2

Basic molecular dynamics

2.1 Introduction

This chapter provides the introductory appetizer and aims to leave the reader new to MD with a feeling for what the subject is all about. Later chapters will address the techniques in detail; here the goal is to demonstrate a working example with a minimum of fuss and so convince the beginner that MD is not only straightforward but also that it works successfully. Of course, the technique for evaluating the forces discussed here is not particularly efficient from a computational point of view and the model is about the simplest there is. Such matters will be rectified later. The general program organization and stylistic conventions used in case studies throughout the book are also introduced.

2.2 Soft-disk fluid

Interactions and equations of motion

The most rudimentary microscopic model for a substance capable of existing in any of the three most familiar states of matter – solid, liquid and gas – is based on spherical particles that interact with one another; in the interest of brevity such particles will be referred to as atoms (albeit without hint of their quantum origins). The interactions, again at the simplest level, occur between pairs of atoms and are responsible for providing the two principal features of an interatomic force. The first is a resistance to compression, hence the interaction repels at close range. The second is to bind the atoms together in the solid and liquid states, and for this the atoms must attract each other over a range of separations. Potential functions exhibiting these characteristics can adopt a variety of forms and, when chosen carefully, actually provide useful models for real substances.

The best known of these potentials, originally proposed for liquid argon, is the Lennard-Jones (LJ) potential [mcq76, mai81]; for a pair of atoms i and j located

https://doi.org/10.1017/CBO9780511816581.005 Published online by Cambridge University Press

at $\boldsymbol{r}_i$ and $\boldsymbol{r}_j$ the potential energy is

$$u(r_{ij}) = \begin{cases} 4\epsilon\left[\left(\dfrac{\sigma}{r_{ij}}\right)^{12} - \left(\dfrac{\sigma}{r_{ij}}\right)^{6}\right] & r_{ij} < r_c \\ 0 & r_{ij} \geq r_c \end{cases} \tag{2.2.1}$$

where $\boldsymbol{r}_{ij} = \boldsymbol{r}_i - \boldsymbol{r}_j$ and $r_{ij} \equiv |\boldsymbol{r}_{ij}|$. The parameter ϵ governs the strength of the interaction and σ defines a length scale; the interaction repels at close range, then attracts, and is eventually cut off at some limiting separation r_c. While the strongly repulsive core arising from (in the language of quantum mechanics) the nonbonded overlap between the electron clouds has a rather arbitrary form, and other powers and functional forms are sometimes used, the attractive tail actually represents the van der Waals interaction due to electron correlations. The interactions involve individual pairs of atoms: each pair is treated independently, with other atoms in the neighborhood having no effect on the force between them.

We will simplify the interaction even further by ignoring the attractive tail and changing (2.2.1) to

$$u(r_{ij}) = \begin{cases} 4\epsilon\left[\left(\dfrac{\sigma}{r_{ij}}\right)^{12} - \left(\dfrac{\sigma}{r_{ij}}\right)^{6}\right] + \epsilon & r_{ij} < r_c = 2^{1/6}\sigma \\ 0 & r_{ij} \geq r_c \end{cases} \tag{2.2.2}$$

with r_c chosen so that $u(r_c) = 0$. A model fluid constructed using this potential is little more than a collection of colliding balls that are both soft (though the softness is limited) and smooth. All that holds the system together is the container within which the atoms (or balls) are confined. While the kinds of system that can be represented quantitatively by this highly simplified model are limited – typically gases at low density – it does nevertheless have much in common with more detailed models, and has a clear advantage in terms of computational simplicity. If certain kinds of behavior can be shown to be insensitive to specific features of the model, in this instance the attractive tail of the potential, then it is clearly preferable to eliminate them from the computation in order to reduce the amount of work, and for this reason the soft-sphere system will reappear in many of the case studies.

The force corresponding to $u(r)$ is

$$\boldsymbol{f} = -\nabla u(r) \tag{2.2.3}$$

so the force that atom j exerts on atom i is

$$\boldsymbol{f}_{ij} = \left(\frac{48\epsilon}{\sigma^2}\right)\left[\left(\frac{\sigma}{r_{ij}}\right)^{14} - \tfrac{1}{2}\left(\frac{\sigma}{r_{ij}}\right)^{8}\right]\boldsymbol{r}_{ij} \tag{2.2.4}$$

https://doi.org/10.1017/CBO9780511816581.005 Published online by Cambridge University Press

provided $r_{ij} < r_c$, and zero otherwise. As r increases towards r_c the force drops to zero, so that there is no discontinuity at r_c (in both the force and the potential); $\nabla \boldsymbol{f}$ and higher derivatives are discontinuous, though this has no real impact on the numerical solution. The equations of motion follow from Newton's second law,

$$m\ddot{\boldsymbol{r}}_i = \boldsymbol{f}_i = \sum_{\substack{j=1 \\ (j \neq i)}}^{N_m} \boldsymbol{f}_{ij} \tag{2.2.5}$$

where the sum is over all N_m atoms (or molecules in the monatomic case), excluding i itself, and m is the atomic mass. It is these equations which must be numerically integrated. Newton's third law implies that $\boldsymbol{f}_{ji} = -\boldsymbol{f}_{ij}$, so each atom pair need only be examined once. The amount of work† is proportional to N_m^2, so that for models in which r_c is small compared with the size of the container it would obviously be a good idea to determine those atom pairs for which $r_{ij} \leq r_c$ and use this information to reduce the computational effort; we will indeed adopt such an approach in Chapter 3. In the present example, which focuses on just the smallest of systems, we continue with this all-pairs approach.

Dimensionless units

At this point we introduce a set of dimensionless, or reduced, MD units in terms of which all physical quantities will be expressed. There are several reasons for doing this, not the least being the ability to work with numerical values that are not too distant from unity, instead of the extremely small values normally associated with the atomic scale. Another benefit of dimensionless units is that the equations of motion are simplified because some, if not all, of the parameters defining the model are absorbed into the units. The most familiar reason for using such units is related to the general notion of scaling, namely, that a single model can describe a whole class of problems, and once the properties have been measured in dimensionless units they can easily be scaled to the appropriate physical units for each problem of interest. From a strictly practical point of view, the switch to such units removes any risk of encountering values lying outside the range that is representable by the computer hardware.

For MD studies using potentials based on the LJ form (2.2.1) the most suitable dimensionless units are defined by choosing σ, m and ϵ to be the units of length,

† Note that for the potential function (2.2.2), or the corresponding force (2.2.4), it is never necessary to evaluate $|\boldsymbol{r}_{ij}|$; only its square is needed, so that the (sometimes costly) square root computation is avoided.

https://doi.org/10.1017/CBO9780511816581.005 Published online by Cambridge University Press

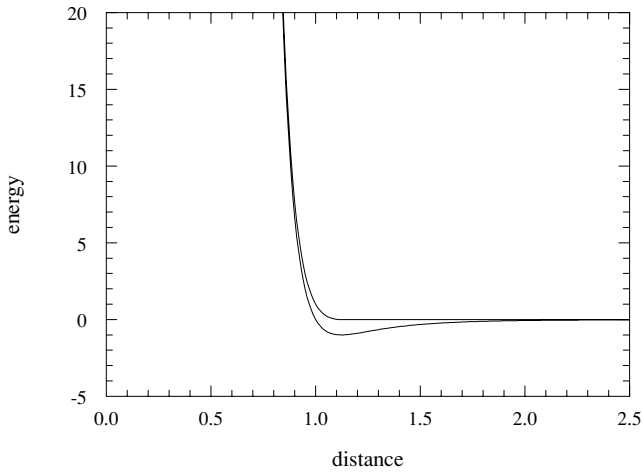

Fig. 2.1. Lennard-Jones and soft-sphere interaction energy (in dimensionless MD units).

mass and energy, respectively, and making the replacements

$$\begin{aligned} &\text{length:} && r \to r\sigma \\ &\text{energy:} && e \to e\epsilon \\ &\text{time:} && t \to t\sqrt{m\sigma^2/\epsilon} \end{aligned} \tag{2.2.6}$$

The resulting form of the equation of motion, now in MD units, is

$$\ddot{\boldsymbol{r}}_i = 48 \sum_{j\,(\neq i)} \left(r_{ij}^{-14} - \tfrac{1}{2} r_{ij}^{-8} \right) \boldsymbol{r}_{ij} \tag{2.2.7}$$

The dimensionless kinetic and potential energies, per atom, are

$$E_K = \frac{1}{2N_m} \sum_{i=1}^{N_m} \boldsymbol{v}_i^2 \tag{2.2.8}$$

$$E_U = \frac{4}{N_m} \sum_{1 \le i < j \le N_m} \left(r_{ij}^{-12} - r_{ij}^{-6} \right) \tag{2.2.9}$$

where $\boldsymbol{v}_i$ is the velocity. The functional forms of the LJ and soft-sphere potentials, in MD units, are shown in Figure 2.1.

