - x_{n-1}$ and the condition follows from the requirement that oscillations remain bounded. In practice one normally chooses a substantially smaller time step to obtain good accuracy as well as stability. Note that there is something else to be learned here; the relevant dimensionless quantity is $\omega\Delta t$. Increasing the stiffness k increases ω and therefore reduces the time step that can be used.

This has a direct analogue in molecular simulations. A system may contain processes occurring on very different time scales. One may be interested, for example, in molecular diffusion over a long time while the simulation also contains a rapidly vibrating bond. The integration time step must nonetheless be sufficiently small to resolve the rapid vibration. A higher formal order does not remove the physical time scale associated with this fastest degree of freedom.

velocity Verlet has a final feature that is not expressed by its $\mathcal{O}(\Delta t^2)$ accuracy. It is a symplectic and time-reversible integrator, thus suited to study conservative dynamics. That is, the energy error in a stable simulation generally oscillates around a bounded value rather than simply accumulating in one direction. The numerical trajectory does not exactly conserve the original Hamiltonian, but it behaves as though it were following a nearby, slightly modified Hamiltonian. By contrast, standard RK4, is not symplectic. Over short times its trajectory may be substantially more accurate because of the $\mathcal{O}(\Delta t^4)$ scaling, but over very long integrations RK4's energy does not exhibit the same bounded behavior. This is one reason why the order of an integrator alone is a poor measure of its usefulness for MD. Accuracy per step, number of force evaluations, stability, reversibility, and long-time behavior must all be considered together.

Exercise: Higher Order Means Better MD?

Consider a one-dimensional harmonic oscillator in reduced units, $m = 1$, $k = 1$, $x(0) = 1$, and $v(0) = 1$. The angular frequency and period are therefore $\omega = 1$ and $\tau = 2\pi$, respectively. Implement both velocity Verlet and RK4.

(a) Integrate the oscillator for one complete period using $\Delta t = \tau/10$, $\tau/20$, $\tau/40$, $\tau/80$, $\tau/160$. For each calculation, determine the position error at $t = \tau$, which is given by

$$E_x = |x_{\text{num}}(\tau) - x_{\text{exact}}(\tau)| \,. \tag{4.22}$$

Plot E_x as a function of Δt on logarithmic axes for both algorithms. Fit the small-Δt region to $E_x \propto \Delta t^p$ and verify that $p \approx 2$ for velocity Verlet and $p \approx 4$ for RK4.

(b) The previous comparison gives both algorithms the same time step, but not the same computational cost. Count the number of force evaluations used by each method. Repeat the comparison at approximately equal numbers of force evaluations. For example, if RK4 uses a time step Δt, compare it with a velocity-Verlet calculation using approximately $\Delta t/4$. Which algorithm now gives the smaller trajectory error for the same approximate force-evaluation budget?

(c) Now examine a much longer trajectory. Integrate for at least 10^3 oscillator periods and calculate the total energy

$$E(t) = \frac{1}{2}mv(t)^2 + \frac{1}{2}kx(t)^2. \tag{4.23}$$

For both integrators, plot

$$\frac{E(t) - E(0)}{E(0)} \tag{4.24}$$

as a function of time. Choose time steps for which both methods appear accurate over the first few periods. Compare the subsequent long-time behavior. Does the energy error remain bounded? Does it drift systematically? Does the method that gave the smallest short-time trajectory error necessarily give the most faithful long-time energy behavior?

(d) Finally, examine the role of the fastest physical time scale. Repeat the velocity-Verlet calculation for oscillators with $\omega = 1$, 5, and 10, while initially keeping Δt fixed. Determine when the numerical solution becomes inaccurate or unstable. Then repeat the calculations while keeping $\omega\Delta t$ fixed instead. Explain the resulting behavior.

What you definitely should not take away from the above is that velocity Verlet is always the best integration algorithm. A useful MD integrator must provide sufficient accuracy at an acceptable cost, remain stable for the fastest motion in the system, and reproduce the qualitative long-time properties of Hamiltonian dynamics. Many higher-order symplectic algorithms exist and may be advantageous in specific situations.

