

# CHE T580: Modern Molecular Simulations

Cameron F. Abrams

## 1 Introduction

### 1.1 Course Objectives

The objective of this course is to provide graduate students with a basic understanding of molecular simulation. The level of understanding we will target involves both the skills to critique current molecular simulation research in the students' respective fields and to develop simple simulation experiments that may contribute to students' original research.

### 1.2 Course Outline

- Statistical Mechanics
- Computing under GNU/Linux
- Monte Carlo Simulation (MC)
- Molecular Dynamics Simulation (MD)
- Modern Open-Source MD Packages
- Visualization
- Course Projects

### 1.3 Prerequisites

Calculus and differential equations through the undergraduate level. Undergraduate physical chemistry (rudimentary quantum and statistical mechanics). Some knowledge of programming in C, FORTRAN, and/or Python will be helpful. A laptop on which various software can be installed (more on that in Assignment 1). You must have a github account.

### 1.4 Introductory Remarks

In this course, we are concerned with systems of many particles. Such a system models a chunk of matter, and the particles are its constituents. *Statistical mechanics* allows us predict the macroscopic properties and behavior of matter when conceptualized as collections of many ( $\sim 10^{23}$ ) particles, and it is the central theme of this course. As we will see, the formalism of statistical mechanics allows straightforward analytical treatment of only a few simple systems. The primary motivation behind molecular simulation is to apply the framework of statistical mechanics in the prediction of macroscopic behavior for relatively "complicated" systems. The two major branches of molecular simulation we will consider are (1) Monte Carlo (MC) and (2) molecular dynamics (MD).

Statistical mechanics is a broad subject, and we will restrict ourselves in this course to stat mech at the introductory level. The lessons of elementary stat mech will be reinforced time and time again throughout the course as we explore aspects of molecular simulation techniques. Stat mech is not normally taught to engineering students outside a general course on physical chemistry. (Undergraduates in physics or chemistry might get a one or two course series on statistical mechanics.)

A second component of this course is programming. Most engineering students take a programming course, so some exposure to computer programming is expected as a prerequisite. It will be necessary to discuss certain simulation algorithms using "pseudo-code" examples, or even examples written in C or FORTRAN or Python. The level of code presented in this course will generally be sufficiently basic

such that novices can understand it line by line. For example, see if you can predict what the following C program does:

```
#include <stdio.h>
int main () {
    int i;
    for (i=0;i<100;i++) printf("Hello, I am number %i\n",i);
}
```

Maybe you'd like it better in Python:

```
for i in range(100):
    print('Hello, I am number',i)
```

If it looks somewhat mysterious to you, don't worry. Part of this course will be explaining how code works. By the end of the course, you would be able to *write* the above program had I told you to write a program to output the numbers 0 - 99. It must be emphasized at this point that I do not intend to turn you all into expert coders (though there are worse things to aim for). This is a survey course from which I hope you gain an accurate picture of the field of molecular simulation from which you can begin your own exploration. In order to achieve this goal, it is necessary to do some *minor* work with actual code.

A closely related aspect of this component is the practical matter of how one *works* with simulation code. My preference, and therefore the manner in which I teach the course, is to use a command-line environment, as opposed to a graphical user environment. It should be emphasized that the concepts and ideas that form the backbone of this course are not operating system-specific. However, the *implementation* of those ideas and concepts, as I hope you will see, is straightforward once you know how to compile and run simple programs at the command-line. Again, this will be at a basic level; this is not a programming course. All of the production MD codes we consider later in the course are *exclusively* used in a command-line environment. For writing and editing code, I recommend VSCode.

Much of this course is based on the book *Understanding Molecular Simulation* by Daan Frenkel and Berend Smit, [1] two chemical engineers from Amsterdam. There are several other books which I have used on occasion, the most useful of which was *Computer Simulation of Liquids* by Allen and Tildesly. [2].

## 2 Statistical Mechanics: A Brief Introduction

This course is centered upon a mathematical statement called an “ensemble average”:

$$\langle G \rangle = \sum_{\nu} P_{\nu} G_{\nu} \quad (1)$$

That is, the *expectation value*,  $\langle G \rangle$ , of some observable property  $G$  is an *average* over all possible microstates available to a system, indexed by  $\nu$ , where  $P_{\nu}$  is the probability of observing the system in microstate  $\nu$ , and  $G_{\nu}$  is the value of the measured property  $G$  when the system is in microstate  $\nu$ . Before even considering how to use computer simulation to make such a measurement of a particular property for a particular system, there are three main issues to consider:

1. What is a microstate?
2. What is meant by observing the system?
3. How do we calculate probabilities?

In the following subsections, we give a cursory treatment of elementary statistical mechanics aimed at answering these questions. The aim is to give the student an appreciation of the basic physics that underlies a majority of current molecular simulation.

### 2.1 Microstates and Degeneracy

A microstate is a full specification of all degrees of freedom of a system. In quantum mechanics, degrees of freedom are *quantum numbers*. The index  $\nu$  in Eq. 1 runs over all unique combinations of quantum number values. Equilibrium (eigen)solutions of the Schrödinger equation define the energy  $E_{\nu}$  of any state  $\nu$ :

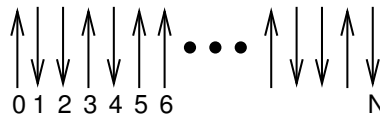
$$\mathcal{H} |\nu\rangle = E_{\nu} |\nu\rangle, \quad (2)$$

where  $\mathcal{H}$  is the Hamiltonian operator,  $|\nu\rangle$  is shorthand for the system wavefunction in state  $\nu$ , and  $E_{\nu}$  is the energy of state  $\nu$  ( $E_{\nu}$  is an eigenvalue of the Schrödinger equation). In contrast to model systems usually considered in elementary quantum mechanics, the number of distinct microstates of systems with energy  $E$  and comprising  $\sim 10^{23}$  particles is *very large*, and this set of eigenstates is in practice impossible to obtain explicitly. This is indeed why we must instead treat this set *statistically*. We refer to the number of states that satisfy a given energy as the degeneracy of the energy level  $E$ , denoted  $\Omega(E, N, V)$ :

$$\Omega(N, V, E) = \begin{array}{l} \text{number of microstates with } N \text{ and } V \text{ and} \\ \text{energy between } E \text{ and } E + \delta E. \end{array}$$

The many states contributing to the count  $\Omega(N, V, E)$  is called a *microcanonical ensemble*. Because one in principle can partition state space into non-overlapping sets of states where each set represents a unique value of  $E$ ,  $\Omega$  is also called the “microcanonical partition function”.

The Ising spin lattice is a simple statistical mechanical model with discrete energy levels which we can now introduce to gain some understanding of what it means to say  $\Omega$  is “large.” Imagine a linear array of  $N$  spins, each pointing either “up” or “down.”



**Figure 1:** A 1-D Ising system.

Let us suppose that the Hamiltonian of this system is given by

$$\mathcal{H} = h \sum_{i=1}^N s_i \quad (3)$$

where  $s_i$  is -1 if spin  $i$  is “down” and +1 if spin  $i$  is “up,” and  $h$  is some unit of energy. Let  $m$  denote the number of spins that are up out of the  $N$  spins, and let  $\Omega(N, m)$  be the number of realizations of the  $N$  spin states for which  $m$  spins are up. The ground state, the state with the lowest energy, has all spins down, so  $\Omega(N, 0) = 1$ . The next state up has one spin up, but there are  $N$  possible microstates that have this energy:  $\Omega(N, 1) = N$ . The next state up has two spins, and there are  $N(N - 1)/2$  such microstates:  $\Omega(N, 2) = N(N - 1)/2$ . For  $m$  spins flipped, there are

$$\Omega_m = \frac{N!}{(N - m)!m!} \quad (4)$$

distinct microstates. You can now easily see that working with  $\Omega$  for statistical mechanical systems means working with *enormous* numbers. For the rather small system of 100 spins, if we ask how many states are there with 50 spins up, we see  $\Omega(100, 50) \approx 10^{29}$ .

Although quantum mechanics tells us that atomic systems have discrete energy levels, when systems contain very large numbers of atoms, these energy levels become so closely spaced relative to their span that they may effectively be considered a continuum. We can thus pass into a *classical* (as opposed to quantum mechanical) representation, where the microstate for a system of  $N$  particles is specified by a point in a  $6N$ -dimensional *phase space*:

$$(r^N, p^N) \equiv (r_1, r_2, \dots, r_N; p_1, p_2, \dots, p_N). \quad (5)$$

where  $r_i$  is the 3-space position of particle  $i$  and  $p_i$  is its momentum. We can denote the number of states in a classical microcanonical ensemble by integrating over all of phase space and plucking out those states for which the energy is  $E$  using the Dirac delta function:

$$\bar{\Omega}(N, V, E) = \frac{1}{h^{3N} N!} \int_V dr_1 \int_V dr_2 \cdots \int_V dr_N \int_{R^3} dp_1 \int_{R^3} dp_2 \cdots \int_{R^3} dp_N \delta[\mathcal{H}(r^N, p^N) - E] \quad (6)$$

$$\equiv \int \int dr^N dp^N \delta[\mathcal{H}(r^N, p^N) - E], \quad (7)$$

(The second line introduces some shorthand notation.) The delta function integrand has units that are the reciprocal of its argument, so  $\bar{\Omega}$  is more precisely termed the “density of states” and is directly related to  $\Omega$ :

$$\bar{\Omega} \delta E = \Omega \quad (8)$$

This definition satisfies the idea that the integral of  $\bar{\Omega}$  over the entire continuous domain of energy  $E$  should equal a complete integral over all of phase space.

The microcanonical ensemble represents a hyperdimensional surface in the phase space dimensioned by  $N$  particles with positions limited by the extent of  $V$ . The factorial in Eq. 6,  $N!$ , takes into account that the particles are indistinguishable; that is, all orderings of particle indices 1, 2, . . . ,  $N$ , are treated identically.  $h$  is Planck’s constant; note that it has units of length•momentum. Think of it as a quantum-mechanically-required “mesh discretization” for continuous space (it arises due to the Heisenberg uncertainty relation). It also *nondimensionalizes* the partition function. We will encounter it again in the next section, but we will also see why these “prefactors” are not essential ingredients of most molecular simulations.

You may wonder why there seem to be two viewpoints of statistical mechanics, quantum and classical. First, there really aren't two viewpoints: the classical picture is an approximation of the more general quantum mechanical picture. But statistical mechanics as a discipline was first formalized by Gibbs and Boltzmann *before* quantum mechanics was widely accepted, so it dealt necessarily with systems of classical particles obeying Newtonian equations of motion; that is, on classical mechanics. There appears to be a general consensus that it is easier to introduce statistical mechanical concepts using the “sum-over-states” notation of quantum statistical mechanics, rather than the apparently more cumbersome (and anyway approximate) “integral-over-phase-space” notation of classical statistical mechanics.

## 2.2 Making Observations: The Ergodic Hypothesis

Scientists are taught early on that when conducting measurements, one must perform repeated experiments and average the results. If one makes  $\mathcal{N}$  independent measurements of some observable  $G$ , one computes the mean value as

$$G_{\text{obs}} = \frac{1}{\mathcal{N}} \sum_{i=1}^{\mathcal{N}} G_i. \quad (9)$$

We now imagine that the time of a measurement is so short that we *know* that the system is in only one of its many possible microstates. This means we can write

$$G_{\text{obs}} = \sum_{\nu} \left[ \frac{1}{\mathcal{N}} \left( \begin{array}{c} \text{number of times state } \nu \text{ is} \\ \text{observed in the } \mathcal{N} \text{ observations} \end{array} \right) \right] G_{\nu}, \quad (10)$$

where  $G_{\nu}$  was introduced previously (Eq. 1) as the value of observable  $G$  when the system is in state  $\nu$ .

Now we have to imagine that our system is evolving in time. As it evolves, its degrees of freedom change values, and the system is thought of as tracing out a *trajectory* in state space. (“State space” is a Hilbert space spanned by all states  $|\nu\rangle$  in the quantum mechanical case, or phase space in the classical case.) How is the system evolving? The system wavefunction evolves according to Schrödinger's equation, while particles in a classical system follow Newtonian mechanics. As the experimenters, we control the system by specifying a handful of variables, such as its total energy,  $E$ , the number of particles,  $N$ , and the volume,  $V$ . These *constraints* force the system's trajectory to remain in a designated partition of state space.

The key assumption we make at this point is that, if we wait long enough, our system will visit every possible state; that is, the trajectory will eventually pass through every available point in state space consistent with our constraints (that is, all states in the partition). If this is true, and we make  $\mathcal{N}$  independent observations, then the number of times we observe the system in state  $\nu$  divided by the number of observations,  $\mathcal{N}$ , is the *probability* of observing state  $\nu$ ,  $P_{\nu}$ , if we happen to make a random observation. So, Eq. 10 above becomes the ensemble average first presented in Eq. 1:

$$G_{\text{obs}} = \sum_{\nu} P_{\nu} G_{\nu} = \langle G \rangle \quad (11)$$

This assumption is important: it is referred to as the *ergodic hypothesis*. A system is “ergodic” if, after a sufficiently long time, it visits all possible state space points consistent with whatever constraints are put on it. We cannot in general prove that any system is ergodic; it is something we are comfortable assuming for most systems based on our physical intuition. There are, however, many systems which are non-ergodic on the time-scales they are measured. The ergodic hypothesis is only strictly true when the number of measurements taken is sufficiently long that the *computed* probabilities  $P_{\nu}$  no

longer change, which can only be guaranteed for an *infinite* number of observations:

$$\langle G \rangle = \lim_{\mathcal{N} \rightarrow \infty} \frac{1}{\mathcal{N}} \sum_{i=1}^{\mathcal{N}} G_{\nu(i)} \quad (12)$$

where  $\nu(i)$  refers to the state of the system at measurement  $i$ .

Another important consideration is the following: How far apart must the  $\mathcal{N}$  independent measurements be from one another in time to be considered truly “independent”? To answer this question, we must introduce the notion of a *relaxation time*,  $\tau_{\text{relax}}$ , which arises naturally due to the presumably chaotic nature of the microscopic system. Given some initial conditions, after a time  $\tau_{\text{relax}}$  has elapsed, the system has “lost memory” of the initial condition. We measure this loss of memory in terms of *correlation functions*, which will be discussed in more detail later. If we wait at least  $\tau_{\text{relax}}$  between successive observations, we know they are more likely to be independent. It turns out that one can use simulation methods to estimate relaxation times (and their spectra; many systems display a broad spectrum of relaxation times, each element corresponding to a particular type of molecular motion). We will pay particularly close attention to  $\tau_{\text{relax}}$  in upcoming sections.

### 2.3 Entropy and Temperature

Eq. 2.1 introduced the quantity  $\Omega(N, V, E)$  as the number of states available to a system under the constraints of constant number of particles,  $N$ , volume,  $V$ , and energy  $E$ . The **fundamental postulate** of statistical mechanics, also called the “rational basis”, is the following:

In statistical equilibrium, all states consistent with the constraints of  $N$ ,  $V$ , and  $E$  are *equally probable*.

or

$$P_{\nu} = 1/\Omega(N, V, E). \quad (13)$$

This relation is often referred to as a statement of the “equal *a priori* probabilities in state space.” Another way of saying the same thing: The probability distribution for states in the microcanonical ensemble is uniform.

This postulate reflects the fact that we are maximally uncertain with regards to the probabilities of any particular arrangements of degrees of freedom inside a *closed system*. A closed system is one which cannot exchange energy, volume, or particles with the environment. As such, there is quite literally no way for us to learn anything at all about how particles are arranged, so we must assume all arrangements that satisfy the given energy, volume and number of particles are equiprobable.

One link between statistical mechanics and classical thermodynamics is given by a definition of entropy:

$$S \equiv k_B \ln \Omega \quad (14)$$

Note two important properties of  $S$ . First, it is extensive: if we consider a compound system made of subsystems  $A$  and  $B$  with  $\Omega_A$  and  $\Omega_B$  as the respective number of states, the total number of states is  $\Omega_A \Omega_B$ , and therefore  $S = S_A + S_B$ . Second, it is consistent with the second law of thermodynamics: putting any constraint on the system lowers its entropy because the constraint lowers the number of accessible states.

Temperature is defined using entropy:  $1/T \equiv (\partial S / \partial E)_{N, V}$ , or

$$\beta = (k_B T)^{-1} = (\partial \ln \Omega / \partial E) \quad (15)$$

Now we will consider constraining our system not with constant  $E$ , but with constant  $T$ . The set of all possible states satisfying constraints of  $N$ ,  $V$ , and  $T$  is called the *canonical ensemble*. We now

ask, what is the probability of any microstate in this ensemble? Consider a closed system divided into a small subsystem  $A$  surrounded by a large “bath”  $B$ . We imagine that these two subsystems exchange only thermal energy, but no particles, and their volumes remain fixed. We seek to compute the probability of finding the total system in a state such that subsystem  $A$  has energy  $E_A$ . The entire system is microcanonical, so the total energy,  $E$  is constant, as is the total number of states available to the system,  $\Omega(E)$  (we omit the  $N$  and  $V$  for simplicity).

When  $A$  has energy  $E_A$ , the total system energy is  $E = E_A + E_B$ , where  $E_B$  is the energy of the bath. By constraining system  $A$ 's energy, we have reduced the number of states available to the whole system to  $\Omega(E - E_A)$ . So, using the fundamental postulate, the probability of observing the closed system in a state in which subsystem  $A$  has energy  $E_A$  is

$$P_A = \frac{\left[ \begin{array}{c} \text{Number of states for which} \\ \text{subsystem } A \text{ has energy } E_A \end{array} \right]}{\left[ \begin{array}{c} \text{Number of states available} \\ \text{to entire system} \end{array} \right]} = \frac{\Omega(E - E_A)}{\Omega(E)} \quad (16)$$

We can expand  $\Omega(E - E_A)$  in a Taylor series around  $E_A = 0$ :

$$P_A \propto \Omega(E - E_A) = \exp[\ln \Omega(E - E_A)] \quad (17)$$

$$= \exp\left[\ln \Omega(E) - E_A \frac{\partial \ln \Omega}{\partial E} + \dots\right], \quad (18)$$

where the partial derivative implies we are holding  $N$  and  $V$  fixed. We can truncate the Taylor expansion at the first-order term, because higher order terms become less and less important as the size of subsystem  $B$  becomes larger and larger. What results is the Boltzmann distribution law for energies of a system at constant temperature:

$$P_A \propto \exp(-\beta E_A) \quad (19)$$

The normalization condition requires that for all energies of subsystem  $A$ ,  $E_A$ ,

$$\sum_A P_A = 1 = \sum_A \exp(-\beta E_A) \equiv Q(N, V, T), \quad (20)$$

which defines the *canonical partition function*,  $Q$ . Therefore,

$$P_A = Q^{-1} \exp(-\beta E_A) \quad (21)$$

Because some energies can correspond to more than one microstate, we should distinguish between “states” and “energy levels.” We can express the canonical partition function as

$$Q = \sum_{\substack{\nu \\ \text{states}}} e^{-\beta E_\nu} = \sum_{\substack{l \\ \text{levels}}} \Omega(E_l) e^{-\beta E_l} \quad (22)$$

where, as we have seen,  $\Omega(E)$  is the number of microstates with energy  $E$ . Moving to the continuum limit, and assuming a reference energy of  $E_{ref} = 0$ ,

$$Q \rightarrow \int_0^\infty dE \bar{\Omega}(E) e^{-\beta E} \quad (23)$$

where  $\bar{\Omega}(E)$  is the density of states. What is this equation telling us? It is telling us that  $Q$  is the *Laplace transform* of  $\bar{\Omega}$ . We know that transform pairs are *unique*, and hence, both  $Q$  and  $\bar{\Omega}$  contain the same information.

We recognize that for a system described by a canonical ensemble, the energy is a fluctuating quantity. And we now have the probability of observing a state with a given energy, so we can use Eq. 1 to compute the average energy,  $\langle E \rangle$ . Consider

$$\langle E \rangle = \langle E_\nu \rangle = \sum_\nu P_\nu E_\nu \quad (24)$$

$$= \frac{\left[ \sum_\nu E_\nu \exp(-\beta E_\nu) \right]}{\left[ \sum_\nu \exp(-\beta E_\nu) \right]} \quad (25)$$

Notice that

$$\sum_\nu E_\nu \exp(-\beta E_\nu) = -(\partial Q / \partial \beta) \quad (26)$$

Recalling that  $d \ln f(x) / dx = 1/f df/dx$ , we see that

$$\langle E \rangle = -(\partial Q / \partial \beta) / Q \quad (27)$$

$$= -(\partial \ln Q / \partial \beta)_{N,V} \quad (28)$$

Now, let us consider the average magnitude of the fluctuations in energy in the canonical ensemble.

$$\langle (\delta E)^2 \rangle = \langle (E - \langle E \rangle)^2 \rangle \quad (29)$$

$$= \langle E^2 \rangle - \langle E \rangle^2 \quad (30)$$

$$= \sum_\nu P_\nu E_\nu^2 - \left( \sum_\nu P_\nu E_\nu \right)^2 \quad (31)$$

$$= Q^{-1} \left( \frac{\partial^2 Q}{\partial \beta^2} \right)_{N,V} - Q^{-2} \left( \frac{\partial Q}{\partial \beta} \right)_{N,V}^2 \quad (32)$$

$$= \left( \frac{\partial \ln Q}{\partial \beta^2} \right)_{N,V} = - \left( \frac{\partial \langle E \rangle}{\partial \beta} \right)_{N,V} \quad (33)$$

Now, noting that the definition of heat capacity at constant volume,  $C_v$ , is

$$C_v = \left( \frac{\partial E}{\partial T} \right) \quad (34)$$

we see that

$$\langle (\delta E)^2 \rangle = k_B T^2 C_v \quad (35)$$

This is an interesting statement. It relates the magnitude of spontaneous fluctuations in the total energy of a system to that system's capacity to store or release energy due to changing its temperature.

The fact (Eq. 28) that the average energy in the canonical ensemble is related to a derivative of the log of the partition function implies that  $\ln Q$  is an important thermodynamic quantity. So, let's go back to our undergraduate thermodynamics course(s) and recall the following statement of the 1st and 2nd Law:

$$dA = -SdT - pdV + \mu dN \quad (36)$$



where  $A$  is the *Helmholtz free energy*, defined in terms of internal energy and entropy as

$$A = \langle E \rangle - TS \quad (37)$$

Now, consider the following derivative of  $A$ :

$$\left( \frac{\partial (A/T)}{\partial (1/T)} \right)_{N,V} = A + \frac{1}{T} \left( \frac{\partial A}{\partial (1/T)} \right)_{N,V} \quad (38)$$

$$= A - T \left( \frac{\partial A}{\partial T} \right)_{N,V} = A + TS = \langle E \rangle. \quad (39)$$

Therefore,

$$\left( \frac{\partial (\beta A)}{\partial \beta} \right)_{N,V} = \langle E \rangle. \quad (40)$$

Considering Eq. 28, we see that

$$\ln Q + C = -\beta A \quad (41)$$

which does indeed suggest an important link between  $\ln Q$  and the important thermodynamic quantity, the Helmholtz free energy. But what is the constant  $C$ ? To evaluate it, consider the “boundary condition” as  $T \rightarrow 0$ :

$$Q = \sum_{\nu} e^{-\beta E_{\nu}} \xrightarrow{T \rightarrow 0} e^{-\beta E_{\text{ground}}}. \quad (42)$$

Here, we have assumed that the degeneracy of the ground state,  $\Omega(E_{\text{ground}})$  is 1. This tells us that

$$\lim_{T \rightarrow 0} \ln Q = -\beta E_{\text{ground}} \quad (43)$$

Using this fact, and combining Eqs. 37 and 41, as  $T \rightarrow 0$ , we see that

$$-\beta E_{\text{ground}} + C = -\beta \underbrace{\langle E \rangle}_{E_{\text{ground}}} - \underbrace{\frac{S}{k_B}}_{\rightarrow 0 (\Omega=1)}, \quad (44)$$

Hence,  $C = 0$ . So,

$$\ln Q = -\beta A. \quad (45)$$

The quantity  $-\beta^{-1} \ln Q$  is the Helmholtz free energy,  $A$ .

## 2.4 Classical Statistical Mechanics

Analogous to the quasi-classical microcanonical partition function of Eq. 6, here is the quasi-classical representation of the canonical partition function:

$$Q_{\text{classical}} = \frac{1}{h^{dN} N!} \int \int d\mathbf{r}^N d\mathbf{p}^N \exp [-\beta \mathcal{H}(\mathbf{r}^N, \mathbf{p}^N)] \quad (46)$$

$\mathcal{H}(\mathbf{r}^N, \mathbf{p}^N)$  is the Hamiltonian function which computes the energy of a point in phase space. The probability of a point in phase space is represented as

$$P(\mathbf{r}^N, \mathbf{p}^N) = (Q_{\text{classical}})^{-1} \exp [-\beta \mathcal{H}(\mathbf{r}^N, \mathbf{p}^N)]. \quad (47)$$

So, the general “sum-over-states” ensemble average of quantum statistical mechanics, first presented in Eq. 1, becomes an integral over phase space in classical statistical mechanics:

$$\langle G \rangle = \frac{\int \int d\mathbf{r}^N d\mathbf{p}^N \exp[-\beta \mathcal{H}(\mathbf{r}^N, \mathbf{p}^N)] G(\mathbf{r}^N, \mathbf{p}^N)}{\int \int d\mathbf{r}^N d\mathbf{p}^N \exp[-\beta \mathcal{H}(\mathbf{r}^N, \mathbf{p}^N)]}, \quad (48)$$

where  $G(\mathbf{r}^N, \mathbf{p}^N)$  is the value of the observable  $G$  at phase space point  $(\mathbf{r}^N, \mathbf{p}^N)$ . Before moving on, it is useful to recognize that we normally simplify this ensemble average by noting that, for a system of classical particles, the usual choice for the Hamiltonian has the form

$$\mathcal{H}(\mathbf{r}^N, \mathbf{p}^N) = \mathcal{K}(\mathbf{p}^N) + U(\mathbf{r}^N) \quad (49)$$

where  $\mathcal{K}$  is the kinetic energy, which is only a function of momenta, and  $U$  is the potential energy, which is only a function of position. The canonical partition function,  $Q$ , can in this case be factorized:

$$Q(N, V, T) = \frac{1}{h^{3N} N!} \left\{ \int d\mathbf{p}^N \exp[-\beta \mathcal{K}(\mathbf{p}^N)] \right\} \left\{ \int d\mathbf{r}^N \exp[-\beta U(\mathbf{r}^N)] \right\} \quad (50)$$

$$= \left\{ \frac{V^N}{h^{3N} N!} \int d\mathbf{p}^N \exp[-\beta \mathcal{K}(\mathbf{p}^N)] \right\} \left\{ V^{-N} \int d\mathbf{r}^N \exp[-\beta U(\mathbf{r}^N)] \right\} \quad (51)$$

$$= Q_{\text{ideal}} Z \quad (52)$$

The quantity in the left-hand braces is the *ideal gas* partition function, because it corresponds to the case when the potential  $U$  is 0. (Note that we have multiplied and divided by  $V^N$ ; this is the equivalent of *scaling* the positions in the integration over positions.) The quantity in the right-hand braces is called the *configurational partition function*,  $Z$ .

Because the kinetic energy  $\mathcal{K}$  has the simple form,

$$\mathcal{K}(\mathbf{p}^N) = \sum_i \frac{\mathbf{p}_i^2}{2m_i}, \quad (53)$$

where  $m_i$  is the mass of particle  $i$ , the integral over particle momenta can be evaluated analytically:

$$\int d\mathbf{p}^N \exp[-\beta \mathcal{K}(\mathbf{p}^N)] = \prod_{i=1}^{3N} \int dp_i \exp\left(-\frac{p_i^2}{2mk_B T}\right) \quad (54)$$

$$= (2\pi mk_B T)^{3N/2}. \quad (55)$$

(We have assumed all particles have the same mass,  $m$ ; in the case of distinct masses, this is just a product of similar factors.)

$Q_{\text{ideal}}$  becomes

$$\frac{V^N}{N! h^{3N}} \int d\mathbf{p}^N \exp[-\beta \mathcal{K}(\mathbf{p}^N)] = \frac{V^N}{N!} \left( \sqrt{\frac{2\pi mk_B T}{h^2}} \right)^{3N} \quad (56)$$

$$= Q_{\text{ideal}}(N, V, T) = \frac{V^N}{N! \Lambda^{3N}} \quad (57)$$

where  $\Lambda$  is the *de Broglie* wavelength, a quantum-mechanical property of a particle inversely propor-

tional to its momentum (and thus inversely proportional to the *square root* of temperature):

$$\Lambda = \sqrt{\frac{h^2}{2\pi m k_B T}} \quad (58)$$

As an example, for a hydrogen atom with mass 1 amu and at room temperature (298 K),  $\Lambda \approx 1 \text{ \AA}$ . The de Broglie wavelength limits the precision by which a particle's position can be determined; for H atoms at room temperature, one is not permitted to specify their positions with a precision finer than about 1 ångstrom without violating the Heisenberg uncertainty principle of quantum mechanics. However, as we will see, in classical molecular simulations, we must lift this restriction, while never forgetting that this makes a classical representation of a molecule somewhat less realistic.

With the momentum degrees of freedom handled at finite temperature, when the observable  $G$  is a function of positions only, the ensemble average becomes a configurational average:

$$\langle G \rangle = Z^{-1} \int d\mathbf{r}^N \exp[-\beta U(\mathbf{r}^N)] G(\mathbf{r}^N). \quad (59)$$

Note that the integration over momentum yields a factor  $Q_{\text{ideal}}$  in both the numerator and denominator, and thus divides out. We can write this configurational average using a probability distribution,  $\rho_{NVT}$ , as

$$\langle G \rangle = \int d\mathbf{r}^N G(\mathbf{r}^N) \rho_{NVT}(\mathbf{r}^N) \quad (60)$$

where

$$\rho_{NVT}(\mathbf{r}^N) \equiv Z^{-1} e^{-\beta U(\mathbf{r}^N)} \quad (61)$$

is called the “canonical probability distribution.” As pointed out on p. 15 of Frenkel & Smit [1], Eq. 59 is “the starting point for virtually all classical simulations of many-body systems”; that is, it is the starting point for almost all simulations discussed in this course.

### 3 Linux and Scientific Computing

#### 3.1 The Linux Ecosystem

Molecular simulations are one class of applications of high-performance computing (HPC). HPC generally refers to the hardware and software environment that allows users to run simulations of many hundreds of thousands of degrees of freedom (or more) distributed across multiple processing elements (PE) and even over multiple nodes, in a shared, general-purpose cluster. Essentially all HPC clusters nowadays run the Linux operating system, and users are (mostly) expected to interact with such clusters via the command-line. For this class, we will mostly restrict our explorations to systems that are small enough NOT to require HPC hardware in order to run them. However, we will focus on building skills with the Linux command-line. If you are already familiar with the Linux command line, you can skip this subsection.

Linux is an operating system. The most basic (and sufficient) way to interact with Linux is via the command-line. The program responsible for monitoring the command-line, allowing the user and the operating system to interact, is called a *shell*. There are many types of shell programs you can choose to run, but the default for most Linux versions nowadays is bash. This is also the default shell when you install Ubuntu on WSL2 in Windows; in macOS X, the default shell is zsh, but this can easily be changed to bash (though this is not strictly necessary). I will demonstrate some simple exercises in bash here. In all the examples, the \$ refers to the bash prompt.

##### 3.1.1 Handling Files and Directories at the Command-Line Interface

Let's first create a subdirectory for holding all your work in this course in your WSL:

```
$ cd
$ mkdir cheT580
$ cd cheT580
```

The `cd` command alone sets the current working directory to your home directory (`/home/username/`). The `mkdir` command makes a new subdirectory under the current working directory, and the second `cd` changes the current working directory to be that directory. (`rmdir` can remove an empty directory.)

Now, let's create a simple file here, just to play around with.

```
$ echo "Hello, world!" > my_file.txt
$ ls
my_file.txt
$ cat my_file.txt
Hello, world!
$ rm my_file.txt
$ cat my_file.txt
cat: my_file.txt: No such file or directory
```

What did we do here? We created a file by *redirecting* the output of the `echo` command to `my_file.txt`. (There are many, many ways to create a file; this is just one.) We then used the `ls` command to show all files and subdirectory names in the current working directory; `my_file.txt` just happens to be the only one. We then displayed the contents of this file to the terminal using the `cat` command. Finally, we removed the file using `rm`, and when we then try to `cat` it, we get an error message indicating the file no longer exists.

Enough playing around. Let's make a directory called `assignment1`:

```
$ mkdir assignment1
```

Maybe you don't like that name; you can destroy it with `rmdir`:

```
$ rmdir assignment1
```

Let's not actually destroy this directory. If you just destroyed it, recreate it. Let's cd into it, and then clone the github repository for assignment1:

```
$ cd assignment1
$ git clone github.com:<repository-name>
```

Here, <repository-name> should be replaced with the actual name. Now, you can follow the instructions in the README.md you are viewing on github.

Some other things: You can cd "up" to the parent directory of the current working directory like this:

```
$ cd ..
```

You can always ask the shell to tell you what the current working directory is using pwd:

```
$ pwd
/home/<username>/cheT580
$
```

No matter what your current working directory is, you can cd to your home directory like this:

```
$ cd
```

Go to your home directory, and let's play with files a bit more. Let's create a new text file with the cat command. Type the following:

```
$ cat > my_file
This is a test file.
Don't panic.
<Ctrl-D>
$
```

<Ctrl-D> means perform the "control-D" key sequence, which signifies to the cat command that you are finished writing to the file. The cat command on the first line waits for you to type some file contents into the terminal, and the > redirects that input to cat to my\_file. Now, we can list the contents of the current directory (which is your home directory here) with the command ls. Guess what we will see?

```
$ ls
cheT580 my_file
$
```

If we use the -F flag with ls, we can easily see which files are files and which are directories:

```
$ ls -F
cheT580/ my_file
$
```

See the "/" after assignment1? That means it is a directory. Now, make a copy of the file my\_file called my\_file2 using the cp command:

```
$ cp my_file my_file2
$ ls
cheT580/ my_file my_file2
$
```

We can rename a file with the mv command. Rename my\_file2 to my\_file3:

```
$ mv my_file2 my_file3
$ ls -F
cheT580/ my_file my_file3
```

Notice that `my_file2` no longer exists. Now, move `my_file3` into the `cheT580/` directory with `mv`:

```
$ mv my_file3 assignment1
$ ls -F
cheT580/ my_file
$ ls -F cheT580
my_file3 assignment1/
$
```

Notice that the last command lists the contents of the `cheT580` directory. We could also `cd` into that directory and just type `ls -F`; we would see the same thing.

Those are all the basic file handling skills you will need to work with code for this course.

### 3.1.2 Pathnames

An important concept that arises because of the directory structure of Linux filesystems is *relative* and *absolute pathname*. “Relative” always refers to the current working directory, while “absolute” always refers to the *root* directory. Suppose that in the `assignment1` subdirectory of your `cheT580` subdirectory of your home directory, there is a file called `my_file`. That file can be referred to from any other directory using either a relative or an absolute pathname. Suppose you are in your home directory and you want to view the contents of that file using `cat`:

```
$ cd
$ cat cheT580/assignment1/my_file
```

The string `cheT580/assignment1/my_file` is the pathname of that specific file *relative* to your home directory. Now, no matter *what* directory you are in, you can always refer to a file using its unique *absolute* pathname:

```
$ cat /home/<username>/cheT580/assignment1/my_file
```

Absolute pathnames are a pain to type, but they have the benefit of being completely unambiguous.

### 3.1.3 (Windows) Keeping your WSL Linux Distribution Up to Date

The Ubuntu 20.04 you installed from the Microsoft store is a stable release version, but individual components of the operating system are constantly being upgraded, sometimes to fix security issues. You should get in the habit of keeping your Ubuntu up to date. This is done using `apt` in superuser mode:

```
$ sudo apt upgrade
[sudo] password for <username>:
Reading package lists... Done
Building dependency tree
Reading state information... Done
Calculating upgrade... Done
0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
```

(In this case, I'm not showing any updates since mine is up to date.) Performing this check once a week is a good idea; the default login message you see when you launch a WSL/Ubuntu terminal also informs you when updates are available.

`apt` is Ubuntu's "package manager" program, and it maintains a database of all packages installed and which ones are upgradable. To do this, it connects periodically to remote repositories (hosted by Ubuntu) in which updated packages are published. You can learn a lot about `apt` by looking at its manual pages using the `man` command:

```
$ man apt
```

### 3.1.4 Working Remotely

Although you will likely not need to do this in this course, a key concept in scientific computing using Linux is the ability to remote in to other computers. This is very easy using the command-line, and is normally done using the "secure shell" protocol's `ssh` command:

```
$ ssh <username>@hostname.domain.edu
```

Here, `hostname.domain.edu` is the fully resolved name of a remote computer on which `<username>` has login privilege. The next thing one normally needs to do is provide a password (and, if necessary, some kind of two-factor authentication, like a one-time code or responding to a push notification on your phone). Typically, once you are logged in you have a command-line interface just like you do locally. Typical workflows for remote work involve uploading data and input files for simulation runs, running the simulations, and then downloading output data back to a local machine.

Often, the "other computers" are actually login nodes that front enormous clusters of "compute nodes". In these settings, execution of simulations is actually scheduled using a batch scheduler, and "running a simulation" actually amounts to submitting the commands necessary to run the simulation to the scheduler. The job of the scheduler is to decide when to run your program based on the availability of system resources. This kind of "batch" processing is typical of high-performance computing. If you are provided an account on a cluster, you will be trained on how to submit jobs to the scheduler (among other things), and a basic working knowledge of Linux is typically assumed for this kind of training.

Many universities maintain their own HPC facilities. Drexel's University Research Computing Facility (URCF) has two main clusters: `proteus.urcf.drexel.edu` and `picotte.urcf.drexel.edu`.

## 3.2 Running Programs at the Command-Line

This is covered in Assignment 1, and I just go over basics here.

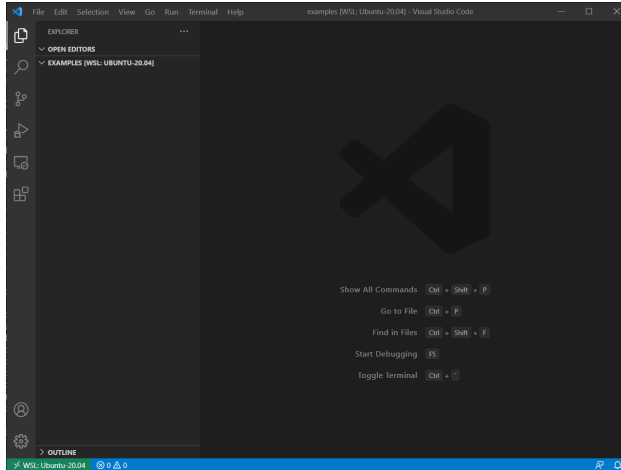
### 3.2.1 Programs that are Compiled: C

Programs written in C or FORTRAN or some other languages must be compiled to generate executable programs. Most programs we will work with are in C, and the default compiler for C in Linux is `gcc`. I'll demonstrate a typical workflow for writing, compiling, and running a C program here.

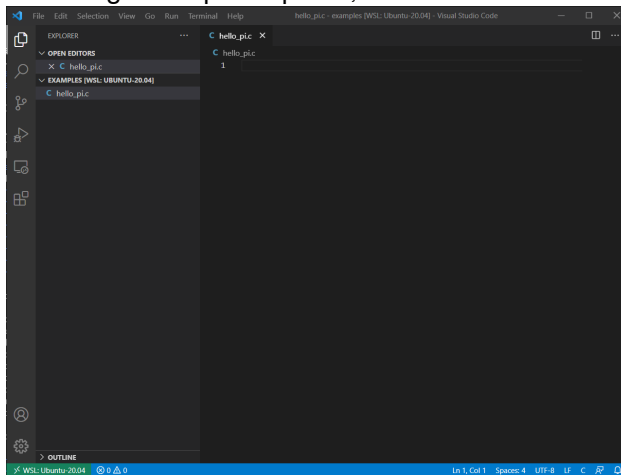
First, `cd` to your `cheT580` subdirectory, and create and `cd` into a subdirectory called `examples`, then launch VSCode:

```
$ cd
$ cd cheT580
$ mkdir examples
$ cd examples
$ code .
```

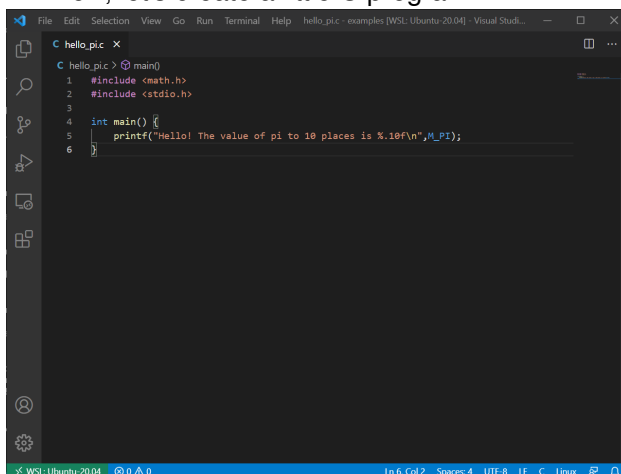
You should see the code window appear something like this:



Using the explorer panel, I can click on the new file icon and create a new file called `hello_pi.c`:



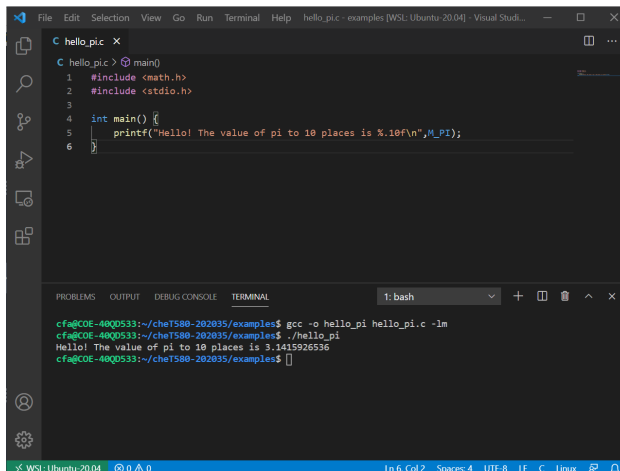
Now, let's create a little C program:



Notice that two libraries are included: `stdio` and `math`. I need `stdio` to use the `printf()` function, and I need `math` to access the constant `M_PI`.

Saving that with `Ctrl-S`, I can now launch a new Terminal inside VSCode (or just go back to the WSL terminal), and compile and run:





```

C hello_pi.c X
C hello_pi.c (main)
1 #include <math.h>
2 #include <stdio.h>
3
4 int main() {
5     printf("Hello! The value of pi to 10 places is %.10f\n", M_PI);
6 }

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: bash
cfa@CFE-4009533:~/che1580-202035/examples$ gcc -o hello_pi hello_pi.c -lm
cfa@CFE-4009533:~/che1580-202035/examples$ ./hello_pi
Hello! The value of pi to 10 places is 3.1415926536
cfa@CFE-4009533:~/che1580-202035/examples$
  
```

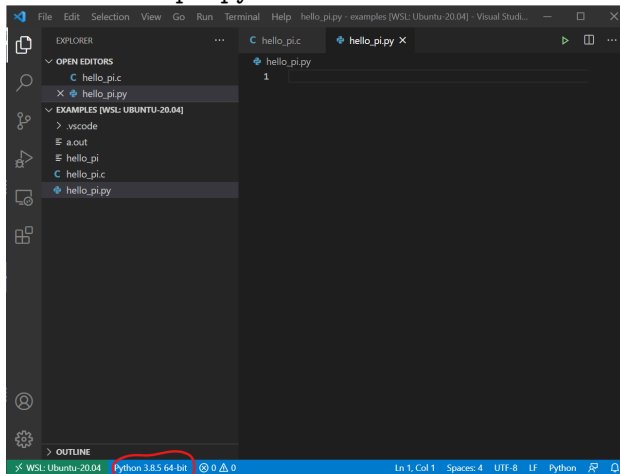
The compile command is `gcc` and its main argument is the name of the C program `hello_pi.c`. The `-o` switch is used to identify the name of the output of the compilation; here, that is the name of the executable, and we're choosing to call that `hello_pi`. If we do not include a `-o` switch, `gcc` calls the output `a.out`. The `-lm` switch instructs `gcc` to include the precompiled standard math library; try omitting this switch and see what happens.

The executable `hello_pi` lives in the same directory as the source code `hello_pi.c`. We can run it by just typing the name of the executable, prepended with `./`. This instructs the shell NOT to go looking in any standard system directories for the name of the command (which it normally does), but instead to run the command whose executable is found in the current directory. The current directory is always signified by `./`. Running this program provides the anticipated result.

### 3.2.2 Programs that are Interpreted: Python

Unlike C, Python is an *interpreted* programming language. This just means that you don't have to compile it yourself before running it. Instead, you feed the program to the Python interpreter and *it* compiles and runs it for you, and then exits.

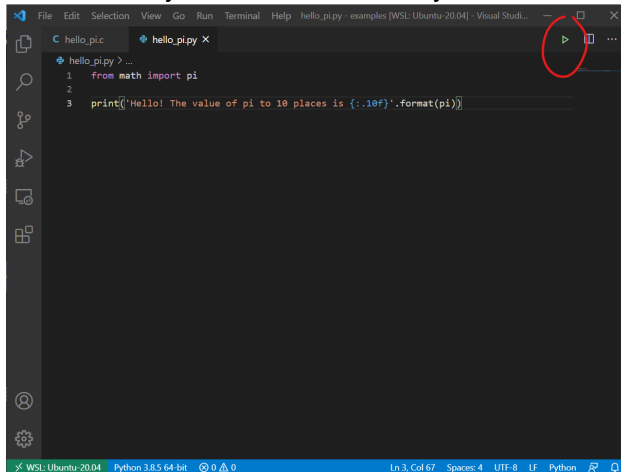
Keeping that instance of VSCode running inside the `examples` subdirectory, let's create a new file called `hello_pi.py`:



In red, I've circled the little message indicating which Python interpreter VSCode will use if you choose to run this program *using* VSCode. You may instead see a message here indicating that you have to select a Python interpreter. (Windows users: If you did not install Python inside your WSL/Ubuntu already, as instructed in Assignment 1, you can do so now using `apt` at the command-

line.)

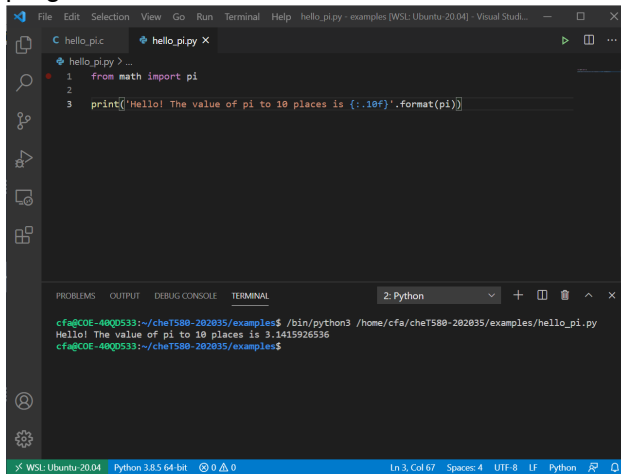
Here is Python that does exactly the same thing as `hello_pi.c`:



```

hello_pi.py > ...
1  from math import pi
2
3  print('Hello! The value of pi to 10 places is {:.10f}'.format(pi))
  
```

Now, notice that little green “play” button in the upper-right? I can just click that to run the Python program:



```

cfa@COE-48Q9533:~/cheT580-202035/examples$ ./bin/python3 /home/cfa/cheT580-202035/examples/hello_pi.py
Hello! The value of pi to 10 places is 3.1415926535
cfa@COE-48Q9533:~/cheT580-202035/examples$
  
```

And we still get the anticipated result. We need not run the Python program inside VSCode; we are free to run it at the command-line, but notice the command that VSCode issued to run the program: the program that VSCode runs is *actually* the *interpreter* `/bin/python3`, and the *argument* of that command is the full pathname of the Python script. You could alternatively issue that command at the bash prompt and the same result would happen.

## 4 Monte Carlo Simulation

The first simulation technique we will study is the Monte Carlo method. The primary “flavor” of MC most appropriate for this introductory level course is the original Metropolis method, explained in Sec. 4.1.

We will also use MC to explain basic technical aspects of molecular simulation code in Sec. 4.3. These aspects are not all restricted to MC, but following the text, we will introduce and discuss these technical aspects here. These include (1) periodic boundary conditions, (2) energy evaluation, (3) representation of data, among others.

Finally, four case studies will be presented and investigated. The first is the Ising magnet (Sec. 4.2). The second considers hard-disks confined in a circle (Sec. 4.4), and the third is hard-disk-dumbbells (Sec. 4.5). The fourth is MC to explore the equation of state of a model liquid known as the Lennard-Jones fluid (Sec. 4.6).