The unit of temperature is ϵ / k_B, and since each translational degree of freedom contributes $k_B T/2$ to the kinetic energy, the temperature of a d-dimensional ($d = 2$

https://doi.org/10.1017/CBO9780511816581.005 Published online by Cambridge University Press

or 3) system is

$$T = \frac{1}{dN_m} \sum_i \boldsymbol{v}_i^2 \tag{2.2.10}$$

We have set $k_B = 1$, so that the MD unit of temperature is now also defined. Strictly speaking, of the total dN_m degrees of freedom, d are eliminated because of momentum conservation, but if N_m is not too small this detail can be safely ignored.

If the model is intended to represent liquid argon, the relations between the dimensionless MD units and real physical units are as follows [rah64]:

- Lengths are expressed in terms of $\sigma = 3.4$ Å.
- The energy units are specified by $\epsilon/k_B = 120\,\mathrm{K}$, implying that $\epsilon = 120 \times 1.3806 \times 10^{-16}$ erg/atom†.
- Given the mass of an argon atom $m = 39.95 \times 1.6747 \times 10^{-24}$ g, the MD time unit corresponds to 2.161×10^{-12} s; thus a typical timestep size of $\Delta t = 0.005$ used in the numerical integration of the equations of motion corresponds to approximately 10^{-14} s.
- Finally, if N_m atoms occupy a cubic region of edge length L, then a typical liquid density of $0.942\,\mathrm{g/cm^3}$ implies that $L = 4.142 N_m^{1/3}$ Å, which in reduced units amounts to $L = 1.218 N_m^{1/3}$.

Suitably chosen dimensionless units will be employed throughout the book. Other quantities, such as the diffusion coefficient and viscosity studied in Chapter 5, will also be expressed using dimensionless units, and these too are readily converted to physical units.

Boundary conditions

Finite and infinite systems are very different, and the question of how large a relatively small system must be to yield results that resemble the behavior of the infinite system faithfully lacks a unique answer. The simulation takes place in a container of some kind, and it is tempting to regard the container walls as rigid boundaries against which atoms collide while trying to escape from the simulation region. In systems of macroscopic size, only a very small fraction of the atoms is close enough to a wall to experience any deviation from the environment prevailing in the interior. Consider, for example, a three-dimensional system with $N_m = 10^{21}$ at liquid density. Since the number of atoms near the walls is of order $N_m^{2/3}$, this amounts to 10^{14} atoms – a mere one in 10^7. But for a more typical MD value of

† Several kinds of units are in use for energy; conversion among them is based on standard relations that include 1.3806×10^{-16} erg/atom $= 1.987 \times 10^{-3}$ kcal/mole $= 8.314$ J/mole.

https://doi.org/10.1017/CBO9780511816581.005 Published online by Cambridge University Press

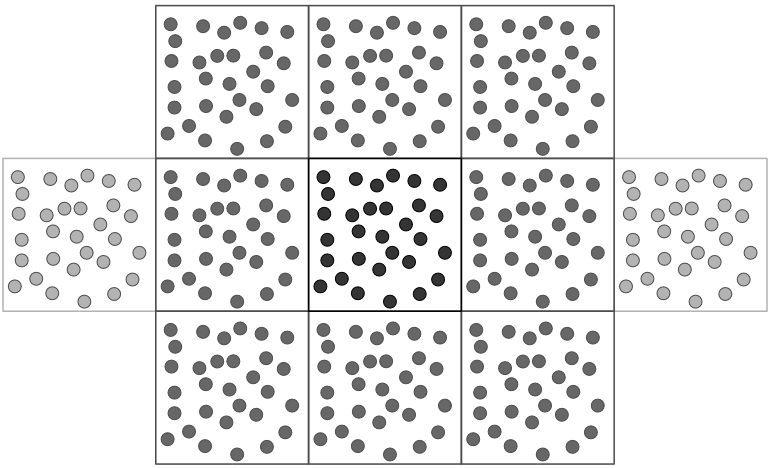

Fig. 2.2. The meaning of periodic boundary conditions (the two-dimensional case is shown).

$N_m = 1000$, roughly 500 atoms are immediately adjacent to the walls, leaving very few interior atoms; if the first two layers are excluded a mere 216 atoms remain. Thus the simulation will fail to capture the typical state of an interior atom and the measurements will reflect this fact. Unless the goal is the study of behavior near real walls, a problem that is actually of considerable importance, walls are best eliminated.

A system that is bounded but free of physical walls can be constructed by resorting to periodic boundary conditions, shown schematically in Figure 2.2. The introduction of periodic boundaries is equivalent to considering an infinite, space-filling array of identical copies of the simulation region. There are two consequences of this periodicity. The first is that an atom that leaves the simulation region through a particular bounding face immediately reenters the region through the opposite face. The second is that atoms lying within a distance r_c of a boundary interact with atoms in an adjacent copy of the system, or, equivalently, with atoms near the opposite boundary – a wraparound effect. Another way of regarding periodic boundaries is to think of mapping the region (topologically, not spatially) onto the equivalent of a torus in four dimensions (a two-dimensional system is mapped onto a torus); then it is obvious that there are no physical boundaries. In this way it is possible to model systems that are effectively bounded but that are nevertheless spatially homogeneous insofar as boundaries are concerned.

The wraparound effect of the periodic boundaries must be taken into account in both the integration of the equations of motion and the interaction computations. After each integration step the coordinates must be examined, and if an atom is

https://doi.org/10.1017/CBO9780511816581.005 Published online by Cambridge University Press

found to have moved outside the region its coordinates must be adjusted to bring it back inside. If, for example, the x coordinate is defined to lie between $-L_x/2$ and $L_x/2$, where L_x is the region size in the x direction, the tests (which can be expressed in various equivalent ways) are:

- if $r_{i\,x} \geq L_x/2$, replace it by $r_{i\,x} - L_x$;
- otherwise, if $r_{i\,x} < -L_x/2$, replace it by $r_{i\,x} + L_x$.

The effect of periodicity on the interaction calculation appears in determining the components of the distance between pairs of atoms; the tests are very similar:

- if $r_{ij\,x} \geq L_x/2$, replace it by $r_{ij\,x} - L_x$;
- otherwise, if $r_{ij\,x} < -L_x/2$, replace it by $r_{ij\,x} + L_x$.

Periodic wraparound may also have to be considered when analyzing the results of a simulation, as will become apparent later.

Periodic boundaries are most easily handled if the region is rectangular in two dimensions, or a rectangular prism in three. This is not an essential requirement, and any space-filling, convex region can be used, although the boundary computations will not be as simple as those just illustrated. The motivation for choosing alternative region shapes is to enlarge the volume to surface ratio, and thus increase the maximum distance between atoms before periodic ambiguity appears (it is obviously meaningless to speak of interatomic distances that exceed half the region size), the most desirable shape in three dimensions – though not space filling – being the sphere. In two dimensions a hexagon might be used, while in three the truncated octahedron [ada80] is one such candidate. Another reason for choosing more complex region shapes is to allow the modeling of crystalline structures with nonorthogonal axes, for example, a trigonal unit cell; there, too, an alternative region shape, such as a sheared cube, might be worth considering.

Although not an issue in this particular case, the use of periodic boundaries limits the interaction range to no more than half the smallest region dimension – in practice the range is generally much less. Long-range forces require entirely different approaches that will be described in Chapter 13. Problems can also arise if there are strong correlations between atoms separated by distances approaching the region size, because periodic wraparound can then lead to spurious effects. One example is the vibration of an atom producing what are essentially sound waves; the disturbance, if not sufficiently attenuated, can propagate around the system and eventually return to affect the atom itself.