4.6 Event-Driven Molecular Dynamics

In the MD algorithms considered thus far, the interaction forces are finite functions of the particle coordinates and the trajectories can be approximated by advancing the system through small time steps. The original MD simulations of simple liquids followed precisely this philosophy [20]. However, discontinuous potentials are an important class of model interactions, for which this is not the natural approach.

Two widely used examples are the hard-sphere and square-well potentials, which read

$$\Phi_{\mathrm{HS}}(r) = \begin{cases} \infty, & r < \sigma \\ 0, & r \geq \sigma \end{cases} \tag{4.25}$$

and

$$\Phi_{\mathrm{SW}}(r) = \begin{cases} \infty, & r < \sigma \\ -\epsilon, & \sigma \leq r < \sigma(1+\lambda) \\ 0, & r \geq \sigma(1+\lambda) \end{cases}, \tag{4.26}$$

respectively, see Fig. 4.4. Here, σ is the particle diameter, $\lambda > 0$ the reduced width of the well, and $\epsilon > 0$ its attraction strength. These simple models have been used extensively in studies of colloidal gels, nucleation, and glass formation [21, 22, 23].

For a hard sphere, it is better not to think in terms of an ordinary force evaluated at contact. The hard-core constraint produces an impulsive change of momentum at the instant of collision. Between collisions, the spheres travel along exactly known straight-line trajectories. At a collision, the velocities change instantaneously according to conservation of momentum and kinetic energy. Decreasing a conventional MD time step does not solve this problem. Unless the hard interaction has first been replaced by a steep but finite potential, there is no finite force to integrate over a small interval.

Equilibrium configurations of hard particles can be sampled efficiently using Monte Carlo (MC) methods [11]. Standard MC moves, however, do not represent Newtonian physical time. If the dynamics and transport are required, event-driven molecular dynamics provides the natural alternative. The method was introduced in early hard-sphere simulations by Alder

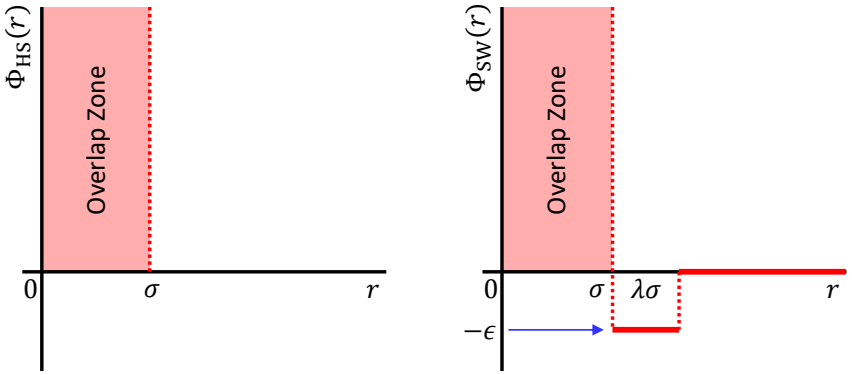

Figure 4.4: The hard-sphere (left) and square-well (right) potentials, $\Phi_{\mathrm{HS}}(r)$ and $\Phi_{\mathrm{SW}}(r)$. The shaded region indicates forbidden overlap. The square-well model adds an attractive region of width $\lambda\sigma$ and depth ϵ outside the hard core. Because the potential changes discontinuously, the dynamics is most naturally described in terms of collision events rather than a finite force evaluated at regular time steps.

and Wainwright [24]. In it, the algorithm advances the system from one collision event to the next. At each step, the time of the next collision is determined, the system is propagated directly to that instant, and the collision is processed before the procedure is repeated.