### 4.1 The Metropolis Monte Carlo Method

Monte Carlo is a type of numerical integration. Consider the following integral:

$$I = \int_a^b f(x) dx \quad (62)$$

Now imagine that we have a second function,  $\rho(x)$ , which is positive in the interval  $[a, b]$ . We can also express  $I$  as

$$I = \int_a^b \frac{f(x)}{\rho(x)} \rho(x) dx \quad (63)$$

If we think of  $\rho(x)$  as a *probability density*, then what we have just expressed is the expectation value of the quantity  $f/\rho$  on  $\rho$  in the interval  $[a, b]$ :

$$I = \left\langle \frac{f(x)}{\rho(x)} \right\rangle_\rho \quad (64)$$

This implies that we can approximate  $I$  by picking  $M$  values  $\{x_i\}$  randomly out of the probability distribution  $\rho(x)$  and computing the following sum:

$$I \approx \frac{1}{M} \sum_{i=1}^M \frac{f(x_i)}{\rho(x_i)} \quad (65)$$

Note that this approximates the mean of  $f/\rho$  as long as we pick a large enough number of random numbers ( $M$  is large enough) such that we “densely” cover the interval  $[a, b]$ . If  $\rho(x)$  is *uniform* on  $[a, b]$ ,

$$\rho(x) = \frac{1}{b-a}, \quad a < x < b \quad (66)$$

and therefore,

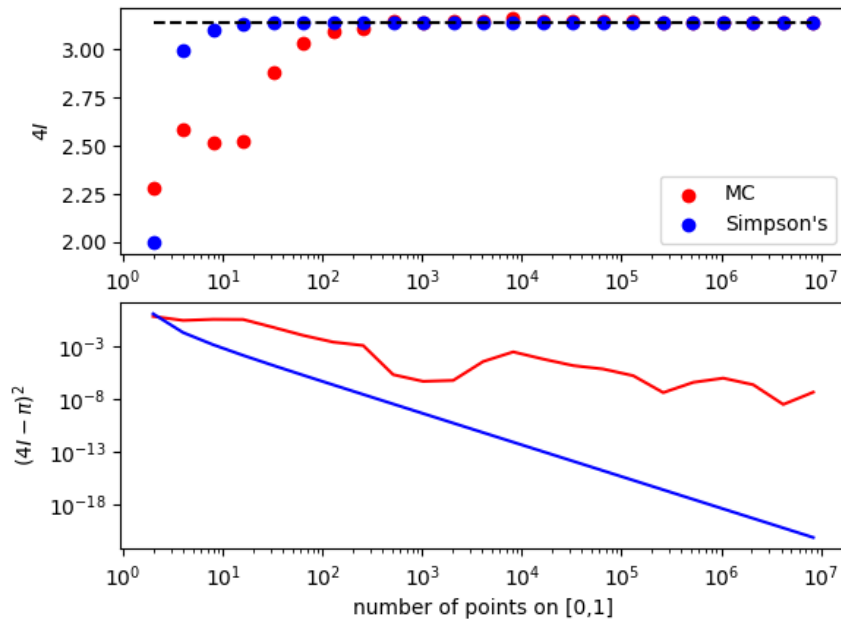
$$I \approx \frac{b-a}{M} \sum_{i=1}^M f(x_i) \quad (67)$$

The next question is, how good an approximation is this, compared with other one-dimensional numerical integration techniques, such as Simpson’s rule and quadrature? A better phrasing of this question is, how expensive is this technique for a given level of accuracy, compared to traditional tech-

niques? Consider this means to compute  $\pi$ :

$$I = \int_0^1 (1 - x^2)^{1/2} dx = \frac{\pi}{4} \quad (68)$$

Allen and Tildesley [2] mention that, in order use Eq. 67 to compute  $\pi$  to an accuracy of one part in  $10^6$  requires  $M = 10^7$  random values of  $x_i$ , whereas Simpson's rule requires *three orders of magnitude* fewer points to discretize the interval to obtain an accuracy of one part in  $10^7$  (Fig. 2). So the answer is, integral estimation using Monte Carlo estimation with uniform random variates is expensive.



**Figure 2:** Integration of Eq. 68 by Monte Carlo and Simpson's rule. Upper:  $4I$  vs number of random points for MC, for which points are uniform random variates on  $[0,1]$ , and for Simpson's rule, for which points are equally spaced along  $[0,1]$ . Lower: Squared error in the estimate of  $\pi$  for each method.

But, the situation changes radically when the dimensionality of the integral is large, as is the case for an ensemble average. For example, for a system of 100 particles comprising 300 coordinates, the configurational average  $\langle G \rangle$  (Eq. 59) could be discretized using Simpson's rule. If we did that, requesting only a modest 10 discrete points per axis in configurational space, we would need to evaluate the integrand  $Ge^{-\beta U} 10^{300}$  times. This is an almost unimaginably large number. Using a direct numerical technique to compute statistical mechanical averages is simply out of the question.

We therefore return to the idea of evaluating the integrand at a discrete set of points selected randomly from a distribution. Here we call upon the idea of *importance sampling*. Let us try to use whatever we know ahead of time about the integrand in picking our random distribution,  $\rho$ , such that we *minimize* the number of points (i.e., the expense) necessary to give an estimate of  $\langle G \rangle$  to a given level of accuracy.

Now, clearly the states that contribute the most to the integrals we wish to evaluate by configurational averaging are those states with large Boltzmann factors; that is, those states for which  $\rho_{NVT}$  is large. It stands to reason that if we randomly select points from  $\rho_{NVT}$ , we will do a pretty good job approximating

the integral. So what we end up computing is the “average of  $G\rho_{NVT}$  over  $\rho_{NVT}$ ”:

$$\langle G\rho_{NVT}/\rho_{NVT} \rangle \approx \langle G \rangle, \quad (69)$$

which should give an excellent approximation for  $\langle G \rangle$ . The idea of using  $\rho_{NVT}$  as the sampling distribution is due to Metropolis et al. [3]. This makes the real work in computing  $\langle G \rangle$  generating states that randomly sample  $\rho_{NVT}$ .

Metropolis et al. [3] showed that an efficient way to do this involves generating a *Markov chain of states* which is constructed such that its *limiting distribution* is  $\rho_{NVT}$ . A Markov chain is just a sequence of trials, where (i) each trial outcome is a member of a finite set and (ii) every trial outcome depends only on the outcome that immediately precedes it. By “limiting distribution,” we mean that the trial acceptance probabilities are tuned such that the probability of observing the Markov chain atop a particular state is defined by some equilibrium probability distribution,  $\rho$ . For the following discussion, it will be convenient to denote a particular state  $n$  using  $\Gamma_n$ , instead of  $\nu$ .

A trial is some perturbation (usually small) of the coordinates specifying a state. For example, in an Ising system, this might mean flipping a randomly selected spin. In a system of particles in continuous space, it might mean displacing a randomly selected particle by a small amount  $\delta r$  in a randomly chosen direction  $(\theta, \phi)$ . There can be a large variety of such “trial moves” for any particular system.

The probability that a trial move results in a successful transition from state  $n$  to  $m$  is denoted  $\pi_{nm}$  and  $\pi$  is called the “transition matrix.” It must be specified ahead of time to execute a traditional Markov chain. Since the probability that a trial results in a successful transition to any state, the rows of  $\pi$  add to unity:

$$\sum_i \pi_{ni} = 1 \quad (70)$$

With this specification, we term  $\pi$  a “stochastic” matrix. Furthermore, for an equilibrium ensemble of states in state space, we require that transitions from state to state *do not alter state weights* as determined by the limiting distribution. So the weight of state  $n$ :

$$\rho_n \equiv \rho_{NVT}(\Gamma_n) \quad (71)$$

must be the result of transitions from all other states to state  $n$ :

$$\rho_n = \sum_m \rho_m \pi_{mn}. \quad (72)$$

For all states  $n$ , we can write Eq. 72 as a post-op matrix equation:

$$\rho\pi = \rho \quad (73)$$

where  $\rho$  is the row vector of all state weights. Eq. 73 constrains our choice of  $\pi$ . This means there is still more than one way to specify  $\rho$ . Metropolis et al. [3] suggested:

$$\rho_m \pi_{mn} = \rho_n \pi_{nm} \quad (74)$$

That is, the probability of transitioning from state  $m$  to  $n$  is exactly equal to the probability of transitioning from state  $n$  to  $m$ . This is called the “detailed balance” condition, and it guarantees that the state weights remain static. Observe:

$$\sum_m \rho_m \pi_{mn} = \sum_m (\rho_n \pi_{nm}) = \rho_n \left( \sum_m \pi_{nm} \right) = \rho_n \quad (75)$$

Detailed balance is, however, overly restrictive; is the only the conceptually simplest way to guarantee that a limiting distribution is obtained. This fact is of little importance in this course, but you may encounter other balance-enforcing conditions in the literature.

Metropolis et al. [3] chose to construct  $\pi$  as

$$\pi_{nm} = \alpha_{nm} \text{acc}(n \rightarrow m) \quad (76)$$

where  $\alpha$  is the probability that a trial move is attempted, and  $\text{acc}$  is the probability that a move is accepted. If the probability of *proposing* a move from  $n$  to  $m$  is equal to that of *proposing* a move from  $m$  to  $n$ , then  $\alpha_{nm} = \alpha_{mn}$ , and the detailed balance condition is written:

$$\rho_n \text{acc}(n \rightarrow m) = \rho_m \text{acc}(m \rightarrow n) \quad (77)$$

from which follows

$$\frac{\text{acc}(n \rightarrow m)}{\text{acc}(m \rightarrow n)} = \frac{\rho_m}{\rho_n} = \frac{e^{-\beta U(\Gamma_m)}}{e^{-\beta U(\Gamma_n)}} \quad (78)$$

giving

$$\frac{\text{acc}(n \rightarrow m)}{\text{acc}(m \rightarrow n)} = \exp\{-\beta[U(\Gamma_m) - U(\Gamma_n)]\} \equiv \exp(-\beta \Delta U_{nm}) \quad (79)$$

where we have defined the change in potential energy as

$$\Delta U_{nm} = U(\Gamma_m) - U(\Gamma_n) \quad (80)$$

There are many choices for  $\text{acc}(n \rightarrow m)$  that satisfy Eq. 79. The original choice of Metropolis is used most frequently:

$$\text{acc}(n \rightarrow m) = \begin{cases} \exp(-\beta \Delta U_{nm}) & \Delta U_{nm} > 0 \\ 1 & \Delta U_{nm} < 0 \end{cases} \quad (81)$$

So, suppose we have some initial configuration  $n$  with potential energy  $U_n$ . We make a trial move, *temporarily* generating a new configuration  $m$ . Now we calculate a new energy,  $U_m$ . If this energy is lower than the original, ( $U_m < U_n$ ) we unconditionally accept the move, and configuration  $m$  becomes the *current* configuration. If it is greater than the original, ( $U_m > U_n$ ) we accept it with a probability consistent with the fact that the states both belong to a canonical ensemble. How does one in practice decide whether to accept the move? One first picks a uniform random variate  $x$  on the interval  $[0, 1]$ . If  $x \leq \text{acc}(n \rightarrow m)$ , the move is accepted.

The next section is devoted to an implementation of the Metropolis Monte Carlo method for a 2D Ising magnet.

## 4.2 Case Study 1: The 2D Ising Magnet

### 4.2.1 Introduction

In the introductory lecture, I introduced state-counting using a simple, one-dimensional Ising system. To be precise, this was a particular case known as an Ising magnet of *noninteracting* spins, because each spin made its own independent contribution to the Hamiltonian. The state of the magnet is specified by specifying each spin variable as either +1 or -1:

$$\Gamma = \{s_1 = \pm 1, s_2 = \pm 1, s_3 = \pm 1, \dots, s_N = \pm 1\} \quad (82)$$

Now, we will consider a more interesting Ising system; namely, that of *interacting* spins on a 2d square lattice. What do we mean by *interacting*? We mean that the Hamiltonian depends on pairs of

spins:

$$\mathcal{H} = -J \sum_{\langle ij \rangle} s_i s_j \quad (83)$$

where the notation  $\langle ij \rangle$  denotes that we are summing over unique pairs of *nearest neighbors*, and, as before a spin  $s_i$  is either +1 or -1.  $J$  is the “coupling constant” and represents our default unit of energy. How many unique pairs of nearest neighbors are there on a lattice of  $N$  spins (assuming periodic boundary conditions)? To answer this question, we must know the *coordination*,  $z$ , of the lattice; that is, how many nearest neighbors does one lattice position have? For a square lattice,  $z = 4$ . Each spin therefore contributes *two* unique spin pairs to the system, so there are  $Nz/2$  unique nearest neighbor pairs.

Think about this Hamiltonian. It says that energy is minimal when all  $N$  spins have the same alignment, either all up or all down. Imagine a microscopic observable called the *magnetization*, or average spin orientation,  $M$ :

$$M \equiv \frac{1}{N} \sum_i s_i \quad (84)$$

Then,  $M = +1$  and  $-1$  are two energetically equivalent microstates. We can then expect that if there is any *thermal* energy in the system which randomly flips spins, the amount of this thermal energy (that is, the temperature) will somehow control the observable magnetization. We “observe” magnetization using an ensemble average:

$$\langle M \rangle = \frac{\sum_{s_1=-1}^{+1} \cdots \sum_{s_N=-1}^{+1} \left[ \sum_i^N s_i \right] \exp[-\beta \mathcal{H}(s_1, \dots, s_N)]}{\sum_{s_1=-1}^{+1} \cdots \sum_{s_N=-1}^{+1} \exp[-\beta \mathcal{H}(s_1, \dots, s_N)]} \quad (85)$$

Our physical intuition tells us that as  $T \rightarrow 0$ ,  $|\langle M \rangle| \rightarrow 1$ , and as  $T \rightarrow \infty$ ,  $\langle M \rangle \rightarrow 0$ . The fascinating thing about an Ising magnet is that there is a finite temperature called the *critical temperature*,  $T_c$ . If we start out with a “hot” system, and cool it to just below  $T_c$ , the *absolute value* of the magnetization spontaneously jumps from 0 to some finite value. In other words, the system undergoes a *phase transition* from a disordered phase to a partially ordered phase. A Metropolis Monte Carlo simulation can allow us to probe the behavior of an Ising system and learn how the system behaves near criticality. The rest of this case study will be devoted to showing you the inner workings of a C program which simulates the Ising magnet using Metropolis MC, as a first implementation of this technique. In the suggested exercises appearing at the end of this case study, you will modify this code slightly to compute averages values of certain observables.

But first, I recommend you check out [this Java implementation](#) of a Monte Carlo simulation of a 2-dimensional Ising magnet (one of many on the web; google “ising simulation” and you’ll get a nice sample.) This is a fun little Java applet that lets you play with an Ising magnet. You can change the temperature of the simulation: making it cold will “freeze” the system, and making it hot “melts” it. Near the *critical temperature*,  $T_c$ , relatively large regions of mostly-up spins compete with regions of mostly down spin. In one of the exercises, we’ll learn how to measure an observable called the *correlation length*, which characterizes the size of these domains and is a useful signature of criticality.

#### 4.2.2 A C Code for the 2D Ising Magnet

In this section, we will dissect piece-by-piece a small C program that implements an NVT Metropolis Monte Carlo simulation of a 2D Ising magnet.

If you have not already done so, clone the Abrams-Teaching/instructional-codes repository from Github. This repository will be updated throughout the term with source code in the `originals` subdirectory. You should create a subdirectory called `my_work` inside this repository and do all editing, compiling, and running in there. This directory is specifically excluded from `git` revision control by its inclusion in the `.gitignore` file. This way, when I put new codes in the repository, you only have to `git pull` to download them.

```
$ cd
$ cd cheT580
$ git clone git@github.com:Abrams-Teaching/instructional-codes.git
$ cd instructional-codes
$ mkdir my_work
$ cd my_work
$ cp ../originals/ising_mc.c .
$ code .
```

From a terminal command-line inside VSCode, or outside, you can compile `ising_mc.c` via

```
$ gcc -O3 -o ising ising_mc.c -lm -lgsl
```

and you can then run it at the command line as `./ising`. It has a lot of options for controlling the size of the magnet (the number of spins), the temperature, and other parameters, which I'll go over now.

`ising_mc.c` conducts a canonical Metropolis Monte Carlo simulation of an Ising magnet of size  $L \times L$  at temperature  $T$  (both specified by the user at run time on the command line), and it computes both the average energy per spin  $\langle \epsilon \rangle$  and the average spin value,  $\langle s_1 \rangle$ .

*Periodic boundaries* are employed in calculating the nearest-neighbor interactions. Consider an  $L \times L$  magnet; each row  $i$  indexed from 0 to  $N - 1$  has  $L$  columns also indexed from 0 to  $N - 1$ . If the cell at (5,5) queries its neighbors, they are at (4,5), (6,5), (5,4), and (5,6). However, the cell at (0,5) would have a southern neighbor at  $(N - 1, 5)$  instead of  $(-1, 5)$ , since there is no row indexed -1! Fig. 3 demonstrates this.

An abbreviated listing of the code follows. Some comments in the full, downloadable code have been omitted for space, and I have instead explained each code fragment in accompanying text.

## Headers

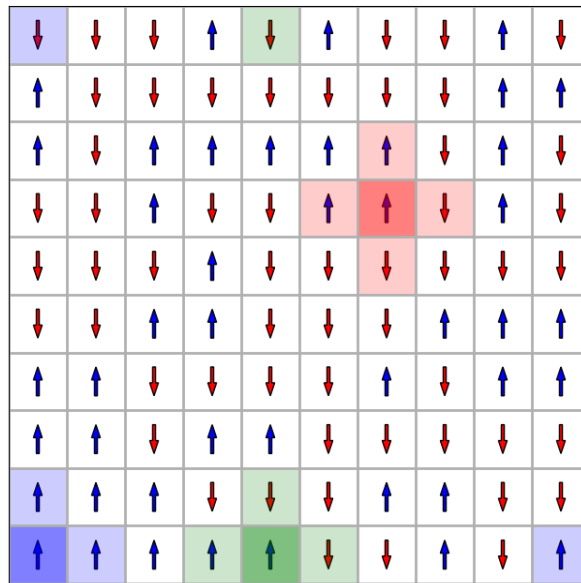
These are some standard headers we include in most C programs, along with the GNU scientific library.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <gsl/gsl_rng.h>
```

## The Change in Energy upon a Proposed Spin Flip

`dE()` is function that accepts as arguments the 2D array of spins, `M`; the side-length, `L`, and a position  $(i, j)$ , and returns the *change* in energy (in units of  $J$ ) upon flipping spin  $(i, j)$ , without *actually* flipping it. All variables are of integer type, `int`. The syntax `** M` means that `M` is a pointer to a pointer to an `int`, signifying that `M` is a 2D array. To access element in row `i` and column `j` in `M`, the syntax is `M[i][j]`. In C, all array indices start at 0, so `M[0][0]` refers to the “upper-left” corner of the magnet (assuming row numbers increase going down the magnet). The inline conditional syntax `i?(i-1):(L-1)` expands as, “If `i` is non-zero, return `i-1`; otherwise, return `L-1`.” This implements periodic boundaries when





**Figure 3:** Schematic Ising magnet highlighting periodic boundary conditions. The neighbors of the red spin can be found by adding or subtracting one from the cell's row and column index. For the green and blue cells, however, at least one of the neighbors appears on the other side of the magnet.

looking to the *west* of spin  $M[i][j]$ . The syntax  $(i+1)\%L$  returns computes  $(i + 1) \bmod L$ , which also implements periodic boundaries when looking to the *east* of spin  $M[i][j]$ .

```
int dE ( int ** M, int L, int i, int j ) {
    return -2*(M[i][j])*(M[i?(i-1):(L-1)][j]+M[(i+1)%L][j]+
        M[i][j?(j-1):(L-1)]+M[i][(j+1)%L]);
}
```

### Initialize all Spin Values

The function `init()` visits every spin and assigns it either +1 or -1 randomly.

```
void init ( int ** M, int L, gsl_rng * r ) {
    int i,j;
    for (i=0;i<L;i++) {
        for (j=0;j<L;j++) {
            M[i][j]=2*(int)gsl_rng_uniform_int(r,2)-1;
        }
    }
}
```

### Main Program: Variable Declarations

In C functions, including `main()`, all variables must be declared by type before they are used. Often, it makes sense to initialize them to some default values upon declaration. Here, we make the default

magnet  $20 \times 20$  in size, set the temperature to 1 (in dimensions of  $J/k_B$ ), and provide some defaults for the number of cycles ( $10^6$ ) and sampling interval ( $10^3$ ) in cycles (a cycle is one round of  $N$  attempted flips). Variables for holding a couple of observables are initialized. Finally, the pseudorandom number generator from the GSL is declared.

```
int main (int argc, char * argv[]) {
    /* System parameters */
    int ** M;          /* The 2D array of spins */
    int L = 20;        /* The sidelength of the array */
    int N;             /* The total number of spins = L*L */
    double T = 1.0;   /* Dimensionless temperature = (T*k)/J */

    /* Run parameters */
    int nCycles = 1000000; /* number of MC cycles to run;
                           one cycle is N consecutive attempted
                           spin flips */
    int fSamp = 1000;    /* Interval between taking samples */

    /* Computational variables */
    int nSamp;          /* Number of samples taken */
    int de;            /* energy change due to flipping a spin */
    double b;          /* Boltzman factor */
    double x;          /* random number */
    int i,j,a,c;       /* loop counters */

    /* Observables */
    double s=0.0, ssum=0.0; /* average magnetization */
    double e=0.0, esum=0.0; /* average energy per spin */

    /* Pseudorandom Number Generator */
    gsl_rng * r = gsl_rng_alloc(gsl_rng_mt19937);
    unsigned long int Seed = 23410981;
```

### Main Program: Argument Parsing

Here, we parse the command line arguments. The user running the program can specify the magnet side-length  $L$ , the temperature (in units of  $J/k_B$ ), the number of cycles, the sampling interval, and the seed value for the pseudorandom number generator. The array `argv[]` and its count of elements `argc` appear as parameters in the definition of `main`, as is customary in C. The built-in function `strcmp()` returns 0 if the two arguments match, so the logical expression `!strcmp(a,b)` evaluates to TRUE if strings `a` and `b` match. The built-in functions `atoi()` and `atof()` convert their string arguments to integers and floating-points, respectively. The notation `argv[++i]` means that *first* the current value of `i` is incremented by 1 and *then* it is used to access the value in `argv[]`. So, for example, if the string `"-L"` is detected at the `i`th position in the array of command-line arguments, the code immediately jumps to the *next* position in the array and tries to interpret that argument as an integer to assign to `L`.

```
for (i=1;i<argc;i++) {
    if (!strcmp(argv[i],"-L")) L=atoi(argv[++i]);
    else if (!strcmp(argv[i],"-T")) T=atof(argv[++i]);
    else if (!strcmp(argv[i],"-nc")) nCycles = atoi(argv[++i]);
```

```
else if (!strcmp(argv[i],"-fs")) fSamp = atoi(argv[++i]);
else if (!strcmp(argv[i],"-s")) {
    Seed = (unsigned long)atoi(argv[++i]);
}
}
```

### Main Program: Initial Outputs

Next, we echo the command entered back to the terminal, and then output any parsed variables values.

```
printf("# command: ");
for (i=0;i<argc;i++) printf("%s ",argv[i]);
printf("\n");
printf("# ISING simulation, NVT Metropolis Monte Carlo\n");
printf("# L = %i, T = %.3lf, nCycles %i, fSamp %i, Seed %lu\n",
    L,T,nCycles,fSamp,Seed);
```

### Main Program: Initialization

Next, we initialize the pseudorandom number generator  $r$  by setting the seed value using `Seed`. Then the number of spins  $N$  is computed assuming a square magnet. We then allocate memory needed to hold the magnet; this is a standard method of allocating a 2D array of integers. Next we call our `init()` function, and finally we convert  $T$  to  $1/T$  since multiplication is more efficient than division.

```
/* Seed the random number generator */
gsl_rng_set(r,Seed);

/* Compute the number of spins */
N=L*L;

/* Allocate memory for the system */
M=(int**)malloc(L*sizeof(int*));
for (i=0;i<L;i++) M[i]=(int*)malloc(L*sizeof(int));

/* Generate an initial state */
init(M,L,r);

/* For computational efficiency, convert T to reciprocal T */
T=1.0/T;
```

### Main Program: Monte Carlo Loop

And now the loop. Note the outer loop counts cycles, and the inner loops counts flip attempts in one cycle. The variables  $i$  and  $j$  are randomly assigned between 0 and  $L-1$ , identifying a random spin. This random spin is tagged and sent to our `dE()` function to calculate the change in energy we *would* expect if that spin were flipped ( $+1 \rightarrow -1$  or  $-1 \rightarrow +1$ ). We then calculate the Boltzmann factor in  $b$ , and then use the Metropolis criterion to decide whether to accept or reject this spin flip: choosing a random variable on  $[0,1]$  and accepting the move if the Boltzmann factor is *greater* than this number. If it is accepted, we actually perform the flip by multiplying the spin value by  $-1$ . Finally, if the current cycle is one in which we collect a sample, we do so by calling the `samp()` function. The logical expression `!(c%fSamp)`

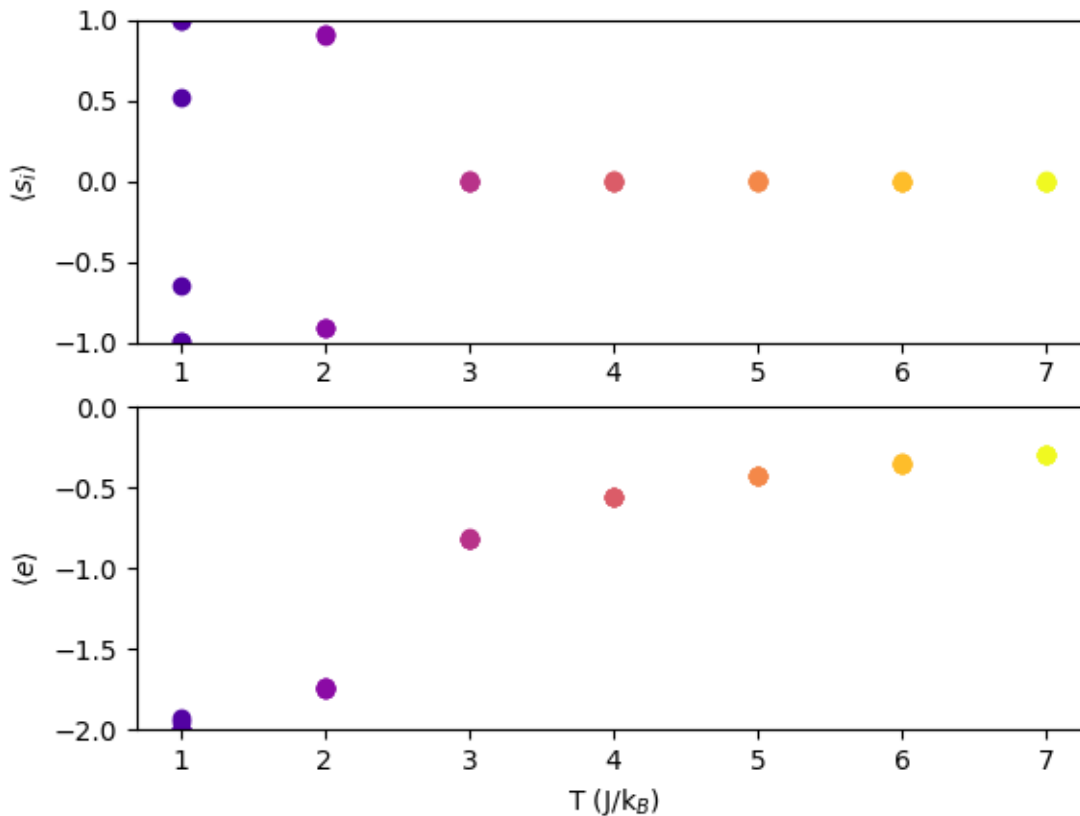
evaluates to TRUE if the cycle counter  $c$  is divisible by  $fSamp$ . In the call to `samp()`, the arguments  $s$  and  $e$  are *passed by reference*, signified by the `&` qualifier. This is necessary because *inside* `samp()` we modify both variables. Each of  $s$  and  $e$  are added to their respective tallies,  $ssum$  and  $esum$ , and the number of samples taken  $nSamp$  is incremented by 1.

```
s = 0.0;
e = 0.0;
nSamp = 0;
for (c=0;c<nCycles;c++) {
    /* Make N flip attempts */
    for (a=0;a<N;a++) {
        /* randomly select a spin */
        i=(int)gsl_rng_uniform_int(r,L);
        j=(int)gsl_rng_uniform_int(r,L);
        /* get the "new" energy as the incremental change due
           to flipping spin (i,j) */
        de = dE(M,L,i,j);
        /* compute the Boltzmann factor; recall T is now
           reciprocal temperature */
        b = exp(de*T);
        /* pick a random number between 0 and 1 */
        x = gsl_rng_uniform(r);
        /* accept or reject this flip */
        if (x<b) { /* accept */
            /* flip it */
            M[i][j]*=-1;
        }
    }
    /* Sample and accumulate averages */
    if (!(c%fSamp)) {
        samp(M,L,&s,&e);
        fprintf(stdout,"%i %.5le %.5le\n",c,s,e);
        fflush(stdout);
        ssum+=s;
        esum+=e;
        nSamp++;
    }
}
```

### Main Program: Final Outputs and Program Termination

After the outer loop finishes, we can finish up by reporting the averages  $\langle s_1 \rangle$  from the tallies of  $s$  and  $e$ .

```
fprintf(stdout,"# The average magnetization is %.5lf\n",
        ssum/nSamp);
fprintf(stdout,"# The average energy per spin is %.5lf\n",
        esum/nSamp);
fprintf(stdout,"# Program ends.\n");
}
```



**Figure 4:** (Upper) Average magnetization  $\langle s_1 \rangle$  and (lower) average energy per spin  $\langle e \rangle$  vs. temperature for a  $40 \times 40$  Ising lattice, computed using six independent 50,000-cycle Metropolis MC simulations with sampling intervals of 100 cycles.

### 4.2.3 Example: Average Energy and Magnetization vs. Temperature

Let's run the code for an  $L=40$  lattice for the following values of temperature: 5.0, 4.0, 3.0, 2.0, 1.0. At each temperature, we'll run six separate simulations with a unique random number generator seed. Fig. 4 shows the average magnetization  $\langle s_1 \rangle$  and the average energy per spin  $\langle e \rangle$  vs. temperature, with each simulation contributing a unique point.

What is happening here? Clearly, as the temperature decreases, the average energy per spin approaches -2; this makes sense because the spins will tend to align with their neighbors. However, when we look at the average magnetization, we see that  $\langle s_1 \rangle$  is zero at high temperatures but then can seemingly approach *either* +1 or -1 as the temperature goes down. This is because of *symmetry breaking*: although either all-up or all-down is equally likely, once it falls toward one it won't ever climb back out and go to the other. So we see *some* magnets go to all -1 and others to all +1.

Symmetry-breaking is an important phenomenon in molecular simulations. The impact it has is to prevent exploration of state space because of barriers that are only extremely rarely overcome when resolving state-to-state transitions microscopically. In the low- $T$  Ising magnet, the all-up and all-down states are equally likely, but once one is committed to, the other will never practically be explored. Why is this significant? Many systems have state spaces like this, where there are two or more high-probability regions separated by large regions of low probability. Sampling based on local moves in

state space can almost never overcome such barriers, meaning such simulations are likely never truly ergodic. None of the MC simulations below about  $T = 2.2$  for an Ising simulation are ergodic!

#### 4.2.4 Suggested Exercises

1. Modify the code so that when samples are taken in accumulating statistics for  $\langle s_1 \rangle$  and  $\langle \epsilon \rangle$ , the current sample values are output to the terminal. You'll want to find the right place to add the following line: `fprintf(stdout, "%i %.51f %.51f\n", c, s, e);`
2. The current version of the code initializes the Ising lattice with random spins. What temperature does this correspond to? Modify the code so that the initial lattice has two well-defined domains, all spin-up for  $i < L/2$  and all spin-down for  $i > L/2$ . Re-run at the various temperatures. Do you see any differences?
3. (Advanced) Modify the code *ising.c* to compute the quantity  $\langle s_i s_j \rangle - \langle s_i \rangle \langle s_j \rangle$  as a function of various distances between spins  $i$  and  $j$ .

### 4.3 Elements of a Continuous-Space MC program

The Ising system serves us well as a simple introduction to the technique of Monte Carlo simulation. Now, we move on to the more advanced case of *continuous-space* Monte Carlo; that is, Monte Carlo on a system composed of particles whose position and velocity vector components are real numbers. We will first consider the simple case of a “hard-sphere” liquid, and then the more realistic Lennard-Jones liquid. These distinctions have to do with the *potential energy function* used to compute the potential energy  $U$  of a configuration  $\mathbf{r}^N$ .

Regardless of the potential used, all continuous-space MC programs have common elements:

1. data representation and input/output;
2. energy calculation;
3. trial move generation.

#### 4.3.1 Data Representation and Input/Output

The information that must be stored and handled in an MC program are the particle positions. In 3D space, each particle has three components of its position. The simplest way to represent this information in a program is by using *parallel arrays*; that is, three arrays, dimensioned from 1 to  $N$ , one for each dimension. In C, we might declare these arrays for 1,000 particles as

```
int N = 1000;
double rx[N], ry[N], rz[N];
```

This data can be stored in standard ASCII text files, or in unformatted binary files. In fact, a large part of most molecular simulation is producing and storing configurations which can be processed “offline” (away from the simulation program) to perform analyses. A simple and widely used ASCII configuration file format is called “XYZ”. A code fragment to write a configuration of  $N$  hydrogen atoms in XYZ format appears below:

```
FILE * fp;
...
fp = fopen("my_config.xyz", "w");
fprintf(fp, "%i\n", N);
fprintf(fp, "This is a comment or just a blank line\n");
for (i=0; i<N; i++)
    fprintf(fp, "%s %.8le %.8le %.8le\n", "H", rx[i], ry[i], rz[i]);
```

The main feature of XYZ files is that one can specify the element type of each atom; for simple visualization purposes of our toy systems, this can be anything, so we’ll just use H for now.

It’s quite easy to read ASCII data in that was written in XYZ format, e.g., using the fragment above, ignoring the element types (for now):

```
FILE * fp;
int N;
double * rx, * ry, * rz;
char dummy[4];
...
fp = fopen("my_config.xyz", "r");
fscanf(fp, "%i\n", &N);
/* allocate if not done already */
rx=(int*)malloc(N*sizeof(int));
ry=(int*)malloc(N*sizeof(int));
rz=(int*)malloc(N*sizeof(int));
```

```
fscanf(fp, "\n"); /* reads the comment line but throws it away */
for (i=0; i<N; i++)
  fscanf(fp, "%s %.81e %.81e %.81e\n", dummy, &rx[i], &ry[i], &rz[i]);
```

### 4.3.2 Analytical Potentials

The total system potential energy is most often computed by means of *analytical potentials*. Such a potential energy can be written as a continuous function of the positions of particle centers-of-mass. (Such potentials are often termed “empirical” when applied to atomic systems because they formally neglect quantum mechanics.) In most molecular simulations, the total system potential can be decomposed in the following way:

$$U(\mathbf{r}^N) = U_0 + \sum_i U_1(\mathbf{r}_i) + \sum_{i<j} U_2(\mathbf{r}_i, \mathbf{r}_j) + \sum_{i<j<k} U_3(\mathbf{r}_i, \mathbf{r}_j, \mathbf{r}_k) + \cdots + U_N(\mathbf{r}_i, \mathbf{r}_j, \dots, \mathbf{r}_N), \quad (86)$$

Here,  $U_0$  is some reference potential.  $U_1$  is a “one-body” potential,  $U_2$  is a “two-body” potential, etc. This generally defines a “many-body” potential, and is the heart of the “many-body” problem of statistical physics. Namely, the Boltzmann factor  $e^{-\beta U}$  correlates every position with every other position, and the configurational integral (Eq. 59) is not factorizable into many separate easily evaluated integrals.

Analytical potentials are most easily understood by considering model systems which are decomposable into spherically-symmetric, pairwise interactions. Consider then the following total system potential energy:

$$U = \sum_{i=1}^N \sum_{j=i+1}^N U_{ij}(r_{ij}) \quad (87)$$

The double-sum runs over all *unique* pairs of particles. The function  $U_{ij}(r_{ij})$  is called a *pair potential*, and it is a function of the scalar distance between particle  $i$  and  $j$ .

The simplest pair potential is the “hard-sphere”:

$$U_{HS}(r) = \begin{cases} \infty & r < \sigma \\ 0 & r > \sigma \end{cases} \quad (88)$$

Here,  $\sigma$  is an arbitrary interaction range that denotes the “size” of the spheres. When the distance between two spheres is less than  $\sigma$ , the energy is infinite; otherwise it is 0. This is the simplest way to enforce *excluded volume* among spherical particles in an MC simulation. We will see an implementation of a hard-sphere liquid in the next section.

The most celebrated pair potential is the Lennard-Jones potential (Fig. 5):

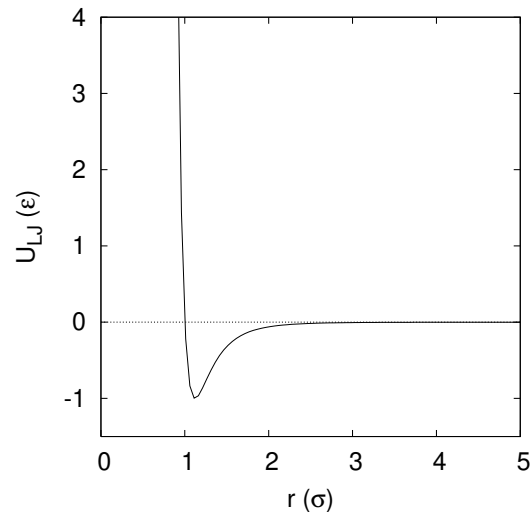
$$U_{LJ}(r) = 4\epsilon \left[ \left( \frac{\sigma}{r} \right)^{12} - \left( \frac{\sigma}{r} \right)^6 \right] \quad (89)$$

$\epsilon$  is the unit of energy, and is the well-depth of the pair potential (that is, it is the *minimum* value of the potential).  $\sigma$  is the unit of length, and is the separation distance when the potential is identically zero. Unlike the hard-sphere potential (Eq. 88), the Lennard-Jones potential is “smooth” (it has a continuous first derivative). This smoothness makes it useful in Molecular Dynamics simulations, as we will see later in the course.

We will encounter more complex potentials than Lennard-Jones, but it serves as a useful tool for introductory molecular simulation.

**Reduced Units.** Because the Lennard-Jones potential is so prevalent in molecular simulations, it is essential that we understand the *unit system* most often chosen for simulations using this potential.





**Figure 5:** The Lennard-Jones pair potential.

For computational simplicity, energy in a Lennard-Jones system is measured in units of  $\epsilon$  and length in  $\sigma$ . This means that everywhere in the code you would expect to see  $\epsilon$  or  $\sigma$ , you find a 1 (often implied).

Now, to compute the total potential  $U$  for a system of particles, the simplest algorithm is to loop over all *unique* pairs of particles. Here is a simple *pair search* C function (the so-called  $N^2$  algorithm because its complexity scales like  $N^2$ ) to compute the total energy of a system of Lennard-Jones particles, in reduced units:

```
double total_e ( double * rx, double * ry, double * rz, int n ) {
    int i,j;
    double dx, dy, dz, r2, r6, r12;
    double e = 0.0;
    for (i=0;i<(n-1);i++) {
        for (j=i+1;j<n;j++) {
            dx = rx[i]-rx[j];
            dy = ry[i]-ry[j];
            dz = rz[i]-rz[j];
            r2 = dx*dx + dy*dy + dz*dz;
            r6 = r2*r2*r2;
            r12 = r6*r6;
            e += 4*(1.0/r12 - 1.0/r6);
        }
    }
    return e;
}
```

Although it is strictly correct, the  $N^2$  pair search algorithm is inefficient if there is a *finite interaction range* in the pair potential. Typically in dense liquid simulations, a Lennard-Jones pair potential is truncated at  $r = 2.5 \sigma$ . There are some potentials that are cut off at even shorter distances. The point is that, when the maximum interaction distance is finite, each particle has a finite maximum number of interaction partners. (This assumes number density is bounded, which is a reasonable assumption.) More advanced techniques (which we discuss later) can be invoked to make the pair search much more

efficient in this case. The two most common are (1) the Verlet list, and (2) the link-cell list. For now, and to keep the presentation simple, we will stick to the inefficient, brute-force  $N^2$  algorithm.

### 4.3.3 Trial Moves

*Particle Displacement.* The most common trial move in continuous-space MC is a particle displacement. First, a small number  $\Delta R$ , representing a maximum displacement, is set. A trial move consists of

1. Randomly select a particle,  $i$ .
2. Displace x-position coordinate of particle  $i$  by a random amount,  $\delta x$ , which is given by

$$\delta x = \Delta R \xi_x \quad (90)$$

where  $\xi_x$  is a uniform random variate on the interval  $[-0.5:0.5]$ .

3. Repeat for the  $y$  and  $z$  coordinates, if applicable.

This move guarantees detailed balance, provided that the random particle selection is uniform; for any given move, selection of all possible particles is equally likely. This means that probability of suggesting a move that displaces a particle, going from a state  $n$  to a new state  $m$ , has the same probability of selecting the same particle while in state  $m$  and giving it a displacement that will return the configuration to state  $n$ . (Do you think such sequential moves ever actually happen?)

For a system of simple particles, random displacements are the only necessary trial moves; thus,  $\alpha_{nm}$  is always unity. For more complicated systems, there are zoos of trial moves all over the literature. We will consider some more complicated systems and trial moves later in the course; one that we consider next is *rigid rotation*.

The question at this point is, how does one choose an appropriate value for  $\Delta R$ ? If  $\Delta R$  is too small, the system will not explore phase space given a reasonable amount of computational effort. If it is too large, displacements will rarely result in new configurations which will be accepted in a Metropolis MC scheme. So it takes a bit of trial and error to find a good value for  $\Delta R$ , and the rule of thumb is to set  $\Delta R$  such that the average probability of accepting a new configuration *during a run* is about 30%. This is termed “tuning  $\Delta R$  to achieve a 30% acceptance ratio.” We will go through the exercise of determining such an appropriate value for  $\Delta R$  for a simple continuous-space system; namely, 2D hard disks confined to a circle.

*Rigid rotation.* A second common type of trial move is used in systems of more structured molecules than just simple, single-center spheres. Consider a diatomic with a rigid bond length  $r_0$ . Clearly, attempting to move one of the two members of the diatomic by a random displacement is likely to result in a new bond length which may be significantly different from  $r_0$ . So, for a system of diatomics, a reasonable *set* of trial moves would include

1. Small displacement of molecule center of mass; and
2. Small rotation around molecule center of mass.

With more than one kind of move, an attempt to generate a new state must be preceded by a random selection of the trial move. We can weight each kind of move and then use a random number to decide which move to attempt. For example, let's say that we choose that 80% of all trial moves be displacements, and the balance rotations (we will see later whether or not this is a good choice). Prior to an attempted move, we select a uniform random variate,  $\xi$ , on the interval  $[0, 1]$ . If  $\xi < 0.8$ , which it will be 80% of the time, we execute a displacement of a randomly chosen molecule; otherwise, we execute a rotation of a randomly chosen molecule.

## 4.4 Case Study 2: MC of Hard Disks

Change directory into your `instructional-codes` repository and issue a pull if you don't already see `hdisk.c` there. This code simulates disks confined to a circle. The Hamiltonian for this system may

be expressed as

$$\mathcal{H} = \sum_{i=1}^N H_1(\mathbf{r}_i) + \sum_{i=1}^N \sum_{j=i+1}^N H_2(\mathbf{r}_i, \mathbf{r}_j) \quad (91)$$

where

$$H_1(\mathbf{r}_i) = \begin{cases} 0 & r_i < R \\ \infty & r_i > R \end{cases} \quad (92)$$

and

$$H_2(\mathbf{r}_i, \mathbf{r}_j) = \begin{cases} 0 & \sqrt{(\mathbf{r}_i - \mathbf{r}_j)^2} > \sigma \\ \infty & \sqrt{(\mathbf{r}_i - \mathbf{r}_j)^2} < \sigma \end{cases} \quad (93)$$

$H_1$  acts to keep the particles confined, and  $H_2$  prevents them from overlapping. One nice thing about using hard-disk Hamiltonians is that there is never a reason to evaluate a Boltzmann factor. Any trial move that results in an overlap or a particle crossing the boundary gives an “infinite”  $\Delta U$ , so  $e^{-\beta\Delta U}$  is identically 0 and the trial is unconditionally rejected.

`hdisk.c` requires as user input any two of the following three parameters:  $R$ , the radius of the circle in  $\sigma$ ;  $\rho$ , the areal number density of particles (# per square  $\sigma$ ); and  $N$ , the number of particles. The user may also specify  $\delta r$ , the scalar displacement, and `nc`, the number of MC cycles, where one cycle is  $N$  attempted particle displacements. Optionally, the code can generate configurational samples as simple text files or as a single XYZ-formatted trajectory (which looks like a concatenation of XYZ files), and in order to generate these samples, `traj_samp` must be set greater than zero. The code reports the acceptance ratio, among other things:

$$\left[ \begin{array}{c} \text{acceptance} \\ \text{ratio} \end{array} \right] = \left[ \frac{\text{number of successful trials}}{\text{number of trials}} \right]$$

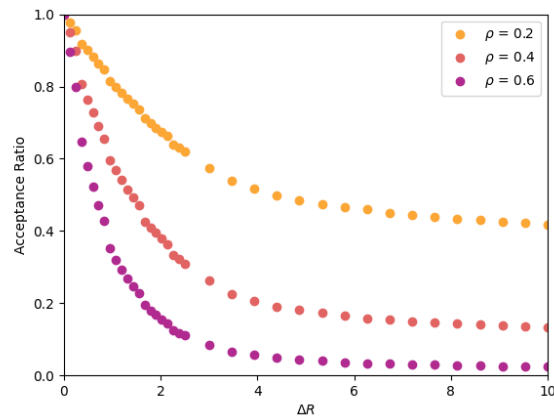
One important aspect of any MC simulation code is how the particle positions are *initialized*. Here, it is best to assign initial positions to the particles such that the initial energy is 0 (i.e., there are no overlaps nor particles out of bounds.) Try to figure out how the function `init()` in the program `hdisk.c` accomplishes this.

As a suggested further exercise, use `hdisk.c` to determine a reasonable displacement to achieve a 30% acceptance ratio at a density of 0.5. Compare your results across differently sized systems and runs with different numbers of cycles. For fewer than  $10^6$  cycles, you will have large acceptance ratios because the initial condition is not yet fully destroyed.

Below is a plot of acceptance ratio vs.  $\Delta R$  for densities  $\rho$  of 0.2, 0.4, 0.6, from a simulations of 200 particles. 2,000 cycles were performed for each run, and each had a unique seed. Are your results consistent with this data?

One advantage of the XYZ format is that we can use VMD to visualize our configurations, and even to make animations of our simulations. For example, suppose we generate a short 1,000-cycle MC trajectory of the hard-disk system at  $\rho = 0.7$  and  $N = 100$ :

```
$ cd
$ cd cheT580-202035/instructional-codes/my_work
$ mkdir hdisk_run
$ cd hdisk_run
$ gcc -O3 -o hdisk ../../originals/hdisk.c -lm -lgs1
$ ./hdisk -xyz traj.xyz -traj_samp 1 -nc 1000 -rho 0.7 -N 100 -dr 0.5
# R = 6.74336; rho = 0.70000; N = 100; seed = 23410981
```



**Figure 6:** Acceptance ratio vs.  $\Delta R$  for various densities in a 200-particle hard-disk system from 2,000-cycle MC simulations.