Even with periodic boundaries, finite-size effects are still present, so how big does the system have to be before they can be neglected? The answer depends on the kind of system and the properties of interest. As a minimal requirement, the size should exceed the range of any significant correlations, but there may be more subtle effects even in larger systems. Only detailed numerical study can hope to resolve this question.

https://doi.org/10.1017/CBO9780511816581.005 Published online by Cambridge University Press

Initial state

In order for MD to serve a useful purpose it must be capable of sampling a representative region of the total phase space of the system. An obvious corollary of this requirement is that the results of a simulation of adequate duration are insensitive to the initial state, so that any convenient initial state is allowed. A particularly simple choice is to start with the atoms at the sites of a regular lattice – such as the square or simple cubic lattice – spaced to give the desired density. The initial velocities are assigned random directions and a fixed magnitude based on temperature; they are also adjusted to ensure that the center of mass of the system is at rest, thereby eliminating any overall flow. The speed of equilibration to a state in which there is no memory of this arbitrarily selected initial configuration is normally quite rapid, so that more careful attempts at constructing a 'typical' state are of little benefit.

2.3 Methodology

Integration

Integration of the equations of motion uses the simplest of numerical techniques, the leapfrog method. The origin of the method will be discussed in §3.5; for the present it is sufficient to state that, despite its low order, the method has excellent energy conservation properties and is widely used.

If $h = \Delta t$ denotes the size of the timestep used for the numerical integration, then the integration formulae applied to each component of an atom's coordinates and velocities are

$$v_{i\,x}(t + h/2) = v_{i\,x}(t - h/2) + h a_{i\,x}(t) \tag{2.3.1}$$

$$r_{i\,x}(t + h) = r_{i\,x}(t) + h v_{i\,x}(t + h/2) \tag{2.3.2}$$

The name 'leapfrog' stems from the fact that coordinates and velocities are evaluated at different times; if a velocity estimate is required to correspond to the time at which coordinates are evaluated, then

$$v_{i\,x}(t) = v_{i\,x}(t - h/2) + (h/2) a_{i\,x}(t) \tag{2.3.3}$$

can be used. The local errors introduced at each timestep due to the truncation of what should really be infinite series in h are of order $O(h^4)$ for the coordinates and $O(h^2)$ for velocities.

The leapfrog method can be reformulated in an alternative, algebraically equivalent manner that enables the coordinates and velocities to be evaluated at the same instant in time, avoiding the need for the velocity adjustment in (2.3.3). To do this, the computations are split into two parts: Before computing the acceleration values,

https://doi.org/10.1017/CBO9780511816581.005 Published online by Cambridge University Press

update the velocities by a half timestep using the old acceleration values, and then update the coordinates by a full timestep using the intermediate velocity values,

$$v_{i\,x}(t + h/2) = v_{i\,x}(t) + (h/2)a_{i\,x}(t) \tag{2.3.4}$$

$$r_{i\,x}(t + h) = r_{i\,x}(t) + hv_{i\,x}(t + h/2) \tag{2.3.5}$$

Now use the new coordinates to compute the latest acceleration values and update the velocities over the second half timestep,

$$v_{i\,x}(t + h) = v_{i\,x}(t + h/2) + (h/2)a_{i\,x}(t + h) \tag{2.3.6}$$

This two-step procedure† is the version of the leapfrog method that will be used throughout the book.

Measurements

The most accessible properties of systems in equilibrium are those introduced in elementary thermodynamics, namely, energy and pressure, each expressed in terms of the independent temperature and density variables T and ρ. Measuring such quantities during an MD simulation is relatively simple, and provides the link between the world of thermodynamics – which predates the recognition of the atomic nature of matter – and the detailed behavior at the microscopic level. However, it is energy rather than temperature that is constant in our MD simulation, so the thermodynamic results are expressed in terms of the average $\langle T \rangle$, rather than T.

In this case study, energy and pressure are the only properties measured. Pressure is defined in terms of the virial expression [han86b] (with $k_B = 1$)

$$PV = N_m T + \frac{1}{d}\left\langle \sum_{i=1}^{N_m} \boldsymbol{r}_i \cdot \boldsymbol{f}_i \right\rangle \tag{2.3.7}$$

In two dimensions, the region volume V is replaced by the area. For pair potentials, (2.3.7) can be written as a sum over interacting atom pairs, namely,

$$PV = N_m T + \frac{1}{d}\left\langle \sum_{i<j} \boldsymbol{r}_{ij} \cdot \boldsymbol{f}_{ij} \right\rangle \tag{2.3.8}$$

and for the force (2.2.4) this becomes (in MD units)

$$PV = \frac{1}{d}\left\langle \sum_i \boldsymbol{v}_i^2 + 48 \sum_{i<j} \left(r_{ij}^{-12} - \tfrac{1}{2} r_{ij}^{-6}\right) \right\rangle \tag{2.3.9}$$

While the total energy per atom $E = E_K + E_U$ is conserved, apart from any numerical integration error, quantities such P and T $(= 2E_K/d)$ fluctuate, and averages

† The first edition used the one-step method of (2.3.1)–(2.3.2).

https://doi.org/10.1017/CBO9780511816581.005 Published online by Cambridge University Press

must be computed over a series of timesteps; such averaging will be included in the program and used for estimating the mean values as well as the statistical measurement errors.

2.4 Programming

Style and conventions

In this section we will be presenting the full listing of the program used in the case study. Not only is the program the tool for getting the job done, it also incorporates a definitive statement of all the computational details. But before addressing these details a few general remarks on matters of organization and programming style are in order. Style, to a considerable degree, is a matter of personal taste; the widely used C language chosen for this work offers a certain amount of flexibility in this respect†, a boon for some, but a bane for others.

A similar form of organization is used for most programs in the book. Parts of the program discussed in this chapter may seem to be expressed in a more general form than is absolutely necessary; this is to provide a basis for extending the program to handle later case studies. We assume that the reader has a reasonable (and easily acquired) familiarity with the C language. C requires that all variables be defined prior to use; all the definitions will be included, but because the material is presented in a 'functional' manner, rather than as a serial listing of the program text, variables may first appear in the recipe before they are formally defined (this is of course not the case in the program sources). Local variables used within functions are not preserved between calls.

We adopt the convention that all variable names begin with a lower case letter; names formed by joining multiple words use intermediate capitals to clarify meaning. Function names begin with an upper case letter, as do macro definitions specified using `#define` statements. Constants specified with `#define` statements are fully capitalized. The format of a C program is also subject to taste. The physical layout used here is fairly standard, with indentation and the positioning of block-delimiting braces used to emphasize the logical structure. The line numbers are of course not part of the program, and are included merely to aid reference.

† In the interest of readability, we have tried to avoid some characteristics of C that allow writing extremely concise code (often bordering on the obfuscated); while the experienced C user may perceive their absence, the efficiency of the compiled program is unlikely to be affected in any serious way. As some readers may notice, the software here differs from the first edition in two key respects: (a) Arrays of C structures are used to represent sets of molecular variables, rather than doubly-indexed arrays that represent individual variables (such as atomic coordinates) in which one of the indices is used to select the component of the vector. (b) The conventional C indexing style is used, in which array indices begin at zero, rather than unity as in the original algebraic formulation of the problem. The programming style of the first edition was aimed at making the software more acceptable to Fortran programmers; with the increasing popularity of C, and other programming languages that borrow much of its syntax, not to mention the changing nature of the Fortran language, this is no longer an issue.

Program organization

The main program♠ of this elementary MD exercise, which forms the basis of most of the subsequent case studies as well, is as follows.

```
int main (int argc, char **argv)
{
  GetNameList (argc, argv);
  PrintNameList (stdout);
  SetParams ();
  SetupJob ();
  moreCycles = 1;
  while (moreCycles) {
    SingleStep ();
    if (stepCount >= stepLimit) moreCycles = 0;
  }
}
```

After the initialization phase (*GetNameList*, *SetParams*, *SetupJob*), in the course of which parameters and other data are input to the program or initialized, and storage arrays allocated, the program enters a loop. Each loop cycle advances the system by a single timestep (*SingleStep*). The loop terminates when *moreCycles* is set to zero; here this occurs after a preset number of timesteps, but in a more general context *moreCycles* can be zeroed once the total processing time exceeds a preset limit, or even by means of an interrupt generated by the user from outside the program when she feels the run has produced enough results (there are also more drastic ways of terminating a program)†.