4.6.1 Pair-Collision Time and Update Rule

Consider two spheres i and j of equal diameter σ with positions $\boldsymbol{r}_i$ and $\boldsymbol{r}_j$ and velocities $\boldsymbol{v}_i$ and $\boldsymbol{v}_j$. Introduce the relative vectors $\boldsymbol{r}_{ij} = \boldsymbol{r}_i - \boldsymbol{r}_j$ and $\boldsymbol{v}_{ij} = \boldsymbol{v}_i - \boldsymbol{v}_j$ as before. Additionally, introduce the scalar value $b_{ij} = \boldsymbol{r}_{ij} \cdot \boldsymbol{v}_{ij}$. Between events, we can update positions according to $\boldsymbol{r}_{ij}(t) = \boldsymbol{r}_{ij} + t\boldsymbol{v}_{ij}$. A collision occurs when $|\boldsymbol{r}_{ij}(t)| = \sigma$. Solving the resulting quadratic gives the earliest candidate collision time

$$\tau_{ij} = \frac{-b_{ij} - \sqrt{b_{ij}^2 - v_{ij}^2(r_{ij}^2 - \sigma^2)}}{v_{ij}^2}. \tag{4.27}$$

Be careful though, a future collision exists only when the particles are approaching, *i.e.*, $b_{ij} < 0$, in which case the discriminant is non-negative, and the resulting τ_{ij} is positive. Additionally, overlaps must be excluded during initialization, otherwise you will obtain rather unusual behavior. The limiting case of a zero discriminant corresponds to a grazing contact.

At contact, define the unit normal $\hat{\boldsymbol{n}} = \boldsymbol{r}_{ij}/\sigma$, pointing from particle j to particle i. For equal masses and a frictionless elastic collision, it is then easy to find $\boldsymbol{v}_i' = \boldsymbol{v}_i - (\boldsymbol{v}_{ij} \cdot \hat{\boldsymbol{n}})\hat{\boldsymbol{n}}$ and $\boldsymbol{v}_j' = \boldsymbol{v}_j + (\boldsymbol{v}_{ij} \cdot \hat{\boldsymbol{n}})\hat{\boldsymbol{n}}$, where the primes denote the quantities after collision. That is, only the normal component of the relative velocity changes during an update. The tangential components remain unchanged because we do not consider rotations and tangential impulses.

Lastly, it is important to note that events are not equally spaced in physical time. An observable A sampled after every event cannot therefore be averaged as a simple arithmetic mean if the desired quantity is a time average. For piecewise-constant quantities, the correct average over total time T is

$$\overline{A} = \frac{1}{T}\sum_k A_k \Delta t_k, \tag{4.28}$$

where Δt_k is the interval for which the value A_k applies. The same idea must be applied carefully when accumulating spatially resolved observables.

4.6.2 The Event Loop

A basic EDMD algorithm is as follows. From the current configuration, determine the future collision time for every admissible particle pair (and, if applicable, for every particle-wall pair). Select the smallest positive time τ^*. Advance all particles ballistically by τ^*, perform the corresponding collision update, and repeat. In your first implementation of this algorithm, it is perfectly acceptable to search all candidate events and simply retain the smallest value. The resulting algorithm is expensive because all pair-collision times are reconsidered after each event, the exercise therefore encourages you to keep the number of particles limited.

Production-level implementations overcome this issue by making use of a suitable structure. They maintain an *event calendar* and recompute only events invalidated by the particles involved in the most recent collision. Spatial cell structures can further restrict which particles can collide, and cell-boundary crossings then become events in their own right. Collision trees and related event-bookkeeping structures provide another route to efficient EDMD [25, 15].

Let us now implement an event calendar. The naive EDMD algorithm has an obvious inefficiency, *i.e.*, after every collision you discard all previously calculated collision times and determine them again from scratch. This is unnecessary. A collision changes the trajectories only of the particles that take part in it. All other particles continue along exactly the same trajectories that were assumed when their future collisions were predicted. In an event calendar, each predicted collision is stored together with the absolute time at which it should occur, and the calendar is ordered such that the earliest event can be retrieved first. In practice, a priority queue is a natural data structure for this purpose, since it allows new events to be inserted while keeping efficient access to the event with the smallest time.