```

Results:
Number of Trial Moves:      100000
Maximum Displacement Length: 0.50000
Acceptance Ratio:         0.48499
Reject Fraction Out-of-bounds: 0.09707
Reject Fraction Overlap:   0.90293
$ ls
hdisk traj.xyz
  
```

Notice I provide the name of the trajectory file, indicating I want one sample per cycle. I also set the magnitude of the maximum displacement at  $0.5 \sigma$ .

After the run finishes, the file `traj.xyz` appears. It looks like this:

```

$ head -10 traj.xyz
100
Generated by hdisk.c; all z-components are zero; all elements are H
H 2.48776 1.78242 0.00000
H 5.80283 0.16579 0.00000
H 2.57789 -4.28715 0.00000
H -4.25852 0.68353 0.00000
H -5.95301 2.69712 0.00000
H 0.28903 -1.04400 0.00000
H 1.86426 -0.93249 0.00000
H -4.16984 1.87756 0.00000
  
```

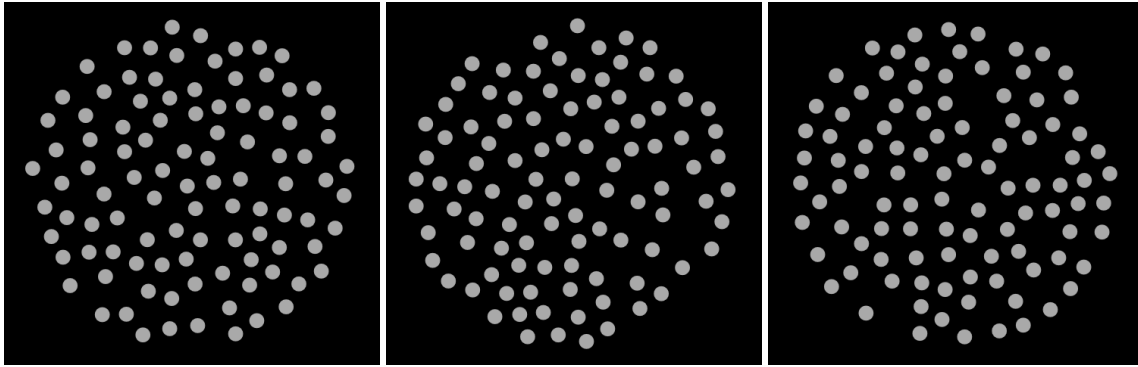
We can use VMD to visualize this trajectory. If you are on a Windows 10 machine and you installed the Windows version of VMD, you can simply launch it from the start menu, and then navigate to

```
\\wsl$\\home\\cheT580-202035\instructional-codes\my
```

and then just click on `traj.xyz` to read it in. If you are on a Mac, you should be able to navigate straight to the file. In Fig. 7, I show three snapshots from this simulation, at cycle 0, 500, and 1000.

As another suggested exercise, consider generating a trial move in the following way:

1. Randomly select a particle  $i$ .



**Figure 7:** Snapshots from a short hard-disk MC simulation at 0, 500, and 1,000 cycles. Images were rendered in VMD in an orthographic view along  $[0,0,-1]$ , and particles were rendered as “points” with size 27.

2. Randomly choose a *direction*. In 2D, this is an angle  $\phi$  chosen uniformly from the interval  $[0, 2\pi]$ . In 3D, this is *two* angles  $\theta$  and  $\phi$ , where  $\cos \theta$  is chosen uniformly from the interval  $[-1, 1]$ , and  $\phi$  is chosen uniformly from the interval  $[0, 2\pi]$ .
3. Compute the displacement vector components. In 2D:  $dx = \Delta R \cos \theta$  and  $dy = \Delta R \sin \theta$ . In 3D:  $dx = \Delta R \sin \theta \cos \phi$ ,  $dy = \Delta R \sin \theta \sin \phi$ ,  $dz = \Delta R \cos \theta$ .
4. Execute the move by adding the displacement vector components to the position of particle  $i$ .

The main difference of this scheme with the first one is that, in 2D, only one random number must be chosen per displacement attempt. Explore the acceptance ratio vs.  $\Delta R$  relationship with this new trial move. Does this scheme generate a 30% acceptance ratio for a *higher* or *lower* maximum displacement that does the original scheme?

#### 4.5 Case Study 3: Hard-Disk Dumbbells in 2D

In this case study, we consider a slightly different system than the hard-disk system in the previous case study. Here, we imagine that pairs of disks are tethered together to form dumbbells. The “bond-length” of a dumbbell is a constant parameter,  $r_0$ . We will again confine the dumbbells to a circle.

What is new is how we have to consider the trial moves for this system. We cannot simply select a random particle and try to displace it, because this is likely to violate the constant bond-length of the dumbbell that particle belongs to. How then do we generate new configurations? A simple idea is to use *two* kinds of trial moves, translation of entire dumbbells and rotation of dumbbells around their centers of mass. This was originally presented in Sec. 4.3.3. In order to implement an MC code with more than one trial move, we must include a “trial move selection rule” which randomly selects a trial move based on their user-defined “weights”.

The code `hdisk-dumbbells.c` implements a MC simulation of hard-disk dumbbells. One specifies a number of particles using `-N #` at the command line, and the number of dimers is assumed to be  $N/2$ . One can specify two of  $N$ ,  $\rho$  (areal particle density), or  $R$  (confining domain diameter). One can also specify  $\delta r$  (maximum dimer displacement distance) and  $\delta\phi$  (maximum dimer rotation angle), as well as the dimer bond length  $r_0$ . An XYZ-format trajectory can be saved every `-fs` cycles using `-traj my_traj.xyz`.

Let’s run this program for  $N = 200$ ,  $\rho = 0.6$ ,  $\delta r = 1$ ,  $\delta\phi = \pi$ , for 10,000 cycles, saving 1000 snapshots in `traj.xyz`:

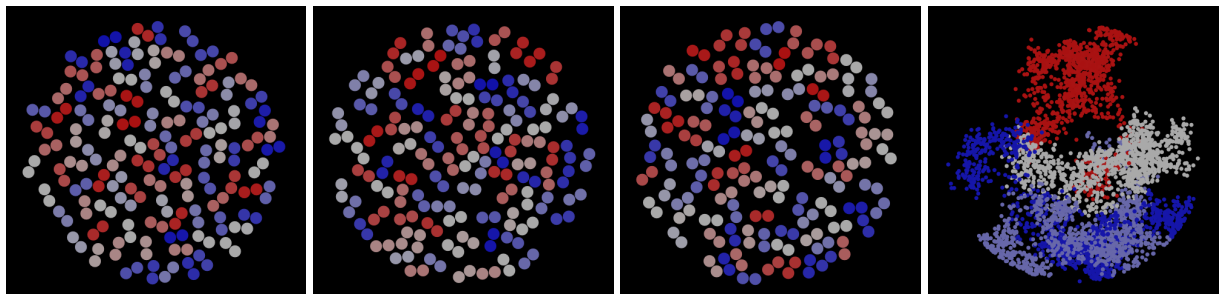
```

$ cd
$ cd cheT580-202035/instructional-codes/my_work
$ mkdir hddb_run
$ cd hddb_run
$ gcc -O3 -o hddb.././originals/hdisk-dumbbells.c -lm -lgs1
$ ./hddb -rho 0.6 -dr 1 -da 3.1 -dw 0.5 -N 200 -traj traj.xyz -fs 10
# R = 10.30; rho = 0.60; N = 200; r_0 = 1.00; s = 1.00; disp_wt = 0.5
Results:
Number of trial moves:           2000000
Maximum displacement length:    1.000
Number of displacement attempts: 999499
Maximum rotation angle (radians): 3.100
Number of rotation attempts:    1000501
Displacement acceptance ratio:  0.344
Rotation acceptance ratio:      0.405
Reject Fraction Out-of-bounds:  0.10037
Reject Fraction Overlap:        0.89963
Trajectory saved to:            traj.xyz
$ ls
hddb traj.xyz

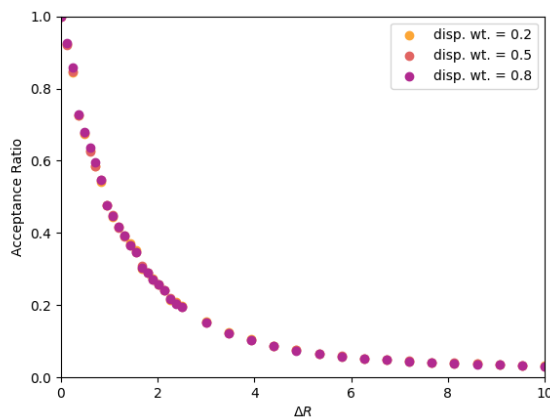
```

As with the hard disks, we can use VMD to visualize frames in this trajectory. Fig. 8 shows three representative snapshots from this simulation, along with a special view showing the histories of four particular dimers. This last rendering serves to indicate that dimers have mostly explored the entire domain for this number of cycles (is this the case for 1,000 cycles?).

Consider the following question: Does the acceptance ratio of rotational moves depend upon the weight given to displacement moves? Why or why not? Below is a plot of the acceptance ratio vs. the maximum displacement for a system of 100 dumbbells at a density of 0.5, for various displacement move weights between 0.1 and 0.9. As you can see, there appears to be no effect on the acceptance



**Figure 8:** Snapshots from a short hard-disk-dumbbell MC simulation at 0, 5000, and 10,000 cycles. Images were rendered in VMD in an orthographic view along  $[0,0,-1]$ , and particles were rendered as “points” and colored according to index. Far right: traces of positions of four particular dimers throughout the trajectory.



**Figure 9:** Displacement acceptance ratio vs. maximum dumbbell displacement for various displacement trial move weights between 0.2, 0.5, and 0.8.  $N = 200$  (100 dumbbells),  $\rho = 0.5$ , and  $R = 11.3$ .

of trial displacements if we change how frequently we perform them relative to trial rotations.

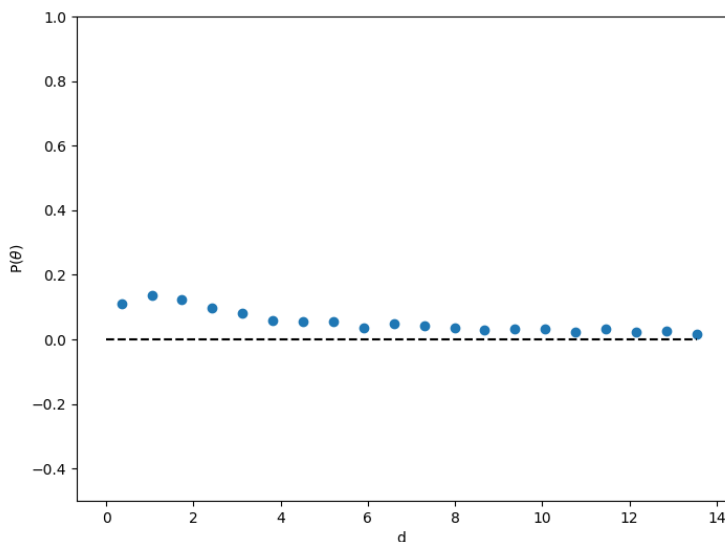
Let’s consider computing an observable function that describes how the molecules order themselves in the circular domain. One such meaningful function we’ll call  $\langle P[\theta(r)] \rangle$ , where

$$P(\theta) = \cos^2 \theta - \frac{1}{2} \tag{94}$$

and  $\theta$  is defined as the angle between a dumbbell’s bond and a vector joining its midpoint to the domain origin.  $\langle P[\theta(r)] \rangle$  is therefore the expectation of  $P$  averaged over all dumbbells located between  $r$  and  $r + \delta r$  from the origin. Why is this a meaningful measure of order for this system? Consider that dumbbells near the periphery like to align so their bonds are tangent to the periphery itself and therefore perpendicular to the radius, and the further away from the periphery you look, the less likely this order is to appear. When two vectors are perpendicular, their angle between them is  $\pi$  and since  $\cos \pi = 0$ , we expect  $\langle P \rangle$  to be -0.5. Is this what we observe for our system?

To explore this function, the code `hdisk_dumbbells_order.c` was copied from `hdisk_dumbbells.c`. In `hdisk_dumbbells_order.c`, I introduce the arrays `theta[]` to hold a tally of  $\cos^2 \theta$  values and `Rcount[]` to hold a count of hits in each radial bin, so that  $\langle P(\theta(r)) \rangle = \text{theta}[i] / \text{Rcount}[i] - 0.5$ , where `i` is the radial bin index set by `R` and a specified number of bins. Fig. 10 shows  $P$  vs  $r$  for a hard-disk dumbbell system at density 0.66 with 400 particles (200 dumbbells), run for 500,000 cycles.

We can see some interesting structure here, notably that it looks like, on average, there is very little ordering at the periphery and it becomes substantially higher as we get closer to the origin, though still not very strong. Notably, at about  $1 \sigma$  from the periphery, we see a dip in  $P$  that might indicate an enrichment in tangentially-oriented dumbbells.



**Figure 10:** Order parameter  $P$  vs radius for a hard-disk dumbbell system confined to a circle of radius  $13.9 \sigma$ .

A final note: Since this is a 2D system, we don't use the second Legendre polynomial of  $\cos \theta$ :

$$P_2(x) = \frac{1}{2}(3x^2 - 1) \quad (95)$$

The reason for this is clear. Suppose  $f(\theta)$  is the probability distribution of the angle  $\theta$ . Now, since the molecules are not polar (they have no head or tail),  $\theta$  is meaningful only on the domain  $[0, \frac{\pi}{2}]$ . Suppose further that there is no preferred angle; i.e.,  $f$  is a constant. In 2D, normalization requires

$$1 = f \int_0^{\pi/2} d\theta \rightarrow f = \frac{2}{\pi}, \quad (96)$$

and in this case, with no preferred orientation, the average of  $\cos^2 \theta$  is

$$\langle \cos^2 \theta \rangle = \frac{2}{\pi} \int_0^{\pi/2} \cos^2 \theta d\theta = \frac{2}{\pi} \frac{\pi}{4} = \frac{1}{2}. \quad (97)$$

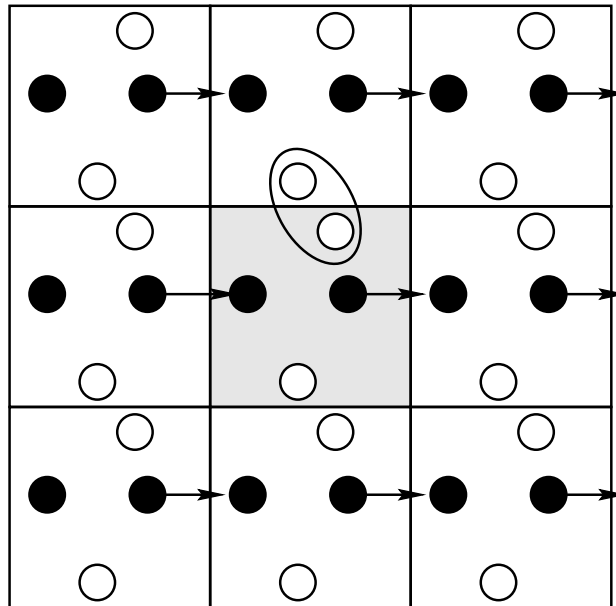
And Eq. 95 will have the described behavior. The second Legendre polynomial evaluates to zero when  $\cos^2 \theta$  is  $\frac{1}{3}$ , which is indeed what  $\langle \cos^2 \theta \rangle$  evaluates to when  $f$  is a constant in *three* dimensions.



#### 4.6 Case Study 4: Equation of State of the Lennard-Jones Fluid

The final case study we will consider in this unit on Monte Carlo simulation is the prototypical system for continuous-space, 3D liquids: The Lennard-Jones fluid. (This is detailed in Sec. 3.4, “Case Study 1” in Frenkel & Smit [1].) The primary objective of the MC code is to predict the *pressure* of a sample of Lennard-Jonesium at a given *density* and *temperature*; that is, we can use MC to map out (in principle) the *phase diagram* of a material. We will use this case study to introduce and discuss another important element of a large number of molecular simulations: *periodic boundary conditions*.

We would like to simulate bulk fluid. The apparently simplest way to approximate bulk behavior in a finite number of particles is to employ periodic boundaries. That is, we imagine the box of length  $L$  is embedded in an infinite space tiled with replicas of the central box. If we focus on the central box, and watch as one particle is displaced “out” of the box, it will reappear in the box at the opposite face. Moreover, particles interact with “images” of other particles in all replica boxes. Periodic boundaries thus allow us to mimic the infinite extent of bulk fluid.



**Figure 11:** A schematic representation of periodic boundary conditions in two dimensions. The black particle leaves the central box by leaving a through right-hand boundary, and consequently re-enters through the left-hand boundary. The two white particles interact through the boundary.

Periodic boundaries require the use of the *minimum image convention* (MIC) when computing inter-particle contributions to the total energy. Below is a modified  $N^2$  loop for a 3-D system of point particles obeying the Lennard-Jones pair potential with periodic boundaries. The function  $e_i()$  computes the sum of all pair interactions between a stipulated particle  $i$  and all particles from  $i_0$  to  $N-1$ . It is called from  $total\_e()$  such that a sum of pairwise energies for all unique pairs is computed.

```
double e_i ( int i, double * rx, double * ry, double * rz, int N,
            double L, double rc2, int tailcorr, double ecor,
            int shift, double ecut, double * vir, int i0 ) {
    int j;
    double dx, dy, dz, r2, r6i;
    double e = 0.0, hL=L/2.0;
    *vir=0.0;
```

```
for (j=i0;j<N;j++) {
  if (i!=j) {
    dx = rx[i]-rx[j];
    dy = ry[i]-ry[j];
    dz = rz[i]-rz[j];
    if (dx>hL)      dx-=L;
    else if (dx<-hL) dx+=L;
    if (dy>hL)      dy-=L;
    else if (dy<-hL) dy+=L;
    if (dz>hL)      dz-=L;
    else if (dz<-hL) dz+=L;
    r2 = dx*dx + dy*dy + dz*dz;
    if (r2<rc2) {
      r6i  = 1.0/(r2*r2*r2);
      e    += 4*(r6i*r6i - r6i) - (shift?ecut:0.0);
      *vir += 48*(r6i*r6i-0.5*r6i);
    }
  }
}
return e+(tailcorr?ecor:0.0);
}

double total_e ( double * rx, double * ry, double * rz,
                 int N, double L,
                 double rc2, int tailcorr, double ecor,
                 int shift, double ecut, double * vir ) {
  int i;
  double tvir;
  double e = 0.0;
  *vir=0.0;
  for (i=0;i<N-1;i++) {
    e += e_i(i,rx,ry,rz,N,L,rc2,
            tailcorr,ecor,shift,ecut,&tvir,i+1);
    *vir += tvir;
  }
  return e;
}
```

The function `e_i()` is also useful when determining whether or not to accept a trial displacement move, since the change in energy  $\Delta\mathcal{U}$  associated with moving particle  $i$  can be found by calling `e_i()` for particle  $i$  before and after the move; the latter energy minus the former is  $\Delta\mathcal{U}$ , since no other particles are displaced. This is much faster than just doing a full-blown  $N^2$  calculation to evaluate each trial move, but it forces you to do careful accounting to keep the total energy up-to-date. If the move is accepted the total energy must be incremented by  $\Delta\mathcal{U}$ ; if the virial is also being tallied, the virial must also be similarly incremented by the change in the virial upon particle displacement.

Note that each  $i$ - $j$  displacement component is subject to the MIC; if  $\Delta x$  is greater than  $L/2$ , we subtract  $L$  from it; if it is less than  $-L/2$ , we add  $L$  to it; same for  $\Delta y$  and  $\Delta z$ . Many caveats come with using periodic boundaries. (A thorough discussion appears in Sec. 3.2.2 of F&S.) The first thing

to realize is that the total potential per particle (as a sum of pair potentials) in principle diverges in an infinite periodic system. This can be circumvented by introducing a *finite interaction range* to the pair potential. We usually work with systems large enough such that the *cutoff* of the pair potential,  $r_c$ , is less than one-half the box-length,  $L$ , in a cubic box. This means that the “image” interactions involve only immediately neighboring replicas.

Truncation of a pair potential is an important idea to understand. The major point is that the cutoff must be spherically symmetric; that is, we can't simply cut off interactions beyond a box length in each direction, because this results in a directional bias in the interaction range of the potential. So, a hard cutoff radius  $r_c$  is required, and it should be less than half the box length. The secondary point is that, once  $r_c$  is chosen, if you wish to mimic a potential with infinite range, you must use the correction terms for energy and pressure described below.

The system we consider is made of  $N$  particles which interact via the Lennard-Jones pair potential (Eq. 89). The particles are confined in a cubic box with side-length  $L$ . Length is measured in units of  $\sigma$  and energy in  $\epsilon$ , and we consider particles with 1  $\sigma$  diameters. A code is provided for simulating this system using Metropolis Monte Carlo: `mc1j.c`. `mj1c.c` will compute the pressure given a temperature and density in the manner discussed in the text. If a cutoff radius is chosen by the user, then a *truncated* and *shifted* pair potential is used, and the following tail corrections are applied:

$$u^{\text{tail}} = \frac{8}{3}\pi\rho\epsilon\sigma^3 \left[ \frac{1}{3} \left( \frac{\sigma}{r_c} \right)^9 - \left( \frac{\sigma}{r_c} \right)^3 \right] \quad (98)$$

$$\Delta P^{\text{tail}} = \frac{16}{3}\pi\rho^2\epsilon\sigma^3 \left[ \frac{2}{3} \left( \frac{\sigma}{r_c} \right)^9 - \left( \frac{\sigma}{r_c} \right)^3 \right] \quad (99)$$

The pressure is computed from

$$P = \rho T + \text{vir}/V \quad (100)$$

where *vir* is the virial:

$$\text{vir} = \frac{1}{3} \sum_{i>j} \mathbf{f}(r_{ij}) \cdot \mathbf{r}_{ij} \quad (101)$$

and  $V$  is the system volume.  $\mathbf{f}(r_{ij})$  is the *force* exerted on particle  $i$  by particle  $j$ , defined as the negative gradient of the pair potential  $u(r_{ij})$  with respect to the position of particle  $i$ :

$$\mathbf{f}(r_{ij}) = -\frac{\partial u(r_{ij})}{\partial \mathbf{r}_i} \quad (102)$$

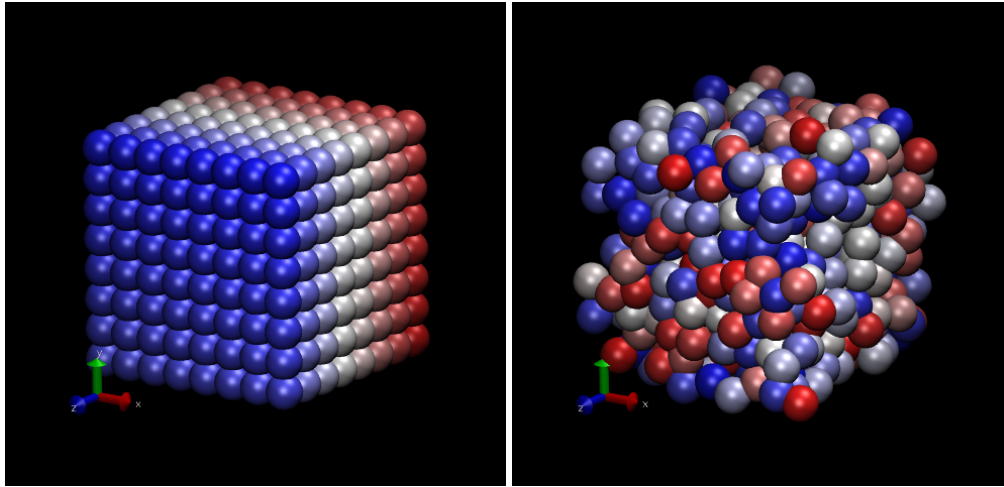
$$= -\frac{\mathbf{r}_{ij}}{r_{ij}} \frac{\partial u(r_{ij})}{\partial r_{ij}} \quad (103)$$

Here we have made use of the fact that

$$\frac{\partial X}{\partial \mathbf{r}} = \frac{\mathbf{r}}{r} \frac{\partial X}{\partial r} \quad (104)$$

when operating on a function  $X$  which depends on relative particle separations. For the Lennard-Jones pair potential (Eq. 89), we see that

$$\frac{\partial u(r_{ij})}{\partial r_{ij}} = 4\epsilon \left[ -12 \frac{\sigma^{12}}{r_{ij}^{13}} + 6 \frac{\sigma^6}{r_{ij}^7} \right] \quad (105)$$



**Figure 12:** Snapshots from a MC simulation of a Lennard-Jones fluid of 512 particles at a density of 0.5 and a temperature of 1.0 with a cutoff of  $2.5 \sigma$ , for 10,000 cycles. Left is initial, right is final.

So,

$$\mathbf{f}_{ij}(r_{ij}) = \frac{\mathbf{r}_{ij}}{r_{ij}^2} \left\{ 48\epsilon \left[ \left( \frac{\sigma}{r_{ij}} \right)^{12} - \frac{1}{2} \left( \frac{\sigma}{r_{ij}} \right)^6 \right] \right\}, \quad (106)$$

and therefore,

$$\text{vir} = \frac{1}{3} \sum_{i>j} \left\{ 48\epsilon \left[ \left( \frac{\sigma}{r_{ij}} \right)^{12} - \frac{1}{2} \left( \frac{\sigma}{r_{ij}} \right)^6 \right] \right\} \quad (107)$$

Notice that any particular pair's contribution to the total virial is *positive* if the members of the pair are repelling one another (positive  $f$  along  $\mathbf{r}_{ij}$ ), and *negative* if the particles are attracting one another.

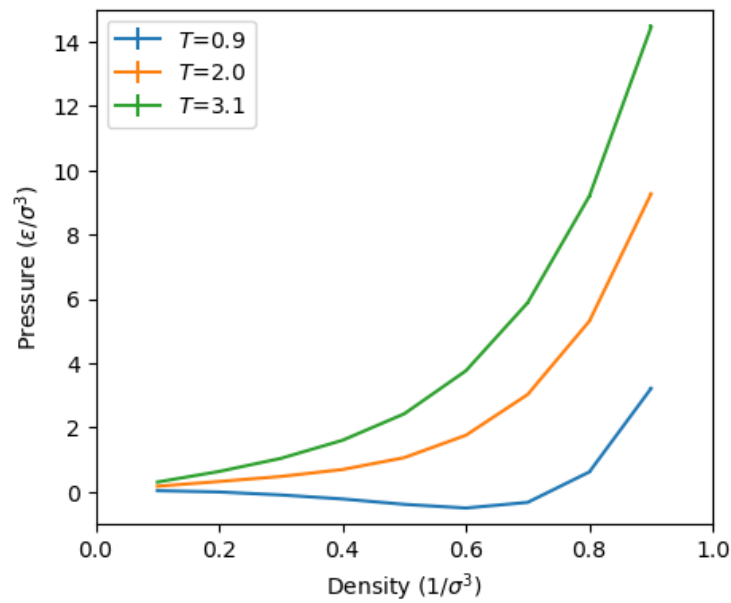
If you read the code `mc1j.c`, you should see that the initialization of positions is accomplished by putting the particles on cubic lattice sites such that an overall density is achieved. It is therefore convenient to run simulations with numbers of particles that are perfect cubes, such as 128, 216, 512, etc, so that the initial state uniformly fills the box.

Another consideration is that a certain number of cycles should be “burned” prior to gathering statistics so this initial state is fully erased. The flag `-ne` allows the user to specify how many equilibration cycles are to be performed before switching to “production” mode.

Fig. 12 shows two snapshots made with VMD of the Lennard-Jones systems with 512 particles.

As a suggested exercise, you can use `mc1j.c` to try to reproduce Figure 3.5 in F&S, which shows  $P$  vs.  $\rho$  at both  $T = 2.0$  and  $T = 0.9$ . How many cycles do you need? How many equilibration cycles? What maximum displacement did you choose?

Below are some of my preliminary results using the code `mc1j.c`. I used only 5,000 cycles for 512 particles for each point, and each point is the result of a single run. These numbers appear to compare well with those in Figure 3.5 in F&S, for which we have no idea how many cycles or independent runs were performed.



**Figure 13:** Pressure vs. density in a Lennard-Jones fluid, measured by Metropolis MC simulation, at reduced temperatures 0.9, 2.0, and 3.1. 600,000 particle-displacement moves were performed per simulation, and five independent simulations per point were performed. Pressures are averages over these five, and standard deviations are smaller than the line width. Each system contained 512 particles. A cutoff of  $3.5\sigma$  was used.

## 5 Molecular Dynamics Simulation

We saw that the Metropolis Monte Carlo simulation technique generates a sequence of states with appropriate probabilities for computing ensemble averages (Eq. 1). Generating states probabilistically is not the only way to explore phase space. The idea behind the Molecular Dynamics (MD) technique is that we can observe our dynamical system explore phase space by solving all particle *equations of motion*. We treat the particles as classical objects that, at least at this stage of the course, obey Newtonian mechanics. Not only does this in principle provide us with a properly weighted sequence of states over which we can compute ensemble averages, it additionally gives us *time-resolved* information, something that Metropolis Monte Carlo cannot provide. The “ensemble averages” computed in traditional MD simulations are in practice *time averages*:

$$\langle G \rangle = \bar{G} = \frac{1}{N_{\text{samp}} \Delta t} \sum_{i=1}^{N_{\text{samp}}} G[r(t)] \quad (108)$$

The *ergodic hypothesis* partially requires that the measurement time,  $\tau_{\text{meas}} = N_{\text{samp}} \Delta t$ , is greater than the longest *relaxation time*,  $\tau_r$ , in the system. The price we pay for this extra information is that we must at least access if not store particle velocities in addition to positions, and we must compute interparticle *forces* in addition to potential energy. We will introduce and explore MD in this section.

### 5.1 MD: Theoretical Background

#### 5.1.1 Newtonian Mechanics and Numerical Integration

The Newtonian equations of motion can be expressed as

$$m \ddot{\mathbf{r}}_i + \nabla_i \mathcal{U} = 0 \quad (109)$$

where  $\ddot{\mathbf{r}}_i$  is the acceleration of particle  $i$ , and the *force* acting on particle  $i$  is given by the negative gradient of the total potential,  $\mathcal{U}$ , with respect to its position:

$$\mathbf{f}_i = -\nabla_i \mathcal{U} = -\frac{\partial \mathcal{U}}{\partial \mathbf{r}_i} \quad (110)$$

Whereas in a typical MC simulation, in which all we really need is the ability to evaluate the potential energy of a configuration, in MD we actually need to evaluate all interparticle *forces* for a configuration.

We first encountered interparticle forces in Sec. 4.6 in a discussion of the *virial* in computing pressure in a standard Metropolis Monte Carlo simulation of the Lennard-Jones liquid. At this point, it suffices to consider a system with generic pairwise interactions, for which the total potential is given by:

$$\mathcal{U} = \sum_j \sum_{k>j} u_{jk}(r_{jk}) \quad (111)$$

where  $r_{jk}$  is the scalar distance between particles  $j$  and  $k$ , and  $u_{jk}$  is the pair potential specific to pair  $(j, k)$ . For a system of  $N$  identical particles, Eq. 111 is a summation of  $\frac{1}{2}N(N-1)$  terms. So, the force on any particular particle,  $i$ , selects  $N$  terms from the above summation; that is, those terms involving particle  $i$ :

$$\mathbf{f}_i = -\sum_{j=1}^N \frac{\partial u_{ij}(r_{ij})}{\partial \mathbf{r}_i} = \sum_{j=1}^N \mathbf{f}_{ij} \quad (112)$$

where we can define the quantity  $\mathbf{f}_{ij}$  is the force exerted on particle  $i$  by virtue of the fact that it interacts

with particle  $j$ . Because  $u_{ij}$  is a function of a scalar quantity, we can break the derivative up:

$$\mathbf{f}_{ij}(r_{ij}) = -\frac{\partial U_{LJ}(r_{ij})}{\partial \mathbf{r}_i} \quad (113)$$

$$= -\frac{\mathbf{r}_{ij}}{r_{ij}} \frac{\partial U_{LJ}(r_{ij})}{\partial r_{ij}} \quad (114)$$

Eq. 114 illustrates that, because  $\mathbf{r}_{ij} = -\mathbf{r}_{ji}$ ,

$$\mathbf{f}_{ij} = -\mathbf{f}_{ji} \quad (115)$$

This leads us to the comforting result that

$$\mathbf{F} = \sum_i \mathbf{f}_i = 0 \quad (116)$$

That is, the total force on the collection of particles is zero. (The same result holds identically for all potentials which are functions of *relative* interatomic positions only.) But the practical advantage of this result is that, when we visit the pair  $(i, j)$  and compute the force on  $i$  due to its interaction with  $j$ ,  $\mathbf{f}_{ij}$ , we automatically have the force on  $j$  due to its interaction with  $i$ ,  $-\mathbf{f}_{ij}$ . Some refer to this as “Newton’s Third Law.”

The other key aspect of a simple MD program is a means of *numerical integration* of the equations of motion of each particle. We first consider the “simple Verlet” algorithm, which is an explicit integration scheme. Let us consider a Taylor-expanded version of one coordinate of the position of a particular particle,  $r(t)$ :

$$r(t + \Delta t) = r(t) + v(t) \Delta t + \frac{f(t)}{2m} (\Delta t)^2 + \frac{(\Delta t)^3}{3!} \ddot{r} + \mathcal{O}[(\Delta t)^4], \quad (117)$$

and, letting  $\Delta t \rightarrow -\Delta t$ ,

$$r(t - \Delta t) = r(t) - v(t) \Delta t + \frac{f(t)}{2m} (\Delta t)^2 - \frac{(\Delta t)^3}{3!} \ddot{r} + \mathcal{O}[(\Delta t)^4]. \quad (118)$$

When we add these together, we obtain:

$$r(t + \Delta t) \approx 2r(t) - r(t - \Delta t) + \frac{f(t)}{2m} (\Delta t)^2. \quad (119)$$

Eq. 119 is termed the “Verlet” algorithm (going back to Verlet’s simulations of liquid argon [4]). Notice that, when one chooses a small  $\Delta t$ , one can predict the position of a particle at time  $t + \Delta t$  given its position at time  $t$  and the force acting on it at time  $t$ . We see that the new position coordinate has an error of order  $(\Delta t)^4$ .  $\Delta t$  is called the “time step” in a molecular dynamics simulation.

A system obeying Newtonian mechanics conserves total energy. For a dynamical system (i.e., a system of interacting particles) obeying Newtonian mechanics, the configurations generated by integration are members of the *microcanonical ensemble*; that is, the ensemble of configurations for which  $NVE$  is constant, constrained to a subvolume  $\Omega$  in phase space. The “natural” ensemble for Metropolis Monte Carlo, you will recall, is canonical; for MD, it is microcanonical. Later, we will consider techniques for conducting MD simulations in other ensembles (at constant temperature and/or pressure, for example).

When the Verlet algorithm is used to integrate Newtonian equations of motion, the total energy

of the system is conserved to within a finite error, so long as  $\Delta t$  is “small enough.” How does one determine a reasonable value for  $\Delta t$ ? Basically the same way we determined reasonable maximum displacements in continuous-space MC simulation: trial and error. We will play with time-step values in the next section, in which we consider MD simulation of the Lennard-Jones liquid.

In saying that the total energy is conserved, we realize that total energy is the sum of potential and kinetic energy. To integrate the equations of motion, we need to compute neither the potential or kinetic energy, so we have to take extra steps in an MD program to make sure total energy is being conserved. Potential energy is easily accumulated during the calculation of forces, but kinetic energy has to be computed using particle *velocities*:

$$\mathcal{K} = \frac{1}{2} \sum_i m_i |\mathbf{v}_i|^2 \quad (120)$$

But where are velocities in the Verlet algorithm? They are not necessary for updating positions, but can be easily “generated” provided one stores previous, current, and next-time-step positions in implementing the algorithm:

$$v(t) = \frac{r(t + \Delta t) - r(t - \Delta t)}{2\Delta t} + \mathcal{O}[(\Delta t)^2] \quad (121)$$

Eq. 121 is used in Algorithm 6 in F&S (Integration of the Equations of Motion) simply in order to compute  $\mathcal{K}$ . Energy conservation can be checked by tracking the total energy,  $\mathcal{U} + \mathcal{K}$ .

While we are considering the instantaneous kinetic energy,  $\mathcal{K}$ , it is useful to recognize a working definition of instantaneous temperature,  $T$ :

$$\frac{3}{2} N k_B T = \mathcal{K} = \frac{1}{2} \sum_{i=1}^N m_i |\mathbf{v}_i|^2 \quad (122)$$

Because  $\mathcal{K}$  fluctuates in time as the system evolves, so does the temperature. So, the actual temperature of a system in a microcanonical MD simulation is a time-average.

Perhaps the most popular integrator is the “velocity Verlet” algorithm [5]. Every MD code I have ever written or used (this totals a dozen or so) has used the velocity Verlet algorithm, so I feel at least it is worth explaining here. The velocity Verlet algorithm requires updates of both positions and velocities:

$$r(t + \Delta t) = r(t) + v(t) \Delta t + \frac{f(t)}{2m} (\Delta t)^2 \quad (123)$$

$$v(t + \Delta t) = v(t) + \frac{f(t + \Delta t) + f(t)}{2m} \Delta t \quad (124)$$

The update of velocities uses an arithmetic average of the force at time  $t$  and  $t + \Delta t$ . This results in a slightly more stable integrator compared to the simple Verlet algorithm, in that one may use slightly larger time-steps to achieve the same level of energy conservation. (Aside: a nice project idea is to quantify this statement.) This might imply that one has to maintain *two* parallel force arrays. In practice, this is not necessary, because the velocity update can be split to either side of the force computation,



forming a so-called “leapfrog” algorithm:

$$r(t + \Delta t) = r(t) + v(t) \Delta t + \frac{f(t)}{2m} (\Delta t)^2 \quad \text{Update positions} \quad (125)$$

$$v\left(t + \frac{1}{2}\Delta t\right) = v(t) + \frac{f(t)}{2m} \Delta t \quad \text{Half-update velocities} \quad (126)$$

$$r(t + \Delta t) \rightarrow f(t + \Delta t) \quad \text{Compute forces.}$$

$$v(t + \Delta t) = v\left(t + \frac{1}{2}\Delta t\right) + \frac{f(t + \Delta t)}{2m} \Delta t \quad \text{Half-update velocities} \quad (127)$$

In the next section (Sec. 5.2), we will consider the velocity Verlet algorithm in the context of an MD simulation of the Lennard-Jones fluid.

As a final tidbit, we must consider periodic boundaries applied in a molecular dynamics simulation. Consider *modes* of a system. Think of a mode as a concerted vibration of collections of particles with a characteristic wavelength. A dense system will have short wavelength (local) modes, and long wavelength modes, like large-scale concerted “sloshing” of the particles in the system. These modes exist naturally in matter, and the partitioning of energy among these various modes is important to understand in describing some transport properties. The key caveat is that modes with wavelengths that are incommensurate with the box size are not permitted in a periodic system because they cancel themselves. A mode is commensurate with the box so long as an integer multiple of its wavelength is the box length. This can actually be very restrictive in systems with a wide span of wavelengths, like amorphous unstructured solids, but is not that important for amorphous liquids.

### 5.1.2 The Liouville Operator Formalism to Generating MD Integration Schemes

In this section, we present an elegant formalism for deriving MD integrators, as discussed by Tuckerman *et al.* [6]. What we present here is essentially the first two parts of the second section of Reference [6], including some of my own elaboration and some of that presented in section 4.3 of F&S.

Imagine a quantity  $f$  which is a function of particle positions  $\mathbf{r}^N$  and momenta  $\mathbf{p}^N$ . Its time derivative is given by

$$\dot{f} = \dot{\mathbf{r}} \frac{\partial f}{\partial \mathbf{r}} + \dot{\mathbf{p}} \frac{\partial f}{\partial \mathbf{p}} \quad (128)$$

We can write down a *formal* solution to this equation. First, define the Liouville operator as

$$iL = \dot{\mathbf{r}} \frac{\partial}{\partial \mathbf{r}} + \dot{\mathbf{p}} \frac{\partial}{\partial \mathbf{p}} \quad (129)$$

As Tuckerman points out, the  $i$  is there by convention and ensures that the operator is *Hermitian*. We can re-express Eq. 128 as

$$\dot{f} = iLf \quad (130)$$

which we solve directly to yield

$$f(t) = \exp(iLt) f(0). \quad (131)$$

If  $f$  is itself a vector quantity identical to the set of positions and momenta,  $\Gamma$ , we have a way to express, formally, the evolution of the system:

$$\Gamma(t) = U(t) \Gamma(0) \quad (132)$$

where  $U(t) = \exp(iLt)$  is the *classical propagator*. The idea with numerical integration is that we find a way to represent the propagator as a *discrete algorithm* for constructing the system at some time  $t + \Delta t$  given the system at time  $t$ .

Let's build our discrete integrator by decomposing the operator:

$$iL = iL_1 + iL_2 \quad (133)$$

This does not necessarily lead to two independent propagators, because the two components do not commute; that is:

$$\exp[(iL_1 + iL_2)t] \neq \exp(iL_1t) \exp(iL_2t) \quad (134)$$

Consider the action of the partial Liouville operator

$$iL_1 \equiv \dot{\mathbf{r}}(0) \frac{\partial}{\partial \mathbf{r}}, \quad (135)$$

which gives

$$f[\mathbf{p}^N(t), \mathbf{r}^N(t)] = \exp\left[\dot{\mathbf{r}}(0) \frac{\partial}{\partial \mathbf{r}} t\right] f[\mathbf{p}^N(0), \mathbf{r}^N(0)] \quad (136)$$

$$= \sum_{n=0}^{\infty} \frac{(\dot{\mathbf{r}}(0)t)^n}{n!} \frac{\partial^n}{\partial \mathbf{r}^n} f[\mathbf{p}^N(0), \mathbf{r}^N(0)] \quad (137)$$

$$= f[\mathbf{p}^N(0), \mathbf{r}^N(0) + \dot{\mathbf{r}}(0)t] \quad (138)$$

The last line is the collapse of the Taylor expansion of the line immediately above it. So, the effect of this operator fragment is a simple shift of coordinates given some initial velocities. This is an interesting fact: we can consider first-order integration as a Taylor expansion.

The next step of Tuckerman was to apply the Trotter identity:

$$\exp[(iL_1 + iL_2)t] = \lim_{P \rightarrow \infty} [\exp(iL_1t/2P) \exp(iL_2t/P) \exp(iL_1t/2P)]^P \quad (139)$$

When  $P$  is large but finite:

$$\exp[(iL_1 + iL_2)t] = [\exp(iL_1t/2P) \exp(iL_2t/P) \exp(iL_1t/2P)]^P \exp[\mathcal{O}(1/P^2)] \quad (140)$$

Now, we define a finite timestep as  $\Delta t = t/P$  and we have then a discrete operator that, when applied to a configuration at time  $t$ , will produce the configuration at time  $t + \Delta t$ :

$$\exp(iL_1\Delta t/2) \exp(iL_2\Delta t) \exp(iL_1\Delta t/2) \Gamma(t) = \Gamma(t + \Delta t) \quad (141)$$

By performing this operation sequentially  $P$  times, we recover a discretized version of the formal solution to generate  $\Gamma(t)$  given  $\Gamma(0)$ .

Now we explicitly consider the decomposition:

$$iL_1 = \dot{\mathbf{p}}(0) \frac{\partial}{\partial \mathbf{p}} \quad (142)$$

$$iL_2 = \dot{\mathbf{r}}(0) \frac{\partial}{\partial \mathbf{r}} \quad (143)$$

We can perform one  $\Delta t$ 's worth of update using the following operation on  $f$ :

$$\exp\left(\dot{\mathbf{p}}(0) \left(\frac{\partial}{\partial \mathbf{p}}\right) \frac{\Delta t}{2}\right) \exp\left(\dot{\mathbf{r}}(0) \left(\frac{\partial}{\partial \mathbf{r}}\right) \Delta t\right) \exp\left(\dot{\mathbf{p}}(0) \left(\frac{\partial}{\partial \mathbf{p}}\right) \frac{\Delta t}{2}\right) f[\mathbf{p}^N(t), \mathbf{r}^N(t)]$$

The action of the rightmost operator,  $\exp\left(\dot{\mathbf{p}}(0) \left(\frac{\partial}{\partial \mathbf{p}}\right) \frac{\Delta t}{2}\right)$ :

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

The action of the next rightmost,  $\exp\left(\dot{\mathbf{r}}(0) \left(\frac{\partial}{\partial \mathbf{r}}\right) \Delta t\right)$ :

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

Then, the action of the final operator:

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

Noting that  $\mathbf{p} = m\dot{\mathbf{r}}$  and  $\mathbf{F} = m\mathbf{a} = \dot{\mathbf{p}}$ , we can summarize the effect of this three-step update of the positions and velocities as

$$\mathbf{r}(\Delta t) = \mathbf{r}(0) + \Delta t\dot{\mathbf{r}}(0) + \frac{(\Delta t)^2}{2} \frac{\mathbf{F}[\mathbf{r}(0)]}{m}, \quad (144)$$

$$\dot{\mathbf{r}}(\Delta t) = \dot{\mathbf{r}}(0) + \frac{\Delta t}{2m} \{\mathbf{F}[\mathbf{r}(0)] + \mathbf{F}[\mathbf{r}(\Delta t)]\} \quad (145)$$

This is the velocity-Verlet algorithm, seen previously in Eqs 125-127. Interestingly, we can also reverse the order of the decomposition; i.e.,

$$iL_1 = \dot{\mathbf{r}}(0) \frac{\partial}{\partial \mathbf{r}} \quad (146)$$

$$iL_2 = \dot{\mathbf{p}}(0) \frac{\partial}{\partial \mathbf{p}} \quad (147)$$

The update algorithm that arises is

$$\dot{\mathbf{r}}(\Delta t) = \dot{\mathbf{r}}(0) + \Delta t\mathbf{F}\left[\mathbf{r}(0) + \frac{\Delta t}{2m}\dot{\mathbf{r}}(0)\right] \quad (148)$$

$$\mathbf{r}(\Delta t) = \mathbf{r}(0) + \frac{\Delta t}{2}[\dot{\mathbf{x}}(0) + \dot{\mathbf{x}}(\Delta t)]. \quad (149)$$

This is termed the *position Verlet* algorithm [6]. Tuckerman *et al.* showed that this new algorithm results in a slightly lower drift in total energy in MD simulation of a simple Lennard-Jones fluid, when the time-step is greater than about 0.004.

## 5.2 Case Study 1: An MD Code for the Lennard-Jones Fluid

### 5.2.1 Introduction

Let us consider a few of the important elements of any MD program:

1. Force evaluation;
2. Integration; and
3. Configuration output.

We will understand these elements by manipulating an existing simulation program that implements the *Lennard-Jones fluid* (which you may recall was analyzed using Metropolis Monte-Carlo simulation

in Sec. 4). The program is called `md1j.c` in the instructional codes repository.

Recall the Lennard-Jones pair potential:

$$u(r) = 4\epsilon \left[ \left( \frac{\sigma}{r} \right)^{12} - \left( \frac{\sigma}{r} \right)^6 \right] \quad (150)$$

When implementing this in an MD code, similar to its implementation in MC, we adopt a reduced unit system, in which we measure length in  $\sigma$ , and energy in  $\epsilon$ . Additionally, for MD, we need to measure mass, and we adopt units of particle mass,  $m$ . This convention makes the force on a particle *numerically equivalent* to its acceleration. With these conventions, time is a derived unit:

$$t[=]\sigma\sqrt{m/\epsilon} \quad (151)$$

We also measure reduced temperature in units of  $\epsilon/k_B$ ; so for a system of identical Lennard-Jones particles:

$$\frac{3}{2}NT = \mathcal{K} = \frac{1}{2} \sum_{i=1}^N |\mathbf{v}_i|^2 \quad (152)$$

(Recall that the mass  $m$  is 1 in Lennard-Jones reduced units.) These conventions obviate the need to perform unit conversions in the code.

We have already encountered interparticle forces for the Lennard-Jones pair potential in the context of computing the pressure from the virial in the MC simulation of the LJ fluid (Sec. 4.6). Briefly, the force exerted on particle  $i$  by virtue of its Lennard-Jones interaction with particle  $j$ ,  $\mathbf{f}_{ij}$ , is given by:

$$\mathbf{f}_{ij}(r_{ij}) = \frac{\mathbf{r}_{ij}}{r_{ij}^2} \left\{ 48\epsilon \left[ \left( \frac{\sigma}{r_{ij}} \right)^{12} - \frac{1}{2} \left( \frac{\sigma}{r_{ij}} \right)^6 \right] \right\} \equiv \mathbf{r}_{ij} f. \quad (153)$$

And, as shown in the previous section, once we have computed the vector  $\mathbf{f}_{ij}$ , we automatically have  $\mathbf{f}_{ji}$ , because  $\mathbf{f}_{ij} = -\mathbf{f}_{ji}$ . The scalar  $f$  is called a “force factor.” If  $f$  is negative, the force vector  $\mathbf{f}_{ij}$  points from  $i$  to  $j$ , meaning  $i$  is *attracted* to  $j$ . Likewise, if  $f$  is positive, the force vector  $\mathbf{f}_{ij}$  points from  $j$  to  $i$ , meaning that  $i$  is being forced *away* from  $j$ .

Below is a C-code fragment for computing both the total potential energy *and* interparticle forces:

```
double forces ( double * rx, double * ry, double * rz,
               double * fx, double * fy, double * fz,
               int n ) {
    int i,j;
    double dx, dy, dz, r2, r6, r12;
    double e = 0.0, f = 0.0;

    for (i=0;i<n;i++) {
        fx[i] = 0.0;
        fy[i] = 0.0;
        fz[i] = 0.0;
    }
    for (i=0;i<(n-1);i++) {
        for (j=i+1;j<n;j++) {
            dx      = rx[i]-rx[j];
            dy      = ry[i]-ry[j];
```

```

    dz      = rz[i]-rz[j];
    r2      = dx*dx + dy*dy + dz*dz;
    r6i     = 1.0/(r2*r2*r2);
    r12i    = r6i*r6i;
    e       += 4*(r12i - r6i);
    f       = 48/r2*(r6i*r6i-0.5*r6i);
    fx[i]   += dx*f;
    fx[j]   -= dx*f;
    fy[i]   += dy*f;
    fy[j]   -= dy*f;
    fz[i]   += dz*f;
    fz[j]   -= dz*f;
  }
}
return e;
}

```

Notice that the argument list now includes arrays for the forces, and because force is a vector quantity, we have three parallel arrays for a three-dimensional system. These forces must of course be initialized, shown in lines 6-10. The  $N^2$  loop for visiting all unique pairs of particles is opened on lines 11-12. The inside of this loop is very similar to the evaluation of potential first presented in the MC simulation of the Lennard-Jones fluid; the only real difference is the computation of the “force factor,”  $f$ , on line 20, and the subsequent increment of force vector components on lines 21-26. Notice as well that there is no implementation of periodic boundary conditions in this code fragment; it was left out for simplicity. What would this “missing” code do? (Hint: look at the code *mdlj.c* for the answer.)

The second major aspect of MD is the integrator. As discussed in class, we will primarily use Verlet-style (explicit) integrators. The most common version is the velocity-Verlet algorithm [5], first presented in Sec. 5.1.1. Below is a fragment of C-code for executing one time step of integration for a system of  $N$  particles:

```

for (i=0;i<N;i++) {
  rx[i]+=vx[i]*dt+0.5*dt2*fx[i];
  ry[i]+=vy[i]*dt+0.5*dt2*fy[i];
  rz[i]+=vz[i]*dt+0.5*dt2*fz[i];
  vx[i]+=0.5*dt*fx[i];
  vy[i]+=0.5*dt*fy[i];
  vz[i]+=0.5*dt*fz[i];
}

PE = total_e(rx,ry,rz,fx,fy,fz,N,L,rc2,ecor,ecut,&vir);

KE = 0.0;
for (i=0;i<N;i++) {
  vx[i]+=0.5*dt*fx[i];
  vy[i]+=0.5*dt*fy[i];
  vz[i]+=0.5*dt*fz[i];
  KE+=vx[i]*vx[i]+vy[i]*vy[i]+vz[i]*vz[i];
}
KE*=0.5;

```

Notice the update of positions (Eq. 125), where  $v_x[i]$  is the  $x$ -component of velocity,  $f_x[i]$  is the  $x$ -component of force,  $dt$  and  $dt^2$  are the time-step and squared time-step, respectively. Notice that there is *no* implementation of periodic boundaries in this code fragment; what would this “missing code” look like? (Hint: see *mdlj.c* for the answer!) Lines 5-7 are the first half-update of velocities (Eq. 126). The force routine computes the new forces on the currently updated configuration on line 10. Then, lines 12-18 perform the second-half of the velocity update (Eq. 127). Also note that the kinetic energy,  $\mathcal{H}$ , is computed in this loop.

### 5.2.2 The Code

The complete C program, *mdlj.c*, contains a complete implementation of the Lennard-Jones force routine and the velocity-Verlet integrator. Compilation instructions appear in the header comments. Let us now consider some sample results from *mdlj.c*. First, *mdlj.c* includes an option that outputs a brief summary of the command line options available:

```
$ mdlj -h
mdlj usage:
mdlj [options]

Options:
  -N [integer]      Number of particles
  -rho [real]       Number density
  -dt [real]        Time step
  -rc [real]        Cutoff radius
  -ns [real]        Number of integration steps
  -T0 [real]        Initial temperature
  -fs [integer]     Sample frequency
  -traj [string]    Trajectory file name
  -prog [integer]   Interval with which logging output is
  -icf [string]     Initial configuration file
  -seed [integer]   Random number generator seed
  -uf               Print unfolded coordinates in traject
  -h               Print this info
```

Let us run *mdlj.c* for 512 particles and 1000 time-steps at a density of 0.85 and an initial temperature of 2.5. We will pick a relatively conservative (small) time-step of 0.001. We will not specify an input configuration, instead allowing the code to create initial positions on a cubic lattice. Here is what we see in the terminal:

```
$ ./mdlj -N 512 -fs 10 -ns 1000 -traj traj.xyz -T0 2.5 -rho 0.85 -rc 2.5
# NVE MD Simulation of a Lennard-Jones fluid
# L = 8.44534; rho = 0.85000; N = 512; rc = 2.50000
# nSteps 1000, seed 23410981, dt 0.00100
#LABELS step time PE KE TE drift T P
0 0.00000 -2429.99610 1919.39622 -510.59988 2.58118e-07 2.49921 4.28896
1 0.00100 -2428.18306 1917.58237 -510.60069 1.84111e-06 2.49685 4.30232
2 0.00200 -2425.15191 1914.54975 -510.60216 4.73104e-06 2.49290 4.32466
3 0.00300 -2420.88660 1910.28230 -510.60430 8.92431e-06 2.48735 4.35604
4 0.00400 -2415.36384 1904.75673 -510.60711 1.44276e-05 2.48015 4.39659
5 0.00500 -2408.55314 1897.94254 -510.61060 2.12521e-05 2.47128 4.44649
6 0.00600 -2400.41688 1889.80212 -510.61476 2.94064e-05 2.46068 4.50593
7 0.00700 -2390.91048 1880.29088 -510.61960 3.88852e-05 2.44830 4.57515
```

```

8 0.00800 -2379.98339 1869.35814 -510.62525 4.99443e-05 2.43406 4.65427
9 0.00900 -2367.57807 1856.94670 -510.63137 6.19292e-05 2.41790 4.74385
10 0.01000 -2353.63308 1842.99526 -510.63782 7.45638e-05 2.39973 4.84412
11 0.01100 -2338.08390 1827.43905 -510.64484 8.83209e-05 2.37948 4.95492
12 0.01200 -2320.86308 1810.21130 -510.65178 1.01908e-04 2.35705 5.07698
...