The function that handles the processing for a single timestep, including calls to functions that deal with the force evaluation, integration of the equations of motion, adjustments required by periodic boundaries, and property measurements, is

```
void SingleStep ()
{
  ++ stepCount;
  timeNow = stepCount * deltaT;
  LeapfrogStep (1);
  ApplyBoundaryCond ();
  ComputeForces ();
  LeapfrogStep (2);
  EvalProps ();
  AccumProps (1);
  if (stepCount % stepAvg == 0) {
```

♠ *pr_02_1* (This is a reference to one of the programs accompanying the book; the full list appears in the Appendix.)

† As a reminder to lapsed C users, *main* is where the program begins, *argc* is the number of arguments passed to the program from the command line (as in Unix), and the array *argv* provides access to the text of each of these arguments.

https://doi.org/10.1017/CBO9780511816581.005 Published online by Cambridge University Press

```
    AccumProps (2);
    PrintSummary (stdout);
    AccumProps (0);
  }
}
```

All the work needed for initializing the computation is concentrated in the following function.

```
void SetupJob ()
{
  AllocArrays ();
  stepCount = 0;
  InitCoords ();
  InitVels ();
  InitAccels ();
  AccumProps (0);
}
```

Having dealt with the top level functions of the program it is appropriate to insert a few comments on the program structure adopted in these recipes. The order of presentation of this introductory case study reflects the organization of the program: the organization is modular, with separate functions being responsible for distinct portions of the computation. In this initial case study, given the simplicity of the problem the emphasis on organization may appear overdone, but, as indicated earlier, our aim is to provide a more general framework that will be utilized later†.

The meaning of most program variables should be apparent from their names, with the same being true for functions. Where the meanings are not obvious, or additional remarks are called for, the text will include further details. An alphabetically ordered summary of the globally declared variables appears in the Appendix. Other questions ought to be resolved by examining functions that appear subsequently.

There are many program elements that are common to MD simulations of various kinds. Some of these already appear in this initial case study, others will be introduced later on. Examples include:

- parameter input with completeness and consistency checks;
- runtime array allocation, with array sizes determined by the actual system size;
- initialization of variables;
- the main loop which cycles through the force computations and trajectory integration, and performs data collection at specified intervals;
- the processing and statistical analysis of various kinds of measurement;

† On the other hand, in order to avoid the risk of tedium, we have not carried this functional decomposition to the extremes sometimes practiced in professional software development.

https://doi.org/10.1017/CBO9780511816581.005 Published online by Cambridge University Press

- storage of accumulated results and condensed configurational snapshots for later analysis;
- run termination based on various criteria;
- provision for checkpointing (or saving) the current computational state of a long simulation run, both as a safety measure, and to permit the run to be interrupted and continued at some later time.

Computational functions

The function `ComputeForces` encountered in the listing of `SingleStep` is responsible for the interaction computations. Before considering the general form of this function we start with a version suitable for a two-dimensional system in order to allow the gradual introduction of data structures and other elements that will be used throughout the book.

This listing differs from conventional C in that a new kind of floating-point variable, `real`, is introduced. To allow flexibility, `real` can be set to correspond to either single or double precision, known respectively in C as `float` and `double`. Single precision saves storage, whereas double precision provides additional accuracy; as for relative computation speed, this depends on the particular processor hardware, and either precision may be faster, sometimes significantly. Double precision will be used throughout by including the declaration

```
typedef double real;
```

at the beginning of the program.

Many of the quantities involved in the calculations, such as the atomic coordinates, are in fact vectors; the programming style used here will reflect this observation in order to enhance the readability of the software. With this goal in mind we introduce the following C structure type to represent a two-dimensional vector quantity with floating-point components

```
typedef struct {
  real x, y;
} VecR;
```

Organizing the variables associated with each atom or molecule is simplified by the introduction of another structure

```
typedef struct {
  VecR r, rv, ra;
} Mol;
```

https://doi.org/10.1017/CBO9780511816581.005 Published online by Cambridge University Press

in which `r`, `rv` and `ra` correspond, respectively, to the coordinate, velocity and acceleration vectors of the atom. An array of such structures will be introduced later on to represent the state of the system.

The initial version of the function for computing the forces (which are identical to the accelerations in the MD units defined earlier), as well as the potential energy `uSum`, can be written in terms of these vector quantities as

```
void ComputeForces ()
{
  VecR dr;
  real fcVal, rr, rrCut, rri, rri3;
  int j1, j2, n;

  rrCut = Sqr (rCut);
  for (n = 0; n < nMol; n ++) {
    mol[n].ra.x = 0.;
    mol[n].ra.y = 0.;
  }
  uSum = 0.;
  for (j1 = 0; j1 < nMol - 1; j1 ++) {
    for (j2 = j1 + 1; j2 < nMol; j2 ++) {
      dr.x = mol[j1].r.x - mol[j2].r.x;
      dr.y = mol[j1].r.y - mol[j2].r.y;
      if (dr.x >= 0.5 * region.x) dr.x -= region.x;
      else if (dr.x < -0.5 * region.x) dr.x += region.x;
      if (dr.y >= 0.5 * region.y) dr.y -= region.y;
      else if (dr.y < -0.5 * region.y) dr.y += region.y;
      rr = dr.x * dr.x + dr.y * dr.y;
      if (rr < rrCut) {
        rri = 1. / rr;
        rri3 = rri * rri * rri;
        fcVal = 48. * rri3 * (rri3 - 0.5) * rri;
        mol[j1].ra.x += fcVal * dr.x;
        mol[j1].ra.y += fcVal * dr.y;
        mol[j2].ra.x -= fcVal * dr.x;
        mol[j2].ra.y -= fcVal * dr.y;
        uSum += 4. * rri3 * (rri3 - 1.) + 1.;
      }
    }
  }
}
```

Periodic boundaries are included by testing whether any of the components of the interatomic separation vector `dr` exceed half the system size, and if they do, performing a wraparound operation.

While C does not provide the capability for defining new operations, in particular operations associated with vector algebra, it does support the use of macro

https://doi.org/10.1017/CBO9780511816581.005 Published online by Cambridge University Press

definitions that can simplify the code considerably. Definitions of this kind will be introduced as necessary, and a complete listing appears in §18.2.

The following definitions can be used for vector addition and subtraction (in two dimensions),

```
#define VAdd(v1, v2, v3)                                    \
   (v1).x = (v2).x + (v3).x,                                \
   (v1).y = (v2).y + (v3).y
#define VSub(v1, v2, v3)                                    \
   (v1).x = (v2).x - (v3).x,                                \
   (v1).y = (v2).y - (v3).y
```

where the extra parentheses are a safety measure to cover the possible ways these definitions might be employed in practice. Other vector operations that will be used here, some of which are specialized instances of preceding definitions, are

```
#define VDot(v1, v2)                                        \
   ((v1).x * (v2).x + (v1).y * (v2).y)
#define VSAdd(v1, v2, s3, v3)                               \
   (v1).x = (v2).x + (s3) * (v3).x,                         \
   (v1).y = (v2).y + (s3) * (v3).y
#define VSet(v, sx, sy)                                     \
   (v).x = sx,                                              \
   (v).y = sy
#define VSetAll(v, s)       VSet (v, s, s)
#define VZero(v)            VSetAll (v, 0)
#define VVSAdd(v1, s2, v2)  VSAdd (v1, v1, s2, v2)
#define VLenSq(v)           VDot (v, v)
```

The definitions have been constructed in a manner that will minimize the changes required when switching to three dimensions – such as defining the scalar product of two vectors and then using this in defining the squared length of a vector. Finally, the expressions for handling the periodic wraparound can be defined as

```
#define VWrap(v, t)                                         \
   if (v.t >= 0.5 * region.t) v.t -= region.t;              \
   else if (v.t < -0.5 * region.t) v.t += region.t
#define VWrapAll(v)                                         \
   {VWrap (v, x);                                           \
    VWrap (v, y);}
```

Note that it is implicitly assumed that atoms will not have moved too far outside the region before the periodic wraparound is applied; the above treatment is clearly inadequate for atoms that have traveled so far that this adjustment does not bring them back inside the region. In practice, it should be impossible for an atom to travel such a distance in just a single timestep; thus the alternative, strictly correct

https://doi.org/10.1017/CBO9780511816581.005 Published online by Cambridge University Press

but more costly computation based on evaluating

$$(r_{xi} + L_x/2) \pmod{L_x} - L_x/2 \tag{2.4.1}$$

is not used.