Suppose that a collision between particles i and j is predicted to occur at time t_{ij}. Before that time is reached, particle i may collide with a third particle k. The collision with k changes the velocity of particle i. The original prediction for the i-j collision was based on the old velocity of i and is therefore no longer valid. More generally, after the i-k collision, any previously predicted event involving either i or k may be discarded, whereas events involving neither particle remain unaffected.

The difficulty lies in efficiently identifying the corresponding obsolete events inside the event calendar. Searching through the complete calendar after every collision would defeat part of the purpose of maintaining it. A simple solution is therefore to associate an integer `version[i]` with every particle, initially set to zero. Whenever a collision changes the trajectory of a particle, its version number is increased by one. When an event is predicted, the current version numbers of the particles involved are stored together with the event:

```
event.time = predicted collision time
event.i = i
event.j = j

event.version_i = version[i]
event.version_j = version[j]

push event into calendar
```

When the event eventually reaches the front of the calendar, the stored version numbers are compared with the current values. If they still agree, neither particle has changed trajectory since the event was predicted and the event may be executed. If either version number differs, at least one of the trajectories used to calculate the event has become obsolete and the event can simply be discarded. This strategy avoids repeatedly searching through the event calendar whenever a collision invalidates earlier predictions and is commonly referred to as *lazy invalidation*. The corresponding event loop is:

```
predict initial events
store them in the event calendar

while time < t_end:
    event = remove earliest event from calendar

    i = event.i
    j = event.j

    if event.version_i != version[i]:
        continue

    if event.version_j != version[j]:
        continue

    advance system to event.time

    perform collision between i and j

    version[i] += 1
    version[j] += 1

    predict new events involving i
    predict new events involving j
```

To make this procedure more concrete, consider four particles, labelled A, B, C, and D. Initially, none of them has collided, so their version numbers are:

```
version[A] = 0
version[B] = 0
version[C] = 0
version[D] = 0
```

Suppose that, from their present positions and velocities, we predict the following ordered set of future collisions:

```
time    particles    stored versions

 6      A - C        (0,0)
10      A - B        (0,0)
12      C - D        (0,0)
14      B - D        (0,0)
```

The earliest event is the collision between A and C at $t = 6$, so this event is removed from the calendar and executed. Their velocities change, and hence so do their future trajectories. We therefore increment their version numbers:

```
version[A] = 1
version[B] = 0
version[C] = 1
version[D] = 0
```

At this point, the previously predicted A-B collision at $t = 10$ can no longer be trusted. It was calculated when particle A still had version 0. The same is true of the C-D collision at $t = 12$. However, the prediction for B-D at $t = 14$ is unaffected, because neither B nor D has changed trajectory.

We could now search through the calendar and explicitly remove the A-B and C-D events. This is precisely the operation we would like to avoid. Instead, we leave them where they are and calculate only new events involving the particles whose trajectories have changed, namely A and C. Suppose this produces:

```
time    particles    stored versions
 9      A - D        (1,0)
11      B - C        (0,1)
```

The calendar now contains the following entries, where we indicate which ones are new and which ones are old using comments:

```
time    particles    stored versions
 9      A - D        (1,0)    # new
10      A - B        (0,0)    # old
11      B - C        (0,1)    # new
12      C - D        (0,0)    # old
14      B - D        (0,0)    # old
```

The next event is therefore A-D at $t = 9$. Its stored versions, $(1, 0)$, agree with the current versions of A and D, so the prediction is still valid and the collision is executed. Afterward, we have that:

```
version[A] = 2
version[D] = 1
```

and new events involving A and D are predicted.

Now consider what happens when the old A-B event at $t = 10$ reaches the front of the calendar. The event contains:

```
stored:     version[A] = 0
current:    version[A] = 2
```

so we immediately know that this collision time was calculated from an obsolete trajectory. The event is discarded without performing a collision. There was no need to find and remove it when it first became invalid. The B-C event at $t = 11$, on the other hand, was predicted after the A-C collision and stored the versions $(0, 1)$. Provided neither B nor C has collided in the meantime, these values agree with their current versions and the event may be executed.