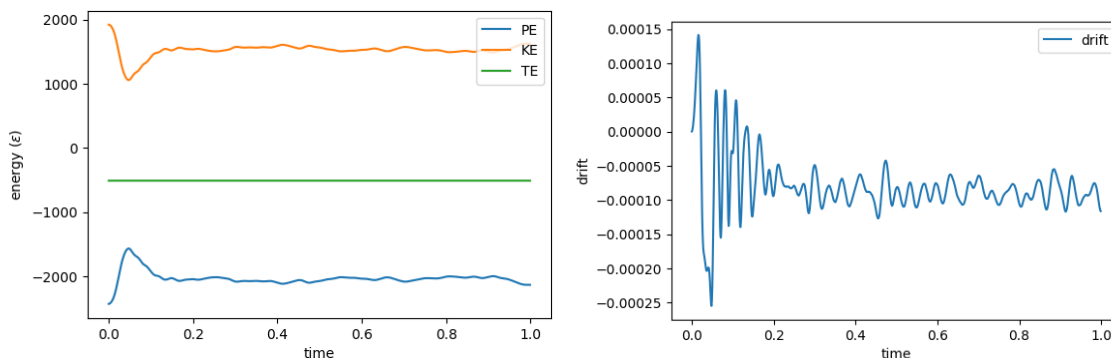
```

Each line of output after the header information corresponds to one time-step. (The header line that begins with the special word `#LABELS` is used by the Python program `plot_md1j_log.py`.) The first column is the time-step, the second is the time value, the third is the potential energy, the fourth is the kinetic energy, the fifth is the total energy, the sixth is the “drift,” the seventh is the instantaneous temperature (Eq. 152), and the eighth is the instantaneous pressure (Eq. 100).

The drift is output to assess the *stability* of the explicit integration. As a rule of thumb, we would like to keep the drift to below 0.01% of the total energy. The drift reported by `md1j.c` is computed as

$$\Delta \mathcal{E}(t) = \frac{\mathcal{E}(t) - \mathcal{E}(0)}{\mathcal{E}(0)} \quad (154)$$

where  $\mathcal{E}$  is the total energy. The plots below show the output trace for the full 1,000 time-steps. We note that with this time-step value (0.001) keeps the total energy conserved to about one part in  $10^4$ .



**Figure 14:** *Left.* Potential (PE), kinetic (KE), and total (TE) energies as functions of time in an NVE MD simulation of the Lennard-Jones fluid at reduced temperature  $T = 2.0$ . (Initial temperature was set at 2.5.) *Right.* Drift in total energy (Eq. 154) vs. time.

Now, this invocation of `md1j.c` produces an XYZ-format trajectory file (just like `mc1j` did). Here, I have expanded my XYZ-format convention to allow inclusion of velocities in the output file. Inclusion of velocities is signified by a 1 on the same line as the number of particles (the first line in each frame). If there is a 1 in that position, then each particle line includes three position coordinates and three velocity components:

```

$ more traj.xyz
512 1
BOX 8.44534 8.44534 8.44534
16 0.527833 0.527833 0.527833 -0.982352 -0.823300 -1.530590
16 1.583500 0.527833 0.527833 -0.549726 -0.942381 1.363996
16 2.639167 0.527833 0.527833 1.145014 -0.616762 -1.172725
16 3.694834 0.527833 0.527833 1.520337 3.057595 -1.522653
16 4.750501 0.527833 0.527833 -0.541350 -0.144665 -0.919845
16 5.806168 0.527833 0.527833 -0.221483 1.143893 -1.015784

```

```

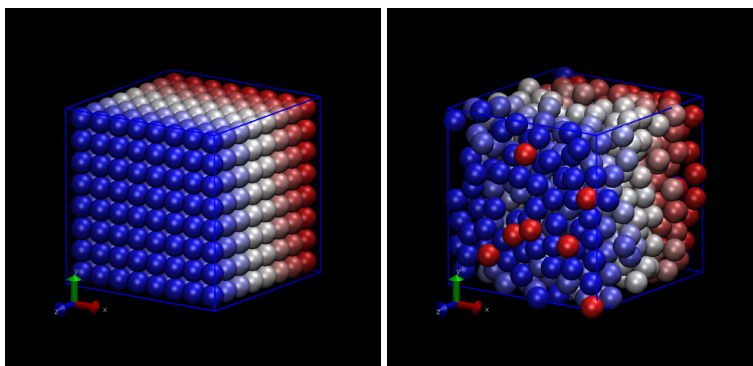
16  6.861835  0.527833  0.527833  5.752205  -1.045603  1.066189
16  7.917502  0.527833  0.527833  0.819532  0.167275  -1.230297
16  0.527833  1.583500  0.527833  -1.062591  2.169692  -1.318921
16  1.583500  1.583500  0.527833  -1.394993  0.055193  1.411530
...

```

The number “16” at the beginning designates the “type” as an atomic number; for simplicity, I have decided to call all of my particles sulfur. (XYZ format is often used for atomically-specific configurations.) The functions `xyz_out()` and `xyz_in()` write and read this format, respectively, in `mdlj.c`. We will use these functions in other programs as well, typically those which *analyze* configuration data. Examples of such analysis codes are the subjects of the next two sections.

At this point, you can’t do much with all this data, except appreciate just how much data an MD code *can* produce. In this example, we generated 100 frames of configuration data (positions and velocities) for a 512-atoms system, and the resulting trajectory file is about 3.5 megabytes in size. Of course, that file size scales with both the number of particles and the number of frames it contains. It is not unusual nowadays for researchers to use MD to produce *hundreds of gigabytes* of configuration data in order to write a single paper. It leads one to think that perhaps a lesson on handling large amounts of data is appropriate for a course on Molecular Simulation; however, I’ll forego that for now by trying to keep our sample exercises small.

One thing we *can* do with this data is make nice pictures using VMD. Below are two renderings, one of the initial snapshot, and the other at time  $t = 1$  (1000 time steps). Notice how the initially perfect crystalline lattice has been wiped out.



**Figure 15:** VMD-generated snapshots of configurations from an NVE MD simulation of the Lennard-Jones fluid at density  $\rho = 0.85$  and average temperature  $\langle T \rangle \approx 2$ . Particles are colored according to their initial  $z$  position.

## 5.3 Case Study 2: Static Properties of the Lennard-Jones Fluid

### 5.3.1 Running the code

The code `mdlj.c` was run on 108 particles with positions initialized on a simple cubic lattice at a density  $\rho = 0.8442$  and temperature (specified by velocity assignment and scaling) initially set at  $T = 0.728$ . A single run of 600,000 time steps was performed, which took about 2 minutes on my laptop. (This is almost  $10^6$  particle-updates per second; not bad for a laptop running a silly  $N^2$  pair search, but it’s only for  $\sim 100$  particles...the same algorithm applied to 10,000 would be slower.) The commands I issued looked like:

```

$ mkdir md_cs2
$ cd md_cs2
$ ../mdlj -N 108 -fs 1000 -ns 600000 -rho 0.8442 -T0 0.728 -rc 2.5 -traj traj1.xyz -prog 100 >& 1.out &
$ tail -f 1.out
468900 468.90000 -475.45590 242.36056 -233.09534 -2.79034e-04 1.49605 5.43064
469000 469.00000 -475.67663 242.58353 -233.09310 -2.88639e-04 1.49743 5.11914

```



```

469100 469.10000 -477.58731 244.49557 -233.09175 -2.94452e-04 1.50923 4.88396
469200 469.20000 -457.41152 224.32176 -233.08975 -3.02989e-04 1.38470 5.82717
469300 469.30000 -492.03036 258.93556 -233.09481 -2.81326e-04 1.59837 4.82093
469400 469.40000 -465.44072 232.34350 -233.09723 -2.70947e-04 1.43422 5.69688
469500 469.50000 -478.52166 245.42514 -233.09652 -2.73957e-04 1.51497 5.01491
469600 469.60000 -469.00769 235.90985 -233.09784 -2.68307e-04 1.45623 5.64466
469700 469.70000 -476.27992 243.18354 -233.09637 -2.74603e-04 1.50113 5.34172
469800 469.80000 -461.91910 228.82734 -233.09176 -2.94382e-04 1.41251 5.68907
469900 469.90000 -469.00227 235.90429 -233.09798 -2.677~C
  
```

The final `\&` puts the simulation “in the background,” and returns the shell prompt to you. The `tail -f` command allows you to “watch” as the log file is written to. You can also verify that the job is running by issuing the “top” command, which displays in the terminal the a listing of processes using CPU, ranked by how intensively they are using the CPU. This command “takes over” your terminal to display continually updated information, until you hit Ctrl-C.

```

top - 13:43:27 up 4 days, 6:19, 0 users, load average: 0.16, 0.45, 0.28
Tasks: 29 total, 2 running, 27 sleeping, 0 stopped, 0 zombie
%Cpu(s): 12.7 us, 0.0 sy, 0.0 ni, 83.3 id, 0.0 wa, 0.0 hi, 4.0 si, 0.0 st
MiB Mem : 12608.9 total, 11169.1 free, 739.2 used, 700.7 buff/cache
MiB Swap: 4096.0 total, 4096.0 free, 0.0 used, 11616.8 avail Mem

  PID USER      PR  NI   VIRT   RES    SHR  S  %CPU  %MEM    TIME+  COMMAND
14668 cfa        20   0   6652   1268   1116  R 100.0  0.0   0:02.83  mdlj
13073 cfa        20   0  863448  47256  28728  S   6.7   0.4   0:19.24  node
   1 root        20   0   1020    648    520  S   0.0   0.0   1:29.57  init
   8 root        20   0    888    76     16  S   0.0   0.0   0:00.00  init
   9 root        20   0    888    76     16  S   0.0   0.0   1:04.20  init
  10 cfa        20   0  10164  4980   3188  S   0.0   0.0   0:00.11  bash
  168 root        20   0    984    172    16  S   0.0   0.0   0:00.00  init
  
```

From the command line arguments shown above, we can see that this simulation run will produce 601 snapshots, beginning with  $t = 0$  and outputting every 1000 steps. Each frame has 108 lines of 69 bytes each, plus another 30 for the header in each frame, so basically every frame is 7,482 bytes. With 601 of them, that is 4,496,682 bytes, or 4.2884 megabytes (1 megabyte is 1,048,576 bytes). We can confirm this calculation using the `du` (disk usage) command:

```

$ du -sh traj1.xyz
4.3M   traj1.xyz
  
```

Thirty years ago, one might have raised an eyebrow at this; nowadays, this is very nearly an insignificant amount of storage.

### 5.3.2 Equilibration and Decorrelation

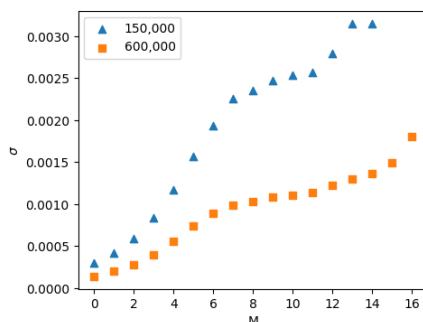
An important purpose of this case study is to quantify the notion of “equilibration” of the system by assessing correlations in (apparently) randomly fluctuating quantities like potential energy. Remember, in order to perform accurate ensemble averaging over an MD trajectory, we have to be sure that correlations in the properties we are measuring have “died out.” This is another way of saying that the length of the time interval over which we conduct the *time average* must be much longer than the correlation time. In this case study, we illustrate the “block averaging” technique of Flyvbjerg and Petersen [7] to determine the equilibration time-scale of the potential energy.

First, compute the variance of the  $L$  samples of  $\mathcal{U}$ :

$$\sigma_0^2(\mathcal{U}) \approx \frac{1}{L} \sum_{i=1}^L [\mathcal{U}_i - \bar{\mathcal{U}}]^2 \quad (155)$$

This is approximate because of so far undetermined time-correlations in  $\mathcal{U}$ ; that is, not all  $L$  samples are uncorrelated. For example, two samples one time-step apart will likely be very close to one another. Now, we block the data by averaging  $L/2$  pairs of adjacent values:

$$\mathcal{U}_i^{(1)} = \frac{\mathcal{U}_{2i-1} + \mathcal{U}_{2i}}{2} \quad (156)$$



**Figure 16:** The standard deviation,  $\sigma$ , of potential energy per particle vs. number of blocking operations  $M$  for a simulations of 150,000 and 600,000 time-steps: initial temperature  $T_0 = 0.729$ , density  $\rho = 0.8442$ , number of particles  $N = 108$ .

**Table 1:** Average potential energy per particle  $\langle \mathcal{U} \rangle / N$ , kinetic energy  $\langle \mathcal{K} \rangle / N$ , total energy  $\langle \mathcal{T} \rangle / N$ , and pressure  $P$  (all in Lennard-Jones units) for three distinct  $N=108$  particle systems run for 600,000 time steps at number density  $\rho = 0.8442$  and initial temperature  $T_0$  of 0.728.

| Run  | $\langle \mathcal{U} \rangle / N$ | $\langle \mathcal{K} \rangle / N$ | $\langle \mathcal{T} \rangle$ | $\langle P \rangle$ |
|------|-----------------------------------|-----------------------------------|-------------------------------|---------------------|
| 1    | $-4.4165 \pm 0.0012$              | $2.2577 \pm 0.0012$               | $1.5051 \pm 0.0008$           | $5.1960 \pm 0.0057$ |
| 2    | $-4.4188 \pm 0.0010$              | $2.2597 \pm 0.0010$               | $1.5064 \pm 0.0007$           | $5.1948 \pm 0.0050$ |
| 3    | $-4.4157 \pm 0.0011$              | $2.2564 \pm 0.0011$               | $1.5043 \pm 0.0008$           | $5.2022 \pm 0.0055$ |
| Avg. | $-4.4170 \pm 0.0011$              | $2.2579 \pm 0.0011$               | $1.5053 \pm 0.0008$           | $5.1977 \pm 0.0054$ |

The (1) superscript indicates that this is a “first-generation” coarsening of the potential energy trace. The variance of this new set,  $\sigma_1^2$  is computed. Then the process is recursively carried out through many subsequent generations. After many blocking operations, the coarsened samples,  $\mathcal{U}_i^{(j)}$ , become uncorrelated, and the variance saturates (temporarily). This means we should observe a plateau in a plot of variance vs generation. The Python program `flyvberg.py` implements this calculation using outputs of `mdlj.c` as inputs.

According to the discussion of this blocking technique, the standard deviation shows a dependence on the blocking degree,  $M$ , when the blocked averages are correlated, and plateaus at a blocking degree for which the averages become uncorrelated. This blocking degree corresponds to a time interval of length  $2^M$ . This data indicates that potential energy decorrelates after a time of approximately  $2^{10} \approx 1000$  steps. This is a reassuring result, as we could have guessed that 1000 steps are required based on the initial transience in the energy traces seen in the previous figure. I ran three independent simulations, each differing only in the random number seed used. The results are shown in Table 1.

### 5.3.3 Radial Distribution Functions and Postprocessing

A second major objective of this case study is to demonstrate how to compute the radial distribution function,  $g(r)$ . The radial distribution function is an important statistical mechanical function that captures the structure of liquids and amorphous solids. We can express  $g(r)$  using the following statement:

$$\rho g(r) = \begin{array}{l} \text{average density of particles at } r \text{ given that} \\ \text{a tagged particle is at the origin} \end{array} \quad (157)$$

The procedure we will follow will be to write a second program (a “postprocessing code”) which will read in the trajectory output produced by the simulation, `mdlj.c`. The general structure of a  $g(r)$

post-processing code could look like this:

1. Determine trajectory time limits: start, stop, and step
2. Initialize histogram.
3. Read in the trajectory as a list of frames.
4. For each frame:
  - (a) Visit all unique pairs of particles, and update histogram for each visit if applicable
5. Normalize histogram and output.
6. End.

We will consider a code, `rdf.c`, that implements this algorithm for computing  $g(r)$ , but first we present a brief argument for post-processing vs on-the-fly processing for computing quantities such as  $g(r)$ . For demonstration purposes, it is arguably simpler to drop in a  $g(r)$ -histogram update sampling function into an existing MD simulation program to enable computation of  $g(r)$  *during* a simulation, compared to writing a wholly separate program. After all, it nominally involves less coding. The counterargument is that, once you have a working (and eventually optimized) MD simulation code, one should be wary of modifying it. The purpose of the MD simulation is to produce samples. One can produce samples once, and use any number of post-processing codes to extract useful information. The counterargument becomes stronger when one considers that, for particularly large-scale simulations, it is simply not convenient to re-run an entire simulation when one discovers a slight bug in the sampling routines. The price one pays is that one needs the disk space to store configurations.

As shown earlier, one MD simulation of 108 particles out to 600,000 time steps, storing configurations every 1,000 time steps, requires less than 5 MB. This is an insignificant price. Given that we *know* that the MD simulation works correctly, it is sensible to *leave it alone* and write a quick, simple post-processing code to read in these samples and compute  $g(r)$ .

The code `rdf.c` is a C-code implementation of just such a post-processing code. This program illustrates a different way to abstractify the trajectory, namely as a list of frames. A frame is an instance of an abstract data type called `frametype` that we define (for now) in `rdf.c`, along with a special function `NewFrame()` to allocate memory for a frame, and another `read_xyz_frame()` to read a frame in from an XYZ-format trajectory:

```
typedef struct FRAME {
    double * rx, * ry, * rz; // coordinates
    double * vx, * vy, * vz; // velocities
    int * typ; // array of particle types 0, 1, ...
    int N; // number of particles
    double Lx, Ly, Lz; // box dimensions
} frametype;

/* Create and return an empty frame */
frametype * NewFrame ( int N, int hv ) {
    frametype * f = (frametype*)malloc(sizeof(frametype));
    f->N=N;
    f->rx=(double*)malloc(sizeof(double)*N);
    f->ry=(double*)malloc(sizeof(double)*N);
    f->rz=(double*)malloc(sizeof(double)*N);
    if (hv) {
        // caller has requested a frame with space for velocities
        f->vx=(double*)malloc(sizeof(double)*N);
        f->vy=(double*)malloc(sizeof(double)*N);
    }
}
```

```
        f->vz=(double*)malloc(sizeof(double)*N);
    } else {
        f->vz=NULL;
        f->vy=NULL;
        f->vx=NULL;
    }
    f->typ=(int*)malloc(sizeof(int)*N);
    return f;
}
/* Read an XYZ-format frame from stream fp; returns the new frame.
   Note the non-conventional use of the first line to indicate
   whether or not the frame contains velocities and the comment
   line to hold boxsize information. */
frametype * read_xyz_frame ( FILE * fp ) {
    int N,i,j,hasvel=0;
    double x, y, z, Lx, Ly, Lz, vx, vy, vz;
    char typ[3], dummy[5];
    char ln[255];
    frametype * f = NULL;
    if (fgets(ln,255,fp)){
        sscanf(ln,"%i %i\n",&N,&hasvel);
        f = NewFrame(N,hasvel);
        fgets(ln,255,fp);
        sscanf(ln,"%s %lf %lf %lf\n",dummy,&f->Lx,&f->Ly,&f->Lz);
        for (i=0;i<N;i++) {
            fgets(ln,255,fp);
            sscanf(ln,"%s %lf %lf %lf %lf %lf %lf\n",
                typ,&f->rx[i],&f->ry[i],&f->rz[i],&vx,&vy,&vz);
            if (hasvel) {
                f->vx[i]=vx;
                f->vy[i]=vy;
                f->vz[i]=vz;
            }
            j=0;
            while(strcmp(elem[j],"NULL")&&strcmp(elem[j],typ)) j++;
            if (strcmp(elem[j],"NULL")) f->typ[i]=j;
            else f->typ[i]=-1;
        }
    }
    return f;
}
```

With this abstract data type, we no longer need to pass all parallel arrays as separate parameters; we can instead just pass a pointer `frametype*`. For example, below is the function `rij` that computes the minimum-image convention distance between particles `i` and `j` in a particular frame:

```
/* Compute scalar distance between particles i and j in frame f;
   note the use of the minimum image convention */
double rij ( frametype * f, int i, int j ) {
```

```

double dx, dy, dz;
double hLx=0.5*f->Lx,hLy=0.5*f->Ly,hLz=0.5*f->Lz;
dx=f->rx[i]-f->rx[j];
dy=f->ry[i]-f->ry[j];
dz=f->rz[i]-f->rz[j];
if (dx<-hLx) dx+=f->Lx;
if (dx> hLx) dx-=f->Lx;
if (dy<-hLy) dy+=f->Ly;
if (dy> hLy) dy-=f->Ly;
if (dz<-hLz) dz+=f->Lz;
if (dz> hLz) dz-=f->Lz;
return sqrt(dx*dx+dy*dy+dz*dz);
}

```

Using `rij()` it is then easy to update the RDF histogram:

```

/* An N^2 algorithm for computing interparticle separations
   and updating the radial distribution function histogram. */
void update_hist ( frametype * f, double rcut,
                  double dr, int * H, int nbins ) {

    int i,j;
    double r;
    int bin;
    for (i=0;i<f->N-1;i++) {
        for (j=i+1;j<f->N;j++) {
            r=rij(f,i,j);
            if (r<rcut) {
                bin=(int)(r/dr);
                if (bin<0||bin>=nbins) {
                    fprintf(stderr,
                            "Warning: %.3lf not on [0.0,%.3lf]\n",
                            r,rcut);
                } else {
                    H[bin]+=2;
                }
            }
        }
    }
}

```

`H` is the histogram. One can see that the bin value is computed by first dividing the actual distance between members of the pair by the *resolution* of the histogram,  $\delta r$ , and casting the result as an integer. This resolution can be specified on the command-line when `rdf.c` is executed. Also notice that the histogram is updated by 2, which reflects the fact that either of the two particles in the pair can be placed at the origin. Also notice the implementation of the minimum image convention.

In the `main()` function of `rdf.c`, a simple block of code can read in a whole trajectory from a file named by `trajfile` and store it in an array of `frametype*` pointers:

```

i=0;
fprintf(stdout,"Reading %s\n",trajfile);

```

```

fp=fopen(trajfile,"r");
while (Traj[i++]=read_xyz_frame(fp));
nFrames=i-1;
fclose(fp);
if (!nFrames) {
    fprintf(stdout,"Error: %s has no data.\n",trajfile);
    exit(-1);
}
fprintf(stdout,"Read %i frames from %s.\n",nFrames,trajfile);

```

Then a second block of code allocates, initializes, and computes the pair correlation histogram:

```

/* Adjust cutoff and compute histogram */
L2min=min(Traj[0]->Lx/2,min(Traj[0]->Ly/2,Traj[0]->Lz/2));
if (rcut>L2min) rcut=L2min;
nbins=(int)(rcut/dr)+1;
H=(int*)malloc(sizeof(int)*nbins);
for (i=0;i<nbins;i++) H[i]=0;
for (i=begin_frame;i<nFrames;i++)
    update_hist(Traj[i],rcut,dr,H,nbins);
nFramesAnalyzed=nFrames-begin_frame;

```

Note that the cutoff `rcut` may not exceed half a box length in any dimension. The number of histogram bins is simply one plus the cutoff divided by the resolution `dr`. The variable `begin_frame` allows the caller to gather statistics only after a certain number of frames in the trajectory in order to “ignore” initial frames where the initial configuration is still “remembered”.

Finally, once the trajectory has been traversed and the histogram computed, it is then normalized by compute  $g(r)$  and save the result to a designated output file:

```

/* Normalize and output g(r) to the terminal */
/* Compute density, assuming NVT ensemble */
fp=fopen(outfile,"w");
fprintf(fp,"# RDF from %s\n",trajfile);
fprintf(fp,"#LABEL r g(r)\n");
fprintf(fp,"#UNITS %s *\n",length_units);
/* Ideal-gas global density; assumes V is constant */
rho=Traj[0]->N/(Traj[0]->Lx*Traj[0]->Ly*Traj[0]->Lz);
for (i=0;i<nbins-1;i++) {
    /* bin volume */
    vb=4./3.*M_PI*((i+1)*(i+1)*(i+1)-i*i*i)*dr*dr*dr;
    /* number of particles in this shell if this were
       an ideal gas */
    nid=vb*rho;
    fprintf(fp,"%0.5lf %0.5lf\n",i*dr,
           (double)(H[i])/(nFramesAnalyzed*Traj[0]->N*nid));
}
fclose(fp);
fprintf(stdout,"%s created.\n",outfile);

```

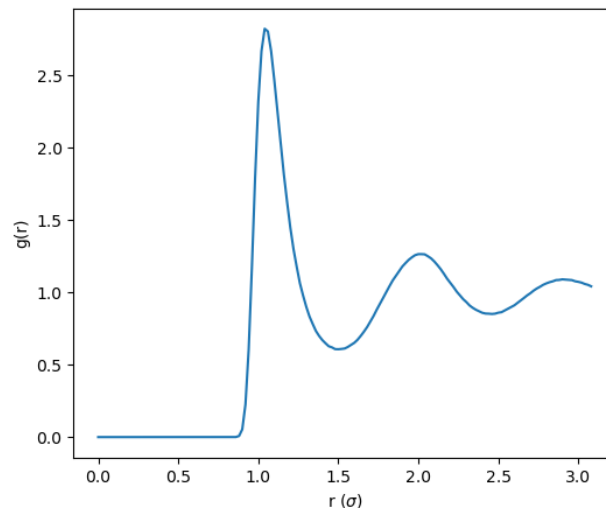
(The variable `length_units` is a string that just labels the length units; by default, this is “sigma”, indicating LJ  $\sigma$ .)

Now, let's execute `rdf` from one of our 600,000-time-step simulations' 6,001-frame trajectory, ignoring the first 1,000 frames (100,000 time steps):

```
$ cd ~/dxu/chet580/instructional-codes/my_work/mdlj/set1
$ gcc -O5 -o rdf ../../originals/rdf.c -lm
$ ./rdf -t traj-rho0.90-rep0.xyz -dr 0.02 -rcut 3.5 \
  -o rdf-rho0.90-rep0.dat -begin-frame 1000
Reading traj-rho0.90-rep0.xyz
Read 6001 frames from traj-rho0.90-rep0.xyz.
rdf-rho0.90-rep0.dat created.
$
```

Using the python program `plot_rdf.py`, we can generate a plot of this RDF (Fig. 17):

```
$ python ../../originals/plot_rdf.py -i rdf-rho0.90-rep0.dat \
  -o rdf-rho0.90-rep0.png
```



**Figure 17:** Radial distribution function of a Lennard-Jones fluid at reduced density 0.90,  $N = 216$ , cutoff of 3.5, NVE MD.

This  $g(r)$  shows a peak at about  $2^{1/6}\sigma$  that corresponds to the LJ well, indicating that there is a dense nearest-neighbor shell out to about  $1.5\sigma$ . How dense? We can use  $g(r)$  to count particles within a distance  $r$  from a central atom:

$$n(r) = \rho \int_0^r \int_0^\pi \int_0^{2\pi} g(r') (r')^2 \sin\theta dr' d\theta d\phi = 4\pi\rho \int_0^r (r')^2 g(r') dr' \quad (158)$$

This integration is enabled in `plot_rdf.py` via the `-rho` and `-R` flags:

```
$ python ../../originals/plot_rdf.py -i rdf-rho0.90-rep0.dat \
  -o rdf-rho0.90-rep0.png -rho 0.9 -R 1.5
n=12.335
```

This indicates the nearest neighbor shell is pretty well packed; spherical close-packing would be exactly 12.

### 5.4 Case Study 3: Transport Properties: The Self-Diffusion Coefficient

This Case Study combines elements of Case Studies 5 and 6 in F&S, which are unfortunately incomplete in their description. The purpose of this Case Study is to demonstrate how one computes a self-diffusion coefficient,  $\mathcal{D}$ , from an MD simulation of a simple Lennard-Jones liquid. There are two means to computing  $\mathcal{D}$ : (1) the mean-squared displacement (“MSD”)  $\langle r^2 \rangle(t)$ , and (2) the velocity autocorrelation function,  $VACF(t)$ . The approaches are equivalent in the sense that the MSD is the integral representation of the VACF; the former is termed the “Einstein” approach, while the latter is the “Green-Kubo” approach [8].

The self-diffusion coefficient governs the evolution of concentration,  $c$ , (or number density) according to a generalized transport equation:

$$\frac{\partial c}{\partial t} = \mathcal{D} \nabla^2 c \quad (159)$$

Einstein showed (details in text) that  $\mathcal{D}$  is related to the mean-squared displacement,  $\langle r^2 \rangle$ :

$$\frac{\partial \langle r^2 \rangle}{\partial t} = 6\mathcal{D} \quad (160)$$

At long times,  $\mathcal{D}$  should be independent of time; hence

$$\langle r^2 \rangle = \lim_{t \rightarrow \infty} 6\mathcal{D}t \quad (161)$$

We can compute  $\langle r^2 \rangle$ , and therefore estimate  $\mathcal{D}$ , easily using MD simulation. There is, however, a very important consideration concerning periodic boundary conditions. Recall that, during integration, immediately after the position update, we test to see if the update has taken the particle outside of the primary box. If it has, we simply shift the particle’s position by a box length in the appropriate dimension and direction. The displacement of the particle during this step is *not* a box length, but if you consider just the coordinates as they appear in the output, you would think that it is. It is therefore important that we work with *unfolded* coordinates when computing mean-squared displacement. This is not adequately explained in the text, so we cover it in some detail here.

“Unfolding” coordinates in a simulation with periodic boundaries requires that we keep track of how many times each particle has crossed a boundary. The code *mdlj.c* allows output of unfolded coordinates in the trajectory output using the `-uf` switch on the command line. Now, generally the array `rx[]` always contains the periodically shifted coordinates, but we can easily generate the *unfolded* coordinates at any time (say, upon output) by performing the following operation:

```
rxu = rx[i]+ix[i]*L;
```

This is because `ix[]` contains a tally of the number of times periodic crossings in the  $x$  direction have occurred: +1 is added to the tally every time a particle’s  $x$  position exceeds  $L$  and is wrapped back in by subtracting  $L$ , and -1 is added to the tally every time a particle’s  $x$  position is below 0 and is wrapped back in by adding  $L$ . Here,  $L$  is the box length (assumed cubic).

The program *msd.c* computes the MSD from a trajectory with unfolded coordinates using a conventional, straightforward algorithm. The C-code for this algorithm appears below.  $M$  is the number of “frames” in the trajectory, and  $N$  is the number of particles.  $\langle r^2 \rangle(t)$  is computed by considering the change in particle position over an interval of size  $t$ . Any frame in the trajectory can be considered an origin for any interval size, provided enough frames come after it in the trajectory. This means that we additionally average over all possible time origins. `dt` is a variable that loops over allowed time *intervals*. `cnt[]` counts the number of time origins for a given interval. `sd[]` is the array in which we accumulate squared displacement at each time interval, and has  $M$  elements, one for each allowed interval.



```

/* Compute the mean-squared displacement using
   the straightforward algorithm */
fprintf(stdout, "# computing MSD...\n"); fflush(stdout);
for (t=begin_frame;t<M;t++) {
  for (dt=1;(t+dt)<M;dt++) {
    cnt[dt]++; /* number of origins for interval length dt */
    for (i=0;i<Traj[0]->N;i++) {
      sd[dt] += rij2_unwrapped(Traj[t+dt],i,Traj[t],i,1);
    }
  }
}

```

The function `rij2_unwrapped(fi,i,fj,j,1)` very simply computes the squared displacement between particle `i` in frame `fi` and particle `j` in frame `fj`:

```

double rij2_unwrapped ( frametype * fi, int i,
                        frametype * fj, int j, int com_corr ) {
  double dx, dy, dz;
  dx=fi->rx[i]-(com_corr?fi->cx:0)-fj->rx[j]+(com_corr?fj->cx:0);
  dy=fi->ry[i]-(com_corr?fi->cy:0)-fj->ry[j]+(com_corr?fj->cy:0);
  dz=fi->rz[i]-(com_corr?fi->cz:0)-fj->rz[j]+(com_corr?fj->cz:0);
  return dx*dx+dy*dy+dz*dz;
}

```

The parameter `com_corr` removes the center of mass drift from the displacement; the center of mass should not move in NVE, but we will use this code for trajectories in which the COM does diffuse. The center of mass of a frame is part of the `frametype` data type used in `msd.c`, and it's computed when the frame is read in.

The code fragment below completes the averaging, and outputs the total mean-squared displacement.

```

fp=fopen(outfile,"w");
fprintf(fp,"# MSD from %s\n",trajfile);
fprintf(fp,"#LABEL time msd\n");
fprintf(fp,"#UNITS %s %s^2\n",time_units,length_units);
for (t=0;t<M-begin_frame;t++) {
  sd[t] /= cnt[t]?(Traj[0]->N*cnt[t]):1;
  fprintf(fp,"% .5lf % .8lf\n",
         t*traj_interval*md_time_step,sd[t]);
}
fclose(fp);

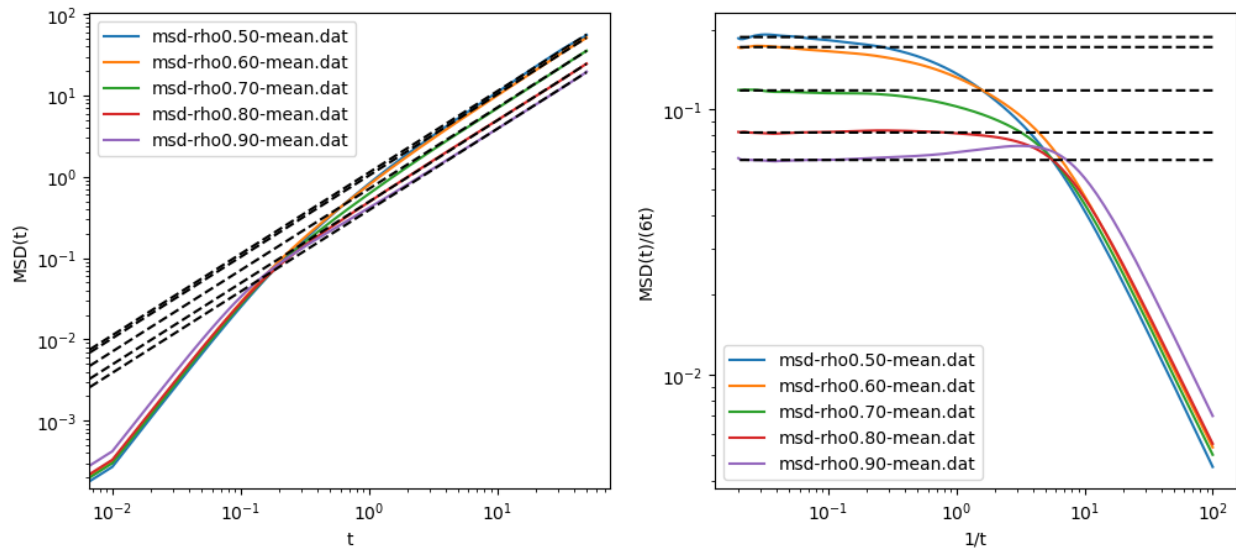
```

Fig. 18 shows MSD vs time from 60,000-step MD simulations at various densities in which frames are saved at intervals of 10 time-steps. In these simulations, the density was constant and there were 216 particles, with velocities initialized at 0.7. The figure shows the data plotted two ways: MSD vs.  $t$  on the left, and  $\text{MSD}/(6t)$  vs  $1/t$  on the right. The program `msd` can post-process an unwrapped trajectory file to generate the MSD:

```

$ ./msd -t traj-rho0.90-rep0.xyz -traj-interval 10 \
-o msd-rho0.90-rep0.dat -begin-frame 1000

```



**Figure 18:** Mean-squared displacement (MSD) vs. simulation time (in reduced LJ units) for a 216-particle, 60,000-step NVE MD simulations at various values of  $\rho$ . Blue curves are MD data and black dashed lines are fits to the Einstein relation to extract  $\mathcal{D}$ . All simulations had velocities initialized at  $T = 0.7$ .

Repeating this process for several densities and several replicas per density builds a nice dataset. MSD at each density was averaged over replicas and plotted using `plot_msd.py`:

```
$ python ../../originals/plot_msd.py -i msd-rho0.50-mean.dat \
-i msd-rho0.60-mean.dat -i msd-rho0.70-mean.dat \
-i msd-rho0.80-mean.dat -i msd-rho0.90-mean.dat \
-o msd-rho-T0.70.png -lowt 1
msd-rho0.50-mean.dat 0.18750243355919102
msd-rho0.60-mean.dat 0.17188252756031633
msd-rho0.70-mean.dat 0.11801146782479009
msd-rho0.80-mean.dat 0.08188412712076683
msd-rho0.90-mean.dat 0.0645241901384243
```

The parameter `-lowt` is the lower time limit beyond which the data is fit to calculate  $\mathcal{D}$ . Values of  $\mathcal{D}$  are output here.

You can see that the MSD transitions from a short-time regime where  $\text{MSD} \propto t^2$  to a long-time regime where  $\text{MSD} \propto t$ . That short-time region displays “ballistic” behavior, and on those time scales particles move ballistically (with constant velocity) between collisions with other particles; you can see by the value of MSD of about 0.02 that they are moving only about 0.1 particle diameters or so before colliding. On the longer, “diffusive” timescales, we can see the expected behavior.

The velocity autocorrelation function route to the diffusion constant begins with the realization that one can reconstruct the displacement of a particle over a time interval  $t$  by simply integrating its velocity:

$$\Delta \mathbf{r} = \int_0^t \mathbf{v}(t') dt' \quad (162)$$

So, the *mean squared* displacement can be expressed

$$\langle r^2 \rangle = \left\langle \left( \int_0^t \mathbf{v}(t') dt' \right)^2 \right\rangle \quad (163)$$

$$= \int_0^t \int_0^t dt' dt'' \langle \mathbf{v}(t') \cdot \mathbf{v}(t'') \rangle \quad (164)$$

$$= 2 \int_0^t \int_0^{t'} dt' dt'' \langle \mathbf{v}(t') \cdot \mathbf{v}(t'') \rangle. \quad (165)$$

The third equality arises because we can swap  $t'$  and  $t''$ . The quantity  $\langle \mathbf{v}(t') \cdot \mathbf{v}(t'') \rangle$  is the velocity autocorrelation function. This is an example of a Green-Kubo relation; that is, a relation between a transport coefficient, and an autocorrelation function of a dynamical variable. Eq. 160 then leads to

$$\mathcal{D} = \frac{1}{3} \int_0^\infty \langle \mathbf{v}(t') \cdot \mathbf{v}(t'') \rangle dt \quad (166)$$

So, the second route to computing  $\mathcal{D}$  requires that we numerically integrate  $\langle \mathbf{v}(t') \cdot \mathbf{v}(t'') \rangle$  out to *very* large times. How large? First, let's try to understand the behavior of  $\langle \mathbf{v}(t') \cdot \mathbf{v}(t'') \rangle$ .

In three dimensions, we compute this by computing the components and adding them together, as we did for mean-squared displacement:

$$\langle \mathbf{v}(t') \cdot \mathbf{v}(t'') \rangle = \langle v_x(t')v_x(t'') \rangle + \langle v_y(t')v_y(t'') \rangle + \langle v_z(t')v_z(t'') \rangle \quad (167)$$

The code to compute the VACF is essentially identical to that for the MSD, with the exception that the quantity we accumulate is the dot product of velocity vectors:

```

/* Compute velocity dot product between particles i and j in
   frame fi and fj, respectively; com_corr removes center of
   mass motion */
double vij2 ( frametype * fi, int i, frametype * fj, int j,
              int com_corr ) {
    double dx, dy, dz;
    dx=(fi->vx[i]-(com_corr?fi->cvx:0))*(fj->vx[j]-(com_corr?fj->cvx:0));
    dy=(fi->vy[i]-(com_corr?fi->cvy:0))*(fj->vy[j]-(com_corr?fj->cvy:0));
    dz=(fi->vz[i]-(com_corr?fi->cvz:0))*(fj->vz[j]-(com_corr?fj->cvz:0));
    return dx+dy+dz;
}

```

Fig. 19 shows the VACF for the same simulation we showed for the MSD above. The right panel is a zoom in by a factor of 20, which allows us to resolve the part of the VACF that dips below zero at short times; this is the same time scale on which we have ballistic motion. The negative VACF indicates “bounce-back” from collisions. That figure was generated using `plot_vacf.py`, which also applies Eq. 166 using `scipy.integrate.simpson()` to compute  $\mathcal{D}$ 's:

```

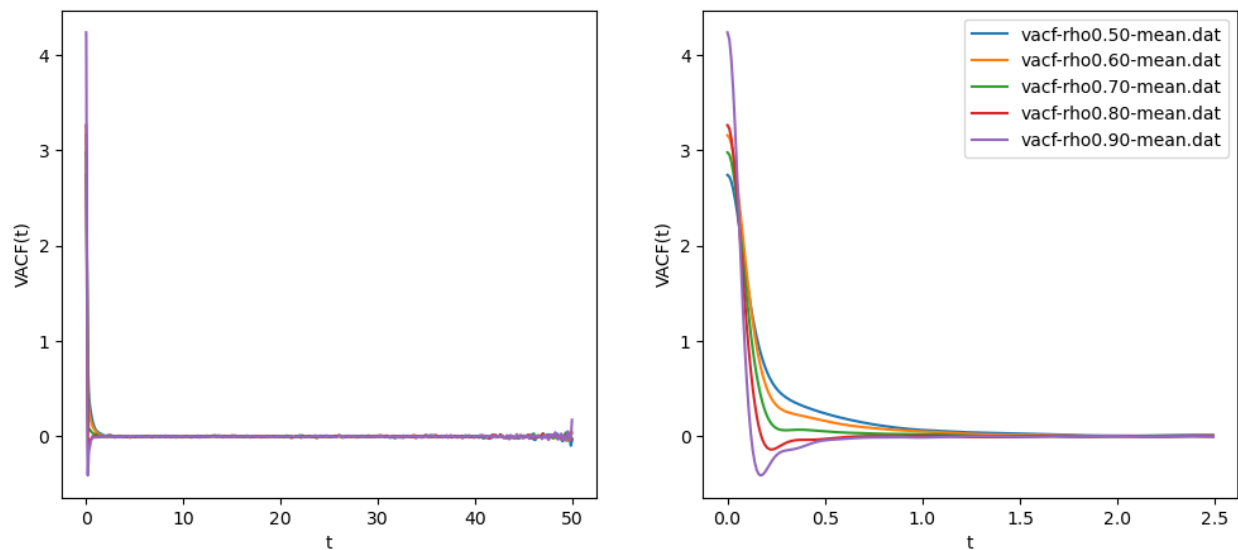
$ python ../../originals/plot_vacf.py -i vacf-rho0.50-mean.dat \
  -i vacf-rho0.60-mean.dat -i vacf-rho0.70-mean.dat \
  -i vacf-rho0.80-mean.dat -i vacf-rho0.90-mean.dat \
  -o vacf-rho-T0.70.png -z 20
vacf-rho0.50-mean.dat 0.162263388888888892
vacf-rho0.60-mean.dat 0.167246988888888884

```

```

vacf-rho0.70-mean.dat 0.11846137777777778
vacf-rho0.80-mean.dat 0.08435750000000004
vacf-rho0.90-mean.dat 0.07054034444444444
  
```

These values agree only weakly with the values computed via fitting to MSD, but either is considered “correct”.



**Figure 19:** Velocity autocorrelation function (VACF) vs. simulation time (in reduced LJ units) for 216-particle, 60,000-step NVE MD simulations at  $\rho = 0.5 - 0.9$ . Right panel is just a close-up of the left panel showing the short-time-scale “bounce-back” behavior.

## 6 Ensembles

### 6.1 Monte Carlo Simulations in the Isothermal-Isobaric and Grand Canonical Ensembles

#### 6.1.1 Isothermal-Isobaric

In this section, we consider how to conduct Monte Carlo simulation in ensembles other than the canonical ensemble. In deriving the partition function for the canonical ensemble (Eq. 46), we imagined our system of constant  $N$ ,  $V$ , and  $T$  in thermal contact with a large reservoir. This thermal contact allowed the system and reservoir to exchange energy such that the system remained at constant  $T$ , and what resulted was the Boltzmann factor. In Section 5.4.1, F&S explain the case when we have the reservoir and the system both thermally and *mechanically* coupled. The mechanical coupling allows the volume of the system to change such that the pressure in the system is the same as the reservoir, which is again considered as an infinite ideal gas. In addition to thermostating our system, the reservoir acts as a *barostat*.

First, for convenience, we express the set of coordinates,  $\mathbf{r}^N$ , scaled by the box length,  $L$ , as  $\mathbf{s}^N$ . The partition function in the NPT ensemble is then

$$Q(N, P, T) = \frac{\beta P}{\Lambda^{3N} N!} \int dV V^N \exp(-\beta PV) \int d\mathbf{s}^N \exp[-\beta \mathcal{U}(\mathbf{s}^N; L)] \quad (168)$$

The free energy associated with this ensemble is the *Gibbs* free energy:

$$G = -k_B T \ln Q(N, P, T) \quad (169)$$

Now, compared to the canonical ensemble, in the NPT ensemble, volume is an additional degree of freedom. We need the probability distribution to include volume:

$$\mathcal{N}(V; \mathbf{s}^N) \propto V^N \exp(-\beta PV) \exp[-\beta \mathcal{U}(\mathbf{s}^N; L)] \quad (170)$$

$$= \exp\{-\beta [\mathcal{U}(\mathbf{s}^N, V) + PV - N\beta^{-1} \ln V]\} \quad (171)$$

We can use this new Boltzmann factor in an acceptance rule for a Monte Carlo trial move involving a simple volume change from  $V$  to  $V + \Delta V$ , where  $\Delta V$  is randomly chosen from  $[-\Delta V_{max}, \Delta V_{max}]$ :

$$\text{acc}(o \rightarrow n) = \min(1, \exp\{-\beta [\mathcal{U}(\mathbf{s}^N, V') - \mathcal{U}(\mathbf{s}^N, V) + P(V - V') - N\beta^{-1} \ln(V'/V)]\}) \quad (172)$$

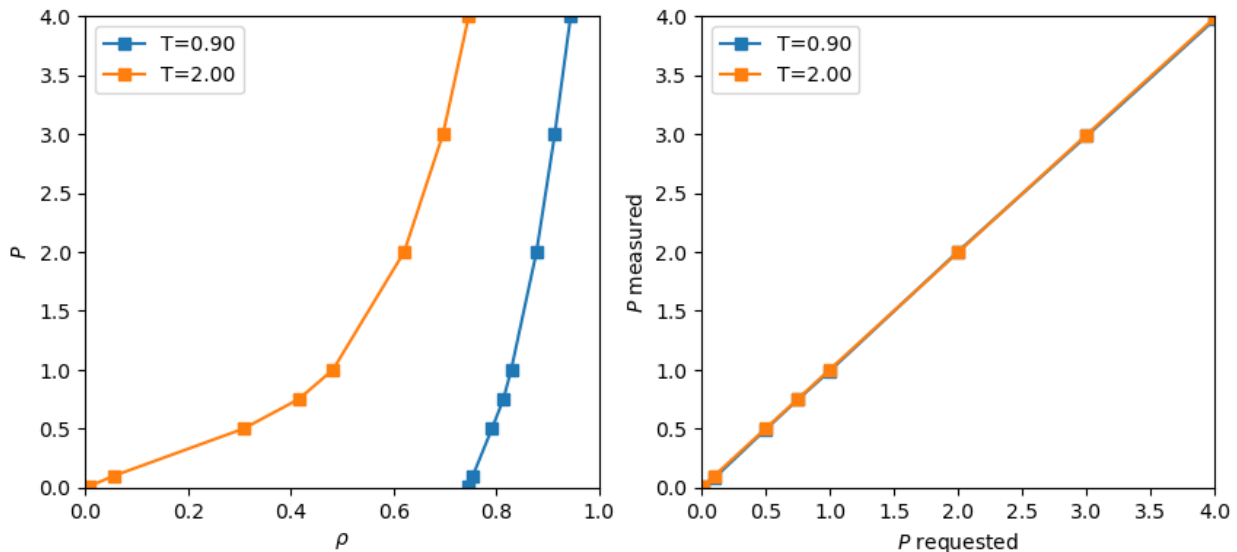
We can also consider trial move that changes the logarithm of the box size from  $\ln V$  to  $\ln V + \Delta(\ln V)$ . In this case, the integral of  $V^N$  over  $dV$  is re-expressed as an integral of  $V^{N+1}$  over  $d \ln V$ , and the acceptance rule is the same as the one above except for a factor of  $(N + 1)$  multiplying  $\beta^{-1}$ , instead of  $N$ .

The C-code `mc1j_npt.c` implements an NPT MC simulation of the Lennard-Jones liquid using both particle displacements and  $\log V$  displacements. For each cycle, there is a  $1/(N + 1)$  probability that a trial move is a volume displacement. The trial move generates a random displacement, computes a new box length, rescales all particle positions, scales the cutoff radius, and recomputes the tail corrections and shift, if applicable. If the Metropolis criterion is not met after a random number is selected, then all of these operations are undone. Otherwise, the new box size with the newly scaled particle positions is kept. The particle displacement algorithm is the same as in `mc1j.c`.

As an exercise, you can use the code to regenerate Figure 5.3 in the text, which is again a slice through the phase diagram of the Lennard-Jones fluid at  $T = 2.0$ . This temperature is above the critical temperature, so we do not anticipate a phase transition at the pressures investigated. However, we saw that when we considered  $T = 0.9$  using the NVT MC simulation, negative pressures were predicted, indicating that the system would have liked to phase separate but couldn't due to its fixed density and

finite size. That is, at the density specified, there might not be enough particles to “nucleate” the denser of the two phases. NPT simulations in principle offer a way around that by allowing the system density to fluctuate.

I ran the code with  $N = 108$  particles for  $10^6$  cycles (Note that I have changed my definition of “cycle”. Before, one “cycle” was  $N$  moves; now it is a single move. This distinction isn’t important for now, but I thought you’d like to be made aware.) The log-volume maximum displacement was set at 0.25, and the maximum particle displacement varied from 0.3 for  $P$ , to 0.5 at the lowest value of  $P$ . You can see from Fig. 20 that the data at  $T = 2.0$  is equally well reproduced here as it was using conventional NVT MC (Fig. 13). However, for  $T = 0.9$ , we notice that the densities which arise are clearly indicate a high-density phase is prevalent. (Indeed, we saw in NVT simulations that forcing a  $T=0.9$  system to exist at densities below about 0.75 resulted in *negative* pressures!) This code also computes the pressure from the virial, and the measured pressure and imposed pressures agreed, as you can see from the right-hand panel in Fig. 20.



**Figure 20:** (Left) Pressure vs. density in a Lennard-Jones fluid at two temperatures computed using NPT MC simulation of systems of  $N = 108$  particles. Each point is the average of three independent simulations, all initialized at a density of 0.5. (Right) Measured pressure vs. requested pressure for all simulations.

For temperatures near the critical temperature, we would expect the fluctuations in density to be maximum. As an exercise, you can modify `mc1j_npt.c` to compute the average fluctuations in  $\rho$ .

### 6.1.2 Grand Canonical

So we see that *volume* exchanges with an ideal gas reservoir can be used to fix the pressure of a test system. Similarly, *particle* exchanges with an ideal gas reservoir can be used to fix the *chemical potential*  $\mu$  of a test system. Chemical potential is defined as the change in free energy with particle number:

$$\mu = \frac{\partial F}{\partial N} \quad (173)$$

Thus, as reciprocal temperature,  $\beta$ , is conjugate to entropy,  $S$ , and pressure,  $P$ , is conjugate to volume,  $V$ , chemical potential,  $\mu$ , is conjugate to number of particles,  $N$ . An ensemble in which  $\mu$ ,  $V$ , and  $T$  are fixed is referred to as the “grand canonical” ensemble.

For an ideal gas, we know that the NVT partition function is given by

$$Q^{i.g.}(N, V, T) = \frac{V^N}{\Lambda^{3N} N!} \quad (174)$$

Because  $F = \beta^{-1} \ln Q$ , it is straightforward to show for the ideal gas that

$$\beta P = \rho \quad (175)$$

and

$$\mu^{i.g.} = k_B T \ln \Lambda^3 \rho = \mu^0 + k_B T \ln \beta P \quad (176)$$

where

$$\mu^0 \equiv k_B T \ln \Lambda^3. \quad (177)$$

The conventional definition of the *excess chemical potential*, or the difference in chemical potential of the material of interest and an ideal gas at the same density, is

$$\mu^{ex} = \mu - \mu^{i.g.} = \mu - \mu^0 - k_B T \ln \beta P \quad (178)$$

To keep things clean, we will specify a *relative* chemical potential defined as

$$\mu' \equiv \mu - \mu^0 \quad (179)$$

giving the definition of the excess as

$$\beta \mu^{ex} \equiv \beta \mu' - \ln \beta P \quad (180)$$

To implement a grand canonical MC simulation, the basic idea is that we allow our system to interact with an ideal gas system at a fixed  $P$  (which is related to a fixed  $\mu$ , as discussed above) by exchanging particles. The appropriate probability density is

$$\mathcal{N}_{\mu VT}(\mathbf{s}^N; N) \propto \frac{\exp(\beta \mu N) V^N}{\Lambda^{3N} N!} \exp[-\beta \mathcal{U}(\mathbf{s}^N)] = \frac{V}{N} \exp[-\beta(\mathcal{U} - \mu' N)] \quad (181)$$

To implement a random walk with this probability distribution, in addition to the normal particle displacement moves, we also have insertion and removal of particles with appropriate acceptance ratios:

$$\text{acc}(N \rightarrow N+1) = \min \left[ 1, \frac{V}{N+1} \exp \left\{ -\beta [\mathcal{U}(N+1) - \mathcal{U}(N) - \mu'] \right\} \right] \quad (182)$$

$$\text{acc}(N \rightarrow N-1) = \min \left[ 1, \frac{N}{V} \exp \left\{ -\beta [\mathcal{U}(N-1) - \mathcal{U}(N) + \mu'] \right\} \right] \quad (183)$$

So, we can specify  $\mu'$  of the ideal gas bath, system volume  $V$  and temperature  $T$ , and conduct a grand canonical MC simulation from which we can observe measure pressure, density, and *excess* chemical potential in our system of interest. The code `mclj_muvt.c` implements grand canonical MC for the Lennard-Jones fluid.

It is instructive to run this code with various values of  $\mu'$ . For example, at  $T = 2.0$  and  $\mu' = -2.0$ , an initially 512-particle system at  $\rho = 0.6$  becomes a 436-particle systems at  $\rho = 0.54$ :