Aided by these definitions, as well as by

```
#define Sqr(x)   ((x) * (x))
#define Cube(x)  ((x) * (x) * (x))
#define DO_MOL   for (n = 0; n < nMol; n ++)
```

we arrive at the following revised version of the interaction function, now also including the contribution of the interactions to the virial.

```
void ComputeForces ()
{
  VecR dr;
  real fcVal, rr, rrCut, rri, rri3;
  int j1, j2, n;

  rrCut = Sqr (rCut);
  DO_MOL VZero (mol[n].ra);
  uSum = 0.;
  virSum = 0.;
  for (j1 = 0; j1 < nMol - 1; j1 ++) {
    for (j2 = j1 + 1; j2 < nMol; j2 ++) {
      VSub (dr, mol[j1].r, mol[j2].r);
      VWrapAll (dr);
      rr = VLenSq (dr);
      if (rr < rrCut) {
        rri = 1. / rr;
        rri3 = Cube (rri);
        fcVal = 48. * rri3 * (rri3 - 0.5) * rri;
        VVSAdd (mol[j1].ra, fcVal, dr);
        VVSAdd (mol[j2].ra, - fcVal, dr);
        uSum += 4. * rri3 * (rri3 - 1.) + 1.;
        virSum += fcVal * rr;
      }
    }
  }
}
```

The code is more concise and transparent, and the use of the vector definitions reduces the scope for typing errors that might otherwise go unnoticed. It should also be noted that by simply changing the definition of the structure `VecR` to

```
typedef struct {
  real x, y, z;
} VecR;
```

https://doi.org/10.1017/CBO9780511816581.005 Published online by Cambridge University Press

and suitably augmenting the vector operations defined above, as well as `VWrapAll`, to include a z component, the code can be used without change for three-dimensional computations. The benefits of this kind of approach will be appreciated more as the problems become increasingly complicated. It is worth reiterating that this approach to the force computations involves all $N_m(N_m - 1)/2$ pairs of atoms, and is not the way to carry out serious simulations of this kind; however, a small performance improvement might be achieved here by testing the magnitudes of the individual `dr` components as they are computed to see if they exceed `rCut`, and bypassing the atom pair as soon as this happens.

The function `LeapfrogStep` handles the task of integrating the coordinates and velocities; it appears twice in the listing of `SingleStep`, with the argument `part` determining which portion of the two-step leapfrog process, (2.3.4)–(2.3.5) or (2.3.6), is to be performed.

```
void LeapfrogStep (int part)
{
  int n;

  if (part == 1) {
    DO_MOL {
      VVSAdd (mol[n].rv, 0.5 * deltaT, mol[n].ra);
      VVSAdd (mol[n].r, deltaT, mol[n].rv);
    }
  } else {
    DO_MOL VVSAdd (mol[n].rv, 0.5 * deltaT, mol[n].ra);
  }
}
```

The function `ApplyBoundaryCond`, called after the first call to `LeapfrogStep`, is responsible for taking care of any periodic wraparound in the updated coordinates.

```
void ApplyBoundaryCond ()
{
  int n;

  DO_MOL VWrapAll (mol[n].r);
}
```

The brevity of these functions, and their applicability in both two and three dimensions, are a result of the vector definitions introduced previously.

Initial state

Preparation of the initial state uses the following three functions, one for the atomic coordinates, the others for the velocities and accelerations. The number of atoms in the system is expressed in terms of the size of the array of unit cells in which the

https://doi.org/10.1017/CBO9780511816581.005 Published online by Cambridge University Press

atoms are initially arranged, the relevant values appear in *initUcell*, which is a vector with integer components defined as

```
typedef struct {
  int x, y;
} VecI;
```

Here a simple square lattice (with the option of unequal edge lengths) is used, so that each unit cell contains just one atom, and the system is centered about the origin.

```
void InitCoords ()
{
  VecR c, gap;
  int n, nx, ny;

  VDiv (gap, region, initUcell);
  n = 0;
  for (ny = 0; ny < initUcell.y; ny ++) {
    for (nx = 0; nx < initUcell.x; nx ++) {
      VSet (c, nx + 0.5, ny + 0.5);
      VMul (c, c, gap);
      VVSAdd (c, -0.5, region);
      mol[n].r = c;
      ++ n;
    }
  }
}
```

The new vector operations used here are

```
#define VMul(v1, v2, v3)                                    \
   (v1).x = (v2).x * (v3).x,                                \
   (v1).y = (v2).y * (v3).y
```

and the corresponding *VDiv*.

The initial velocities are set to a fixed magnitude *velMag* that depends on the temperature (see below), and after assigning random velocity directions the velocities are adjusted to ensure that the center of mass is stationary. The function *VRand* (§18.4) serves as a source of uniformly distributed random unit vectors, here in two dimensions. The accelerations are simply initialized to zero.

```
void InitVels ()
{
  int n;

  VZero (vSum);
  DO_MOL {
```

https://doi.org/10.1017/CBO9780511816581.005 Published online by Cambridge University Press

```
    VRand (&mol[n].rv);
    VScale (mol[n].rv, velMag);
    VVAdd (vSum, mol[n].rv);
  }
  DO_MOL VVSAdd (mol[n].rv, - 1. / nMol, vSum);
}

void InitAccels ()
{
  int n;

  DO_MOL VZero (mol[n].ra);
}
```

New vector operations used here are

```
#define VScale(v, s)                                    \
   (v).x *= s,                                          \
   (v).y *= s
#define VVAdd(v1, v2)  VAdd (v1, v1, v2)
```

Variables

It is debatable which should be discussed first, the program, or the variables on which it operates. Here we have picked the former in order to provide some motivation for a discussion of the latter.

The scheme we have chosen is that all variables needed by more than one function are declared globally; this implies that they are accessible to all functions†. The alternative is to make extensive use of argument lists, perhaps using structures to organize the data transferred between functions; while offering a means of regulating access to variables, it makes the program longer and more tedious to read, so we forgo the practice.

Having settled this issue, what are the global variables used by the program? The list of declarations – each type ordered alphabetically – follows:

```
Mol *mol;
VecR region, vSum;
VecI initUcell;
Prop kinEnergy, pressure, totEnergy;
real deltaT, density, rCut, temperature, timeNow, uSum, velMag,
   virSum, vvSum;
int moreCycles, nMol, stepAvg, stepCount, stepEquil, stepLimit;
```

† This is not an approach recommended for large software projects because it is difficult to keep track of (and control) which variables are used where.

https://doi.org/10.1017/CBO9780511816581.005 Published online by Cambridge University Press

A new C structure is introduced for representing property measurements that will undergo additional processing,

```
typedef struct {
  real val, sum, sum2;
} Prop;
```

The three elements here are an actual measured value, a sum accumulated over several such measurements in order to evaluate the average, and a sum of squares used in evaluating the standard deviation (more on this below).

The following definition is also included, both to ensure the correct dimensionality of the vectors, and for use in formulae that depend explicitly on whether the system is two- or three-dimensional,

```
#define NDIM  2
```

Most of the names should be self-explanatory. The variable *mol* is actually a pointer to a one-dimensional array that is allocated dynamically at the start of the run and sized according to the value of *nMol*. From a practical point of view, writing **mol* in the above list of declarations is equivalent to *mol[...]* with a specific array size, except that in the former case the array size is established when the program is run rather than at compilation time†. The vector *region* contains the edge lengths of the simulation region. The other quantities, as well as a list of those variables supplied as input to the program, will be covered by the remaining functions below.

All dynamic array allocations are carried out by the function *AllocArrays*. In this example there is just a single array,

```
void AllocArrays ()
{
  AllocMem (mol, nMol, Mol);
}
```

where *AllocMem* is defined as

```
#define AllocMem(a, n, t)  a = (t *) malloc ((n) * sizeof (t))
```

and provides a convenient means of utilizing the standard C memory allocation function *malloc* while ensuring the appropriate type casting.