There is a final optimization worth mentioning. The simple algorithm above advances all N particle positions whenever an event occurs. This is unnecessary because the trajectory of a particle is known analytically between collisions. If particle i was last updated at time t_i, then at a later time t, we have $\boldsymbol{r}_i(t) = \boldsymbol{r}_i(t_i) + \boldsymbol{v}_i(t - t_i)$. One may therefore store a separate last-update time for every particle and evaluate its current position only when that particle is actually needed. This removes an otherwise $\mathcal{O}(N)$ position update from every event. The price is additional bookkeeping, however, so it is sensible to first construct and verify the simpler implementation in which all particle positions are advanced after every event.

4.6.3 Cells and Dynamic Linked Lists in EDMD

Cells play a slightly different role in EDMD than in ordinary time-stepped MD. Their main purpose is to reduce the number of candidate collision partner. That is, a particle can only collide next with particles in its own cell or nearby cells, so there is no need to predict collision times with every other particle in the system. Particles move ballistically between events, so a particle may leave its present cell before colliding. The time at which it crosses a cell boundary must therefore be predicted and inserted into the event calendar. When such a crossing event occurs, the particle is removed from one cell and inserted into the adjacent one. After this, possible collisions with particles newly brought into its local neighborhood are scheduled according to the above procedure.

This is one situation in which dynamically maintaining a linked-cell structure is genuinely useful. Store `head[c]`, `next[i]`, `prev[i]`, and `cell_of[i]`. To remove particle i from its present cell c, we first reconnect its predecessor to its successor. If i has no predecessor, then it is the first particle in the chain, so the head of the cell must instead be advanced to its successor:

```
if prev[i] != -1:
next[prev[i]] = next[i]
else:
head[c] = next[i]
```

We then reconnect the successor back to the predecessor. If i has no successor, it is already the last particle in the chain and no update is needed:

```
if next[i] != -1:
prev[next[i]] = prev[i]
```

Together, these operations simply splice particle i out of the doubly linked list. Insertion into a new cell c_{new} is similarly straightforward. We place i at the head of the new chain, so it has no predecessor and its successor becomes the particle that previously occupied the head:

```
prev[i] = -1
next[i] = head[c_new]
```

If the cell was not empty, the previous head must now point back to i:

```
if head[c_new] != -1:
prev[head[c_new]] = i
```

Finally, i becomes the new head of the cell and its stored cell index is updated:

```
head[c_new] = i
cell_of[i] = c_new
```

Because both removal and insertion involve only a fixed number of pointer updates, thus moving a particle from one cell to another takes constant time.

Exercise: A Simple Event-Driven MD Simulation

In this exercise, we will establish a basic three-dimensional EDMD simulation for identical hard spheres. Use reduced units $\sigma = m = k_{\text{B}} = 1$ as per usual, and start with a modest system, *e.g.*, $N = 32$ particles at packing fraction $\phi = 0.10$. The associated edge length is

$$L = \left(\frac{N\pi\sigma^3}{6\phi} \right)^{1/3} . \tag{4.29}$$

For the core exercise, use stationary reflecting walls rather than periodic boundaries. This avoids mixing two different collision geometries before the event-driven algorithm itself has been verified.

(a) Derive Eq. (4.27) from $|\boldsymbol{r}_{ij} + \tau \boldsymbol{v}_{ij}|^2 = \sigma^2$. Identify explicitly the conditions under which a real positive root corresponds to a future collision. Derive the collision update from conservation of linear momentum and kinetic energy. Why do simultaneous three-sphere collisions have probability zero for generic continuous initial conditions?

(b) Initialize non-overlapping spheres, for example on a simple cubic lattice followed by a small random displacement that preserves non-overlap. Draw velocities from a Maxwell distribution at $T = 1$ and remove any initial center-of-mass drift. This removal is only an initialization step. Because collisions with the reflecting walls exchange momentum with the particles, zero total particle momentum is not a constraint preserved by the subsequent dynamics. We must therefore count all $3N$ translational degrees of freedom and use the kinetic-temperature convention

$$T_{\mathrm{kin}} = \frac{2K}{3Nk_{\mathrm{B}}} \tag{4.30}$$

and rescale once so that $T_{\mathrm{kin}} = 1$. That is, we do not use the center-of-mass correction to the degrees of freedom that we saw in Eq. (4.6).