```
$ ./mclj_muvt -N 512 -rho 0.6 -T 2 -mu -2 \
  -disp-wt 0.5 -nc 50000 -dr 0.5 -s 124521 \
  -ne 1000 -rc 3.5 -prog 0
```

```

# muVT MC Simulation of a Lennard-Jones fluid
# L = 9.48505; rho0 = 0.60000; mu' = -2.00000; N0 = 512; rc = 3.50000
# nCycles 50000, nEq 1000, seed 124521, dR 0.50000
NPT Metropolis Monte Carlo Simulation of the Lennard-Jones fluid in t
-----
Number of cycles:                51000
Maximum particle displacement:   0.50000
Displacement weight:            0.50000
Temperature:                     2.00000
Relative chemical potential:     -2.00000
Initial number of particles:     512
Tail corrections used?           Yes
Shifted potentials used?         No
Results:
Final number of particles:       408
Displacement attempts:          26058
Insertion attempts:             12384
Deletion attempts:              12558
Acceptance ratio, ptcl displ:   0.47981
Acceptance ratio, insertion:    0.08648
Acceptance ratio, deletion:     0.09357
Overall acceptance ratio:       0.28920
Energy/particle:                -3.31911
Density:                        0.50063
Computed pressure:              0.96980
Excess chemical potential:      -0.61622
Program ends.

```

Why does the number of particles go down? The system is being asked to find an equilibrium in which the chemical potential is negative, yet we are apparently starting it at a state where it is more positive, so the system sheds particles. That is, our initial density corresponds to a system of higher chemical potential than what we are asking for. Conversely, if initialize at a lower density, say 0.4, then we see the number of particles increases:

```

$ ./mclj_muvt -N 512 -rho 0.4 -T 2 -mu -2 \
  -disp-wt 0.5 -nc 50000 -dr 0.5 -s 124521
  -ne 1000 -rc 3.5 -prog 0
# muVT MC Simulation of a Lennard-Jones fluid
# L = 10.85767; rho0 = 0.40000; mu' = -2.00000; N0 = 512; rc = 3.50000
# nCycles 50000, nEq 1000, seed 124521, dR 0.50000
NPT Metropolis Monte Carlo Simulation of the Lennard-Jones fluid in t
-----
Number of cycles:                51000
Maximum particle displacement:   0.50000
Displacement weight:            0.50000
Temperature:                     2.00000
Relative chemical potential:     -2.00000
Initial number of particles:     512
Tail corrections used?           Yes