† The advantage of such dynamic allocation (in addition to bypassing any size limitations that some compilers might impose on arrays whose limits are included in the program source) is that it enhances program flexibility by eliminating any arbitrary built-in size assumptions.

https://doi.org/10.1017/CBO9780511816581.005 Published online by Cambridge University Press

Other variables required for the simulation (expressed in MD units when appropriate), excluding those that form part of the input data, are set by the function *SetParams*,

```
void SetParams ()
{
  rCut = pow (2., 1./6.);
  VSCopy (region, 1. / sqrt (density), initUcell);
  nMol = VProd (initUcell);
  velMag = sqrt (NDIM * (1. - 1. / nMol) * temperature);
}
```

which uses the additional definitions

```
#define VSCopy(v2, s1, v1)                                    \
   (v2).x = (s1) * (v1).x,                                    \
   (v2).y = (s1) * (v1).y
#define VProd(v)  ((v).x * (v).y)
```

The evaluation of *nMol* and *region* assumes just one atom per unit cell, and allowance is made for momentum conservation (which removes *NDIM* degrees of freedom) when computing *velMag* from the temperature.

Measurements

In this introductory case study the emphasis is on demonstrating a minimal working program. The measurements of the basic thermodynamic properties of the system that are included are covered by the following functions. The quantity *vSum* is used to accumulate the total velocity (or momentum, since all atoms have unit mass) of the system; the fact that this should remain exactly zero serves as a simple – but only partial – check on the correctness of the calculation.

The first of the functions computes the velocity and velocity-squared sums and the instantaneous energy and pressure values.

```
void EvalProps ()
{
  real vv;
  int n;

  VZero (vSum);
  vvSum = 0.;
  DO_MOL {
    VVAdd (vSum, mol[n].rv);
    vv = VLenSq (mol[n].rv);
    vvSum += vv;
  }
```

https://doi.org/10.1017/CBO9780511816581.005 Published online by Cambridge University Press

```
  kinEnergy.val = 0.5 * vvSum / nMol;
  totEnergy.val = kinEnergy.val + uSum / nMol;
  pressure.val = density * (vvSum + virSum) / (nMol * NDIM);
}
```

The second function collects the results of the measurements, and evaluates means and standard deviations upon request.

```
void AccumProps (int icode)
{
  if (icode == 0) {
    PropZero (totEnergy);
    PropZero (kinEnergy);
    PropZero (pressure);
  } else if (icode == 1) {
    PropAccum (totEnergy);
    PropAccum (kinEnergy);
    PropAccum (pressure);
  } else if (icode == 2) {
    PropAvg (totEnergy, stepAvg);
    PropAvg (kinEnergy, stepAvg);
    PropAvg (pressure, stepAvg);
  }
}
```

Depending on the value of the argument *icode* (0, 1 or 2), *AccumProps* will initialize the accumulated sums, accumulate the current values, or produce the final averaged estimates (which overwrite the accumulated values). The following operations† are defined for use with the *Prop* structures (*Max* is defined in §18.2):

```
#define PropZero(v)                                          \
   v.sum = 0.,                                               \
   v.sum2 = 0.
#define PropAccum(v)                                         \
   v.sum += v.val,                                           \
   v.sum2 += Sqr (v.val)
#define PropAvg(v, n)                                        \
   v.sum /= n,                                               \
   v.sum2 = sqrt (Max (v.sum2 / n - Sqr (v.sum), 0.))
#define PropEst(v)                                           \
   v.sum, v.sum2
```

† While the argument of the square-root function evaluated here should never be negative, the *Max* test is included to guard against computer rounding error in cases where the result is close to zero.

https://doi.org/10.1017/CBO9780511816581.005 Published online by Cambridge University Press

Input and output

The function *GetNameList*, called from *main*, reads all the data required to specify the simulation from an input file. It uses a Fortran-style (almost) 'namelist' to group all the data conveniently and automate the input task. It also checks that all requested data items have been provided. For this case study the list of variables is specified in the following way:

```
NameList nameList[] = {
  NameR (deltaT),
  NameR (density),
  NameI (initUcell),
  NameI (stepAvg),
  NameI (stepEquil),
  NameI (stepLimit),
  NameR (temperature),
};
```

The C macro definitions *NameR* and *NameI* are used to signify real and integer quantities (either single variables, or entire structures with all members of that type). The name of the data file from which the input values are read is derived from the name of the program (if the program happens to be called *md_prog* then the data file should be named *md_prog.in*). The function *PrintNameList*, also called by *main*, outputs an annotated copy of the input data. Full details of these functions and macros appear in §18.5; *VCSum* simply adds the vector components.

Output from the run is produced by

```
void PrintSummary (FILE *fp)
{
  fprintf (fp,
     "%5d %8.4f %7.4f %7.4f %7.4f %7.4f %7.4f %7.4f %7.4f\n",
     stepCount, timeNow, VCSum (vSum) / nMol, PropEst (totEnergy),
     PropEst (kinEnergy), PropEst (pressure));
}
```

Data are written to a file, which in the present case is just the user's terminal because the call to *PrintSummary* in *SingleStep* used the argument *stdout*. By calling this function twice, with different arguments, output can be sent both to the terminal and to a file that logs all the output†.

† The reader unfamiliar with standard C library functions will find *fprintf* – and numerous other functions used later – described in any text on the C language, or in generally available C documentation.

https://doi.org/10.1017/CBO9780511816581.005 Published online by Cambridge University Press

2.5 Results

In this section we present a few of the results that can be obtained from simulations of the two-dimensional soft-disk fluid. In view of the fact that the MD algorithm described here is far from efficient, the results will mostly be confined to short simulation runs of small systems, just to give a foretaste of what is to come. More detailed results based on more extensive computations will appear later.

The input file used in the first demonstration contains the following entries:

```
deltaT          0.005
density         0.8
initUcell       20 20
stepAvg         100
stepEquil       0
stepLimit       10000
temperature     1.
```

The initial configuration is a 20×20 square lattice so that there are a total of 400 atoms. The timestep value `deltaT` is determined by the requirement that energy be conserved by the leapfrog method (to be discussed in §3.5). The initial temperature is $T = 1$; temperature will fluctuate during the run, and no attempt will be made here to set the mean temperature to any particular value.

Conservation laws

The most obvious test that the computation must pass is that of momentum and energy conservation. While the former is intrinsic to the algorithm and, assuming periodic boundaries, its violation would suggest a software error, the latter is sensitive to the choice of integration method and the size of Δt. One quantity that is not conserved is angular momentum; a conservation law requires the system to be invariant under some change, such as translation, but, because of the periodic boundaries, the rotational invariance needed for angular momentum conservation is not applicable. Programming errors can sometimes (but not always) be detected by the violation of a conservation law; when this occurs the effect can be gradual, intermittent, or catastrophic, depending on the cause of error.

In Table 2.1 we show an edited version of the output† of the run specified above; the results listed are the sum of the velocity components, the mean energy and kinetic energy per atom, their standard deviations, and the mean pressure. Clearly, energy and momentum are conserved as expected, kinetic energy fluctuates by a

† Note that the higher-order digits of some of the values listed here – and elsewhere in the book – may vary, depending on the computer, compiler and level of optimization; this is an expected consequence of the trajectory sensitivity, discussed later in this section.

https://doi.org/10.1017/CBO9780511816581.005 Published online by Cambridge University Press

Table 2.1. Edited output from a short MD run.

timestep	$\sum v$	$\langle E \rangle$	$\sigma(E)$	$\langle E_K \rangle$	$\sigma(E_K)$	$\langle P \rangle$
100	0.0000	0.9952	0.0002	0.6555	0.0910	4.5751
200	0.0000	0.9951	0.0001	0.6493	0.0118	4.5802
300	0.0000	0.9951	0.0001	0.6398	0.0168	4.6445
400	0.0000	0.9951	0.0000	0.6476	0.0155	4.5685
500	0.0000	0.9951	0.0000	0.6599	0.0167	4.4682
1000	0.0000	0.9950	0.0000	0.6481	0.0256	4.5489
2000	0.0000	0.9951	0.0001	0.6500	0.0125	4.5370
3000	0.0000	0.9951	0.0001	0.6301	0.0166	4.6898
5000	0.0000	0.9952	0.0001	0.6410	0.0139	4.6254
10000	0.0000	0.9949	0.0001	0.6535	0.0205	4.4886

limited amount, and it is also apparent that as a result of some of the initial kinetic energy being converted to potential energy the temperature of the system (here $T = E_K$) has dropped considerably below the initial setting.