(c) Derive the collision time with the six stationary walls. If the geometrical walls lie at x, y, and $z = \pm L/2$ and the sphere radius is $a = \sigma/2$, the center coordinates are restricted to $[-L/2 + a, L/2 - a]$. For a wall with unit inward normal $\hat{\boldsymbol{n}}$, show that an elastic reflection changes the velocity according to

$$\boldsymbol{v}' = \boldsymbol{v} - 2(\boldsymbol{v} \cdot \hat{\boldsymbol{n}})\hat{\boldsymbol{n}}. \tag{4.31}$$

Implement separate routines for particle-particle and particle-wall collision times.

(d) For every admissible event, compute the future collision time and retain only the smallest positive value and the identities of the particles or wall involved. Do not sort the complete event list.

(e) Advance all particle positions by the selected time τ^* and then apply the corresponding collision rule. Because round-off may leave colliding particles infinitesimally overlapping or numerically still at contact, guard against spurious events with $\tau^* \leq 0$. It may be tempting to separate such particles by an additional small displacement, but this artificially changes the trajectory and can hide an error in the collision-detection logic rather than correct it.

(f) Verify the simulation. For particle-particle collisions, check conservation of total momentum and kinetic energy to numerical precision. For wall collisions, check conservation of kinetic energy. Throughout the run, verify that no two spheres overlap and that all centers remain within the allowed wall positions.

(g) Measure the mean kinetic temperature and the particle-particle collision frequency. Define the collision frequency per particle as

$$\nu = \frac{2N_{\mathrm{pp}}}{Nt_{\mathrm{run}}}, \tag{4.32}$$

where N_{pp} is the number of particle-particle collisions during physical run time t_{run}; you should count wall collisions separately. Repeat the simulation from several independent

initial velocity sets. Because EDMD has no integration time step, the conserved kinetic energy should not exhibit the systematic time-step drift seen in conventional MD.

(h) Optional extension: replace the reflecting boundaries in one or more directions by periodic boundaries. This requires care because a future collision may involve a periodic image rather than the present minimum-image separation. A robust implementation should treat boundary crossings or image changes consistently with the event schedule.

(i) Challenge: replace the repeated global minimum search by an event calendar. Store events in a min-heap together with the collision counters of the particles involved. After each collision, increment the appropriate counters and schedule only events involving the particles whose velocities changed. Verify that stale events are discarded correctly by deliberately printing the first few invalidations in a very small system.

(j) Challenge: introduce spatial cells and predict cell-boundary crossings as events. Compare the number of pair predictions per physical unit of simulated time with the naive all-pairs implementation.

References

[1] C. Dekker. “Solid-state nanopores”. In: *Nat. Nanotechnol.* 2 (2007), p. 209.

[2] G. Hummer, J.C. Rasaiah, and J.P. Noworyta. “Water conduction through the hydrophobic channel of a carbon nanotube”. In: *Nature* 414 (2001), p. 188.

[3] M. Majumder, N. Chopra, R. Andrews, and B.J. Hinds. “Enhanced flow in carbon nanotubes”. In: *Nature* 438 (2005). Erratum published in Nature 438, 930 (2005), DOI 10.1038/438930b, p. 44.

[4] S. Gravelle, L. Joly, C. Ybert, and L. Bocquet. “Large permeabilities of hourglass nanopores: From hydrodynamics to single file transport”. In: *J. Chem. Phys.* 141 (2014), p. 18C526.

[5] D.M. Huang, C. Sendner, D. Horinek, R.R. Netz, and L. Bocquet. “Water Slippage versus Contact Angle: A Quasiuniversal Relationship”. In: *Phys. Rev. Lett.* 101 (2008), p. 226101.