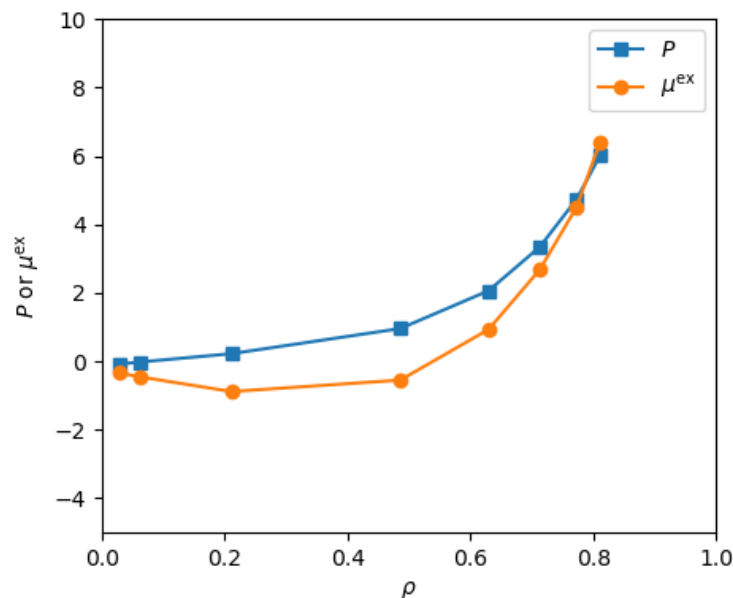
```



```

Shifted potentials used?      No
Results:
Final number of particles:    598
Displacement attempts:       26058
Insertion attempts:          12384
Deletion attempts:           12558
Acceptance ratio, ptcl displ: 0.54885
Acceptance ratio, insertion:  0.12766
Acceptance ratio, deletion:   0.11905
Overall acceptance ratio:     0.34075
Energy/particle:              -2.62249
Density:                       0.44318
Computed pressure:             0.88387
Excess chemical potential:    -0.37246
Program ends.
  
```

Note that while these two runs purportedly aim for the same equilibrium state, they don't converge there. The second run converges to a lower pressure and more positive excess chemical potential than the first run does. This is partially due to the fact that they are different sizes so there are random errors, but it is also due to the fact that grand canonical MC simulations take a relatively long time to reach equilibrium compared to NVT MC. In practice, it can take many hundreds of thousands of cycles to generate reproducible measurements in  $\mu$ VT MC. Fig. 21 shows isotherms at  $T = 2.0$  of pressure and excess chemical potential for the LJ fluid computed using `mclg_muvt.c`. These were computed using three independent trials per data point. This matches Fig. 5.8 in F&S [1].



**Figure 21:** Isotherms of pressure and excess chemical potential at  $T = 2.0$  from  $\mu$ VT MC simulation of the Lennard-Jones fluid. Each point is the average of three independent, 600,000-move simulations.

## 6.2 Molecular Dynamics at Constant Temperature

Conventional MD simulation conserves total energy; hence, the time averages computed from MD simulation, if it is long enough, are equivalent to ensemble averages computed from the microcanonical ensemble. The flexibility of MD is greatly enhanced by noting that it is not restricted to NVE. There exist techniques by which MD can simulate in the NVT or NPT ensembles as well. We will consider some popular temperature control schemes, and one popular pressure control scheme, in this section.

There are essentially three ways to control the temperature in an MD simulation:

1. Velocity rescaling;
2. Stochastic forces and/or velocities; and
3. “Extended Lagrangian” formalisms.

Each of these classes of schemes has advantages and disadvantages, depending on the application. In the following subsections, we consider several examples of thermostats, and attempt to discuss their advantages and drawbacks. A simple barostat is also described in the last section.

### 6.2.1 Temperature Fluctuations in the Canonical Ensemble

Before considering how to fiddle with particle velocities or forces to enforce constant temperature, it is worth considering what statistical thermodynamics has to say about temperature. When we have direct knowledge of instantaneous particle velocities, we know that the kinetic energy is

$$\mathcal{K} = \sum_{i=1}^N \sum_{\alpha \in \{x,y,z\}} \frac{p_{i,\alpha}^2}{2m} \equiv \sum_{j=1}^{3N} \frac{p_j^2}{2m} \quad (184)$$

where we recognize that all momentum components are independent variables. We also know that temperature (in reduced units) is directly proportional to kinetic energy:

$$\frac{3}{2}NT = \mathcal{K} \quad (185)$$

With this equivalence, we can consider the “instantaneous” temperature as

$$T = \frac{2}{3N} \sum_{j=1}^{3N} \frac{p_j^2}{2m} \quad (186)$$

Since momenta must fluctuate it is necessarily the case that instantaneous temperature *also* fluctuates. Let’s see how much.

First, since all momenta are independent, it follows from the definition of the canonical partition function that a particle momentum component  $p_j$  follows the Maxwell-Boltzmann distribution:

$$\rho(p_j) = \left( \frac{\beta}{2\pi m} \right)^{\frac{3}{2}} \exp \left( -\frac{\beta p_j^2}{2m} \right) \quad (187)$$

We can characterize fluctuations in  $p_j^2$  by dividing its variance  $\sigma_{p_j^2}$  to the square of its average  $\langle p_j^2 \rangle^2$ . The variance is defined

$$\sigma_{p_j^2} = \langle p_j^4 \rangle - \langle p_j^2 \rangle^2 \quad (188)$$

Using the Maxwell-Boltzmann distribution:

$$\langle p_j^2 \rangle = \int_{-\infty}^{\infty} dp_j p_j^2 \exp \left( -\frac{\beta p_j^2}{2m} \right) = \frac{3m}{\beta} \quad (189)$$

and

$$\langle p_j^4 \rangle = \int_{-\infty}^{\infty} dp_j p_j^4 \exp\left(-\frac{\beta p_j^2}{2m}\right) = 15 \left(\frac{m}{\beta}\right)^2 \quad (190)$$

Thus,

$$\frac{\langle p_j^4 \rangle - \langle p_j^2 \rangle^2}{\langle p_j^2 \rangle} = \frac{2}{3}. \quad (191)$$

Now, let's compute fluctuations in the instantaneous temperature. First, the average:

$$\langle T \rangle = \frac{2}{3N} \sum_{j=1}^{3N} \frac{\langle p_j^2 \rangle}{2m} = \frac{2}{3N} \frac{3N}{2m} \langle p_j^2 \rangle = \frac{\langle p_j^2 \rangle}{m} \quad (192)$$

Now the variance,  $\langle T^2 \rangle - \langle T \rangle^2$ , starting with  $\langle T^2 \rangle$ :

$$\langle T^2 \rangle = \left(\frac{2}{3N}\right)^2 \left\langle \left(\sum_{j=1}^{3N} \frac{p_j^2}{2m}\right) \left(\sum_{k=1}^{3N} \frac{p_k^2}{2m}\right) \right\rangle \quad (193)$$

$$= \left(\frac{2}{3N}\right)^2 \sum_{jk} \frac{\langle p_j^2 p_k^2 \rangle}{(2m)^2} \quad (194)$$

$$= \left(\frac{2}{3N}\right)^2 \left[ 9N \frac{\langle p_j^4 \rangle}{(2m)^2} + 9N(N-1) \frac{\langle p_j^2 \rangle^2}{(2m)^2} \right] \quad (195)$$

$$= \frac{1}{(mN)^2} [N \langle p_j^4 \rangle + N(N-1) \langle p_j^2 \rangle^2] \quad (196)$$

Note that in going from Eq. 195 to 196, we note the fact that

$$\langle p_j^2 p_k^2 \rangle = \langle p_j^2 \rangle \langle p_k^2 \rangle = \langle p_j^2 \rangle^2 \quad (197)$$

since momenta are not correlated to each other. Putting these together:

$$\frac{\langle T^2 \rangle - \langle T \rangle^2}{\langle T \rangle^2} = \frac{\frac{1}{(mN)^2} [N \langle p_j^4 \rangle + N(N-1) \langle p_j^2 \rangle^2] - \left(\frac{\langle p_j^2 \rangle}{m}\right)^2}{\left(\frac{\langle p_j^2 \rangle}{m}\right)^2} \quad (198)$$

$$= \frac{N \langle p_j^4 \rangle + N(N-1) \langle p_j^2 \rangle^2 - N^2 \langle p_j^2 \rangle^2}{N^2 \langle p_j^2 \rangle^2} \quad (199)$$

$$= \frac{1}{N} \frac{\langle p_j^4 \rangle - \langle p_j^2 \rangle^2}{\langle p_j^2 \rangle^2} = \frac{2}{3N} \quad (200)$$

So clearly temperature fluctuates in the canonical ensemble. Of course, in the thermodynamic limit ( $N \rightarrow \infty$ ), these fluctuations vanish and we perceive a “constant” temperature, but in a simulation in which we resolve the momenta of a set of  $N$  particles, we *must* observe that  $T$  fluctuates as shown above. We can use this fact to decide whether or not a temperature-control scheme in MD is actually resulting in sampling the canonical ensemble.

### 6.2.2 Velocity Rescaling: Isokinetics and the Berendsen Thermostat

“Isokinetics” refers to altering velocities on the fly to keep kinetic energy (and therefore temperature) constant. The relationship between kinetic energy and temperature results from the application of the equipartition theorem to velocity (or, equivalently, momentum) degrees of freedom:

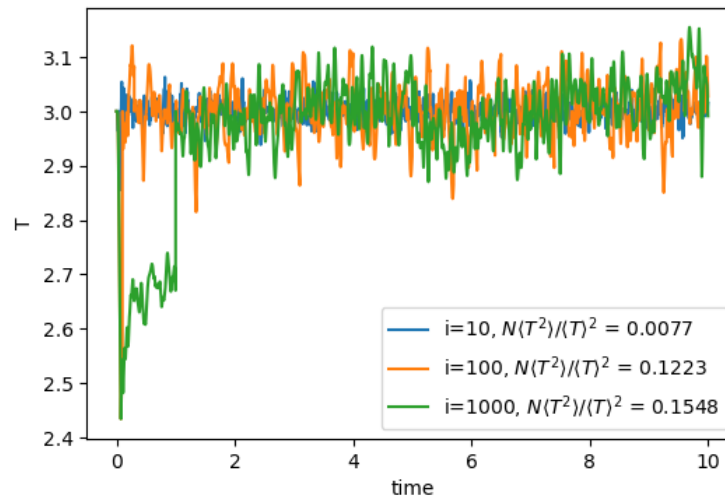
$$\frac{3}{2}Nk_B T = \frac{1}{2} \sum_i m_i v_i^2 = \sum_i \frac{p_i^2}{2m} \quad (201)$$

Scaling every particle velocity by a factor  $\lambda$  will yield a new temperature  $T'$ :

$$\lambda = \sqrt{(T'/T)} \quad (202)$$

An isokinetic thermostat computes  $\lambda$  and rescales velocities at every time step. Such a thermostat **cannot** be used to conduct a simulation in the canonical ensemble, since it totally suppresses the required temperature fluctuations. However, isokinetics is perfectly fine to use in a warmup or initialization phase in order to prevent numerical instabilities.

The code `mdlj_isok.c` illustrates implementation of an isokinetic thermostat for constant- $T$  simulation of a Lennard-Jones fluid. The implementation uses a two parameters, `isoKT` and `isoKi`, that specify the desired temperature and the number of time steps between applications of the rescaling. It is worth asking whether or not *occasional* velocity rescaling (rather than at every step) might allow us to preserve the correct statistics. Fig. 22 shows traces of temperature vs time for three MD simulations of the Lennard-Jones fluid with 512 particles at a density of 0.50. Each simulation used a different interval size  $i$ . The quantity  $N \frac{\langle T^2 \rangle - \langle T \rangle^2}{\langle T \rangle^2}$  is computed for each set and values are shown in the legend. For pure canonical statistics, we know this should be  $2/3$ ; clearly, isokinetics even at a very modest frequency utterly fails to preserve canonical statistics.



**Figure 22:** Temperature vs time (output every time step) for three isokinetic MD simulations of the LJ fluid at density 0.5 with 512 particles.  $i$  indicates the number of steps between velocity rescalings; the setpoint temperature is 3.0, and all simulations were initialized with velocities consistent with a temperature of 2.0.

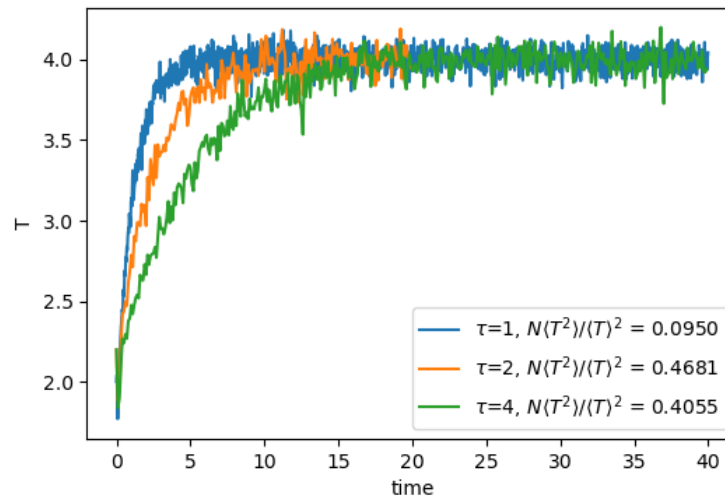
Berendsen realized that velocity scaling could be reformulated to model energy exchange with a

bath at constant  $T$  [9]. His scale factor is defined as

$$\lambda = \left[ 1 + \frac{\Delta t}{\tau_T} \left( \frac{T_0}{T} - 1 \right) \right]^{\frac{1}{2}} \quad (203)$$

Here,  $T_0$  is the setpoint temperature,  $\Delta t$  is the integration time step, and  $\tau_T$  is a constant called the “rise time” of the thermostat. It describes the strength of the coupling of the system to a hypothetical heat bath. The larger  $\tau_T$ , the weaker the coupling; in other words, the larger  $\tau_T$ , the longer it takes to achieve a given  $T_0$  after an instantaneous change from some previous  $T_0$ .

The code `mdlj_ber.c` implements the Berendsen thermostat. The two relevant parameters are `berT`, the setpoint temperature, and `ber_tau`, the rise time. Fig. 23 shows traces of temperature vs time for three MD simulations of the Lennard-Jones fluid with 512 particles at a density of 0.50, with temperature controlled using the Berendsen thermostat with various values of  $\tau$ . Larger  $\tau$  clearly results in longer approach times to the setpoint temperature. Note also that the relative fluctuations of the temperature reported indicate that canonical statistics are not being held.



**Figure 23:** Temperature vs time (output every time step) for three Berendsen-thermostatted MD simulations of the LJ fluid at density 0.5 with 512 particles.  $\tau$  indicates the rise time; the setpoint temperature is 3.0, and all simulations were initialized with velocities consistent with a temperature of 2.0.

Though relatively simple, velocity scaling thermostats are not recommended for use in production MD runs because they do not strictly conform to the canonical ensemble.

### 6.2.3 Stochastic NVT Thermostats: Andersen, Langevin, and Dissipative Particle Dynamics

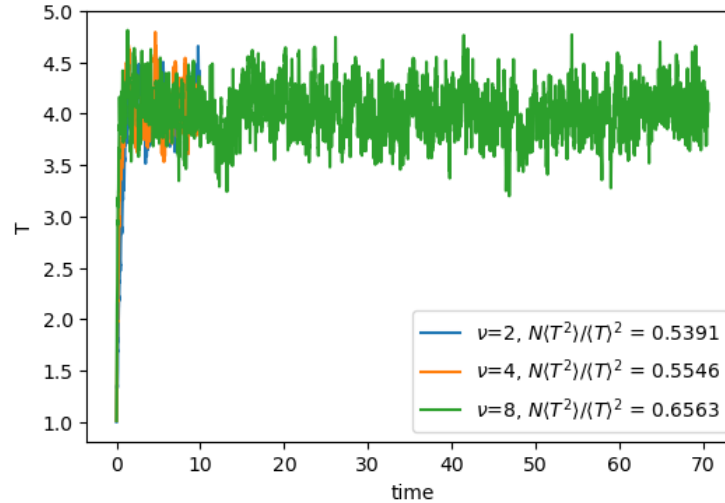
**The Andersen Scheme.** Perhaps the simplest thermostat which does correctly sample the NVT ensemble is due to Andersen [10]. Here, at each step, some prescribed number of particles is selected, and their momenta (actually, their velocities) are drawn from a Gaussian distribution at the prescribed temperature (otherwise known as the Maxwell-Boltzmann distribution):

$$\mathcal{P}(p) = \left( \frac{\beta}{2\pi m} \right)^{3/2} \exp \left[ -\beta p^2 / (2m) \right] \quad (204)$$

This is intended to mimic collisions with bath particles at a specified  $T$ . The strength of the coupling to the heat bath is specified by a collision frequency,  $\nu$ . For each particle, a random variate is selected

between 0 and 1. If this variate is less than  $\nu\Delta t$ , then that particle's momenta are reset.

The code `mdlj_and.c` implements the Andersen thermostat for the Lennard-Jones fluid. The two relevant parameters are `and_T`, the setpoint temperature, and `and_nu`, the rise time. Fig. 24 shows traces of temperature vs time for three MD simulations of the Lennard-Jones fluid with 512 particles at a density of 0.50, with temperature controlled using the Andersen thermostat with various values of collision frequency  $\nu$ . Larger  $\nu$  results in longer approach times to the setpoint temperature, but it is also clear that for these values of  $\nu$ , the Andersen thermostat acts much more quickly than the Berendsen thermostat. Note also that the relative fluctuations of the temperature reported indicate that canonical statistics are in fact being held.



**Figure 24:** Temperature vs time (output every time step) for three Berendsen-thermostatted MD simulations of the LJ fluid at density 0.5 with 512 particles.  $\tau$  indicates the rise time; the setpoint temperature is 3.0, and all simulations were initialized with velocities consistent with a temperature of 2.0.

Although temperature fluctuations match the canonical ensemble, the Andersen thermostat destroys momentum transport because of the random reassignment of velocities; hence, there is no continuity of momentum in an Andersen LJ fluid, and therefore no proper  $\mathcal{D}$  or viscosity. Fig. 6.3 in Frenkel & Smit clearly shows that  $\mathcal{D}$ , if measured from an Andersen MD run, is incorrect.

**The Langevin thermostat.** In the “Langevin” thermostat, at each time step all particles receive a random force and have their velocities lowered using a constant friction. [11] The average magnitude of the random forces and the friction are related in a particular way, which guarantees that the “fluctuation-dissipation” theorem is obeyed, thereby guaranteeing NVT statistics.

In this formalism, the particle- $i$  equation of motion is modified:

$$m\ddot{\mathbf{x}}_i = -\nabla_i U - m\Gamma\dot{\mathbf{r}}_i + \mathbf{W}_i(t) \quad (205)$$

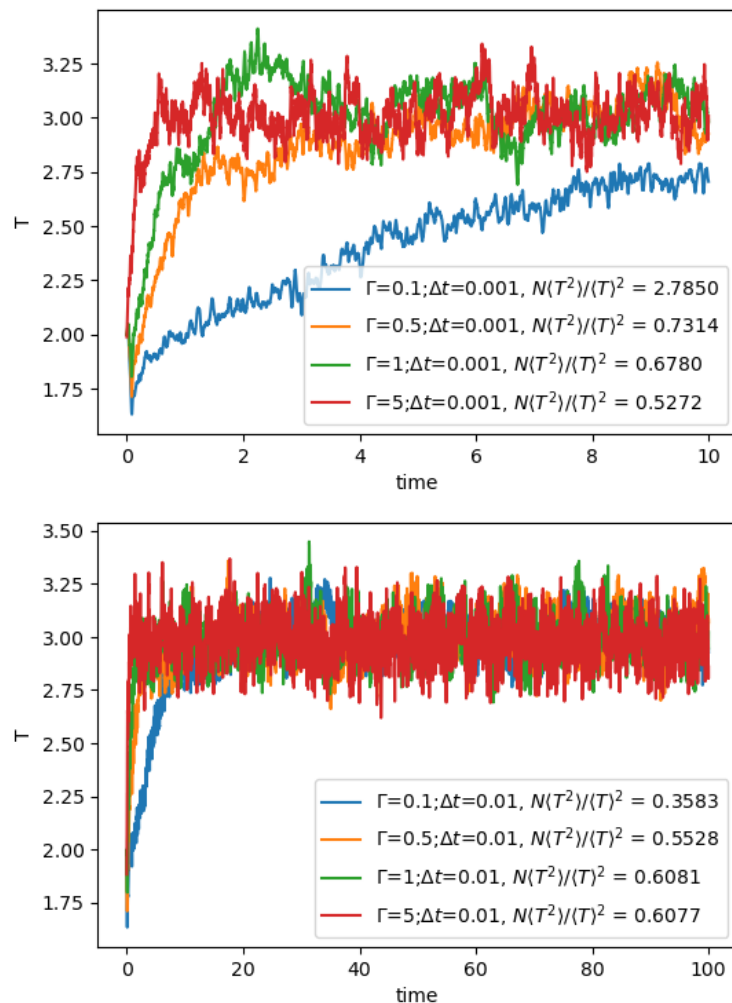
Here,  $\Gamma$  is a friction coefficient with units of  $\tau^{-1}$ , and  $\mathbf{W}_i$  is a random force that is uncorrelated in time and across particles, with a mean given by

$$\langle \mathbf{W}_i(t), \mathbf{W}_j(t') \rangle = \delta_{ij}\delta(t-t') 6k_B m T \Gamma \quad (206)$$

The code `mdlj_lan.c` implements the Langevin thermostat. The two relevant parameters are `lanT`, the setpoint temperature, and `lan_friction`, the friction  $\Gamma$ . The two major elements are a force initialization at each time step that adds in the random forces,  $\mathbf{W}$ , and a slight modification to the

update equations in the integrator to include the effect of  $\Gamma$ . Note that the initialization of forces to zero in the force routine has been removed.

Fig. 25 shows temperature vs time for several MD simulations of a 512-particle LJ fluid at a density of 0.5; the upper plot shows data from runs with  $\Delta t=10^{-3}$ , and the lower plot  $\Delta t=10^{-2}$ , each showing four values of  $\Gamma$ . Relative temperature fluctuations indicate weak agreement with canonical statistics that improves for the lower values of  $\Delta t$ .



**Figure 25:** Temperature vs time (output every time step) for eight Langevin-thermostatted MD simulations of the LJ fluid at density 0.5 with 512 particles.  $\Gamma$  indicates the friction; the setpoint temperature is 3.0, and all simulations were initialized with velocities consistent with a temperature of 2.0. Upper plot shows  $\Delta t$  of 0.001, lower 0.01.

One advantage of the Langevin thermostat (and to a limited extent, the Andersen thermostat and other stochastic-based thermostats) is that we can get away with a larger time step than in NVE simulations. At a density of  $\rho = 0.8442$  and a mean temperature  $T = 1.0$ , an NVE simulation is unstable for time-steps above about  $\Delta t = 0.004$ . We can, however, run a Langevin dynamics simulation with a friction  $\Gamma = 1.0$  stably with a time-step as large as  $\Delta t = 0.01$  or even higher. This has proven invaluable in simulations of more complicated systems than simple liquids, namely linear polymers, which have very long relaxation times. MD with the Langevin thermostat is the method of choice for equilibrating

samples of liquids of long bead-spring polymer chains.

Of course, the drawback of most stochastic thermostats (one exception is discussed next) is that momentum transfer is destroyed. So again, it is unadvisable to use Langevin or Andersen thermostats for runs in which you wish to compute diffusion coefficients. The recommendation stands: use NVE to compute properties, and use thermostats for equilibration only.

**The Dissipative Particle Dynamics thermostat.** The DPD thermostat [12, 13] adds pairwise random and dissipative forces to all particles, and has been shown to preserve momentum transport. Hence, it is the only stochastic thermostat so far that should even be considered for use if one wishes to compute transport properties.

The DPD thermostat is implemented by slight modification of the force routine to add in the pairwise random and dissipative forces. For the  $ij$  pair, the dissipative force is defined as

$$\mathbf{f}_{ij}^D = -\gamma\omega^D(r_{ij})(\mathbf{v}_{ij} \cdot \hat{\mathbf{r}}_{ij})\hat{\mathbf{r}}_{ij} \quad (207)$$

Here,  $\gamma$  is a friction coefficient,  $\omega$  is a cut-off function for the force as a function of the scalar distance between  $i$  and  $j$  which simply limits the interaction range of the dissipative (and random) forces,  $\mathbf{v}_{ij} = \mathbf{v}_i - \mathbf{v}_j$  is the relative velocity of  $i$  to  $j$ , and  $\hat{\mathbf{r}}_{ij} = \mathbf{r}_{ij}/r_{ij}$  is the unit vector pointing from  $j$  to  $i$ . The random force is defined as

$$\mathbf{f}_{ij}^R = \sigma\omega^R(r_{ij})\zeta_{ij}\hat{\mathbf{r}}_{ij} \quad (208)$$

Here,  $\sigma$  is the strength of the random force,  $\omega^R$  is a cut-off, and  $\zeta_{ij}$  is a Gaussian random number with zero mean and unit variance, and  $\zeta_{ij} = \zeta_{ji}$ .

The update of velocity uses these new forces:

$$\mathbf{v}_i(t + \Delta t) = \mathbf{v}_i(t) - \frac{\Delta t}{m}\nabla_i U + \frac{\Delta t}{m}\mathbf{f}_i^D + \frac{\sqrt{\Delta t}}{m}\mathbf{f}_i^R \quad (209)$$

where

$$\mathbf{f}_i^D = \sum_{j \neq i} \mathbf{f}_{ij}^D \quad (210)$$

$$\mathbf{f}_i^R = \sum_{j \neq i} \mathbf{f}_{ij}^R \quad (211)$$

The parameters  $\gamma$  and  $\sigma$  are linked by a fluctuation-dissipation theorem:

$$\sigma^2 = 2\gamma k_B T \quad (212)$$

So, in practice, one must specify either  $\gamma$  or  $\sigma$ , and then a setpoint temperature,  $T$ , in order to use the DPD thermostat.

The cutoff functions are also related:

$$\omega^D(r_{ij}) = [\omega^R(r_{ij})]^2 \quad (213)$$

This is the only real constraint on the cutoffs; we are otherwise allowed to use any cutoff we like. The simplest uses the cutoff radius of the pair potential,  $r_c$ :

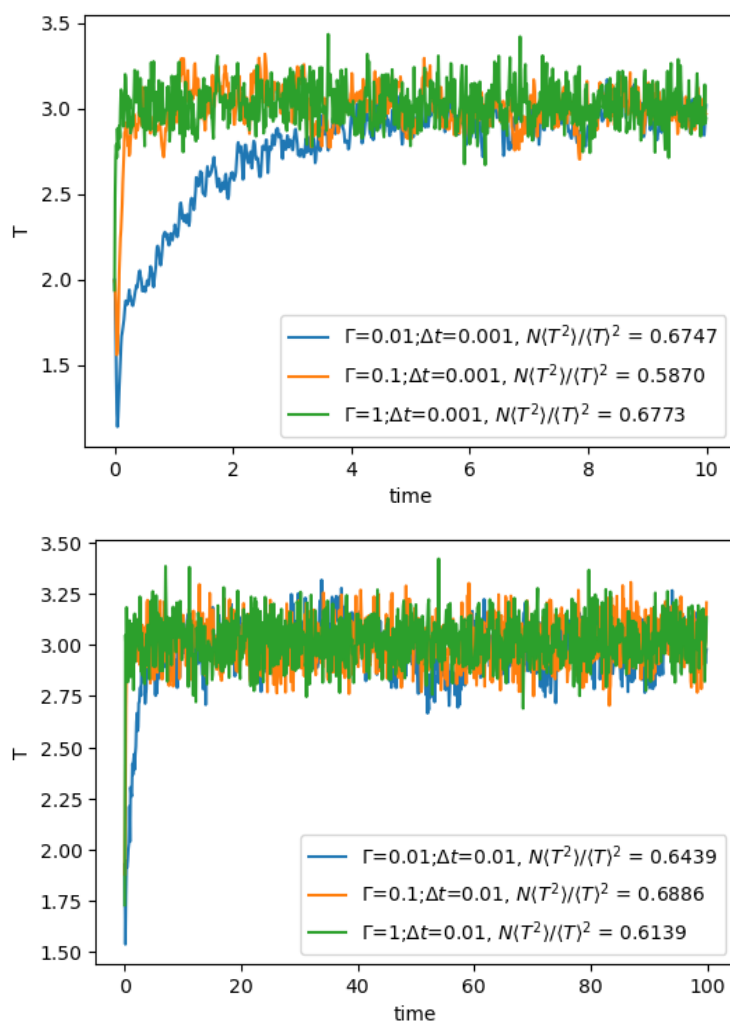
$$\omega(r) = \begin{cases} 1 & r < r_c \\ 0 & r > r_c \end{cases} \quad (214)$$



Note that, with this choice,  $[\omega^R(r_{ij})]^2 = \omega^R(r_{ij}) = \omega^D(r_{ij}) = \omega$ .

The code `mdlj_dpd.c` implements the DPD thermostat in an MD simulation of the Lennard-Jones liquid. The major changes (compared to `mdlj.c`) are to the force routine, which now requires several more arguments, including particle velocities, and parameters for the thermostat. Inside the pair loop, the force on each particle is updated by the conservative, dissipative, and random pairwise force components. The random force is divided by  $\sqrt{\Delta t}$  so that the velocity Verlet algorithm need not be altered to implement Eq. 209.

The behavior of the DPD thermostat can be assessed in a similar fashion as was the Berendsen thermostat above. Here I've run several MD simulations of the LJ fluid at a density of 0.84 with 512 particles for 10,000 steps, with various values of  $\Gamma$  and  $\Delta t$ . Fig. 26 shows the temperature vs time for these various runs. We see that increased friction leads to faster approach to the setpoint temperature, and that temperature fluctuations seem to conform to canonical statistics pretty well.



**Figure 26:** Temperature vs time (output every time step) for eight DPD-thermostatted MD simulations of the LJ fluid at density 0.84 with 512 particles.  $\Gamma$  indicates the friction; the setpoint temperature is 3.0, and all simulations were initialized with velocities consistent with a temperature of 2.0. Upper plot shows  $\Delta t$  of 0.001, lower 0.01.

### 6.2.4 The Nosé-Hoover Chain

The final thermostat we consider is one based on the extended Lagrangian formalism, which leads to a deterministic trajectory; i.e., there are no random forces or velocities to deal with. The most common and so far most reliable thermostat of this kind is the Nosé-Hoover thermostat. This thermostat can be implemented as a “single” or a “chain”; here, we consider a chain.

The basic idea of the Nosé-Hoover thermostat is to use a friction factor to control particle velocities. This friction factor is actually the scaled velocity,  $v_{\xi_1}$ , of an additional and dimensionless degree of freedom,  $\xi_1$ . This degree of freedom has an associated “mass”,  $Q_1$ , which effectively determines the strength of the thermostat. The equations of motion obeyed by this additional degree of freedom guarantee that the *original* degrees of freedom ( $\mathbf{r}^N, \mathbf{p}^N$ ) sample a canonical ensemble. This degree of freedom is the terminus of a chain of similar degrees of freedom, each with their own mass. The chain has a total of  $M$  “links.” The overall set of equations of motion are:

$$\dot{\mathbf{r}}_i = \frac{\dot{\mathbf{P}}_i}{m_i} \quad (215)$$

$$\dot{\mathbf{P}}_i = \mathbf{F}_i - \frac{p_{\xi_1}}{Q_1} \mathbf{p}_i \quad (216)$$

$$\dot{\xi}_k = \frac{p_{\xi_k}}{Q_k} \quad k = 1, \dots, M \quad (217)$$

$$p_{\xi_1} = \left( \sum_i \frac{p_i^2}{m_i} - Lk_B T \right) - \frac{p_{\xi_2}}{Q_2} p_{\xi_1} \quad (218)$$

$$p_{\xi_k} = \left( \frac{p_{\xi_{k-1}}^2}{Q_{k-1}} - k_B T \right) - \frac{p_{\xi_{k+1}}}{Q_{k+1}} p_{\xi_k} \quad (219)$$

$$p_{\xi_k} = \left( \frac{p_{\xi_{M-1}}^2}{Q_{M-1}} - k_B T \right) \quad (220)$$

The main advantage of the Nosé-Hoover chain thermostat is that the dynamics of all degrees of freedom are deterministic and time-reversible. No random numbers are used. The code `mdlj_nhc.c` implements an  $M = 2$  Nosé-Hoover chain thermostat in an MD simulation of an Lennard-Jones fluid, by implementing Algorithms 30, 31, and 32 from Frenkel & Smit. The relevant parameters are `nhcT`, the setpoint temperature, and `nhcQ`, the two masses. Fig. 27 illustrates the use of the NHC thermostat on an  $N=512$ ,  $\rho = 0.84$  LJ system.

### 6.3 Molecular Dynamics at Constant Pressure: The Berendsen Barostat

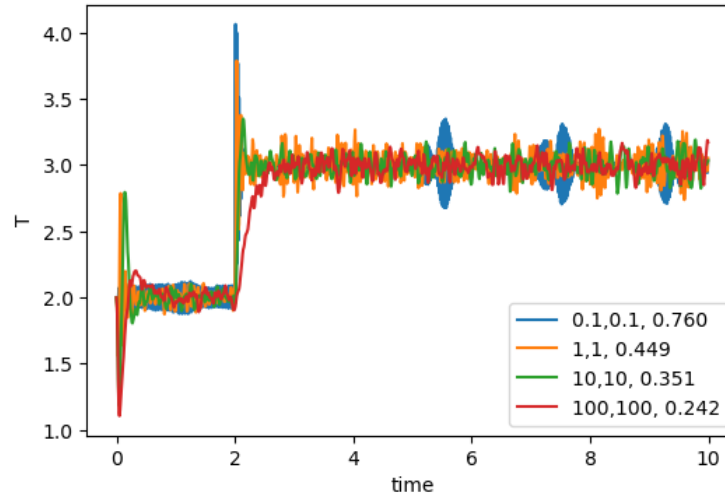
As with temperature control, there are different classes of pressure control for MD simulation. The only one we consider here is the length-scaling technique of Berendsen. It should be noted that one can also use the extended Nosé-Hoover (extended Lagrangian) formalism of Martyna, which is mentioned in F&S; in the interest of time, we will forego a discussion of this technique.

Here we consider implementation of the Berendsen barostat [9]. Recall that the working definition of instantaneous pressure,  $P$ , is given by:

$$P = \rho T + \text{vir}/V \quad (221)$$

where `vir` is the virial:

$$\text{vir} = \frac{1}{3} \sum_{i>j} \mathbf{f}(r_{ij}) \cdot \mathbf{r}_{ij} \quad (222)$$



**Figure 27:** Temperature vs time (output every 10 time steps) for four MD simulations of the LJ fluid at density 0.84 with 512 particles with initial velocities assigned to give an initial temperature of 2.0. A 2-mass Nosé-Hoover chain with masses indicated in the legend is used to maintain the temperature at 2 until  $t = 2$ , at which time the setpoint temperature is instantaneously raised to 3. The third number in the legend label is the product of the number of particles and the relative fluctuation in instantaneous temperature measured for the second half of each respective simulation, which in the canonical ensemble should be  $2/3$ .

and  $V$  is the system volume.  $\mathbf{f}(r_{ij})$  is the *force* exerted on particle  $i$  by particle  $j$ .

Consider a cubic system, where  $V = L^3$ . The Berendsen barostat uses a scale factor,  $\mu$ , which is a function of  $P$ , to scale lengths in the system:

$$\mathbf{r}_i \rightarrow \mu \mathbf{r}_i \quad (223)$$

$$L \rightarrow \mu L \quad (224)$$

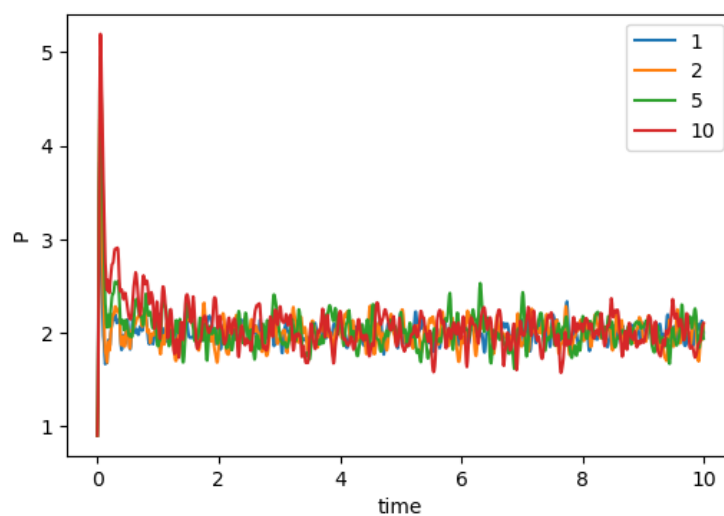
$\mu$  is given by

$$\mu = \left[ 1 - \frac{\Delta t}{\tau_P} (P_0 - P) \right]^{1/3} \quad (225)$$

Here,  $\Delta t$  is the integrator time-step,  $\tau_P$  is the “rise time” of the barostat, and  $P_0$  is the setpoint pressure. Berendsen discusses the tensor-based analog for non-cubic systems [9].

The code `mdlj_berp.c` implements the Berendsen barostat. The relevant parameters are `berP`, the setpoint pressure, and `ber_tau`, the rise time. Fig. 28 shows pressure vs time for four MD simulations of 512 particles with a setpoint pressure of 2.

Length scaling at each time step using a global scale factor, while effective in this instance, can lead to violent oscillations of pressure in more ordered systems, and is therefore not recommended for production MD runs. However, it is common to find length scaling barostats used in the literature without reporting how effective they are, measured at least in terms of pressure and its fluctuations. But they can be useful for pre-equilibrating samples at some  $P$  prior to beginning an NVE simulation during which one hopes the instantaneous pressure fluctuates about the previous setpoint. It is easy to implement both the Berendsen thermostat and barostat in the same simulation program, to allow pre-equilibration at setpoint  $T$  and  $P$ .



**Figure 28:** Pressure vs time (output every time step) for three Berendsen-barostatted MD simulations of the LJ fluid at initial density 0.84 with 512 particles. The rise time  $\tau$  is indicated for each system in the legend. The setpoint pressure is 2.0, and all simulations were initialized with velocities consistent with a temperature of 2.0.

## 7 Long-Range Interactions: The Ewald Summation

So far, we have considered interparticle interactions that are *short-ranged* by construction. Because the Lennard-Jones potential decays so strongly with distance (as  $r^{-6}$ ), it is acceptable to cut off this interaction at moderate distances and, if desired, add a correction factor which is the result of integrating the potential over a uniform particle density out to  $r = \infty$ . However, Coulomb interactions, common in molecular simulation, decay relatively much more slowly (as  $r^{-1}$ ) and as a consequence, we cannot compute a correction factor; the integral diverges. There are several ways to handle long-ranged interactions, but the most popular is the Ewald summation [14], which we discuss here. This discussion is drawn primarily from F&S chapter 12 [1], and the excellent paper by Markus Deserno and Christian Holm [15, 16].

### 7.1 The Ewald Coulombic energy

First, assume we have a collection of charged particles in a cubic box with side length  $L$ , with periodic boundary conditions. The collection is assumed neutral; there is an equal number of positive and negative charges. The total Coulombic energy in this system is given by:

$$\mathcal{U}_{\text{Coul}} = \frac{1}{2} \sum_{i=1}^N q_i \phi(r_i) \quad (226)$$

$\phi(r_i)$  is the electrostatic potential at position  $r_i$ :

$$\phi(r_i) = \sum_{j=1}^{N'} \sum_{\mathbf{n} \in \mathbb{Z}^3} \frac{q_j}{|\mathbf{r}_{ij} + \mathbf{n}L|} \quad (227)$$

$\mathbf{n}$  is a three dimensional integer vector. The prime on the first summation indicates that we do not admit the term for which  $j = i$  if  $\mathbf{n} = (0, 0, 0)$ . That is, we allow each particle to interact with its periodic images, but not with itself.

To evaluate  $\mathcal{U}$  efficiently, we break it into two parts:

- A short-ranged potential treated with a simple cutoff;
- A long-ranged potential which is periodic and slowly varying, which can therefore be represented to an acceptable level of accuracy by a finite Fourier series.

How can we do this? The idea of Ewald is to do two things: first, screen each point charge using a diffuse cloud of opposite charge around each point charge, and then compensate for these screening charges using a smoothly varying, periodic charge density. The screening charge is constructed to make the electrostatic potential due to a charge at position  $r_j$  decay rapidly to near zero at a prescribed distance. These interactions are treated in real space. The compensating charge density, which is the sum of all screening densities except with opposite charges, is treated using a Fourier series.

The standard choice for a screening potential is Gaussian:

$$\rho_s(r) = -q_i(\alpha/\pi)^{3/2} e^{-\alpha r^2} \quad (228)$$

So for each charge, we add such a screening potential. Now, to evaluate  $\mathcal{U}_{\text{Coul}}$ , we have to evaluate the potential of a charge density that compensates for the screening charge densities at each particle. This is done in Fourier space.

The potential of a given charge distribution is given by Poisson's equation:

$$-\nabla^2 \phi(r) = 4\pi \rho(r) \quad (229)$$

Now, the compensating charge distribution, denoted  $\rho_1$ , can be written:

$$\rho_1(r) = \sum_{j=1}^N \sum_{\mathbf{n} \in \mathbb{Z}^3} q_j (\alpha/\pi)^{3/2} \exp[-\alpha |\mathbf{r} - (\mathbf{r}_j + \mathbf{n}L)|] \quad (230)$$

Notice that the sum over  $j$  includes the self-interaction when we include the potential due to this charge density in the calculation of the total Coulombic energy (i.e., we have omitted the prime on the outer summation over particle indices).

Now, consider the Fourier transform of Poisson's equation:

$$k^2 \tilde{\phi}(k) = 4\pi \tilde{\rho}(k) \quad (231)$$

The Fourier transform of  $\rho_1(r)$  is given by

$$\rho_1(k) = \int_V d\mathbf{r} e^{-i\mathbf{k} \cdot \mathbf{r}} \rho(\mathbf{r}) \quad (232)$$

$$= \sum_{j=1}^N q_j e^{-i\mathbf{k} \cdot \mathbf{r}_j} e^{-k^2/4\alpha} \quad (233)$$

(The math required to show this involves noting that the the Fourier transform of a Gaussian is another Gaussian, and that the integral over all space of a normalized Gaussian is unity.) The  $\mathbf{k}$ -vectors are given by

$$\mathbf{k} = \frac{2\pi}{L} \mathbf{l} \quad \mathbf{l} \in \mathbb{Z}^3 \quad (234)$$

We can use Eq. 229 to solve for  $\tilde{\phi}(k)$ :

$$\tilde{\phi}(k) = \frac{4\pi}{k^2} \sum_{j=1}^N q_j e^{-i\mathbf{k} \cdot \mathbf{r}_j} e^{-k^2/4\alpha} \quad (235)$$

Note that this solution is not defined for  $k = 0$ . In fact, we have to assume that  $\tilde{\phi}(0) = 0$ , which is consistent with the notion that our system and all its periodic images is embedded in a medium of infinite dielectric constant (a perfect conductor; the "tin foil" boundary condition).

Fourier inverting  $\tilde{\phi}(k)$  gives

$$\phi_i(r) = \frac{1}{V} \sum_{\mathbf{k} \neq 0} \tilde{\phi}(k) e^{i\mathbf{k} \cdot \mathbf{r}} \quad (236)$$

which, when we substitute for  $\tilde{\phi}(k)$  from Eq. 235 yields

$$\phi_i(r) = \sum_{\mathbf{k} \neq 0} \sum_{j=1}^N \frac{4\pi q_j}{V k^2} e^{i\mathbf{k} \cdot (\mathbf{r} - \mathbf{r}_j)} e^{-k^2/4\alpha} \quad (237)$$

So, the total Coulombic energy due to the compensating charge distribution is

$$\mathcal{U}_1 = \frac{1}{2} \sum_i q_i \phi_1(r_i) \quad (238)$$

$$= \frac{1}{2} \sum_{\mathbf{k} \neq 0} \sum_{j=1}^N \frac{4\pi q_i q_j}{V k^2} e^{i\mathbf{k} \cdot (\mathbf{r} - \mathbf{r}_j)} e^{-k^2/4\alpha} \quad (239)$$

$$= \frac{1}{2V} \sum_{\mathbf{k} \neq 0} \frac{4\pi}{k^2} |\rho(\mathbf{k})|^2 e^{-k^2/4\alpha} \quad (240)$$

where

$$\rho(\mathbf{k}) = \sum_{i=1}^N q_i e^{i\mathbf{k} \cdot \mathbf{r}_i} \quad (241)$$

Notice that this does indeed include a spurious self-self interaction, because the point charge at  $\mathbf{r}_i$  interacts with the compensating charge cloud also at  $\mathbf{r}_i$ . This self-interaction is the potential at the center of a Gaussian charge distribution. First, we solve Poisson's equation for the potential due to a Gaussian charge distribution (details in F&S):

$$-\frac{1}{r} \frac{\partial^2 r \phi_{\text{Gauss}}}{\partial r^2} = 4\pi \rho_{\text{Gauss}}(r) \quad (242)$$

yielding

$$\phi_{\text{Gauss}}(r) = \frac{q_i}{r} \text{erf}(\sqrt{\alpha} r) \quad (243)$$

where erf is the error function:

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-r^2} dr \quad (244)$$

At  $r = 0$ , we have

$$\phi_{\text{self}} = \phi_{\text{Gauss}}(0) = 2 \left( \frac{\alpha}{\pi} \right)^{1/2} q_i \quad (245)$$

So the total self-interaction energy becomes

$$\mathcal{U}_{\text{self}} = \left( \frac{\alpha}{\pi} \right)^{1/2} \sum_{i=1}^N q_i^2 \quad (246)$$

which must be *subtracted* from the total Coulombic energy.

Finally, the real-space contribution of the point charge at  $\mathbf{r}_i$  is the screened potential:

$$\phi_{\text{short}}(r) = \frac{q_i}{r} - \frac{q_i}{r} \text{erf}(\sqrt{\alpha} r) \equiv \frac{q_i}{r} \text{erfc}(\sqrt{\alpha} r) \quad (247)$$

where erfc is the complementary error function. The total real-space Coulombic potential energy is therefore

$$\mathcal{U}_{\text{short}} = \frac{1}{2} \sum_{i \neq j}^N \frac{q_i q_j}{r_{ij}} \text{erfc}(\sqrt{\alpha} r_{ij}) \quad (248)$$

Putting it all together:

$$\mathcal{U}_{\text{Coul}} = \frac{1}{2V} \sum_{\mathbf{k} \neq 0} \frac{4\pi}{k^2} |\rho(\mathbf{k})|^2 e^{-k^2/4\alpha} \quad (249)$$

$$- \left(\frac{\alpha}{\pi}\right)^{1/2} \sum_{i=1}^N q_i^2 \quad (250)$$

$$+ \frac{1}{2} \sum_{i \neq j}^N \frac{q_i q_j}{r_{ij}} \text{erfc}(\sqrt{\alpha} r_{ij}) \quad (251)$$

where

$$\rho(\mathbf{k}) = \sum_{i=1}^N q_i e^{i\mathbf{k} \cdot \mathbf{r}_i}. \quad (252)$$

Now, the arbitrariness left to us at this point is in a choice for the parameter  $\alpha$ . Clearly, very small alphas make the Gaussians tighter and therefore the compensating charge distribution less smoothly varying. This means a Fourier series representation of  $\mathcal{U}_1$  with a given number of terms is more accurate for larger  $\alpha$ . We'll evaluate choice of  $\alpha$  in Sec. 7.3.

## 7.2 Ewald Forces

Now, we can use Eq. 249 in a Monte Carlo simulation of a system of charges, provided that periodic boundary conditions are used and the domain is cubic. (Extensions to non-cubic boxes and slab geometries are discussed to a limited extent in F&S.) We can also use the Ewald technique to calculate forces for use in molecular dynamics simulations.

The force on particle  $i$  due to the charges in the system is given by

$$\mathbf{F}_i = -\frac{\partial}{\partial \mathbf{r}_i} \mathcal{U}_{\text{Coul}} \quad (253)$$

For our purposes, the two contributions to  $\mathbf{F}_i$  are due to the  $k$ -space energy and the short-ranged, real-space energy:

$$\mathbf{F}_i = \mathbf{F}_i^{(k)} + \mathbf{F}_i^{(r)} \quad (254)$$

Notice that there is no change in  $\mathcal{U}_{\text{self}}$  when  $\mathbf{r}_i$  changes, so no forces arise from  $\mathcal{U}_{\text{self}}$ .

The  $k$ -space contribution is given by

$$\mathbf{F}_i^{(k)} = q_i \sum_j q_j \frac{1}{V} \sum_{\mathbf{k} \neq 0} \frac{4\pi \mathbf{k}}{k^2} e^{-k^2/4\alpha} \sin(\mathbf{k} \cdot \mathbf{r}_{ij}) \quad (255)$$

The real-space contribution is given by

$$\mathbf{F}_i^{(r)} = q_i \sum_j q_j \left[ 2\sqrt{\frac{\alpha}{\pi}} e^{-\alpha r_{ij}^2} + \frac{1}{r_{ij}} \text{erfc}(\sqrt{\alpha} r_{ij}) \right] \frac{\mathbf{r}_{ij}}{r_{ij}^2} \quad (256)$$

## 7.3 Implementation and Evaluation

We will consider an Ewald implementation which is a modified version of the `ewald` code written for Berend Smit's Molecular Simulation course. (All of Prof. Smit's codes are available in the `FrenkelSmitCodes` directory of the `instructional-codes` repository.) This code simply computes



the Ewald energy for a cubic lattice, given an appropriate number of particles, and a value for  $\sqrt{\alpha}$  (which is called  $\alpha$  in the code), and a value for  $k_{\max}$ , the maximum integer index for enumerating  $k$ -vectors.<sup>1</sup>

The units used in a system with electrostatics differ depending on community. So far, we have assumed that the units of electrostatic potential are charge  $\mathcal{C}$ , divided by length  $\mathcal{L}$ , because we write potential as  $\phi = q/|\mathbf{r}|$ , where  $q$  is measured in units of  $\mathcal{C}$  and distance in units of  $\mathcal{L}$ . Energy is therefore written in units of  $\mathcal{C}^2$  over  $\mathcal{L}$ , and force in units of  $\mathcal{C}^2$  over  $\mathcal{L}^2$ . If we want the final energy in more familiar units, we can choose  $\mathcal{C}$  and  $\mathcal{L}$ , and use the standard prefactor  $1/4\pi\epsilon_0$  to convert from “charge squared per length” to “energy”. For example, in SI units,  $\epsilon_0 = 8.85419 \times 10^{-12}$  (C<sup>2</sup>/m)/J. In this implementation, we use a length of  $\mathcal{L} = 1$  and  $\mathcal{C} = 1$  and measure energy such that  $1/4\pi\epsilon_0 = 1$ .

We will examine two configurations, both with  $N = 8^3 = 512$  particles, with alternating + and - charges. One configuration has the particle on a cubic lattice with lattice spacing  $r_0 = 1$ , which is the standard NaCl crystal structure. We will call this the “crystal” configuration. The other is like the crystal, only each particle is displaced by a random amount from its Self Part lattice position with a maximum displacement of 0.3. We will call this the “liquid” configuration. We compute the total electrostatic energy via the Ewald sum technique for various values of  $1/\sqrt{\alpha}$  and maximum  $k$ -vector index. As we increase the number of  $k$ -vectors taken in the sum, we would like to show that the total energy converges to a certain value. We will measure this in terms of the Madelung constant,  $M$ :

$$\mathcal{U}_{\text{Coul}} = -\frac{Nq^2M}{4\pi\epsilon_0r_0} \quad (257)$$

Table 2 shows results of Ewald summation for the perfect lattice, and Table 3 shows results for the “liquid”. We see several interesting things from these example calculations:

1. The self-energy is the dominant contributor from the long-range compensating charge smears.
2. The Fourier-space contribution is relatively small, and is much smaller for the perfect lattice than for the liquid. This means the precision of the Fourier transform of the charge distribution (which is larger for larger  $k_{\max}$ ) is relatively unimportant to the calculation.
3. The overall Coulomb energy is insensitive to the real-space cutoff for the perfect lattice, while it is weakly decreasing in magnitude (it is negative overall) for increasing real-space cutoff in the liquid.

---

<sup>1</sup>One modification of this code is a necessary one: the implementation of the real-space energy was left as an exercise. Other modifications made easily include embedding the main program inside a double loop over desired  $\alpha$  and  $k_{\max}$  values.

| $1/\sqrt{\alpha}$ | $k_{\max}$ | $\mathcal{U}_{\text{short}}$ | $\mathcal{U}_1$          | $\mathcal{U}_{\text{self}}$ | $\mathcal{U}_{\text{Coul}}$ | $M$    |
|-------------------|------------|------------------------------|--------------------------|-----------------------------|-----------------------------|--------|
| 1.00              | 4          | -0.3106                      | $1.0354 \times 10^{-3}$  | -0.5642                     | -0.8738                     | 1.7476 |
| 1.00              | 8          | -0.3106                      | $1.0354 \times 10^{-3}$  | -0.5642                     | -0.8738                     | 1.7476 |
| 1.00              | 16         | -0.3106                      | $1.0354 \times 10^{-3}$  | -0.5642                     | -0.8738                     | 1.7476 |
| 1.50              | 4          | -0.4958                      | $1.1708 \times 10^{-7}$  | -0.3780                     | -0.8738                     | 1.7476 |
| 1.50              | 8          | -0.4958                      | $1.1708 \times 10^{-7}$  | -0.3780                     | -0.8738                     | 1.7476 |
| 1.50              | 16         | -0.4958                      | $1.1708 \times 10^{-7}$  | -0.3780                     | -0.8738                     | 1.7476 |
| 2.00              | 4          | -0.5917                      | $2.3491 \times 10^{-13}$ | -0.2821                     | -0.8738                     | 1.7476 |
| 2.00              | 8          | -0.5917                      | $2.3491 \times 10^{-13}$ | -0.2821                     | -0.8738                     | 1.7476 |
| 2.00              | 16         | -0.5917                      | $2.3491 \times 10^{-13}$ | -0.2821                     | -0.8738                     | 1.7476 |
| 2.50              | 4          | -0.6481                      | $1.3732 \times 10^{-20}$ | -0.2257                     | -0.8738                     | 1.7476 |
| 2.50              | 8          | -0.6481                      | $1.3732 \times 10^{-20}$ | -0.2257                     | -0.8738                     | 1.7476 |
| 2.50              | 16         | -0.6481                      | $1.3732 \times 10^{-20}$ | -0.2257                     | -0.8738                     | 1.7476 |
| 3.00              | 4          | -0.6876                      | $5.1260 \times 10^{-30}$ | -0.1862                     | -0.8738                     | 1.7476 |
| 3.00              | 8          | -0.6876                      | $5.1260 \times 10^{-30}$ | -0.1862                     | -0.8738                     | 1.7476 |
| 3.00              | 16         | -0.6876                      | $5.1260 \times 10^{-30}$ | -0.1862                     | -0.8738                     | 1.7476 |
| 3.50              | 4          | -0.7102                      | $1.2018 \times 10^{-36}$ | -0.1636                     | -0.8738                     | 1.7476 |
| 3.50              | 8          | -0.7102                      | $1.2018 \times 10^{-36}$ | -0.1636                     | -0.8738                     | 1.7476 |
| 3.50              | 16         | -0.7102                      | $1.2018 \times 10^{-36}$ | -0.1636                     | -0.8738                     | 1.7476 |
| 4.00              | 4          | -0.7328                      | $2.5866 \times 10^{-37}$ | -0.1410                     | -0.8738                     | 1.7476 |
| 4.00              | 8          | -0.7328                      | $2.5866 \times 10^{-37}$ | -0.1410                     | -0.8738                     | 1.7476 |
| 4.00              | 16         | -0.7328                      | $2.5866 \times 10^{-37}$ | -0.1410                     | -0.8738                     | 1.7476 |

**Table 2:** Using the Ewald summation to compute total Coulomb energy of an 8x8x8 “NaCl” lattice (using `ewald.f` from `FrenkelSmitCodes/Exercises`). For various values of the real-space cutoff  $1/\sqrt{\alpha}$  and maximum number of  $k$ -vectors  $k_{\max}$ , the values of the real-space energy  $\mathcal{U}_{\text{short}}$ , the Fourier-space energy  $\mathcal{U}_1$ , and the self-energy correction  $-\mathcal{U}_{\text{self}}$  are shown, together with the Madelung constant  $M$ .

| $1/\sqrt{\alpha}$ | $k_{\max}$ | $\mathcal{U}_{\text{short}}$ | $\mathcal{U}_1$         | $-\mathcal{U}_{\text{self}}$ | $\mathcal{U}_{\text{Coul}}$ | $M$    |
|-------------------|------------|------------------------------|-------------------------|------------------------------|-----------------------------|--------|
| 1.00              | 4          | -0.3322                      | $3.1339 \times 10^{-2}$ | -0.5642                      | -0.8650                     | 1.7300 |
| 1.00              | 8          | -0.3322                      | $3.2193 \times 10^{-2}$ | -0.5642                      | -0.8642                     | 1.7283 |
| 1.00              | 16         | -0.3322                      | $3.2193 \times 10^{-2}$ | -0.5642                      | -0.8642                     | 1.7283 |
| 1.50              | 4          | -0.4960                      | $9.8621 \times 10^{-3}$ | -0.3780                      | -0.8642                     | 1.7283 |
| 1.50              | 8          | -0.4960                      | $9.8652 \times 10^{-3}$ | -0.3780                      | -0.8642                     | 1.7283 |
| 1.50              | 16         | -0.4960                      | $9.8652 \times 10^{-3}$ | -0.3780                      | -0.8642                     | 1.7283 |
| 2.00              | 4          | -0.5859                      | $3.8735 \times 10^{-3}$ | -0.2821                      | -0.8642                     | 1.7283 |
| 2.00              | 8          | -0.5859                      | $3.8735 \times 10^{-3}$ | -0.2821                      | -0.8642                     | 1.7283 |
| 2.00              | 16         | -0.5859                      | $3.8735 \times 10^{-3}$ | -0.2821                      | -0.8642                     | 1.7283 |
| 2.50              | 4          | -0.6402                      | $1.7194 \times 10^{-3}$ | -0.2257                      | -0.8642                     | 1.7283 |
| 2.50              | 8          | -0.6402                      | $1.7194 \times 10^{-3}$ | -0.2257                      | -0.8642                     | 1.7283 |
| 2.50              | 16         | -0.6402                      | $1.7194 \times 10^{-3}$ | -0.2257                      | -0.8642                     | 1.7283 |
| 3.00              | 4          | -0.6787                      | $7.4067 \times 10^{-4}$ | -0.1862                      | -0.8641                     | 1.7282 |
| 3.00              | 8          | -0.6787                      | $7.4067 \times 10^{-4}$ | -0.1862                      | -0.8641                     | 1.7282 |
| 3.00              | 16         | -0.6787                      | $7.4067 \times 10^{-4}$ | -0.1862                      | -0.8641                     | 1.7282 |
| 3.50              | 4          | -0.7008                      | $3.7594 \times 10^{-4}$ | -0.1636                      | -0.8640                     | 1.7281 |
| 3.50              | 8          | -0.7008                      | $3.7594 \times 10^{-4}$ | -0.1636                      | -0.8640                     | 1.7281 |
| 3.50              | 16         | -0.7008                      | $3.7594 \times 10^{-4}$ | -0.1636                      | -0.8640                     | 1.7281 |
| 4.00              | 4          | -0.7230                      | $1.5044 \times 10^{-4}$ | -0.1410                      | -0.8639                     | 1.7278 |
| 4.00              | 8          | -0.7230                      | $1.5044 \times 10^{-4}$ | -0.1410                      | -0.8639                     | 1.7278 |
| 4.00              | 16         | -0.7230                      | $1.5044 \times 10^{-4}$ | -0.1410                      | -0.8639                     | 1.7278 |

**Table 3:** Using the Ewald summation to compute total Coulomb energy of an 8x8x8 “NaCl” liquid (using `ewald.f` from `FrenkelSmitCodes/Exercises`). For various values of the real-space cutoff  $1/\sqrt{\alpha}$  and maximum number of  $k$ -vectors  $k_{\max}$ , the values of the real-space energy  $\mathcal{U}_{\text{short}}$ , the Fourier-space energy  $\mathcal{U}_1$ , and the self-energy correction  $-\mathcal{U}_{\text{self}}$  are shown, together with the Madelung constant  $M$ .

## 8 All-atom Potential Energy Functions

The modeling of molecular structure and inter/intra-molecular interactions is the job of the potential energy function  $\mathcal{U}(\mathbf{r}^N)$ . Modern potential energy functions are actively maintained and carefully curated and optimized by several devoted groups around the world, and for the most part they are made freely available to the research community. In this section, we'll consider a few of the more popular potentials out there. There are many good reviews out there about all-atom potential energy functions; most of what I present here is adapted from the recent review by Harrison et al. [17]

First, a note on terminology. A potential energy function, or just "potential", is conceptually just a function that can compute potential energy from all atomic positions. Pairwise Lennard-Jones is an example we have used extensively already. When used specifically by MD simulations, potentials are often referred to as "force-fields", since it is really their *gradients* that are used in MD; potential energy itself is more of a diagnostic in most standard MD simulations (though it is extremely important in some advanced free-energy methods). Because of this, potentials that are used in MD simulations have to be differentiable, for the most part. (There are a few MD methods that can use so-called "discontinuous forces" but no large-scale MD codes can.)

Second, a note on physical reality vs what potentials actually model. Interactions among atoms are quantum-mechanical in nature. Modeling them accurately involves in-depth quantum chemical calculations that can be very expensive. Potentials used in most molecular simulations are *empirical* functions that generally only very roughly approximate true interatomic interactions. Nuclei are treated as point masses and electrons aren't treated at all. Because of this simplification, empirical potentials always have parameters that must be tuned against more accurate quantum chemical calculations and experimental observations. Parameter tuning in potentials leads to specialization that reflects where such potentials are most in demand. This will become evident as we start discussing important examples.

### 8.1 Class-I Potentials

Class-I potentials break down interatomic interactions between bonded and non-bonded interactions. Bonded interactions include bonds, valence angles, and both proper and improper dihedrals (Fig. 29). Non-bonded interactions include electrostatic and Lennard-Jones interactions. Class-I potentials apply for systems of fixed bonded topology; i.e., bonds between atoms are permanent, never breaking or forming. A simulation using a Class-I potential is therefore unable to model "chemistry". Examples of Class-I potentials include CHARMM [18–20], AMBER [21], Gromos [22], OPLS [23, 24], and TraPPE [25].

The general form of a Class-I potential is

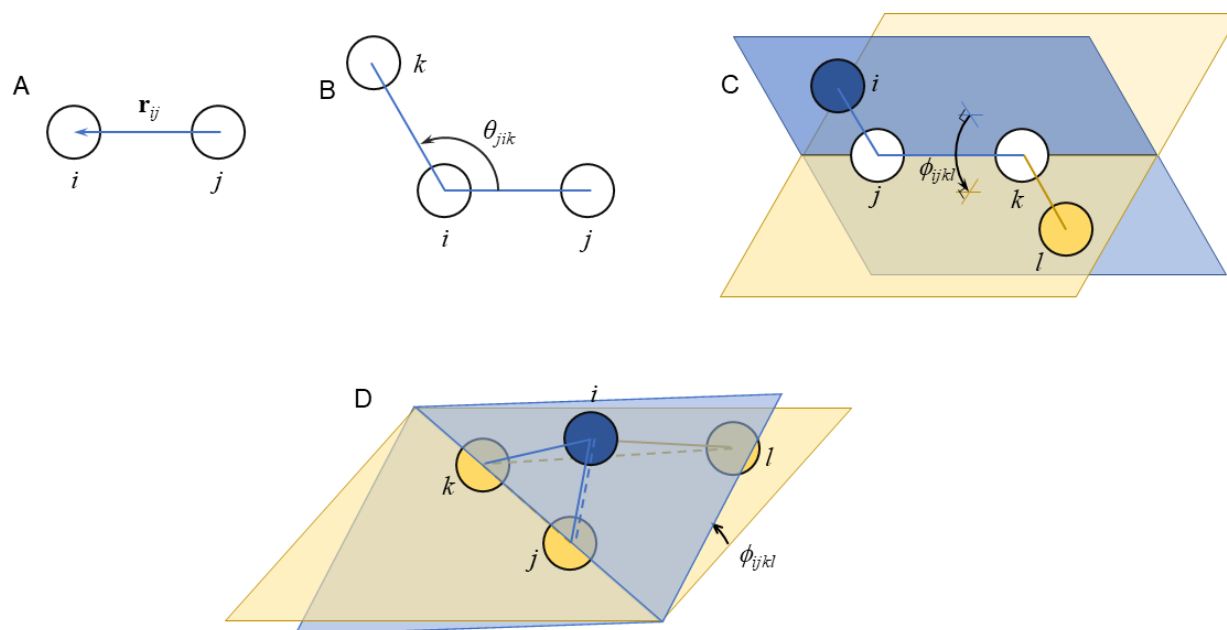
$$\mathcal{U} = \sum_{\text{bonds}} K_b(l - l_0)^2 + \sum_{\text{angles}} K_\theta(\theta - \theta_0)^2 \quad (258)$$

$$+ \sum_{\text{dihedrals}} K_{\phi,n} [1 + \cos(n\phi - \delta_n)] \quad (259)$$

$$+ \sum_{\text{impropers}} K_\phi(\phi - \phi_0)^2 \quad (260)$$

$$+ \sum_{i < j} \left\{ \frac{q_i q_j}{r_{ij}} + 4\epsilon_{ij} \left[ \left( \frac{\sigma}{r_{ij}} \right)^{12} - \left( \frac{\sigma}{r_{ij}} \right)^6 \right] \right\} \quad (261)$$

The first four terms are the "bonded" potential. Each bond is treated like a harmonic spring, with a specific spring constant  $2K_b$  and equilibrium length  $l_0$ . Similarly, every angle is also harmonic. Dihedrals (also called torsions) are periodic and therefore modeled with a trigonometric expansion; each



**Figure 29:** Schematic representation of (A) the  $i$ - $j$  bond vector  $\mathbf{r}_{ij}$ , (B) the  $j$ - $i$ - $k$  angle  $\theta_{jik}$ , (C) the  $i$ - $j$ - $k$ - $l$  torsion  $\phi_{ijkl}$  around the  $j$ - $k$  bond, and (D) the  $i$ - $j$ - $k$ - $l$  improper  $\phi_{ijkl}$ , comprising bonds  $i$ - $j$ ,  $i$ - $k$ , and  $i$ - $l$  (solid), with vectors  $\mathbf{r}_{ij}$ ,  $\mathbf{r}_{jk}$ ,  $\mathbf{r}_{kl}$  (dashed) defining the dihedral angle. In (C), the angle  $\phi_{ijkl}$  is acute and positive by IUPAC/IUB convention; if we sight down the  $j$ - $k$  bond with  $j$  nearer to the eye, the  $i$ - $j$  bond must rotate in a clockwise direction by an angle less than  $\pi$  to eclipse the  $k$ - $l$  bond.

dihedral may have a specific  $K_{\phi,n}$  for one or more values of  $n$ . Finally, improper (improper dihedrals) refer to groupings of four atoms meant to be coplanar, such as three atoms bound to a nitrogen atom in a trigonal-planar configuration.

Fig. 29 depicts representative bonded features with atoms labeled to permit presentation of formulas for computing bond lengths, valence angles, dihedrals, and improper. The measures are all defined conventionally using vector arithmetic and the right-hand rule for vector cross-products. The bond vector  $\mathbf{r}_{ij}$  is defined

$$\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j \quad (262)$$

with magnitude

$$r_{ij} = |\mathbf{r}_{ij}| = |\mathbf{r}_i - \mathbf{r}_j| \quad (263)$$

$$= [(r_{i,x} - r_{j,x})^2 + (r_{i,y} - r_{j,y})^2 + (r_{i,z} - r_{j,z})^2]^{\frac{1}{2}}. \quad (264)$$

For atoms  $j$  and  $k$  both bonded to  $i$ , the valence angle  $\theta_{jik}$  is found via

$$\cos \theta_{jik} = \frac{\mathbf{r}_{ij} \cdot \mathbf{r}_{ik}}{r_{ij}r_{jk}} \quad (265)$$

For atoms  $j$  and  $k$  bonded to each other, the dihedral angle defined by atom  $i$  bonded to  $j$  and atom  $l$  bonded to  $k$  is found by computing the angle of intersection between the two planes defined by the  $i$ - $j$ / $j$ - $k$  bonds and the  $j$ - $k$ / $k$ - $l$  bonds.

$$\cos \phi_{ijkl} = \frac{(\mathbf{r}_{ji} \times \mathbf{r}_{kj}) \cdot (\mathbf{r}_{kj} \times \mathbf{r}_{lk})}{|\mathbf{r}_{ji} \times \mathbf{r}_{kj}| |\mathbf{r}_{kj} \times \mathbf{r}_{lk}|} \quad (266)$$

Now, for improper dihedrals, this formula is usually sufficient because (a)  $\phi_0$  is zero for coplanarity and thus  $\phi$  should be close to zero, and (b)  $\arccos()$  always returns an angle on  $[0, \pi]$ . However, for proper dihedrals (torsions), since these are fully periodic and not guaranteed to have symmetry of reflection about 0 or  $\pi$ , we must define  $\phi$  on  $[-\pi, \pi]$ , and the  $\arccos()$  is not enough. So, the convention used to determine the sign of the torsion angle is to compute its sine:

$$\sin \phi_{ijkl} = \frac{[(\mathbf{r}_{ji} \times \mathbf{r}_{kj}) \cdot (\mathbf{r}_{kj} \times \mathbf{r}_{lk})] \cdot \mathbf{r}_{kj}}{|\mathbf{r}_{ji} \times \mathbf{r}_{kj}| |\mathbf{r}_{kj} \times \mathbf{r}_{lk}| r_{kj}} \quad (267)$$

The sign of  $\phi_{ijkl}$  is determined by its sine thusly:

$$\phi_{ijkl} = \begin{cases} -\cos^{-1} \cos \phi_{ijkl}, & \text{if } \sin_{ijkl} < 0 \\ \cos^{-1} \cos \phi_{ijkl}, & \text{otherwise} \end{cases} \quad (268)$$

Most Class-I potentials are based on the concept of context-specific “atom types”. For example, a carbon atom in an aliphatic chain is of a different type than a carbon atom in an aromatic ring, even though both are “carbon”. Then, bonds, angles, dihedrals, and impropers are classified by the types of their member atoms. This results in huge databases of potential parameters and the necessity to fully specify all bonded parameters for any system that is to be simulated. Luckily, most MD packages have helper routines to do just that.

Determining these parameters for a given force field is context-specific and therefore a bit of an art. Bonded potential parameters are mostly found by fitting the given analytical forms to series of single-point energy calculations performed using high-level quantum-mechanical simulations. For example, parameters for the C1-C2-C3-C4 torsion in olefins was parameterized by sweeping the torsion angle of a quantum mechanical model of butane. Because most molecules involve a superposition of such bonded potentials, their use confers a vibrational mode spectrum to molecules that can also be matched with experimental spectra for parameter adjustment.

Non-bonded parameters like charges and LJ parameters are adjusted so that errors in thermodynamics property calculations are minimized. This includes things like vacancy energies in solids, heats of vaporization in liquids, and volumetric equations of state. Most force-field developers strive to make their parameters sets as transferrable as possible, but one must always be aware of the context a force-field is most appropriate for. CHARMM and AMBER are used mainly to model biological molecules, while TRAPPE is pretty much strictly used for fluids. OPLS and GROMOS are used in a variety of settings.

It is important to note that all potentials considered so far assume each atom has a fixed partial charge (which may be zero). In fact, for class-I potentials, “charge” is an atom attribute wholly separate from its type, and they are usually derived from QM calculations on small fragments. Potentials for which the charge on each atom is a degree of freedom rather than a fixed quantity are described as “polarizable”. Polarizable potentials are not widely used, but there are good arguments for their requirement in situations where atoms with high polarizabilities might lead to local dipole moments that influence interatomic interactions.

One important polarizable model is the classical Drude oscillator, in which each atom is assigned a “ghost” particle tethered to its center that carries some of the charge. This allows each atom to behave like a literal “fluctuating dipole”, but it requires an extended Hamiltonian approach where the Drude particles’ equations of motion are solved along with those of the atom nuclei, making Drude-enabled

simulations substantially more expensive than fixed-charge simulations. For example, the polarizable version of CHARMM is called CHARMM-Drude [26, 27].

## 8.2 Reactive Potentials

There are of course whole classes of problems for which the lack of ability to model bond breaking and forming is a show-stopper. Though in principle quantum calculations could be done in these cases, the number of atoms typically included in a system usually precludes this. Instead, a lot of work has gone into developing reactive potentials. The three main classes we consider here are the embedded-atom method, bond-order potentials and ReaxFF.

### 8.2.1 Embedded Atom Method (EAM)

The embedded-atom method (EAM) was originally developed to model metals. [28] The basic idea of EAM is that atoms interact in a pairwise manner with the nearest neighbors but they also interact with a global field of electron density that is explicitly many-body in nature. For a system of  $N$  atoms, the EAM potential is

$$\mathcal{U}(\mathbf{r}^N) = \sum_{i=1}^N \left[ F_i(\rho_{h,i}) + \frac{1}{2} \sum_{j \neq i} \phi_{ij}(r_{ij}) \right] \quad (269)$$

where

$$F(\rho) = -\sqrt{\rho}, \quad (270)$$

$$\rho_i = \sum_{j \neq i} g(r_{ij}), \quad (271)$$

$$g(r) = \exp(-\beta r), \quad (272)$$

$$\phi(r) = V_2(r) - 2F[g(r)], \quad (273)$$

$$V_2(r) = \begin{cases} V_{\text{ZBL}}(r), & r < r_1, \\ \alpha_0 + \alpha_1 r + \alpha_2 r^2 + \alpha_3 r^3, & r_1 \leq r < r_2, \\ -2c \exp\left(-\frac{\beta}{2}r\right) + \Phi_0 \exp(-\alpha r), & r \geq r_2, \end{cases} \quad (274)$$

$$V_{\text{ZBL}}(r) = \frac{Z_1 Z_2 e^2}{4\pi\epsilon_0 r} \sum_{i=1}^4 c_i \exp\left(-d_i \frac{r}{a}\right) \quad (275)$$

The second term in Eq. 273 accounts for the fact that the electron density into which an atom is embedded does not include electrons from that atom itself. Eq. 275 is the Ziegler-Biersack-Littmark (ZBL) screened nuclear repulsion potential used for modeling high-energy collisions between atoms. The three branches that make up  $V_2$  produce a spline-connection between the ZBL and a Morse-like attractive tail.

Among the many systems simulated using EAM is the sputtering of copper. [29]

### 8.2.2 Bond-Order Potentials

Bond-order potentials aim to capture the effect of the nearest-neighbor environment on the behavior of any bond. Such potentials began with the work of Abell [30]. Here I only very lightly gloss over this very deep field of research. In the bond-order formalism, the total potential energy due to covalent bonds is:

$$U = \sum_i \sum_{j>i} \phi_{ij}, \quad (276)$$

where the bond energy  $\phi_{ij}$  between atoms  $i$  and  $j$  has repulsive and attractive components:

$$\phi_{ij} = V_R(r_{ij}) - \bar{b}_{ij}V_A(r_{ij}), \quad (277)$$

where  $V_R$  and  $V_A$  are Morse-type pair potentials:

$$V_R(r_{ij}) = f_{ij}(r_{ij})A_{ij} \exp(-\lambda_{ij}r_{ij}) \quad \text{and} \quad (278)$$

$$V_A(r_{ij}) = f_{ij}(r_{ij})B_{ij} \exp(-\mu_{ij}r_{ij}) \quad (279)$$

Here,  $r_{ij}$  is the scalar separation between atoms  $i$  and  $j$ .  $A_{ij}$ ,  $B_{ij}$ ,  $\lambda_{ij}$ , and  $\mu_{ij}$  are all parameters specific to the two elements participating in the bond. (I use the following  $ij$ -subscript convention: when  $ij$  appears on a variable,  $i$  and  $j$  refer to the individual atom indices; when  $ij$  appears on a parameter or function,  $i$  and  $j$  are specific only to the elements of atoms  $i$  and  $j$ .) The cutoff function  $f_{ij}$  decays smoothly from 1 at some “inner” radius to 0 at some “outer” radius.

The bond order  $\bar{b}_{ij}$  models all of the many-body chemistry:

$$\bar{b}_{ij} = \frac{1}{2} [b_{ij} + b_{ji} + \text{corrections}], \quad (280)$$

where  $b_{ij}$  is the contribution of atoms that neighbor atom  $i$  to the bond order of the  $ij$  bond. These involve complicated 3- and 4-body interactions. The corrections arise from the need to expand set of thermochemical data which the potentials are fit against.

Bond-order potentials first applied to metals, but expanded into silicon and hydrocarbons with the work of and Tersoff [31] and Brenner [32], respectively, producing what is called the “reactive empirical bond order potential (REBO). The more recent adaptive intermolecular reactive empirical bond-order (AIREBO) potential combines REBO with Lennard-Jones interactions and specific torsional potentials for better modeling of hydrocarbon chains. [33]

Polarizable versions of reactive potentials have also been developed. The charge-optimized many-body (COMB) potential is an extension of REBO in which the charge on each atom is allowed to change according to energies dictated by input parameters such as atom electronegativity. [34] Adding oxygen to the AIREBO hydrocarbon potential also necessitated including polarizability, leading to the qAIREBO potential. [35]

### 8.2.3 ReaxFF

ReaxFF was introduced in 2001 as a new type of reactive force field for hydrocarbons, and its formalism has now greatly expanded to describe reactive interatomic chemistries for a wide swath of the periodic table [36, 37]. ReaxFF adopts the co-called “central-force” formalism wherein all pairs of atoms interact and there are no switching functions. ReaxFF breaks down the potential into seven distinct additive contributions:

$$\mathcal{U}_{\text{ReaxFF}} = U_{\text{bond}} + U_{\text{over}} + U_{\text{angle}} + U_{\text{torsion}} + U_{\text{VdW}} + U_{\text{Coulomb}} + U_{\text{specific}} \quad (281)$$

The role of the “overcoordination penalty”  $U_{\text{over}}$  is to monitor atom valencies and penalize deviations from an ideal. This keeps hydrogens from binding to more than one partner, and carbons more the four. The bond, angle, and torsion terms mimic those of class-I potentials but essentially determine parameters on the fly based on geometries. The Coulomb and VdW terms represent electrostatics and dispersion/excluded-volume interactions, while atom charges are forced to equilibrate given atom electronegativities. The “specific” term is the “secret sauce” where a lot of things are put to specialize ReaxFF for a particular system.



ReaxFF has been used for a large variety of reactive systems to date. It is fairly expensive to run and requires a lot of tuning when applying it to systems it has never been applied to before.

### 8.3 Case Study: Stillinger-Weber Silicon

As a precursor to the Tersoff formalism, perhaps the earliest attempt to use molecular simulation with a reactive potential to study a realistic atomic-scale model of silicon was due to Stillinger and Weber [38]. The C-code that implements the Stillinger-Weber (SW) potential for silicon is `mdswsi.c`. This code computes in reduced units as well;  $\sigma = 0.20951$  nm,  $\epsilon = 2.1678$  eV, and  $m = 28.085$  amu, which are appropriate for a system of pure silicon. One reduced unit of temperature,  $\epsilon/k_B$ , corresponds to 25156.74 K.

The main reason to introduce Stillinger-Weber silicon here is to give you a historical example of an implementation of a simple reactive force-field. Silicon forms 4-coordinated tetrahedral bonded structures. The SW potential includes three-body interactions that enforce this symmetry as well as permit breaking and reforming of bonds to produce defects, for example.

The total potential is expressed as two sums, one for unique pair interactions, and another for unique triplet interactions:

$$\mathcal{U} = \sum_{i < j} v_2(r_{ij}) + \sum_{i < j < k} v_3(\mathbf{r}_i, \mathbf{r}_j, \mathbf{r}_k) \quad (282)$$

The two-body term models the bonds:

$$v_2(r) = \begin{cases} \epsilon A (B r^{-p} - r^{-q}) \exp \left[ (r - a)^{-1} \right], & r < a \\ 0 & r \geq a \end{cases} \quad (283)$$

It is very much like a Lennard-Jones potential, only with different exponents and a “smooth cutoff” as the interatom separation distance,  $r$ , approaches some cutoff,  $a$ , given by the factor  $\exp \left[ (r - a)^{-1} \right]$ .

The three body models the angles, and is the sum of functions of each of the three angles of a triplet,  $ijk$ :

$$v_3(\mathbf{r}_i, \mathbf{r}_j, \mathbf{r}_k) = h_{jik} + h_{ijk} + h_{ikj} \quad (284)$$

Here I have employed the shorthand notation  $h_{jik} \equiv h(r_{ij}, r_{ik}, \theta_{jik})$ . Note that, in the notation of this potential,  $\theta_{jik}$  is subtended at  $\mathbf{r}_i$ , and  $\cos \theta_{jik}^c = -\frac{1}{3}$ :

$$h(r_{ij}, r_{ik}, \theta_{jik}) \equiv \begin{cases} \epsilon \lambda \exp \left[ \frac{\gamma}{r_{ij}-a} + \frac{\gamma}{r_{ik}-a} \right] \left( \cos \theta_{jik} - \cos \theta_{jik}^c \right)^2 & \text{if } r_{ij} < a, \text{ and} \\ 0 & \text{otherwise} \end{cases} \quad (285)$$

One computes the angle- $j$  term,  $h_{ijk}$ , and the angle- $k$  term,  $h_{ikj}$ , by permuting the indices appropriately.

The parameters used in the original study by Stillinger and Weber are:

$$A = 7.049556277 \quad (286)$$

$$B = 0.6022245584 \quad (287)$$

$$p = 4 \quad (288)$$

$$q = 0 \quad (289)$$

$$a = 1.80 \quad (290)$$

$$\lambda = 21.0 \quad (291)$$

$$\gamma = 1.20 \quad (292)$$

As in any MD simulation, one computes the force on any particle  $i$  from the negative gradient of the total potential:

$$\mathbf{f}_i = -\nabla_{\mathbf{r}_i} \mathcal{U} \quad (293)$$

$$= -\sum_{j \neq i} \nabla_{\mathbf{r}_i} v_2(r_{ij}) - \sum_{j \neq i} \sum_{k \neq i, j} \nabla_{\mathbf{r}_i} v_3(\mathbf{r}_i, \mathbf{r}_j, \mathbf{r}_k) \quad (294)$$

The two-body term for the  $i$ - $j$  interaction is only slightly more complicated than Lennard-Jones, due to the smooth cutoff. Here, assuming  $i$  and  $j$  are within interaction range ( $r_{ij} < a$ ), we have for the force on  $i$  due to  $j$ :

$$-\nabla_{\mathbf{r}_i} v_2(r_{ij}) = -\frac{\partial}{\partial \mathbf{r}_i} \left\{ \epsilon A \left( B r_{ij}^{-p} - r_{ij}^{-q} \right) \exp \left[ (r_{ij} - a)^{-1} \right] \right\} \quad (295)$$

$$= -\epsilon A \frac{\mathbf{r}_{ij}}{r_{ij}} \left( \left[ \frac{\partial}{\partial r} \left( B r^{-p} - r^{-q} \right) \right] \Big|_{r_{ij}} \exp \left[ (r_{ij} - a)^{-1} \right] \right. \quad (296)$$

$$\left. + \left( B r_{ij}^{-p} - r_{ij}^{-q} \right) \left\{ \frac{\partial}{\partial r} \exp \left[ (r - a)^{-1} \right] \right\} \Big|_{r_{ij}} \right) \quad (297)$$

$$= -\epsilon A \frac{\mathbf{r}_{ij}}{r_{ij}} \left\{ \left( -p B r_{ij}^{-p-1} + q r_{ij}^{-q-1} \right) \exp \left[ (r_{ij} - a)^{-1} \right] \right. \quad (298)$$

$$\left. - \left( B r_{ij}^{-p} - r_{ij}^{-q} \right) \exp \left[ (r_{ij} - a)^{-1} \right] (r_{ij} - a)^{-2} \right\} \quad (299)$$

$$= v_2 \frac{\mathbf{r}_{ij}}{r_{ij}} \left[ \frac{p B r_{ij}^{-p-1} - q r_{ij}^{-q-1}}{B r_{ij}^{-p} - r_{ij}^{-q}} + (r_{ij} - a)^{-2} \right] \quad (300)$$

$$\equiv \mathbf{f}_{ij} \quad (301)$$

Note that it is still true that  $\mathbf{f}_{ij} = -\mathbf{f}_{ji}$ . As an exercise, you should be able to show that the right-hand-side of Eq. 300 is well-behaved (that is, it does not diverge, and in fact vanishes) as  $r_{ij} \rightarrow a^-$ .

It is comparatively much more tedious to evaluate the three-body gradients:


$$-\nabla_{\mathbf{r}_i} v_3(\mathbf{r}_i, \mathbf{r}_j, \mathbf{r}_k) = -\nabla_{\mathbf{r}_i} (h_{jik} + h_{ijk} + h_{ikj}) \quad (302)$$

$$= -\left( \frac{\partial h_{jik}}{\partial \mathbf{r}_i} + \frac{\partial h_{ijk}}{\partial \mathbf{r}_i} + \frac{\partial h_{ikj}}{\partial \mathbf{r}_i} \right) \quad (303)$$


$$\equiv \mathbf{f}_{i \leftarrow jk} \quad (304)$$

The triplet forces on the other members of the triplet,  $\mathbf{f}_{j \leftarrow ik}$  and  $\mathbf{f}_{k \leftarrow ij}$ , are defined analogously.


Each of the partials in Eq. 303 is unique:



$$\begin{aligned} \frac{\partial h_{jik}}{\partial \mathbf{r}_i} &= -\gamma h_{jik} \left[ \frac{\mathbf{r}_{ij}}{r_{ij}} \frac{1}{(r_{ij} - a)^2} + \frac{\mathbf{r}_{ik}}{r_{ik}} \frac{1}{(r_{ik} - a)^2} \right] \\ &+ 2\lambda \exp \left( \frac{\gamma}{r_{ij} - a} + \frac{\gamma}{r_{ik} - a} \right) (\cos \theta_{jik} - \cos \theta_{jik}^c) \\ &\times \left[ \frac{\mathbf{r}_{ij}}{r_{ij}} \frac{1}{r_{ik}} + \frac{\mathbf{r}_{ik}}{r_{ik}} \frac{1}{r_{ij}} - \left( \frac{\mathbf{r}_{ij}}{r_{ij}} \frac{1}{r_{ik}} + \frac{\mathbf{r}_{ik}}{r_{ik}} \frac{1}{r_{ij}} \right) \cos \theta_{jik} \right] \end{aligned} \quad (305)$$



$$\begin{aligned} \frac{\partial h_{ijk}}{\partial \mathbf{r}_i} &= -\gamma_j h_{ijk} \left[ \frac{\mathbf{r}_{ij}}{r_{ij}} \frac{1}{(r_{ij} - a)^2} \right] \\ &+ 2\lambda_j \exp \left( \frac{\gamma_j}{r_{ij} - a} + \frac{\gamma_j}{r_{jk} - a} \right) (\cos \theta_{ijk} - \cos \theta_{ijk}^c) \\ &\times \left[ \frac{\mathbf{r}_{jk}}{r_{jk}} \frac{1}{r_{ij}} + \frac{\mathbf{r}_{ij}}{r_{ij}} \frac{1}{r_{jk}} \cos \theta_{ijk} \right] \end{aligned} \quad (306)$$



$$\begin{aligned} \frac{\partial h_{ikj}}{\partial \mathbf{r}_i} &= -\gamma_k h_{ikj} \left[ \frac{\mathbf{r}_{ik}}{r_{ik}} \frac{1}{(r_{ik} - a)^2} \right] \\ &+ 2\lambda_k \exp \left( \frac{\gamma_k}{r_{ik} - a} + \frac{\gamma_k}{r_{jk} - a} \right) (\cos \theta_{ikj} - \cos \theta_{ikj}^c) \\ &\times \left[ -\frac{\mathbf{r}_{jk}}{r_{jk}} \frac{1}{r_{ik}} + \frac{\mathbf{r}_{ik}}{r_{ik}} \frac{1}{r_{jk}} \cos \theta_{ikj} \right] \end{aligned} \quad (307)$$

Note that the right hand sides of each of Eqns. 305, 306, and 307 vanish when the appropriate  $r$ 's are greater than  $a$ 's, as in Eq. 212. As an exercise, it is easy to show that  $\mathbf{f}_{i \leftarrow jk} + \mathbf{f}_{j \leftarrow ik} + \mathbf{f}_{k \leftarrow ij} = 0$ .

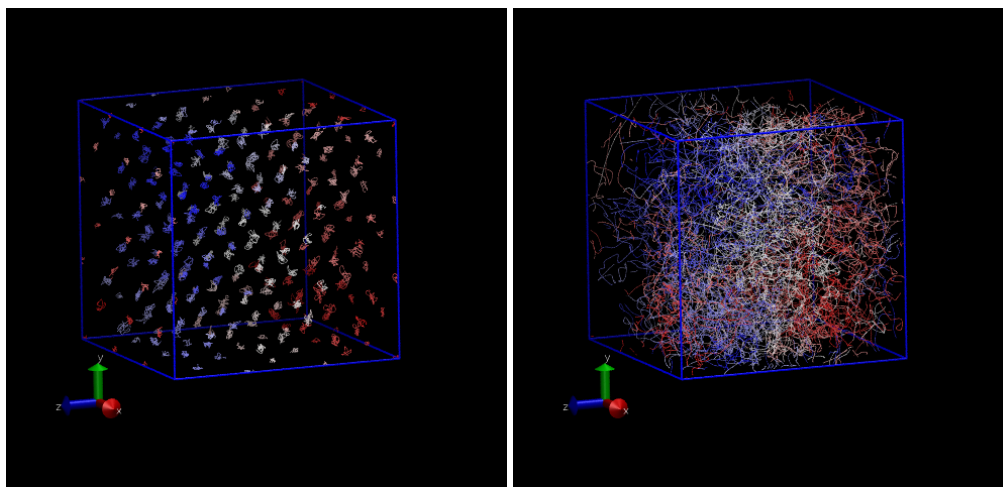
Let us now run two MD simulations for 10,000 time steps; one with and the other without three body forces, at reduced initial temperature  $T = 0.12$  and reduced density  $\rho = 0.46$  for a small system of 216 atoms. The code `mdswsi.c` can initialize atoms on a diamond-cubic lattice; the DC unit cell has 8 atoms, so the number of atoms one should specify for a perfect crystal is 8 times any product of three integers. Using the `-nc #,#,#` switch we indicate how many DC unit cells we want in the  $x$ ,  $y$ , and  $z$ , directions, respectively. The code also has a switch `-do-three-body` which is by default on, but can be turned off by passing a zero. Running this twice generates two logs and two trajectories:

```
$ ./mdswsi -nc 3,3,3 -ns 10000 -fs 10 -prog 10 \
  -do-three-body 1 -traj yes3.xyz > yes3.log
$ ./mdswsi -nc 3,3,3 -ns 10000 -fs 10 -prog 10 \
  -do-three-body 0 -traj no3.xyz > no3.log
```

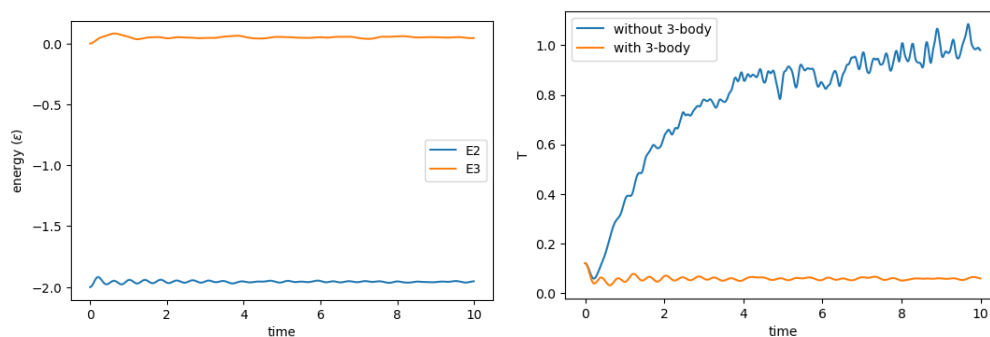
Fig. 30 shows two 3-D system representations. The first is a perfect DC lattice with 216 atoms, where a dot is drawn every 10 time steps for each atom, and each atom is colored uniquely. The second shows the same view of a system for which the three-body forces are turned off; notice that it appears liquid-like. Three-body forces are required to stabilize the DC lattice since each atom only has four nearest neighbors.

Fig. 31 shows a plot of two-body and three-body potential energy vs. time from the first simulation, and a plot of instantaneous temperature vs. time from both simulations on the right. Notice that the three-body forces act to keep the system oscillating in its crystalline state, and the lack of three-body forces results in system melting. This latter occurrence is because the DC lattice is a fairly unfavorable configuration for a system with only two-body forces active.

The code `mdswsi.c` is very slow when three-body forces are turned on because it uses a brute-



**Figure 30:** Two views of a 10,000-time-step NVE MD simulation of a system of 216 silicon atoms initialized on a diamond-cubic lattice; (left) with three-body force and (right) without three-body forces. Each point is an atom position, and atom positions are rendered every 10 time-steps, and each atom is colored uniquely.



**Figure 31:** Results of a 216-atom NVE MD simulation of SW silicon initialized on a DC lattice. (Left) Two-body and three-body contributions to the potential energy vs time. (Right) Temperature vs. time for systems with and without three-body forces active.

force  $N^3$  loop. In practice, any code that implements two- and three-body potentials with short-range cutoffs will use data structures like the neighbor list and the link-cell algorithm to make the pair and triplet loops more efficient. And although SW silicon is historically important, there are much better (and more complicated) potentials out there for silicon-based systems, such as COMB [34].

## 9 Open-source Production MD: Gromacs and NAMD

In this section, I'll illustrate a few examples for using two of the most popular MD packages for academic research: Gromacs [39] and NAMD [40]. There are many other packages in use, and I in no way mean to say these are any better or worse, just that they are popular.

### 9.1 Gromacs

The Gromacs source code is available officially from [www.gromacs.org](http://www.gromacs.org), though some Linux distributions offer pre-compiled versions. In most high-performance computing settings, Gromacs is compiled from source code in order to link in hardware-specific libraries for things like internode communication and compiler-specific math libraries. For this introductory survey though, we can just run the precompiled version on a laptop. (If you have macOS, you might have to compile Gromacs from source.) In Ubuntu under WSL:

```
$ sudo apt install gromacs
```

Gromacs is a suite of tools that include an MD engine along with tools for system preparation and simulation analysis. All tools are invoked using the pre-command 'gmx'. 'gmx -help' will give a lot of information.

Before proceeding with a couple of practical examples, I must convey the importance of reading the documentation if you want to use Gromacs in your own research. The [official Gromacs documentation](#) is extensive, but accessible to beginners. The [official tutorials](#) by Justin Lemkul are also a must if you want to learn how to use Gromacs.

Like any MD simulation, using Gromacs breaks down into three main steps:

1. **Prepare system.**
  - (a) Get relevant atomic coordinates;
  - (b) Decide on a force-field and set the topology;
  - (c) Add solvent, ions, other atoms as necessary;
  - (d) Minimize the potential energy.

2. **Run MD.**

3. **Analyze results.**

As may be inferred, the first step is often the most difficult. It usually requires a lot of care and thought to generate an initial condition for MD. Here we'll consider just two test cases for which this is not so difficult, but which illustrate the workflow.

#### 9.1.1 A Box of Water

Here I illustrate a workflow for generating and simulating a small box of water molecules.

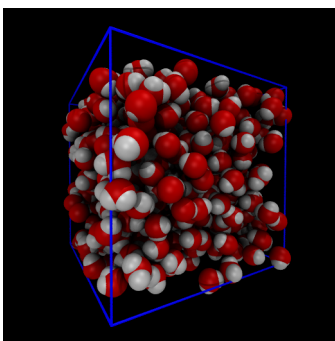
First, we can use 'gmx insert-molecules' to generate a system with waters randomly positioned inside a box:

```
$ gmx insert-molecules -ci /usr/share/gromacs/top/spc216.gro \  
-nmol 100 -box 3 3 3 -o water_box.gro
```

The `-box` switch allows us to specify a box that is  $3 \times 3 \times 3$  nm<sup>3</sup>, and `-ci` refers to a special file containing template atomic coordinates for a single water molecule. The `-nmol` switch asks Gromacs to try to insert more molecules than a simple volume/density calculation would suggest (not many more can be put in). This creates the file `water_box.gro`, which for me contains 432 water molecules, as depicted using VMD in Fig. 32.

Next, we need to use `pdb2gmx` to generate the Gromacs topology file.