Equilibration

Characterizing equilibrium is by no means an easy task, especially for small systems whose properties fluctuate considerably. Averaging over a series of timesteps will reduce the fluctuations, but different quantities relax to their equilibrium averages at different rates, and this must also be taken into account when trying to establish when the time is ripe to begin making measurements. Fortunately, relaxation is generally quite rapid, but one must always beware of those situations where this is not true. Equilibration can be accelerated by starting the simulation at a higher temperature and later cooling by rescaling the velocities (this is similar, but not identical, to using a larger timestep initially); too high a temperature will, however, lead to numerical instability.

One simple measure of equilibration is the rate at which the velocity distribution converges to its expected final form. Theory [mcq76] predicts the Maxwell distribution

$$f(\boldsymbol{v}) = \frac{\rho}{(2\pi T)^{d/2}} e^{-v^2/2T} \tag{2.5.1}$$

(in MD units) which, after angular integration, becomes

$$f(v) \propto v^{d-1} e^{-v^2/2T} \tag{2.5.2}$$

https://doi.org/10.1017/CBO9780511816581.005 Published online by Cambridge University Press

The distribution can be measured by constructing a histogram of the velocity values $\{h_n \mid n = 1, \ldots N_b\}$, where h_n is the number of atoms with velocity magnitude between $(n-1)\Delta v$ and $n\Delta v$, $\Delta v = v_m/N_b$, and v_m is a suitable upper limit to v. The normalized histogram represents a discrete approximation to $f(v)$.

The function that carries out this computation♠ is

```
void EvalVelDist ()
{
  real deltaV, histSum;
  int j, n;

  if (countVel == 0) {
    for (j = 0; j < sizeHistVel; j ++) histVel[j] = 0.;
  }
  deltaV = rangeVel / sizeHistVel;
  DO_MOL {
    j = VLen (mol[n].rv) / deltaV;
    ++ histVel[Min (j, sizeHistVel - 1)];
  }
  ++ countVel;
  if (countVel == limitVel) {
    histSum = 0.;
    for (j = 0; j < sizeHistVel; j ++) histSum += histVel[j];
    for (j = 0; j < sizeHistVel; j ++) histVel[j] /= histSum;
    PrintVelDist (stdout);
    countVel = 0;
  }
}
```

in which the definitions `Min` (§18.2) and

```
#define VLen(v)  sqrt (VDot (v, v))
```

are used. Depending on the value of `countVel`, the function will, in addition to adding the latest results to the accumulated total, either initialize the histogram counts, or carry out the final normalization. Other kinds of analysis in subsequent case studies will involve functions that operate in a similar manner.

In order to use this function storage for the histogram array must be allocated, and a number of additional variables declared and assigned values. The variables are

```
real *histVel, rangeVel;
int countVel, limitVel, sizeHistVel, stepVel;
```

♠ `pr_02_2`

https://doi.org/10.1017/CBO9780511816581.005 Published online by Cambridge University Press

and those included in the input data must be added to the array *nameList*,

```
NameI (limitVel),
NameR (rangeVel),
NameI (sizeHistVel),
NameI (stepVel),
```

Allocation of the histogram array is included in *AllocArrays*,

```
AllocMem (histVel, sizeHistVel, real);
```

Initialization, in *SetupJob*, requires the additional statement

```
countVel = 0;
```

and the histogram function is called from *SingleStep* by

```
if (stepCount >= stepEquil &&
   (stepCount - stepEquil) % stepVel == 0) EvalVelDist ();
```

Histogram output is provided by the function

```
void PrintVelDist (FILE *fp)
{
  real vBin;
  int n;

  printf ("vdist (%.3f)\n", timeNow);
  for (n = 0; n < sizeHistVel; n ++) {
    vBin = (n + 0.5) * rangeVel / sizeHistVel;
    fprintf (fp, "%8.3f %8.3f\n", vBin, histVel[n]);
  }
}
```

To demonstrate the way in which the velocity distribution evolves over time during the early portion of the simulation we study a system with $N_m = 2500$. Use of a larger system than before produces smoother results, and these are further improved by averaging over five separate runs with different random initial velocities; to simulate a system of this size efficiently we resorted to methods that will be introduced in §3.4, although this has no effect on the results.

The initial velocities are based on random numbers generated using a default initial seed; to change this value introduce a new integer variable *randSeed* (whose default value is arbitrarily set to 17) and in *SetupJob* use this value to initialize a

https://doi.org/10.1017/CBO9780511816581.005 Published online by Cambridge University Press

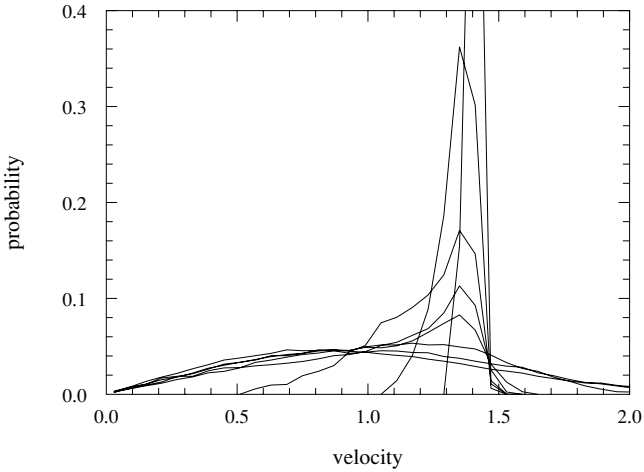

Fig. 2.3. Velocity distribution as a function of time; successively broader graphs are at times 0.02, 0.04, 0.06, 0.08, 0.1, 0.2, 0.4, and 1.0 (the zero-time state – not shown – is a spike at the initial velocity $\sqrt{2}$).

different random sequence by the call

```
InitRand (randSeed);
```

Also add

```
NameI (randSeed),
```

to the array `nameList`. The input data are as above, except for

```
deltaT          0.001
initUcell       50 50
limitVel        4
randSeed        17
rangeVel        3.
sizeHistVel     50
stepVel         5
```

and `randSeed` is different for each run. The results are shown in Figure 2.3; the final distribution develops rapidly and is reached within about 0.4 time units. From results of this kind it is clear that there is no need to assign an initial velocity distribution carefully – the system takes care of this matter on its own (for very small systems there will be deviations from the theoretical distribution [ray91]).

https://doi.org/10.1017/CBO9780511816581.005 Published online by Cambridge University Press

The Boltzmann H-function occupies an important position in the development of statistical mechanics [hua63]. It is defined as

$$H(t) = \int f(\boldsymbol{v}, t) \log f(\boldsymbol{v}, t)\, d\boldsymbol{v} \tag{2.5.3}$$

and it can be proved that $\langle dH/dt \rangle \leq 0$, with equality only applying when $f(\boldsymbol{v})$ is the Maxwell distribution. In order to compute $H(t)$ we use the velocity histogram $\{h_n\}$ obtained previously; if we neglect constants, $H(t)$ can be approximated by

$$h(t) = \sum_n h_n \log(h_n / v_n^{d-1}) \tag{2.5.4}$$

An additional variable is required for this computation, namely,

```
real hFunction;
```

and the following code must be added to the summary phase of `EvalVelDist` (for the two-dimensional case),

```
hFunction = 0.;
for (j = 0; j < sizeHistVel; j ++) {
  if (histVel[j] > 0.) hFunction += histVel[j] * log (histVel[j] /
     ((j + 0.5) * deltaV));
}
```

For output, add the extra line to `PrintVelDist`

```
fprintf (fp, "hfun: %8.3f %8.3f\n", timeNow, hFunction);
```

In Figure 2.4 we show the results of this analysis for several densities, using the above system, but with a quarter the number of atoms to enhance the fluctuations. The long-time limit of the H-function depends on T (as well as ρ), and since no attempt is made to force the system to a particular temperature the limiting values will differ. Convergence is fastest at high density, while at lower density $h(t)$ does not begin to change until atoms come within interaction range. Finite systems lack the monotonicity suggested by the theorem, but the overall trend is clear and, strictly speaking, the theorem only addresses average quantities. A computation of this kind was carried out in the early days of MD [ald58]; Boltzmann would presumably have found the results much to his liking.