[6] T. Krüger, H. Kusumaatmaja, A. Kuzmin, O. Shardt, G. Silva, and E.M. Viggen. *The Lattice Boltzmann Method: Principles and Practice*. Graduate Texts in Physics. Cham: Springer International Publishing, 2017.

[7] G. Gompper, T. Ihle, D.M. Kroll, and R.G. Winkler. “Multi-Particle Collision Dynamics: A Particle-Based Mesoscale Simulation Approach to the Hydrodynamics of Complex Fluids”. In: *Advanced Computer Simulation Approaches for Soft Matter Sciences III*. Ed. by C. Holm and K. Kremer. Vol. 221. Advances in Polymer Science. Berlin / Heidelberg: Springer, 2009, p. 1.

[8] C.H. Mak. “Large-scale simulations of the two-dimensional melting of hard disks”. In: *Phys. Rev. E* 73 (2006), 065104(R).

[9] M. Engel, J.A. Anderson, S.C. Glotzer, M. Isobe, E.P. Bernard, and W. Krauth. “Hard-disk equation of state: First-order liquid-hexatic transition in two dimensions with three simulation methods”. In: *Phys. Rev. E* 87 (2013), p. 042134.

[10] P.P. Ewald. “Die Berechnung optischer und elektrostatischer Gitterpotentiale”. In: *Ann. Phys. (Leipzig)* 369 (1921), p. 253.

[11] D. Frenkel and B. Smit. *Understanding Molecular Simulation*. 2nd ed. Vol. 1. Computational Science Series. San Diego: Academic Press, 2002.

[12] M.P. Allen and D.J. Tildesley. *Computer Simulation of Liquids*. 2nd ed. Oxford: Oxford University Press, 2017.

[13] L. Verlet. “Computer “Experiments” on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules”. In: *Phys. Rev.* 159 (1967), p. 98.

[14] M.P. Howard, J.A. Anderson, A. Nikoubashman, S.C. Glotzer, and A.Z. Panagiotopoulos. “Efficient neighbor list calculation for molecular simulation of colloidal systems using graphics processing units”. In: *Comput. Phys. Commun.* 203 (2016), p. 45.

[15] D.C. Rapaport. *The Art of Molecular Dynamics Simulation*. 2nd ed. Cambridge: Cambridge University Press, 2004.

[16] S. Plimpton. "Fast Parallel Algorithms for Short-Range Molecular Dynamics". In: *J. Comput. Phys.* 117 (1995), p. 1.

[17] J.D. Weeks, D. Chandler, and H.C. Andersen. "Role of Repulsive Forces in Determining the Equilibrium Structure of Simple Liquids". In: *J. Chem. Phys.* 54 (1971), p. 5237.

[18] C.P. Lowe. "An alternative approach to dissipative particle dynamics". In: *Europhys. Lett.* 47 (1999), p. 145.

[19] S.D. Stoyanov and R.D. Groot. "From molecular dynamics to hydrodynamics: a novel Galilean invariant thermostat". In: *J. Chem. Phys.* 122 (2005), p. 114112.

[20] A. Rahman. "Correlations in the Motion of Atoms in Liquid Argon". In: *Phys. Rev.* 136 (1964), A405.

[21] G. Foffi, C. De Michele, F. Sciortino, and P. Tartaglia. "Scaling of Dynamics with the Range of Interaction in Short-Range Attractive Colloids". In: *Phys. Rev. Lett.* 94 (2005), p. 078301.

[22] S. Auer and D. Frenkel. "Prediction of absolute crystal-nucleation rate in hard-sphere colloids". In: *Nature* 409 (2001), p. 1020.

[23] R.J. Speedy. "The hard sphere glass transition". In: *Mol. Phys.* 95 (1998), p. 169.

[24] B.J. Alder and T.E. Wainwright. "Studies in Molecular Dynamics. I. General Method". In: *J. Chem. Phys.* 31 (1959), p. 459.

[25] D.C. Rapaport. "The event scheduling problem in molecular dynamic simulation". In: *J. Comput. Phys.* 34 (1980), p. 184.