```
$ gmx pdb2gmx -f water_box.gro -o new_water_box.gro -p topol.top
```



**Figure 32:** A cubic box containing 432 water molecules created using `gmx insert-molecules`.

In running this command, I selected the CHARMM27 force field (which won't matter since we have nothing but water here) and the TIP3P water model. We now have the topology file and a new gromacs coordinate file (which just renames the water molecules to HOH to conform to the TIP3P naming scheme).

Now, using the `minim.mdp` parameter file given, we can build and run the energy minimization:

```
$ gmx grompp -f minim.mdp -c new_water_box.gro -p topol.top -o min.tpr
$ gmx mdrun -v -deffnm min
```

This will create a lot of output files that all begin with `min`. One of them contains all the energy-like data: `min.edr`. This file is binary, so the tool `gmx energy` is used to extract data from it:

```
$ gmx energy -f min.edr
```

This will create `energy.xvg` with column-oriented time-series of whatever data is selected interactively. Fig. 33 shows what the potential energy looks like for this minimization.

(This is a very, very minimized system; the initial water box had no overlaps really at all.) Now we can run the NVT simulation using the `nvt.mdp` parameter file:

```
$ gmx grompp -f nvt.mdp -c min.gro -p topol.top -o nvt.tpr
$ gmx mdrun -v -deffnm nvt
```

Fig. 34 shows a plot of the energies vs time for this short, short simulation.

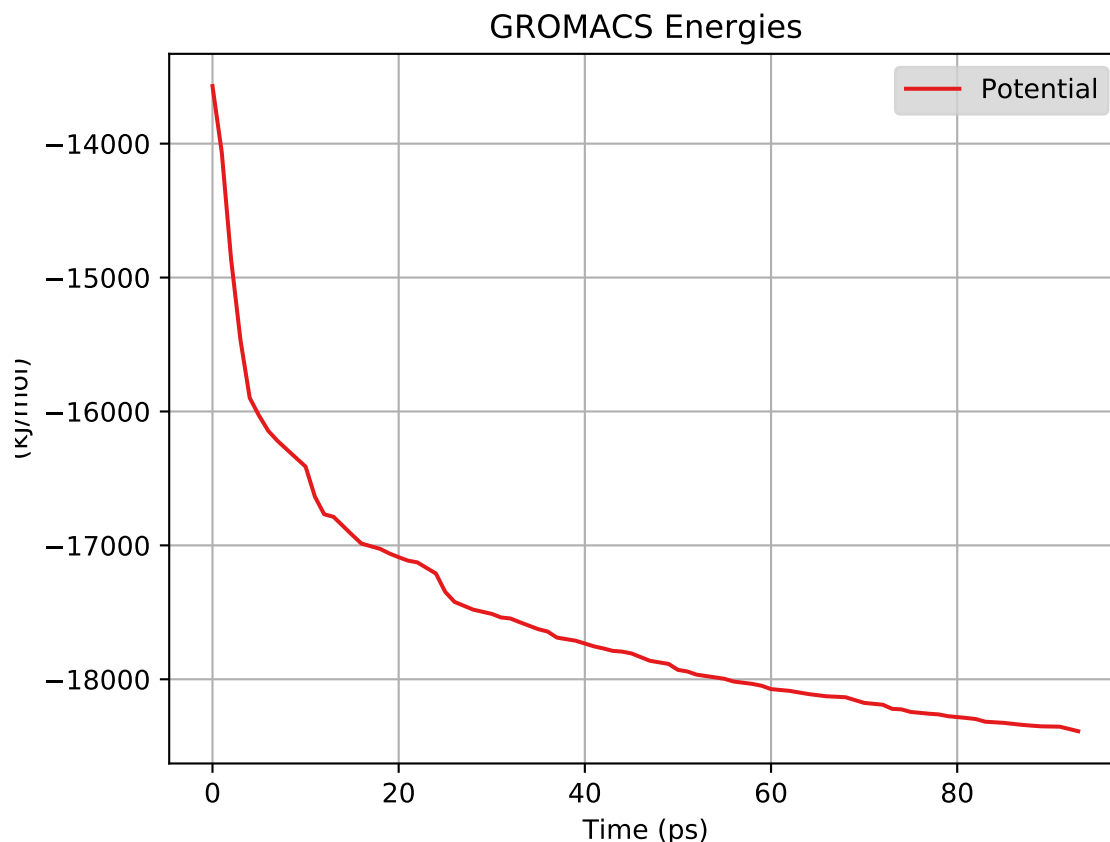
Now let's try using the output configuration from this NVT simulation as an input for an NPT simulation.

```
$ gmx grompp -f npt.mdp -c nvt.gro -t nvt.cpt -p topol.top -o npt.tpr
$ gmx mdrun -v -deffnm npt
```

Fig. 35 shows a plot of the density vs time for this short, short simulation.

### 9.1.2 The SARS-CoV-2 spike protein receptor binding domain (RBD)

As a quick example of how to build a protein simulation system, we can consider a very recent example from the Protein Data Bank. The SARS-CoV-2 spike glycoprotein complex is an enormous protein, but a really important part of it are the domains that bind to the ACE2 receptors on the surfaces of epithelial cells. These are called Receptor Binding Domains (RBDs). A lot of recent structural biology has gone into understanding the details of the RBD-ACE2 interface. As an example, take a look at the PDB entry 7c8j, which is the X-ray crystallographic structure of a recombinant construct of the SARS-CoV-2 RBD and the ACE2 ectodomain of the bat [41]. We can use simulations to answer a lot of interesting questions about this structure, but let's just use it now as a source of coordinates to run a simulation of just the RBD alone.



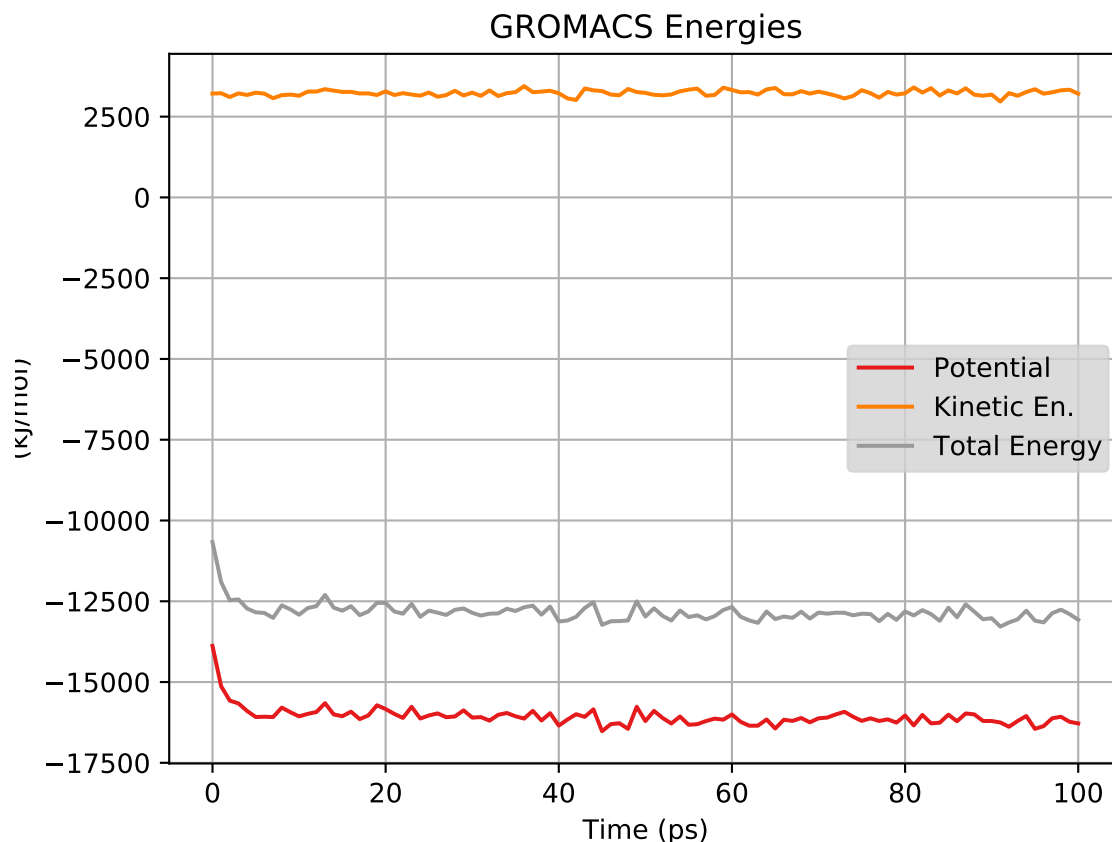
**Figure 33:** Potential energy vs cycle in a Gromacs minimization of a box of 432 waters.

The first thing to do is to download the PDB file for this entry; there are several ways to do this, but I like to use an interactive VMD session and just put 7c8j in the new molecule file browser. Once VMD has it loaded, the following Tcl command in the terminal will create the stripped-down PDB file for just the RBD:

```
[atomselect top "chain B"] writepdb my_7c8j.pdb
```

Now we can pretty much follow Justin Lemkul's lysozyme tutorial here:

```
# generate the topology; use OPLS-AA force field (sel. 15)
$ gmx pdb2gmx -f my_7c8j.pdb -o my_7c8j_processed.pdb -water spce
# enlarge box
$ gmx editconf -f my_7c8j_processed.pdb -o my_7c8j_newbox.gro \
  -c -d 1.0 -bt cubic
# solvate
$ gmx solvate -cp my_7c8j_newbox.gro
# copy JK's ions.mdp; add ions (replace group 13)
$ gmx grompp -f ions.mdp -c my_7c8j_solv.gro -p topol.top \
  -o ions.tpr
$ gmx genion -s ions.tpr -o my_7c8j_solv.gro -p topol.top \
  -pname NA -nname CL -neutral
```



**Figure 34:** Energies vs time in an NVT MD simulation (300 K) of a box of 432 water molecules.

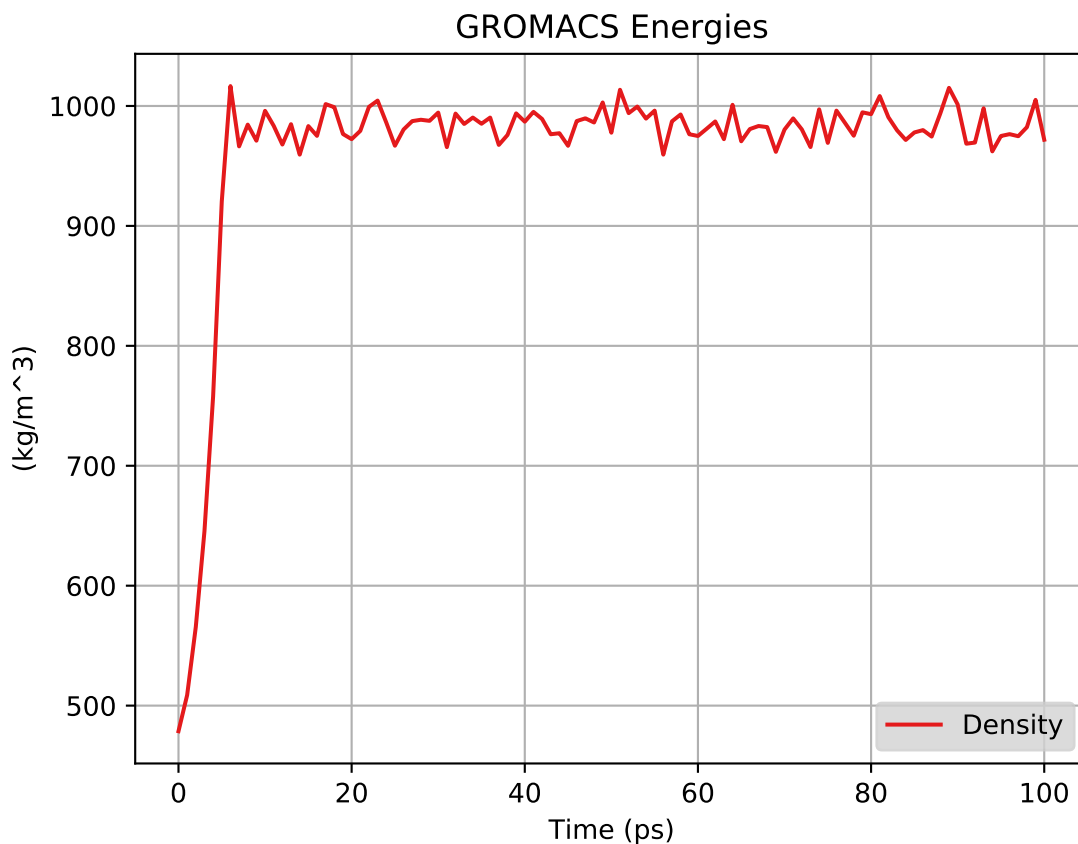
```
# minimize potential energy using JK's minim.mdp
$ gmx grompp -f minim.mdp -c my_7c8j_solv.gro -p topol.top -o em.tpr
$ gmx mdrun -v -deffnm em
# have a look at the potential energy
$ gmx energy -f em.edr -o potential.xvg
# copy JK's nvt.mdp; run NVT MD for 50,000 steps -- this will take a
$ gmx grompp -f nvt.mdp -c em.gro -r em.gro -p topol.top -o nvt.tpr
$ gmx mdrun -v -deffnm nvt
```

This system has about 60,000 atoms. I show a couple of views from VMD in Fig. 36. For such a large system, we won't be able to run very long on a laptop; 100 ps takes about 3 hours on mine. Note that we'd typically want to simulate for hundreds of nanoseconds, which would be several thousand hours on my laptop, or a day or so on a supercomputer. After about 30 ps (an hour), I went ahead and made a plot of the energies (Fig. 37).

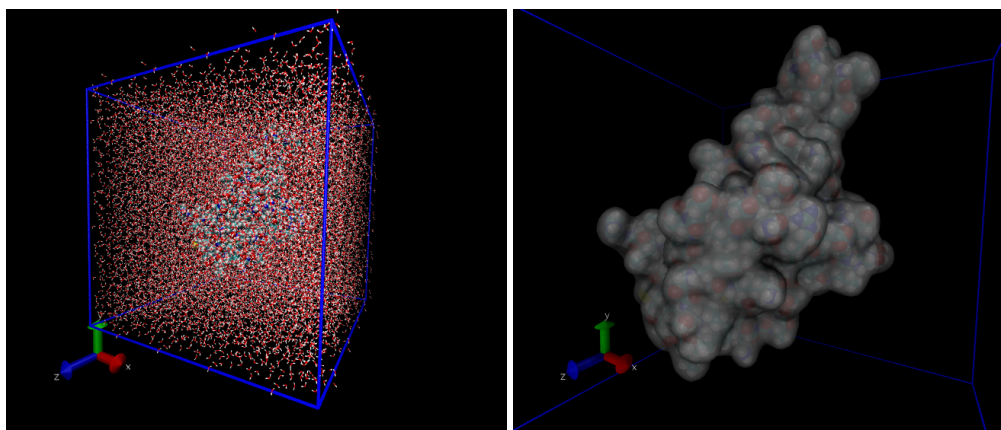
## 9.2 NAMD

The NAMD source code and many precompiled binaries are available from [the official download site](#) at the University of Illinois. Like Gromacs, NAMD is also available in many supercomputing centers. Compiling NAMD from source is not for the faint-hearted, so I won't cover that here; fortunately for us, the standard 'Linux-multicore' executable will work just fine in Ubuntu on WSL. (If you have a mac, you'll





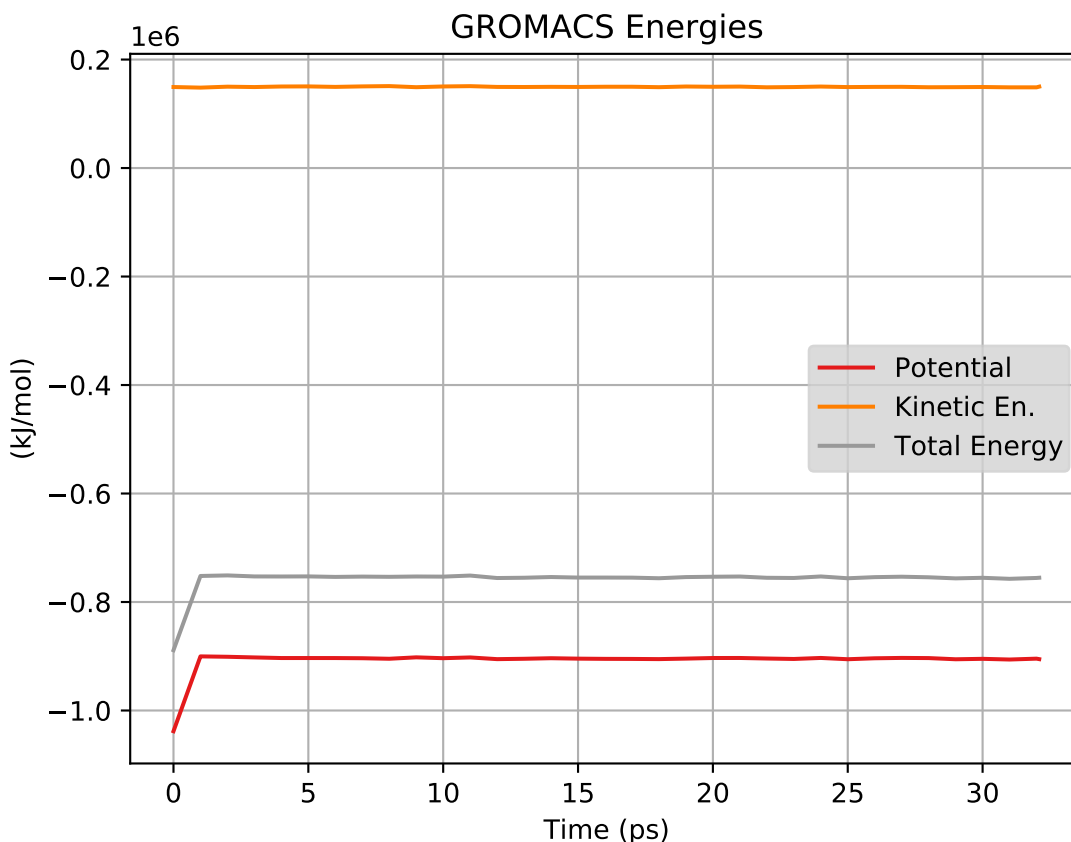
**Figure 35:** Density vs. time in an NPT MD simulation (1 bar, 300 K) of a box of 432 water molecules.



**Figure 36:** (Left) Simulation box for the SARS-CoV-2 RBD simulation showing all waters. (Right) Just the protein.

have to download a mac-specific executable.) For now, just download the tarball, extract it, and copy the namd2 executable to `/usr/local/bin/`:

```
$ tar zxf NAMD_2.14_Linux-x86_64-multicore.tar.gz
$ cd NAMD_2.14_Linux-x86_64-multicore/
```



**Figure 37:** Energies vs. time for a simulation of the SARS-CoV-2 S RBD in explicit water using Gromacs on a dinky little Dell laptop.

```
$ sudo cp namd2 /usr/local/bin/
```

If you have not done so already, download the Linux version of VMD and install it inside WSL as well – we need to use that rather than the native Windows version here.

To build a system to simulation with NAMD with the CHARMM force field, one must use the `psfgen` utility of VMD. This will generate an input PDB file along with a PSF file that contains all the topological information. I cannot stress enough how important it is to use the `psfgen` [manual](#); I'm only showing a little bit of here.

Let's revisit the SARS-CoV-2 S RBD for this example. Starting with the same coordinates extracted from the PDB entry and put into `my_7c8j.pdb`, we use a series of VMD scripts to build a solvated system.

First is `mkpsf.tcl` (it's name is not important; what's important is what is inside it). This script contains Tcl commands that implement PSF generation using VMD as an interpreter. To run it:

```
$ vmd -dispdev text -e mkpsf.tcl
```

This generates `my_7c8j.psf` which is an X-PLOR Protein Structure Format (PSF) file, along with the PDB file `my_78cj_processed.pdb`, which contains all the missing H's. It is customary to run a short energy minimization in vacuum just in case some of those H's are a little too close to each other by accident. As with Gromacs, this is done by invoking the main MD program with an input configuration

file requesting a minimization. Here, this is `vac.namd`. This can be run:

```
$ namd2 vac.namd >& vac.log &
$ tail -f vac.log
```

The `tail` command allows you to watch the simulation write to `vac.log` as it runs. Once this finishes, the next step is to solvate and neutralize. This is accomplished with another script, `solv.tcl`, which uses both the `solvate` and `autoionize` VMD packages:

```
$ vmd -dispdev text -e solv.tcl
```

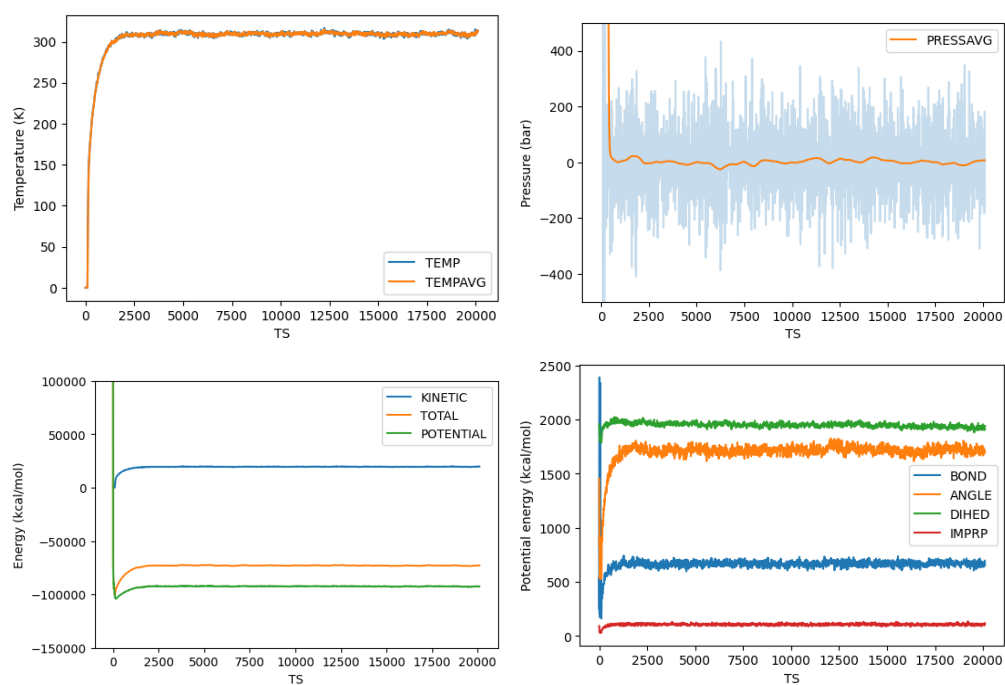
Similar to the `editconf` in Gromacs, `solv.tcl` computes and saves the resulting box dimensions, here in a file called `cell.inp`. Once this is done, we see the new PSF/PDB pair `my_7c8j_i.psf` and `my_7c8j_i.pdb` that contain the protein and all the solvent. These files, along with `cell.inp` and the CHARMM parameter files are now inputs for a 100-ps NPT MD simulation described by `prod.namd`.

```
$ namd2 +p8 prod.namd >& prod.log &
$ tail -f prod.log
```

This takes about 1.5 h on my Dell Latitude 7400 2-in-1 Intel Core-i7 laptop on WSL2 running Ubuntu. I ran this in a directory called `instructional-codes/my_work/namd`, so the directory `originals` is two up from the run directory. Fig. 38 shows plots of energies, temperature, and pressure, vs time-step from this MD simulation, generated by the following four python commands:

```
$ python3 ../../originals/plot_mdj_log.py -fmt NAMD \
-llogs prod.log -d 11 14 -ylabel "Temperature (K)" -o temp.png
$ python3 ../../originals/plot_mdj_log.py -fmt NAMD \
-llogs prod.log -d 18 -ylabel "Pressure (bar)" -o pres.png \
-ylim -500 500 -every 100
$ python3 ../../originals/plot_mdj_log.py -fmt NAMD \
-llogs prod.log -d 9 10 12 -ylabel "Energy (kcal/mol)" \
-ylim -150000 100000 -o kin-pot-tot.png
$ python3 ../../originals/plot_mdj_log.py -fmt NAMD \
-llogs prod.log -d 1 2 3 4 -ylabel "Potential energy (kcal/mol)" \
-o bonded.png
```

It should be stressed that system generation in both Gromacs and NAMD requires surmounting a significant learning curve if one wants to venture from only the simplest systems shown here. The Gromacs workflow is (to me) a bit more uniform than NAMD, which requires essentially a bunch of script-writing. (VMD offers an automated PSF builder that also works for simple systems.)



**Figure 38:** (Clockwise from top-left) Temperature; pressure; bonded potential energies; kinetic, potential and total energies, vs time-step from a NAMD simulation of solvated SARS-CoV-2 S RBD.

## 10 Free Energy Methods

In this lecture, we will consider aspects of computing free energy differences from molecular simulations. The umbrella terms “free-energy methods” or “free-energy calculations” cover a wide and growing array of methods for computing free energies [42]. Some, like the Widom test-particle insertion method and metadynamics, can be computed using a single simulation, while others require a series of simulations that carefully span the space between metastable states. In some cases, that space is an artificial mixture of two Hamiltonians, as in classical thermodynamics integration (TI), or it is a region of “feature space” for a single Hamiltonian that separates two actual metastable states, as in potential of mean force (PMF) calculations. In these remaining sections, we will touch on a few of these methods.

### 10.1 Excess Chemical Potential via the Widom Method

We recall that the free energy of the canonical ensemble, termed the Helmholtz free energy and denoted  $F$ , is defined by

$$F = -k_B T \ln Q(N, V, T) \quad (308)$$

$$= -k_B T \ln \left( \left\{ \frac{V^N}{\Lambda^{3N} N!} \right\} \left\{ V^{-N} \int d\mathbf{r}^N \exp[-\beta \mathcal{U}(\mathbf{r}^N)] \right\} \right) \quad (309)$$

$$= -k_B T \ln \left( \frac{V^N}{\Lambda^{3N} N!} \right) - k_B T \ln \left( \int d\mathbf{s}^N \exp[-\beta \mathcal{U}(\mathbf{s}^N; L)] \right) \quad (310)$$

$$\equiv F_{\text{id}}(N, V, T) + F_{\text{ex}}(N, V, T) \quad (311)$$

Here,  $F_{\text{id}}$  is the “ideal gas” free energy, and  $F_{\text{ex}}$  is the “excess” free energy. The chemical potential is defined as the change in free energy upon addition of a particle:

$$\mu = \left( \frac{\partial F}{\partial N} \right)_{VT} \quad (312)$$

For large  $N$ ,

$$\mu = -k_B T \ln(Q_{N+1}/Q_N) \quad (313)$$

$$\begin{aligned} &= -k_B T \ln \left( \frac{V/\Lambda^d}{N+1} \right) - k_B T \ln \left( \frac{\int d\mathbf{s}^{N+1} \exp[-\beta \mathcal{U}(\mathbf{s}^{N+1}; L)]}{\int d\mathbf{s}^N \exp[-\beta \mathcal{U}(\mathbf{s}^N; L)]} \right) \\ &= \mu_{\text{id}}(\rho) + \mu_{\text{ex}} \end{aligned} \quad (314)$$

which defines the excess chemical potential,  $\mu_{\text{ex}}$ . We see that we can express the quotient of configurational integrals in  $\mu_{\text{ex}}$  as an integration of the ensemble average of  $\Delta \mathcal{U} \equiv \mathcal{U}(\mathbf{s}^{N+1}) - \mathcal{U}(\mathbf{s}^N)$  over  $\mathbf{s}_{N+1}$ , the scaled coordinates of the  $(N+1)$ ’th particle, or “test” particle:

$$\mu_{\text{ex}} = -k_B T \ln \int d\mathbf{s}_{N+1} \langle \exp(-\beta \Delta \mathcal{U}) \rangle_N \quad (315)$$

This equation implies that we can measure  $\mu_{\text{ex}}$  by performing a brute force sampling of  $\exp(-\beta \Delta \mathcal{U})$  in an otherwise normal NVT MC simulation. That is, we can at frequent intervals in a normal MC program “create” a test particle with a position sampled uniformly in the box, compute  $\mathcal{U}(\mathbf{s}^{N+1}) - \mathcal{U}(\mathbf{s}^N)$ , and accumulate  $\exp(-\beta \Delta \mathcal{U})$ . This is the Widom method.

The code `mclj_widom.c` implements the Widom method for the Lennard-Jones fluid in an NVT simulation. Below is a code fragment for sampling  $\Delta\mathcal{U}$  using the Lennard-Jones pair potential 89:

```

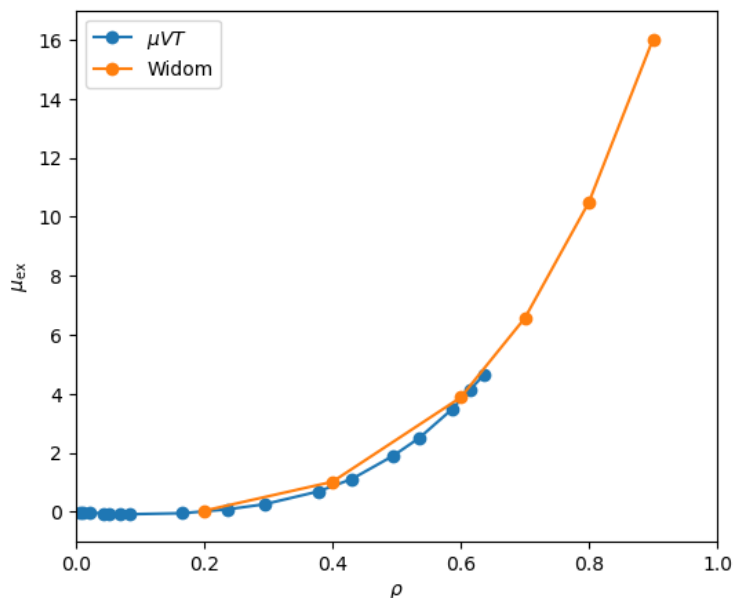
rx[N]=(gsl_rng_uniform(r)-0.5)*L;
ry[N]=(gsl_rng_uniform(r)-0.5)*L;
rz[N]=(gsl_rng_uniform(r)-0.5)*L;