Thermodynamics

To provide a glimpse of what can be done, we show a few measurements made during some short test runs using as input data,

https://doi.org/10.1017/CBO9780511816581.005 Published online by Cambridge University Press

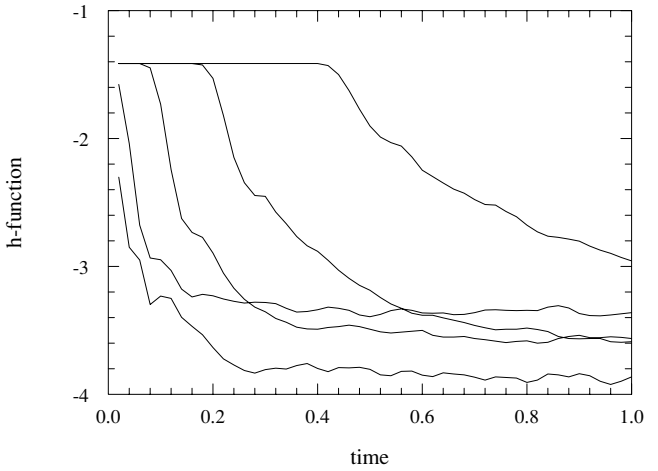

Fig. 2.4. Time dependence of the Boltzmann H-function (neglecting constants) starting from an ordered state, at densities 0.2–1.0; convergence is faster at higher density.

```
deltaT          0.005
density         0.8
initUcell       20 20
stepAvg         1000
stepEquil       1000
stepLimit       3000
temperature     1.
```

Various values of `density` are used; any data items not explicitly shown take values specified previously. The output is summarized in Table 2.2.

It is unlikely that the temperature (here just $\langle E_K \rangle$) is the one wanted, and the value will certainly not be the one used to create the initial state. To obtain a particular $\langle T \rangle$, the velocities must be adjusted over a series of timesteps until the system settles down at the correct state point. The actual velocity rescaling should be based on $\langle T \rangle$, and not on the instantaneous T values that may be subject to considerable fluctuation. Though not apparent here, the energy can gradually drift upward because of the numerical error in the leapfrog method; the drift rate for a given temperature depends on Δt and is negligible for sufficiently small values. We will return to these matters in Chapter 3.

https://doi.org/10.1017/CBO9780511816581.005 Published online by Cambridge University Press

Table 2.2. Measurements from soft-disk simulations at different densities: total energy, kinetic energy and pressure are shown.

ρ	$\langle E \rangle$	$\langle E_K \rangle$	$\sigma(E_K)$	$\langle P \rangle$	$\sigma(P)$
0.4	0.9935	0.917	0.014	0.803	0.056
0.6	0.9936	0.823	0.016	1.955	0.099
0.8	0.9952	0.645	0.022	4.578	0.165

Trajectories

The first opportunity for using MD to provide results that are unobtainable by other means is in the study of the trajectories followed by individual atoms. Clearly, a single trajectory conveys very little information, but if the trajectories of groups of nearby atoms are examined a clear picture emerges of the different behavior in the solid, liquid and gaseous states of matter. In the solid phase the atoms are confined to small vibrations around the sites of a lattice, the gas is distinguished by trajectories that are ballistic over relatively long distances, while the liquid is characterized by generally small steps, occasional rearrangement, and no long-range positional order. The differences in the trajectories are reflected at the macroscopic level by the values of the diffusion coefficient. Diffusion is just the mean-square atomic displacement (after allowing for periodic wraparound in the MD case), and is one example of a transport process that MD can examine directly; we will return to this in Chapter 5.

The best way to observe these features is by running an MD simulation interactively and watching the trajectories as they develop for different T and ρ. Trajectories can be shown on a computer display screen by simply drawing a line between the atomic positions every few timesteps; whenever a periodic boundary is crossed simply interrupt the trajectory drawing and restart it from the opposite boundary. Suitable graphics functions are readily added to the program; all that is required, apart from setting up the display functions and arranging for atomic coordinates to be converted to screen coordinates, is the decision as to how frequently the display should be updated. Typical trajectories obtained in the solid and dense fluid phases appear in Figure 2.5.

An example of a simple interactive MD simulation is shown in Figure 2.6. Here the user interface permits realtime control of the computation, including the choice of system size, altering the values of T and ρ, and changing the display update rate. The details involved in writing such programs depend on the computer and software environment; this two-dimensional example is described in [rap97], although

https://doi.org/10.1017/CBO9780511816581.005 Published online by Cambridge University Press

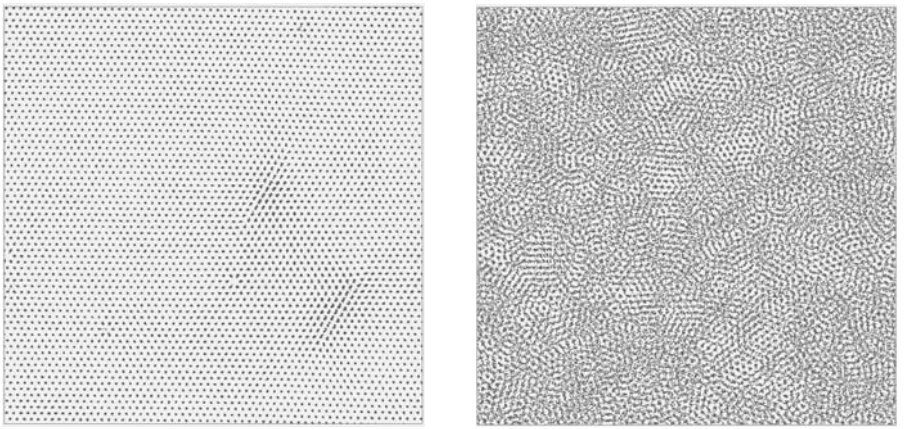

Fig. 2.5. Trajectory plots at densities of 1.05 and 0.85 showing the difference between solid and dense fluid phases, namely, localized and diffusing trajectories.

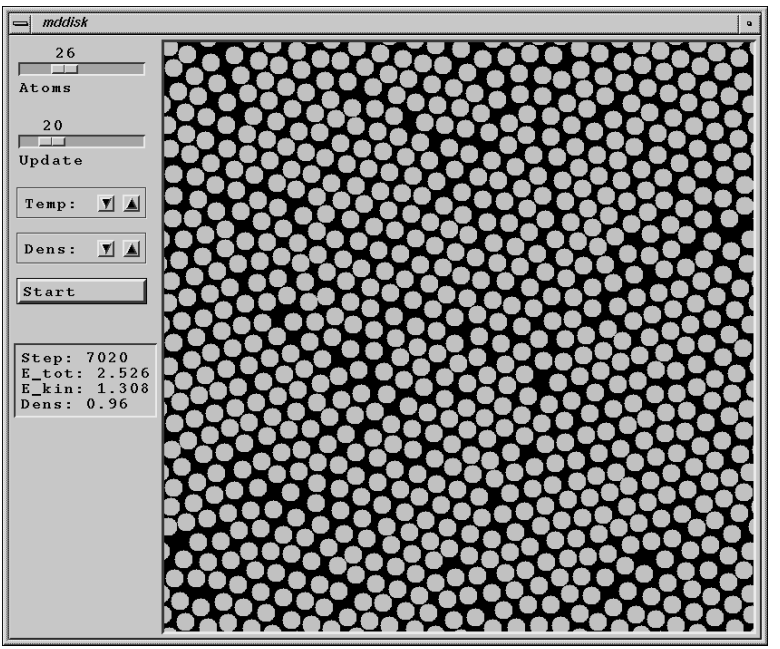

Fig. 2.6. Example of an interactive simulation.

a little more effort would be required for the corresponding three-dimensional case. Visualization plays an essential role in many kinds of problem, and the ability to interact with the simulation while in progress can prove to be of considerable value.

https://doi.org/10.1017/CBO9780511816581.005 Published online by Cambridge University Press

2.6 Further study

2.1 Compare the observed velocity distribution with the theoretical result (2.5.2).

2.2 Check that the correct limiting values of $H(t)$, defined in (2.5.3), are obtained.

2.3 Extend the graphics capability of the interactive MD program so that trajectories can be displayed.

https://doi.org/10.1017/CBO9780511816581.005 Published online by Cambridge University Press