for (j=0;j<N;j++) {
  dx = rx[N]-rx[j];
  dy = ry[N]-ry[j];
  dz = rz[N]-rz[j];
  r2 = dx*dx + dy*dy + dz*dz;
  r6i = 1.0/(r2*r2*r2);
  du += 4*(r6i*r6i - r6i);
}

```

The particle with index  $N$  is assumed to be the “test particle”; the other particles are indexed 0 to  $N - 1$ . In the first bit, the position of the test particle is generated as a uniformly random location inside a cubic box of side length  $L$ . Then we loop over the particles 0 to  $N - 1$  and accumulate  $\Delta\mathcal{U}$ .

Using the code `mclj_widom.c`, we can measure  $\mu_{\text{ex}}(\rho, T)$  in NVT MC simulations. This represents an alternate way of computing  $\mu_{\text{ex}}$  to that of  $\mu VT$  MC, in which  $\rho$  is an observable. In Fig. 39, I show  $\mu_{\text{ex}}$  vs.  $\rho$  at  $T = 3.0$  computed using both  $\mu VT$  MC and the NVT Widom method. The  $\mu VT$  simulations were initialized with 512 particles initially, while the Widom simulations were run with  $N = 216$  particles. Each point is an average of three independent calculations.



**Figure 39:**  $\mu_{\text{ex}}$  vs.  $\rho$  for the Lennard-Jones fluid at  $T = 3.0$  computed using a grand canonical  $\mu VT$  Monte Carlo simulation and an NVT simulation with the Widom sampling method.

It would be useful to know how to determine which of these apparently competing methods is best for computing  $\mu_{\text{ex}}$ . They are both similar in computational requirements (this is not further qualified here; if someone wants to make this comparison, he or she is welcome to do this as a project). On the one hand, we have an inherent limitation of the grand canonical simulation: one cannot specify the system

density exactly; rather it is an observable with some mean and fluctuations. The Widom method does allow one to specify the density precisely, and in this regard, it is probably more trustworthy in computing  $\mu_{\text{ex}}$ . On the other hand, the Widom method suffers the limitation that it is not generally applicable to systems with any potential energy function. For example, for hard-sphere systems, the Widom method would always predict that  $\mu_{\text{ex}}$  is 0, a clearly nonsensical answer. The “overlapping distribution method” of Bennett, discussed in Section 7.2.3 of F&S, offers a means to overcome this particular limitation. We do not cover this method in lecture, but you are encouraged to explore the overlapping distribution method on your own (maybe as a project).

## 10.2 Thermodynamic Integration

Thermodynamic integration is a conceptually simple, albeit expensive, way to calculate free energy differences from MC or MD simulations. In this example, we will consider the calculation (again) of chemical potential in a Lennard-Jones fluid at a given temperature and density, a task performed very well already by the Widom method (so long as the densities are not too high.) More details of the method can be found in the work of Tironi and van Gunsteren [43].

We begin with the relation derived in the book for a free energy difference,  $F_{II} - F_I$ , between two systems which are identical (same number of particles, density, temperature, etc.) *except* that they obey two different potentials. System I obeys  $\mathcal{U}_I$  and System II  $\mathcal{U}_{II}$ . To measure this free energy difference, we must integrate along a reversible path from I to II. So let us suppose that we can write a “metapotential” that uses a switching parameter,  $\lambda$ , to measure distance along this path. So, when  $\lambda = 0$ , we are in System I, and when  $\lambda = 1$  we are in System II. One way we might encode this (though this is not necessarily a general splitting, as we shall see below) is

$$\mathcal{U}(\lambda) = (1 - \lambda) \mathcal{U}_I + \lambda \mathcal{U}_{II} \quad (316)$$

Let us consider the canonical partition function for a system obeying a general potential  $\mathcal{U}(\lambda)$ :

$$Q(N, V, T, \lambda) = \frac{1}{\Lambda^{3N} N!} \int d\mathbf{r}^N \exp[-\beta \mathcal{U}(\lambda)] \quad (317)$$

Recalling that the free energy is given by  $F = -k_B T \ln Q$ , we can express the derivative of the Helmholtz free energy with respect to  $\lambda$ :

$$\left( \frac{\partial F(\lambda)}{\partial \lambda} \right)_{N, V, T} = -\frac{1}{\beta} \frac{\partial}{\partial \lambda} \ln Q(N, V, T, \lambda) \quad (318)$$

$$= -\frac{1}{\beta Q(N, V, T, \lambda)} \frac{\partial Q(N, V, T, \lambda)}{\partial \lambda} \quad (319)$$

$$= \frac{\int d\mathbf{r}^N (\partial \mathcal{U}(\lambda) / \partial \lambda) \exp[-\beta \mathcal{U}(\lambda)]}{\int d\mathbf{r}^N \exp[-\beta \mathcal{U}(\lambda)]}$$

The free energy difference between I and II is given by:

$$F_{II} - F_I = \int_{\lambda=0}^{\lambda=1} \left\langle \frac{\partial \mathcal{U}}{\partial \lambda} \right\rangle_{\lambda} d\lambda \quad (320)$$

where,  $\left\langle \frac{\partial \mathcal{U}}{\partial \lambda} \right\rangle_{\lambda}$  is the ensemble average of the derivative of  $\mathcal{U}$  with respect to  $\lambda$ .

To compute  $\mu_{\text{ex}}$ , we imagine two systems: System I has  $N - 1$  “real” particles, and 1 ideal gas particle, and system II has  $N$  real particles. The two free energies can be written:

$$F_{\text{I}} = F_{\text{id}}(N, V, T) + F_{\text{ex}}(N - 1, V, T) \quad (321)$$

$$F_{\text{II}} = F_{\text{id}}(N, V, T) + F_{\text{ex}}(N, V, T) \quad (322)$$

For large values of  $N$ , we see that  $\mu_{\text{ex}} = F_{\text{II}} - F_{\text{I}}$ . So, we have another route to compute  $\mu_{\text{ex}}$ . First, we tag a particle  $i_\lambda$ , call it the “ $\lambda$ -particle”, and apply the following modified potential to its pairwise interactions:

$$U_{\text{LJ},\lambda}(r; \lambda) = 4(\lambda^5 r^{-12} - \lambda^3 r^{-6}) \quad (323)$$

So, the total potential is given by

$$\mathcal{U} = \sum'_{i < j} U_{\text{LJ}}(r_{ij}) + \sum_i U_{\text{LJ},\lambda}(r_{i,i_\lambda}; \lambda) \quad (324)$$

where the prime denotes that we ignore particle  $i_\lambda$  in the sum. When  $\lambda = 1$ , all particles interact fully, and we have System II.

Next, we conduct many independent MC simulations at various values of  $\lambda$  and a given value of  $\rho$  and  $T$ , generating for each  $(T, \rho)$  a table of  $\langle \partial \mathcal{U} / \partial \lambda \rangle_\lambda$  vs.  $\lambda$  which can be integrated to yield a single value for  $\mu_{\text{ex}}$ . The code `mclj_ti.c` implements this sampling when a value for  $\lambda$  is specified. To demonstrate its use to compute  $\mu_{\text{ex}}$ , the following protocol was used:

1. Run 600,000-cycle NVT  $\lambda$ -MC simulations at given density  $\rho$  and temperature  $T$  (with  $N = 216$  particles) for values of  $\lambda \in \{0, 0.1, 0.2, \dots, 1\}$  to compute  $\langle d\mathcal{U} / d\lambda \rangle$ . Three independent simulations are run for each  $N$ - $\rho$ - $T$ - $\lambda$  point, and  $\langle d\mathcal{U} / d\lambda \rangle$  is an average over these three.
2. Use Simpson’s rule in `scipy.integrate.simpson` to numerically integrate  $\langle d\mathcal{U} / d\lambda \rangle$  vs.  $\lambda$  to obtain  $\mu_{\text{ex}}$ .

This turns out to be an expensive way to compute the chemical potential for a Lennard-Jones fluid, compared to the Widom method (Sec. 10.1) or grand canonical MC (Sec. 6.1), for at least low to moderate densities. At very high densities, however, particle insertion moves in grand canonical and Widom-method simulations become difficult. Fig. 40 shows a plot of  $\langle d\mathcal{U} / d\lambda \rangle$  vs.  $\lambda$  for various densities, all at  $T = 3.0$  (left), with values of  $\mu_{\text{ex}}$  found from integrating those curves shown together with the data from Fig. 39 showing  $\mu_{\text{ex}}$  from both grand canonical MC and the Widom method.

The curves of  $\langle d\mathcal{U} / d\lambda \rangle$  vs  $\lambda$  are not completely noise-free, but integrating each of these curves to produce a single value of  $\mu_{\text{ex}}$  produces values that are not too off from the grand canonical and Widom-method simulations.

### 10.3 The Method of Overlapping Distributions

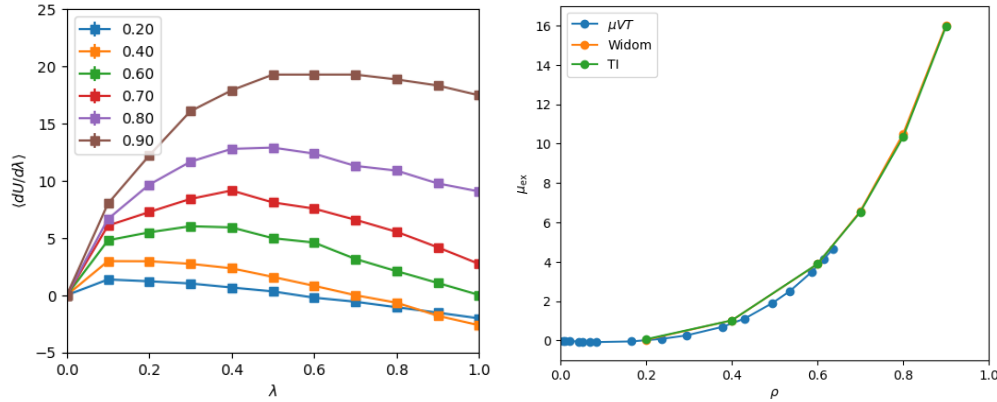
One interesting feature of the Widom method is that the only trial move is *insertion*; however, the free-energy difference between an  $N$ -particle system and an  $N + 1$ -particle system should not depend on which direction the trial moves take. If we imagine a “Widom real-particle removal” method, we’d write the chemical potential as

$$\mu = +k_B T \ln(Q_N / Q_{N+1}) = \mu_{\text{id}} + k_B T \ln \langle \exp(+\beta \Delta \mathcal{U}) \rangle \quad (325)$$

Sampling  $\langle \exp(+\beta \Delta \mathcal{U}) \rangle$  in a straightforward NVT MC simulation won’t work, however, because...

There is, however, a right way to use bidirectional energy changes to compute free-energy differences, termed the “overlapping distribution method” and attributed to Bennett [44]. Consider two





**Figure 40:** (left)  $\langle d\mathcal{U}/d\lambda \rangle$  vs.  $\lambda$  for the Lennard-Jones fluid for three values of  $\rho$  at  $T = 3.0$ , computed from MC simulations of 216 particles for  $6 \times 10^6$  cycles. Five independent simulations were run for each  $(\rho, \lambda)$ , and values of  $\langle d\mathcal{U}/d\lambda \rangle$  are averaged over these five. Standard deviations a smaller than the symbol size. (right) Overlay of  $\mu_{\text{ex}}$  vs  $\rho$  at  $T = 3.0$  using grand canonical MC, the Widom method, and thermodynamic integration.

systems 0 and 1, obeying potentials  $\mathcal{U}_0$  and  $\mathcal{U}_1$ , respectively. Let  $q_i$  be the scaled configurational integral of the Boltzmann factor:

$$q_i = \int d\mathbf{s}^N e^{-\beta\mathcal{U}_i} \quad (326)$$

We can then express the free energy difference between these systems as (assuming for simplicity they have the same volumes):

$$\beta\Delta F = -\ln \frac{q_1}{q_0} \quad (327)$$

Consider next we run an NVT MC simulation on  $\mathcal{U}_1$  and sample  $\Delta\mathcal{U} \equiv \mathcal{U}_1 - \mathcal{U}_0$ . Formally, the probability density of  $\Delta\mathcal{U}$  from this simulation is

$$p_1(\Delta\mathcal{U}) = \frac{\int d\mathbf{s}^N e^{-\beta\mathcal{U}_1} \delta(\mathcal{U}_1 - \mathcal{U}_0 - \Delta\mathcal{U})}{q_1} \quad (328)$$

$$= \frac{\int d\mathbf{s}^N e^{-\beta(\mathcal{U}_0 + \Delta\mathcal{U})} \delta(\mathcal{U}_1 - \mathcal{U}_0 - \Delta\mathcal{U})}{q_1} \quad (329)$$

$$= \frac{q_0}{q_1} e^{-\beta\Delta\mathcal{U}} \frac{1}{q_0} \int d\mathbf{s}^N e^{-\beta\mathcal{U}_0} \delta(\mathcal{U}_1 - \mathcal{U}_0 - \Delta\mathcal{U}) \quad (330)$$

$$= \frac{q_0}{q_1} e^{-\beta\Delta\mathcal{U}} p_0(\Delta\mathcal{U}) \quad (331)$$

In going from Eq. 329 to 330 we have used the fact that the Dirac delta function will only permit one value of  $\Delta\mathcal{U}$  to survive integration. Taking the log of both sides gives

$$\ln p_1(\Delta\mathcal{U}) = -\ln \frac{q_1}{q_0} - \beta\Delta\mathcal{U} + \ln p_0(\Delta\mathcal{U}) \quad (332)$$

$$= \beta(\Delta F - \Delta\mathcal{U}) + \ln p_0(\Delta\mathcal{U}) \quad (333)$$

This equation provides a way to estimate  $\Delta F$  at *any one value* of  $\Delta \mathcal{U}$  so long as good estimates of both  $p_1(\Delta \mathcal{U})$  and  $p_0(\Delta \mathcal{U})$  are available. That means we must be able to sample a sufficiently large domain of  $\Delta \mathcal{U}$  from both a simulation run on  $\mathcal{U}_1$  and *another* run on  $\mathcal{U}_0$ . That is, there must be a domain of  $\Delta \mathcal{U}$  where  $p_1$  and  $p_0$  overlap.

Bennett<sup>[44]</sup> suggests the following transformation of  $p_1$  and  $p_0$  to permit easy calculation of  $\Delta F$ . Letting

$$f_0(\Delta \mathcal{U}) \equiv \ln p_0(\Delta \mathcal{U}) - \frac{\beta \Delta \mathcal{U}}{2}, \quad \text{and} \quad (334)$$

$$f_1(\Delta \mathcal{U}) \equiv \ln p_1(\Delta \mathcal{U}) + \frac{\beta \Delta \mathcal{U}}{2} \quad (335)$$

gives

$$\beta \Delta F = f_1(\Delta \mathcal{U}) - f_0(\Delta \mathcal{U}). \quad (336)$$

This means we can measure  $f_1$  and  $f_0$  in *separate* simulations, and then observe a constant offset between them to be  $\beta \Delta F$ .

Suppose we now take the example of system 1 with  $N$  real particles and system 0 with  $N - 1$  real particles and one ideal-gas particle. The free-energy change from 0 to 1 is the excess chemical potential (yet again!). Fig. 41 illustrates using Bennett's method to compute  $\mu_{\text{ex}}$  of the Lennard-Jones fluid at  $T = 1.2$  for a few different densities. For each density, two simulations were run: simulation-0 computes the distribution of  $\Delta \mathcal{U}$ , the energy associated with converting the ideal-gas particle to a real particle, while simulation-1 computes the same distribution for converting a randomly chosen particle from being an ideal-gas particle to being a real particle. This latter  $\Delta \mathcal{U}$  is easily computed using the single-particle energy function  $e_{\text{i}}$ . It is important to note that the direction of the  $\Delta$  is from ideal-gas to real for *both* simulations. Note too that since we sample  $\Delta \mathcal{U}$  for particle insertion in simulation-0, we can just as easily compute the expectation  $\langle \exp(-\beta \Delta \mathcal{U}) \rangle$  and thereby get a direct estimate of  $\beta \mu_{\text{ex}}$ .

At the moderately low density of  $\rho = 0.7$ , we see a clear constant offset  $\beta \mu_{\text{ex}}$  between  $f_0$  and  $f_1$ . Note clear agreement between the offset over a finite-size domain of  $U$  and the single-point Widom estimate. For the somewhat higher density of 0.9, the offset is a bit noisier, reflecting somewhat poorer sampling. For the highest density, the sampling in simulation-0 is so poor that it is nearly impossible to detect an overlap domain.

#### 10.4 Histogram Reweighting

To further generalize our discussion of free-energy methods, it is convenient to introduce the concept of the “order parameter”  $z$  which is computed by a mapping function  $\theta(\mathbf{r}^N)$ , which is generally just any function of configuration (for example,  $\Delta \mathcal{U} = \mathcal{U}_1 - \mathcal{U}_0$ ). For a system obeying the potential  $\mathcal{U}_0$ , we can define the order parameter probability density as

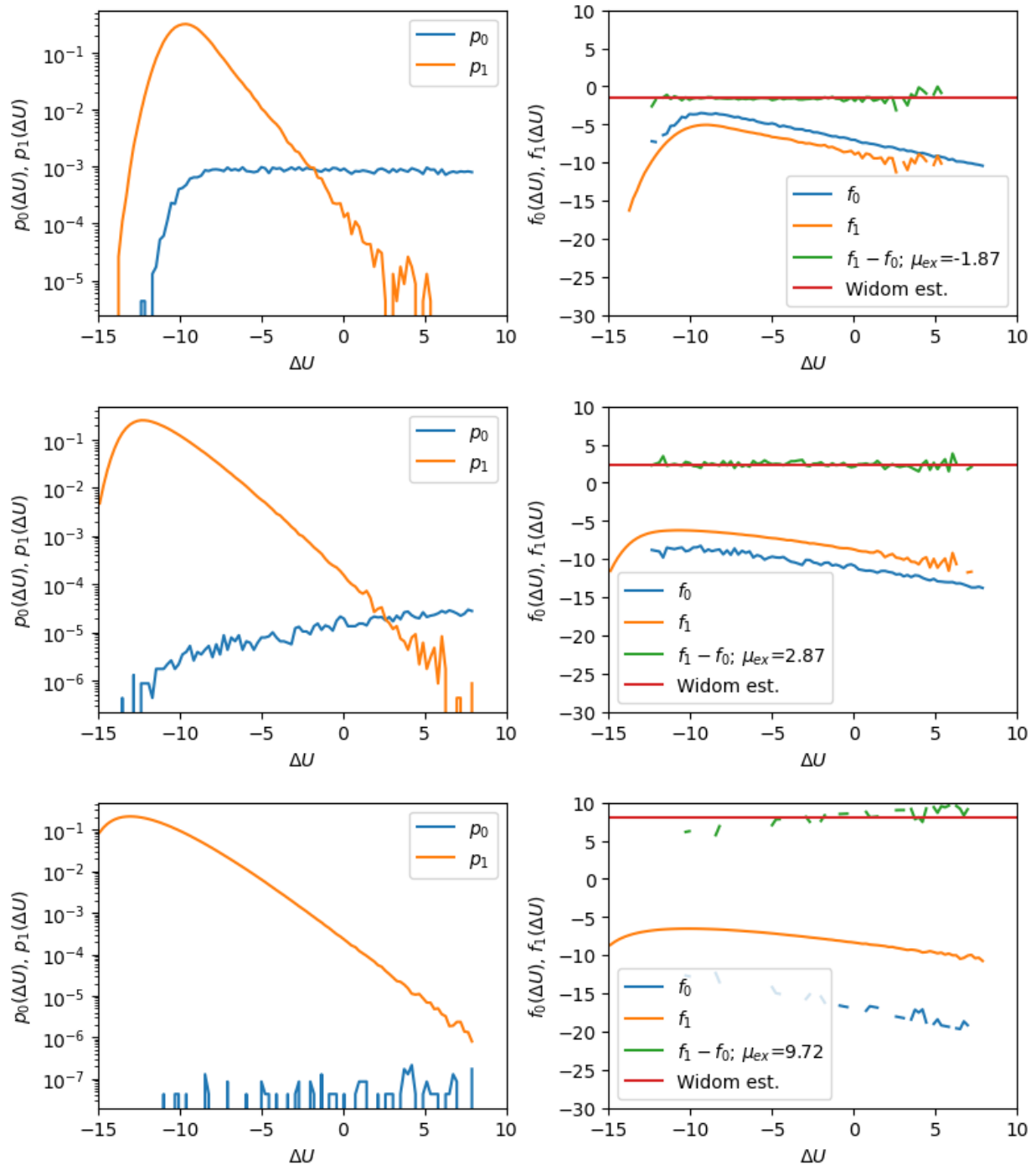
$$p_0 = \frac{\int \mathbf{r}^N e^{-\beta \mathcal{U}_0} \delta [z - \theta(\mathbf{r}^N)]}{Z_0} \quad (337)$$

where we are using the shorthand

$$Z_0 = \int \mathbf{r}^N e^{-\beta \mathcal{U}_0} \quad (338)$$

to represent (now the unscaled) configurational integral of the Boltzmann factor. The “Landau” free energy is then expressed as

$$F_{\text{Landau}}(z) = -k_B T \ln p_0(z) \quad (339)$$



**Figure 41:**  $p_0$  and  $p_1$  vs  $\Delta\mathcal{U}$  (left), and  $f_0$ ,  $f_1$ ,  $f_1 - f_0$  vs  $\Delta\mathcal{U}$  (right) computed using NVT MC simulations at  $T = 1.2$  of system-0 and system-1, for densities of 0.7 (top), 0.9 (middle), and 1.0 (bottom). Estimates of  $\beta\mu_{ex}$  from Widom test-particle insertion are shown as red horizontal lines in the plots on the right.

We'd generally like to be able to compute  $p_0$  or, alternatively,  $F_{\text{Landau}}$ , but as we have already seen, it is often impossible to perform adequate sampling over an entire range of order parameter. Suppose we restrict sampling to a region in order parameter space using a "bias potential"  $W_i(\theta(\mathbf{r}^N))$ . Under the

action of this bias, we can compute directly a biased probability density:

$$p_i(z) = \frac{\int \mathbf{r}^N e^{-\beta[\mathcal{Z}_0 + W_i(z)]} \delta[z - \theta(\mathbf{r}^N)]}{Z_i} \quad (340)$$

where we use the shorthand

$$Z_i = \int \mathbf{r}^N e^{-\beta[\mathcal{Z}_0 + W_i(\theta(\mathbf{r}^N))]} \quad (341)$$

Note that in Eq. 340, we have made use of the fact that the Dirac delta function only permits  $\theta(\mathbf{r}^N)$  to equal  $z$ , so we do not need to express the argument of  $W_i$  as  $\theta(\mathbf{r}^N)$  in the numerator. This is important, because it allows us to express the *unbiased* probability density:

$$p_0(z) = \exp(+\beta W_i(z)) \frac{Z_i}{Z_0} p_i(z) \quad (342)$$

Let's now consider that we compute  $p_i$  by histogramming into bins of constant width  $\Delta z$ . Let's call the histogram  $H_i$  and the total number of hits in the histogram  $M_i$ . Then one way to write the relation between the approximate probability density and the histogram is

$$p_i \Delta z = \frac{H_i}{M_i} \quad (343)$$

Now we imagine we can use several different  $W_i$ 's together, each of which focus sampling on a particular region of order-parameter space, and we propose that the unbiased probability density can be constructed by a superposition of the biased probabilities:

$$p_0(z) = \sum_{i=1}^n a_i(z) \exp(+\beta W_i(z)) \frac{Z_i}{Z_0} p_i(z) \quad (344)$$

$$\Rightarrow p_0(z) \Delta z = \sum_{i=1}^n a_i(z) \exp(+\beta W_i(z)) \frac{Z_i}{Z_0} \frac{H_i(z)}{M_i} \quad (345)$$

subject to

$$\sum_{i=1}^n a_i(z) = 1 \quad \text{for all } z \quad (346)$$

In Eq. 345, we have supposed that  $p_0 \Delta z$  is a normalized histogram with the same bin size  $\Delta z$  as all  $H_i$ .

Via minimization of the squared fluctuations of  $p_0(z)$  (since it will fluctuate in a simulation), one can arrive at an expression for  $a_i$  (math not shown):

$$a_i(z) = \alpha(z) \exp[-\beta W_i(z)] M_i \frac{Z_0}{Z_i} \quad (347)$$

where the normalization condition is met by

$$\alpha(z) = \frac{1}{\sum_{i=1}^n \exp[-\beta W_i(z)] M_i \frac{Z_0}{Z_i}} \quad (348)$$

This gives a reweighted histogram of

$$p_0(z)\Delta z = \frac{\sum_{i=1}^n H_i}{\sum_{i=1}^n \exp[-\beta W_i(z)] M_i \frac{Z_0}{Z_i}} \quad (349)$$

Now, let's define  $F_i$  via

$$\exp \beta F_i = -\frac{Z_i}{Z_0} \quad (350)$$

$$= \frac{\int d\mathbf{r}^N e^{-\beta(\mathcal{U}_0 + W_i)}}{Z_0} \quad (351)$$

$$= \frac{\int dz \int \mathbf{r}^N e^{-\beta(\mathcal{U}_0 + W_i)} \delta(\theta(\mathbf{r}^N) - z)}{Z_0} \quad (352)$$

$$= \int dz e^{-\beta W_i(z)} \frac{\int \mathbf{r}^N e^{-\beta \mathcal{U}_0} \delta[z - \theta(\mathbf{r}^N)]}{Z_0} \quad (353)$$

$$= \int dz p_0(z) e^{-\beta W_i(z)} \quad (354)$$

$$\Rightarrow F_i = -\frac{1}{\beta} \ln \int dz p_0(z) e^{-\beta W_i(z)} \quad (355)$$

This allows us to write the unbiased histogram as

$$p_0(z)\Delta z = \frac{\sum_{i=1}^n H_i}{\sum_{j=1}^n \exp[-\beta(W_j(z) - F_j)] M_j} \quad (356)$$

Given that we have a finite bin width, the integration in Eq. 355 can be approximated as

$$F_j = -\frac{1}{\beta} \ln \sum_{k=0}^{n_{\text{bins}}-1} p_0(z_k) \exp[-\beta W_j(z_k)] \Delta z \quad (357)$$

where  $z_k = z_{\text{min}} + (k + \frac{1}{2})\Delta z$ , with  $z_{\text{min}}$  the lower bound of the histogram domain and  $n_{\text{bins}}$  is the number of bins.

Eq. 356 and 357 are called (by some) the WHAM equations (for Weighted Histogram Analysis Method [45]). They can be solved iteratively to yield an unbiased probability provided with a set of biased histograms  $H_i$  obtained from running MC simulations on  $\mathcal{U}_0 + W_i$ . The standard WHAM approach is

- Set  $F_j = 0$  for all  $j$ .
- Compute the first estimate of  $p_0$  using Eq. 356 from the  $H_i$ 's and  $M_i$ 's.
- Compute a new estimate for the  $F_j$ 's using Eq. 357.
- Repeat the last two steps until the  $F_j$ 's stop changing within some tolerance.

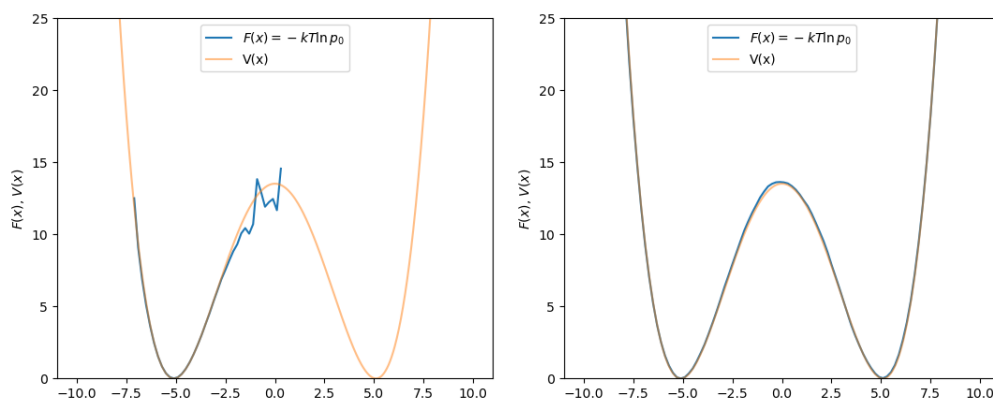
As an example, consider a single degree of freedom  $x$  (our “particle”) moving under Brownian dynamics (BD) on a quartic two-well potential:

$$\dot{x} = -\frac{1}{\gamma} \frac{dV}{dx} + \sqrt{2k_B T / \gamma} \eta, \quad \text{where} \quad (358)$$

$$V(x) = a(x^2 - 1)^2 + bx^2 + cx \quad (359)$$

Here,  $\gamma$  is the BD friction,  $T$  is the temperature, and  $\eta$  is a random variate centered on zero with unit variance. A symmetrical two-well potential with a barrier at  $x = 0$  of about 13 can be had by using  $a = 0.02$ ,  $b = -1$ , and  $c = 0$ . Since there is only one degree of freedom, it can easily be shown that  $F(x) = V(x)$ . So, if we run BD, sampling  $x$  and then histogramming it into  $p_0$ , then we should see that  $F = -k_B T \ln p_0$  and  $V$  agree. Computing  $F$  from a directly histogrammed variable is sometimes called “Boltzmann inversion”.

Fig. 42 shows the calculation of  $F$  using this approach, with BD on  $V$  using the code `bd-w.c`. We show that sampling with a reduced temperature of 10 requires about 2 billion BD timesteps to reproduce  $V$ . If we reduce the temperature to 1, the large barrier between the two wells is not surmounted and  $V$  is therefore not correctly reproduced.



**Figure 42:** Boltzmann inversion from 1-D Brownian dynamics simulations obeying the potential  $V$  in Eq. 359. (Left) BD run at  $T = 1.0$ ; (right) BD run at  $T = 10.0$ . Each simulation was run for  $2 \times 10^9$  steps of length  $1 \times 10^{-3}$ .

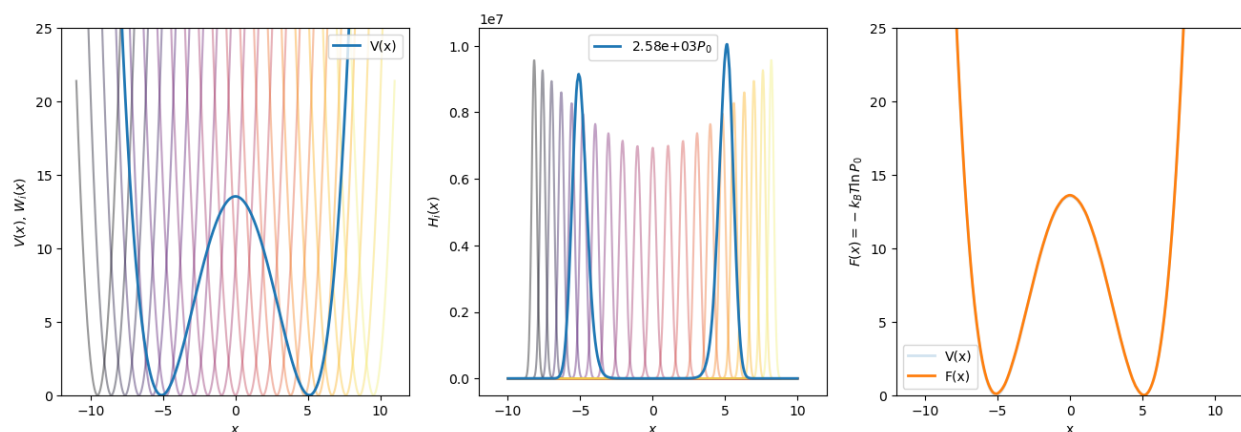
Fig. 43 shows that  $V$  can be reproduced at  $T$  of 1.0 using WHAM using 21 windows and harmonic bias potentials  $W_i$  spaced uniformly along  $z$ , each with a  $k$  of 20. That is, each bias potential is of the form

$$W_i(z) = \frac{1}{2} k (z - z_{0,i})^2 \quad (360)$$

where  $z_{0,i}$ ’s are uniformly spaced between -10 and 10. (Actually, -10 and 10 are the domain boundaries; this domain of size 20 is divided into 21 windows, and each  $z_0$  is in the *center* of its window. So,  $z_{0,0} = -10 + 0.5(20/21) = -9.5238$ , etc. An odd number of windows is a good idea so that we can better resolve the tippy top of the barrier.)

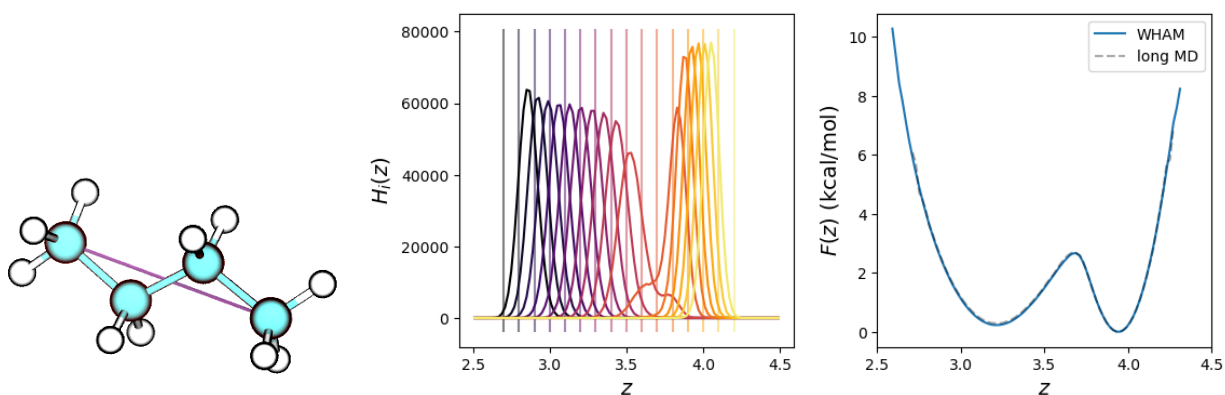
Here, only two million BD steps were run per window, for a total of 42 million steps. This means this WHAM dataset required more than twenty times less BD sampling at a temperature of 1.0 than the brute force sampling approach could at a temperature of 10.0.

As a second example of WHAM, consider MD simulation of butane molecule in vacuum at  $T = 310$  K. One order parameter we may consider here is the  $C_1$ - $C_4$  distance. When the molecule is in *trans*, the  $C_1$ - $C_4$  distance is about 4.5 Å, while when it is in *gauche*, the  $C_1$ - $C_4$  distance is about 3.2 Å. We expect there to be a small free-energy barrier between these two states when characterized using the  $C_1$ - $C_4$



**Figure 43:** WHAM results from BD simulations on  $V$  from Eq. 359. (Left)  $V(x)$  and all  $W_i(x)$ ; each  $z_0$  is uniformly spaced along  $z$ , and  $k$  is 20. (Center) All biased histograms  $H_i(x)$  from BD simulations of  $2 \times 10^6$  steps at  $T = 1.0$ , with a scaled unbiased histogram  $p_0$  (scaled for visibility). (Right) Reconstructed  $F(x)$  and  $V(x)$  (again).

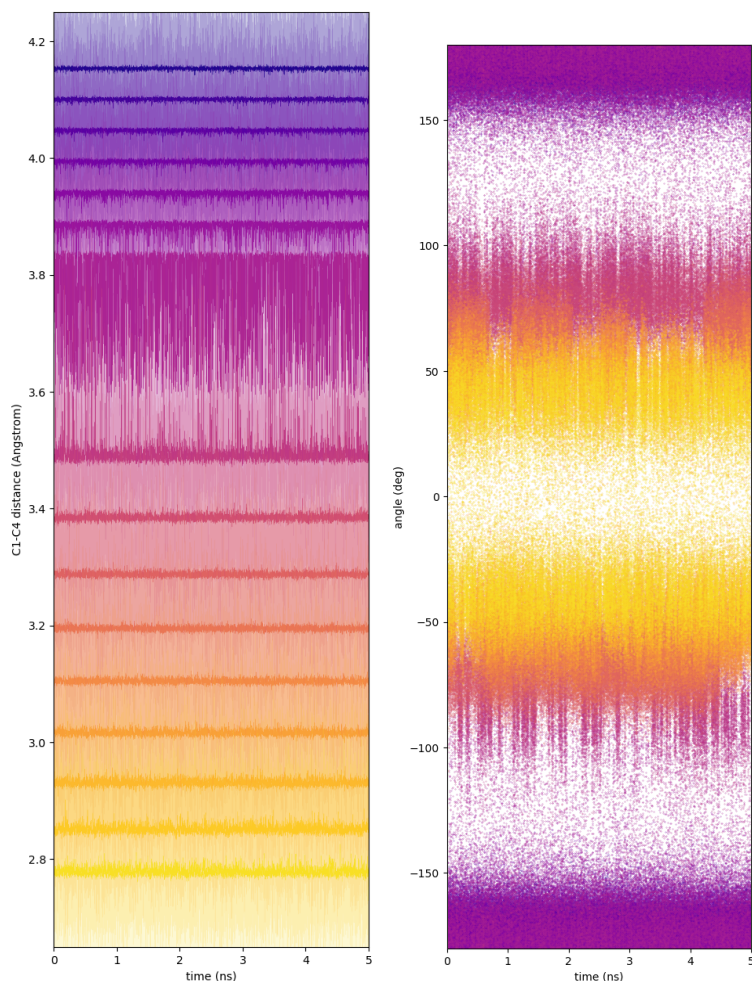
distance. Fig. 44 shows results of both WHAM and long MD simulations, showing indeed that the *trans* and *gauche* states are separated by a small barrier of about 2 kcal/mol at 310 K, easily surmountable in MD. WHAM reconstruction of  $F$  from 20 biased histograms generated from MD restrained using harmonic window potentials with spring constants of  $100 \text{ kcal/mol-Å}^2$  shows perfect agreement with the Boltzmann-inverted result.



**Figure 44:** (Left) A butane molecule with the  $C_1-C_4$  distance indicated by a purple line. (Center) Biased histograms from window-restrained MD simulations of butane in vacuum at 310 K. Windows are distributed uniformly along the  $C_1-C_4$  distance variable. (Right) Free energy vs  $C_1-C_4$  distance computed using Boltzmann inversion from a long MD simulation (grey dashes) and from WHAM (blue solid).

Butane is an almost trivially simple case; the  $C_1-C_4$  distance is relatively easily sampled in standard MD at 310 K in few million time-steps. This is of course evident because we can generate a Boltzmann-inverted free energy directly from the histogram of this distance generated by MD. But it is important to note that the order parameter here, the  $C_1-C_4$  distance, has a small amount of degeneracy: every value of this parameter except for its minimum and maximum has two major realizations determined by positive and negative senses of the dihedral angle. (There are of course some minor realizations due to angle stretching.) That means that each window's simulation should sample both the positive and negative regions of the order parameter space, and we must take care that the window potential is not

so strong that these dihedral angle fluctuations cannot occur; if so, we would undersample the gauche states for sure. Fig. 45 shows traces of the C1-C4 distance for each window and corresponding traces of the dihedral angle; clearly for this value of  $k$ , each window is indeed able to sample both positive and negative values of the dihedral angle.



**Figure 45:** (Left) C1-C4 distance vs. time for each of 16 windows. (Right) Dihedral angle vs. time for each of 16 windows.

### 10.5 Adaptive Free-Energy Methods

The histogram reweighting approach has a lot of parameters that have to be optimized in order to generate a reliable  $F$ : the number of windows, the spring constant for the window potentials, how the windows are spaced, how much sampling in each window, and more. This is often a problem because without knowing something about  $F$  it is hard to guess the best set of WHAM parameters. This has led to the development of “adaptive” approaches that aim to overcome this supposed weakness of histogram reweighting.

The two most well-known adaptive biasing approaches are metadynamics [46] and the adaptive-biasing forces (ABF) method [47].



## 10.5.1 Metadynamics

### 10.5.1.1 Original Metadynamics

In metadynamics, a time-dependent bias potential is “grown” during the course of the simulation that acts to enhance sampling of the order parameter [48]. Rather than confining to local regions of order parameter space as in umbrella sampling, the metadynamics potential pushes the system away from easily sampled regions of order parameter space. This bias can be expressed:

$$V_b(\theta, t) = w \sum_{t' < t} \exp \left( -\frac{[\theta(t') - \theta(t)]^2}{2\sigma^2} \right) \quad (361)$$

Here,  $w$  is a weight of each Gaussian kernel deposited, and  $\sigma$  is its width. Apart from them, another key parameter in running metadynamics is how frequently a new Gaussian kernel is added, i.e., what is the list of values for  $t'$ ? Apart from an irrelevant constant, the free energy along the order parameter is the time-average of the bias potential

$$F(\theta) = -\frac{1}{t_f - t_i} \sum_{t=t_i}^{t_f} V_b[\theta(t)] \quad (362)$$

NAMD includes native support for metadynamics using the `colvars` module. By default, kernels are deposited every 1,000 steps. To illustrate metadynamics, we return to the system of a single molecule of butane, this time at 273 K. It requires a 100-million time-step MD simulation to generate a smooth histogram for the C1-C4 distance at this temperature. Fig. 46 shows the free energy vs C1-C4 distance computed using a 10-million time-step metadynamics simulation for which  $w = 0.1$  kcal/mol, and  $\sigma = 0.1$  Å. We see excellent reconstruction of the true free energy at a much lower computational cost with metadynamics.

This  $10^7$ -time-step metadynamics simulation deposited 10,000 Gaussian kernels in total. Generally, it is most efficient for the simulation to keep track of the bias potential on a grid rather than as an explicit sum of Gaussians. Here, the order-parameter line was divided into increments of 0.02 Å between 1.5 and 5.5 Å, or roughly 400 points. Fig. 47 shows the evolution of the bias potential  $V_b(t)$  from this simulation.

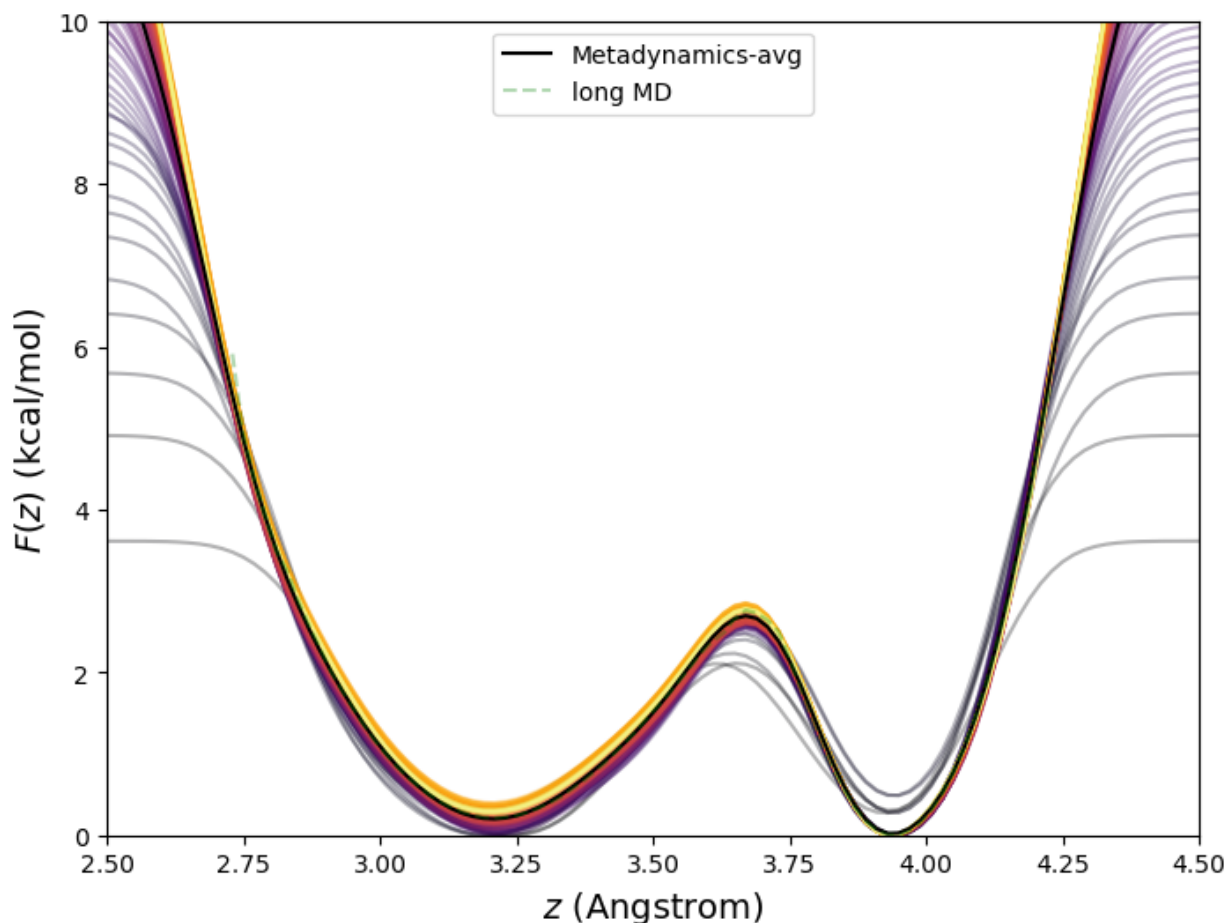
The accuracy of metadynamics is fairly sensitive to  $w$  and  $\sigma$ . Fig. ?? shows free energies for the butane system at 273 K computing using metadynamics (and the long MD for reference) using various combinations of  $w$  and  $\sigma$ , all for  $10^7$  steps. Among those considered, it appears  $w = 0.1$  kcal/mol, and  $\sigma = 0.1$  Å are the best choices. Generally, smaller  $w$  will yield more accurate free energies, but at larger computational cost.

### 10.5.1.2 Well-Tempered Metadynamics

In the well-tempered variant [49] of metadynamics, The weight  $w$  is augmented with a Boltzmann factor that diminishes exponentially as  $V_b$  builds up:

$$V_b(\theta, t) = w \sum_{t' < t} \exp \left[ -\frac{V_b(\theta(t'))}{\Delta T} \right] \exp \left( -\frac{[\theta(t') - \theta(t)]^2}{2\sigma^2} \right) \quad (363)$$

$\Delta T$  is the so-called “bias temperature”, and it acts to diminish the contribution to  $V_b$  at any  $\theta$  as time progresses, leading to a converged bias potential. The free energy is reconstructed by an inversion of



**Figure 46:** Free energy in kcal/mol vs. C1-C4 distance in Å, for butane in vacuum at 273 K computed using MD (green dash) and metadynamics (black solid). The MD simulation was run for  $10^8$  timesteps, and the metadynamics for  $10^7$ . Intermediate values of the metadynamics free energy are also shown color-coded from purple (early) to yellow (late). The final metadynamics free energy is the average over all free-energy snapshots (i.e., it is the time-average negative bias potential).

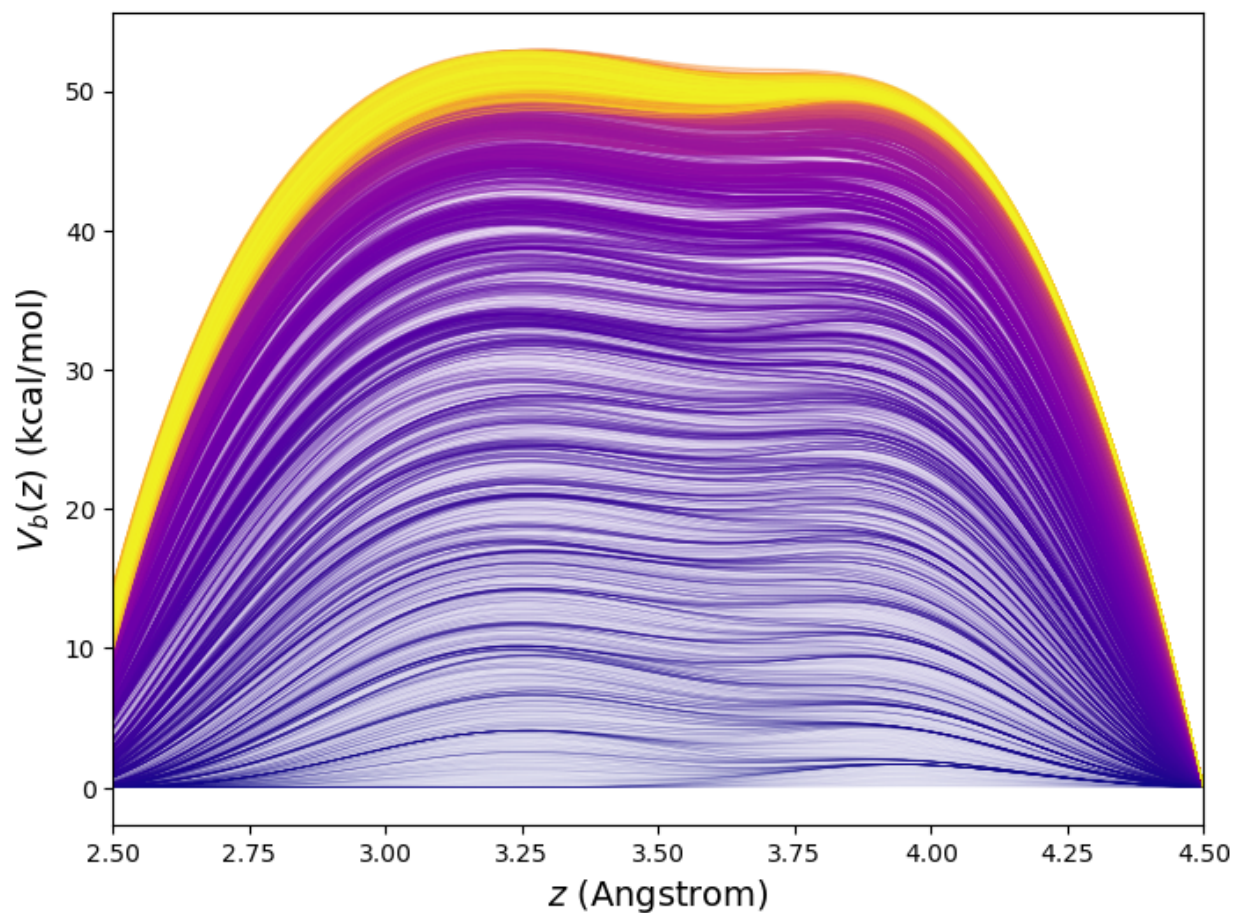
the converged bias potential that requires the bias temperature:

$$F = -\frac{T + \Delta T}{\Delta T} V_b \quad (364)$$

Fig. 48 shows the free energy vs. C1-C4 distance for butane at 273 K computed using well-tempered metadynamics with parameters identical to the standard metadynamics run of the previous section but with a bias temperature of 1000 K.

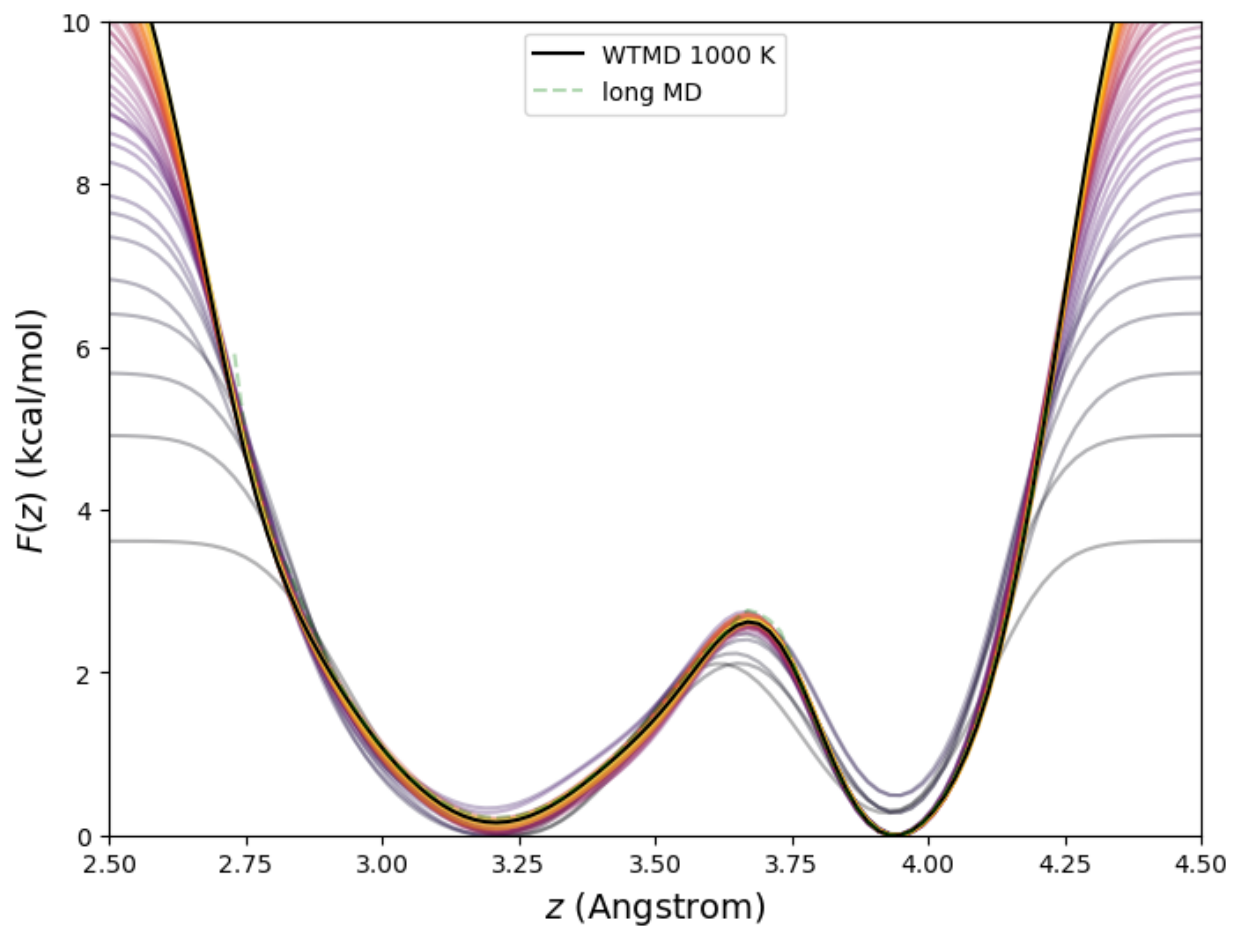
### 10.5.2 ABF

The adaptive biasing force method is based on recovery of free energies as functions of order parameters via thermodynamic integration of mean forces. These mean forces are traditionally computed using MD simulations restrained (or constrained, depending) to particular values of the order parameter. Indeed, one way of doing this is tethering the system to a reference point with a harmonic bias potential, just as we did for the histogram reweighting approach above. In the TI formalism, however, it can be shown that the *gradient* of the free energy along order parameter is

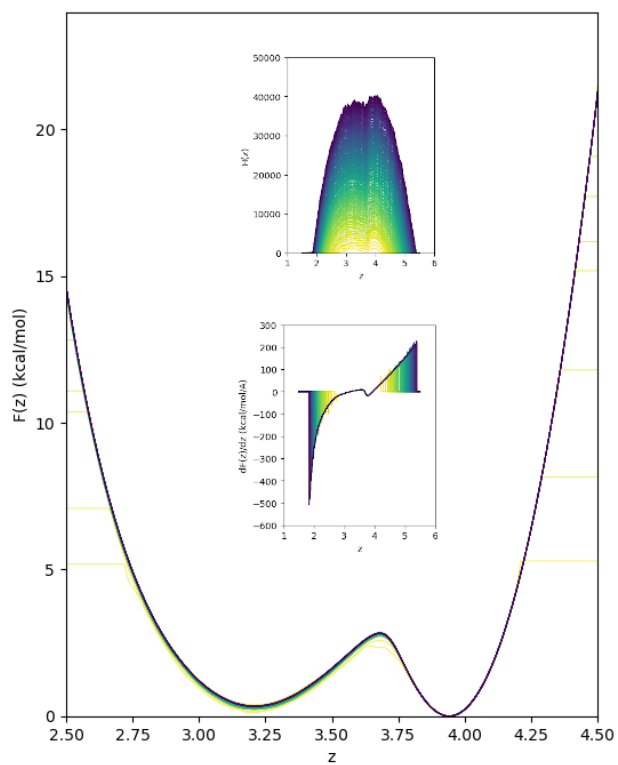


**Figure 47:** Evolution of the bias potential from the  $10^7$ -step metadynamics simulation of butane along the C1-C4 distance. The instantaneous bias potential is drawn every  $10^5$  time-steps and shifted to its current minimum value. Each curve is drawn with an  $\alpha$  of 0.2, so where the curves appear opaque signifies a temporarily static portion of the bias as kernels are being deposited elsewhere.

Fig. 49 shows the evolution of the PMF, along with the order-parameter histogram and free-energy gradients, from a single ABF simulation of butane at 273 K using NAMD.



**Figure 48:** Free energy in kcal/mol vs. C1-C4 distance in Å, for butane in vacuum at 273 K computed using MD (green dash) and well-tempered metadynamics (black solid). The MD simulation was run for  $10^8$  timesteps, and the metadynamics for  $10^7$ . The bias temperature for well-biased metadynamics was 1000 K. Intermediate values of the metadynamics free energy are also shown color-coded from purple (early) to yellow (late). The final metadynamics free energy is the average over all free-energy snapshots (i.e., it is the time-average negative bias potential).



**Figure 49:** Potentials of mean force converging to a final form in an ABF simulation of butane at 273 K, where the order parameters  $z$  is the C1-C4 distance. Insets show the histogram of  $z$  (top) and the convergence of the free-energy gradients (bottom).

**References**

- [1] D. Frenkel and B. Smit. *Understanding Molecular Simulation: From Algorithms to Applications*. Academic Press, San Diego, 2 edition, 2002.
- [2] M. P. Allen and D. J. Tildesley. *Computer Simulation of Liquids*. Oxford University Press, New York, 1987.
- [3] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953. doi: 10.1063/1.1699114. URL <https://doi.org/10.1063/1.1699114>.
- [4] Loup Verlet. Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. *Physical Review*, 159(1):98–103, 7 1967. ISSN 0031-899X. doi: 10.1103/PhysRev.159.98. URL <https://link.aps.org/doi/10.1103/PhysRev.159.98>.
- [5] William C. Swope, Hans C. Andersen, Peter H. Berens, and Kent R. Wilson. A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters. *The Journal of Chemical Physics*, 76(1):637–649, 1982. doi: 10.1063/1.442716. URL <https://doi.org/10.1063/1.442716>.
- [6] M. Tuckerman, B. J. Berne, and G. J. Martyna. Reversible multiple time scale molecular dynamics. *The Journal of Chemical Physics*, 97(3):1990, 1992. ISSN 00219606. doi: 10.1063/1.463137. URL <http://link.aip.org/link/JCPSA6/v97/i3/p1990/s1&Agg=doi>.
- [7] H. Flyvbjerg and H. G. Petersen. Error estimates on averages of correlated data. *The Journal of Chemical Physics*, 91(1):461–466, 1989. doi: 10.1063/1.457480. URL <https://doi.org/10.1063/1.457480>.
- [8] Edward J. Maginn, Richard A. Messerly, Daniel J. Carlson, Daniel R. Roe, and J. Richard Elliott. Best Practices for Computing Transport Properties 1. Self-Diffusivity and Viscosity from Equilibrium Molecular Dynamics [Article v1.0]. *Living Journal of Computational Molecular Science*, 1(1):1–20, 2019. doi: 10.33011/livecoms.1.1.6324.
- [9] H. J. C. Berendsen, J. P. M. Postma, W. F. van Gunsteren, A. DiNola, and J. R. Haak. Molecular dynamics with coupling to an external bath. *The Journal of Chemical Physics*, 81(8):3684–3690, 10 1984. ISSN 0021-9606. doi: 10.1063/1.448118. URL <http://aip.scitation.org/doi/10.1063/1.448118>.
- [10] Hans C. Andersen. Molecular dynamics simulations at constant pressure and/or temperature. *The Journal of Chemical Physics*, 72(1980):2384, 1980. ISSN 00219606. doi: 10.1063/1.439486. URL <http://scitation.aip.org/content/aip/journal/jcp/72/4/10.1063/1.439486>.
- [11] GS Grest and K Kremer. Structure of many arm star polymers: a molecular dynamics simulation. *Macromolecules*, 20:1376–1383, 1987. URL <http://pubs.acs.org/doi/abs/10.1021/ma00172a035>.
- [12] P Nikunen. How would you integrate the equations of motion in dissipative particle dynamics simulations? *Computer Physics Communications*, 153(3):407–423, 7 2003. ISSN 00104655. doi: 10.1016/S0010-4655(03)00202-9. URL <http://linkinghub.elsevier.com/retrieve/pii/S0010465503002029>.

- [13] Thomas Soddemann, Burkhard Dünweg, and Kurt Kremer. Dissipative particle dynamics: A useful thermostat for equilibrium and nonequilibrium molecular dynamics simulations. *Physical Review E*, 68(4):1–8, 10 2003. ISSN 1063-651X. doi: 10.1103/PhysRevE.68.046702. URL <http://link.aps.org/doi/10.1103/PhysRevE.68.046702>.
- [14] P. P. Ewald. Die Berechnung optischer und elektrostatischer Gitterpotentiale. *Annalen der Physik*, 369(3):253–287, 1921. ISSN 00033804. doi: 10.1002/andp.19213690304. URL <http://doi.wiley.com/10.1002/andp.19213690304>.
- [15] Markus Deserno and Christian Holm. How to mesh up Ewald sums. I. A theoretical and numerical comparison of various particle mesh routines. *The Journal of chemical physics*, 109(18):7678, 1998. URL <http://link.aip.org/link/?JCP/109/7678/1>.
- [16] Markus Deserno. *3 Efficient electrostatics : Ewald Sum and Particle Mesh Ewald Algorithms*. PhD thesis, 1999.
- [17] Judith A. Harrison, J. David Schall, Sabina Maskey, Paul T. Mikulski, M. Todd Knippenberg, and Brian H. Morrow. Review of force fields and intermolecular potentials used in atomistic computational materials research. *Applied Physics Reviews*, 5(3), 2018. ISSN 19319401. doi: 10.1063/1.5020808. URL <http://dx.doi.org/10.1063/1.5020808>.
- [18] Bernard R. Brooks, Robert E. Bruccoleri, Barry D. Olafson, David J. States, S. Swaminathan, and Martin Karplus. CHARMM: A program for macromolecular energy, minimization, and dynamics calculations. *Journal of Computational Chemistry*, 4(2):187–217, 1983. ISSN 1096987X. doi: 10.1002/jcc.540040211.
- [19] B. R. Brooks, C. L. Brooks, A. D. Mackerell, L. Nilsson, R. J. Petrella, B. Roux, Y. Won, G. Archontis, C. Bartels, S. Boresch, A. Caffisch, L. Caves, Q. Cui, A. R. Dinner, M. Feig, S. Fischer, J. Gao, M. Hodoscek, W. Im, K. Kuczera, T. Lazaridis, J. Ma, V. Ovchinnikov, E. Paci, R. W. Pastor, C. B. Post, J. Z. Pu, M. Schaefer, B. Tidor, R. M. Venable, H. L. Woodcock, X. Wu, W. Yang, D. M. York, and M. Karplus. CHARMM: The biomolecular simulation program. *Journal of Computational Chemistry*, 30(10):1545–1614, 7 2009. ISSN 0192-8651. doi: 10.1002/jcc.21287. URL <https://onlinelibrary.wiley.com/doi/abs/10.1002/jcc.21287>.
- [20] A. D. MacKerell, D. Bashford, M. Bellott, R. L. Dunbrack, J. D. Evanseck, M. J. Field, S. Fischer, J. Gao, H. Guo, S. Ha, D. Joseph-McCarthy, L. Kuchnir, K. Kuczera, F. T. K. Lau, C. Mattos, S. Michnick, T. Ngo, D. T. Nguyen, B. Prodhom, W. E. Reiher, B. Roux, M. Schlenkrich, J. C. Smith, R. Stote, J. Straub, M. Watanabe, J. Wiórkiewicz-Kuczera, D. Yin, and M. Karplus. All-Atom Empirical Potential for Molecular Modeling and Dynamics Studies of Proteins †. *The Journal of Physical Chemistry B*, 102(18):3586–3616, 4 1998. ISSN 1520-6106. doi: 10.1021/jp973084f. URL <http://pubs.acs.org/doi/abs/10.1021/jp973084f> <https://pubs.acs.org/doi/10.1021/jp973084f>.
- [21] Paul K. Weiner and Peter A. Kollman. AMBER: Assisted model building with energy refinement. A general program for modeling molecules and their interactions. *Journal of Computational Chemistry*, 2(3):287–303, 1981. ISSN 0192-8651. doi: 10.1002/jcc.540020311. URL <http://doi.wiley.com/10.1002/jcc.540020311>.
- [22] Xavier Daura, Alan E. Mark, and Wilfred F. Van Gunsteren. Parametrization of aliphatic CH<sub>n</sub> united atoms of GROMOS96 force field. *Journal of Computational Chemistry*, 19(5):535–547, 1998. ISSN 01928651. doi: 10.1002/(SICI)1096-987X(19980415)19:5<535::AID-JCC6>3.0.CO;2-N.

- [23] William L. Jorgensen, Jeffrey D. Madura, and Carol J. Swenson. Optimized Intermolecular Potential Functions for Liquid Hydrocarbons. *Journal of the American Chemical Society*, 106(22):6638–6646, 1984. ISSN 15205126. doi: 10.1021/ja00334a030.
- [24] William L. Jorgensen, David S. Maxwell, and Julian Tirado-Rives. Development and testing of the OPLS all-atom force field on conformational energetics and properties of organic liquids. *Journal of the American Chemical Society*, 118(45):11225–11236, 1996. ISSN 00027863. doi: 10.1021/ja9621760.
- [25] Marcus G. Martin and J. Ilja Siepmann. Transferable potentials for phase equilibria. 1. United-atom description of n-alkanes. *Journal of Physical Chemistry B*, 102(14):2569–2577, 1998. ISSN 15206106. doi: 10.1021/jp972543+.
- [26] Pedro E.M. Lopes, Benoit Roux, and Alexander D. Mackerell. Molecular modeling and dynamics studies with explicit inclusion of electronic polarizability: Theory and applications. *Theoretical Chemistry Accounts*, 124(1-2):11–28, 2009. ISSN 1432881X. doi: 10.1007/s00214-009-0617-x.
- [27] K. Vanommeslaeghe and A. D. Mackerell. CHARMM additive and polarizable force fields for biophysics and computer-aided drug design. *Biochimica et Biophysica Acta - General Subjects*, 1850(5):861–871, 2015. ISSN 18728006. doi: 10.1016/j.bbagen.2014.08.004. URL <http://dx.doi.org/10.1016/j.bbagen.2014.08.004>.
- [28] Murray S. Daw and M. I. Baskes. Embedded-atom method: Derivation and application to impurities, surfaces, and other defects in metals. *Physical Review B*, 29(12):6443–6453, 1984. ISSN 01631829. doi: 10.1103/PhysRevB.29.6443.
- [29] Cameron F. Abrams and David B. Graves. Three-dimensional spatiokinetic distributions of sputtered and scattered products of Ar<sup>+</sup> and Cu<sup>+</sup> impacts onto the Cu surface: Molecular dynamics simulations. *IEEE Transactions on Plasma Science*, 27(5):1426–1432, 1999. ISSN 00933813. doi: 10.1109/27.799821.
- [30] G. C. Abell. Empirical chemical pseudopotential theory of molecular and metallic bonding. *Physical Review B*, 31(10):6184–6196, 5 1985. ISSN 0163-1829. doi: 10.1103/PhysRevB.31.6184. URL <https://link.aps.org/doi/10.1103/PhysRevB.31.6184>.
- [31] J. Tersoff. New empirical model for the structural properties of silicon. *Physical Review Letters*, 56(6):632–635, 2 1986. ISSN 0031-9007. doi: 10.1103/PhysRevLett.56.632. URL <https://link.aps.org/doi/10.1103/PhysRevLett.56.632>.
- [32] Donald W. Brenner. Empirical potential for hydrocarbons for use in simulating the chemical vapor deposition of diamond films. *Physical Review B*, 42(15):9458–9471, 1990. ISSN 01631829. doi: 10.1103/PhysRevB.42.9458.
- [33] Steven J. Stuart, Alan B. Tutein, and Judith A. Harrison. A reactive potential for hydrocarbons with intermolecular interactions. *Journal of Chemical Physics*, 112(14):6472–6486, 2000. ISSN 00219606. doi: 10.1063/1.481208.
- [34] Jianguo Yu, Susan B. Sinnott, and Simon R. Phillpot. Charge optimized many-body potential for the Si/SiO<sub>2</sub> system. *Physical Review B - Condensed Matter and Materials Physics*, 75(8):1–13, 2007. ISSN 10980121. doi: 10.1103/PhysRevB.75.085311.



- [35] M Todd Knippenberg, Paul T Mikulski, Kathleen E Ryan, Steven J Stuart, Guangtu Gao, and Judith A Harrison. Bond-order potentials with split-charge equilibration: Application to C-, H-, and O-containing systems. *The Journal of Chemical Physics*, 136(16):164701, 4 2012. ISSN 0021-9606. doi: 10.1063/1.4704800. URL <http://aip.scitation.org/doi/10.1063/1.4704800>.
- [36] Thomas P. Senftle, Sungwook Hong, Md Mahbulul Islam, Sudhir B. Kylasa, Yuanxia Zheng, Yun Kyung Shin, Chad Junkermeier, Roman Engel-Herbert, Michael J. Janik, Hasan Metin Aktulga, Toon Verstraelen, Ananth Grama, and Adri C.T. Van Duin. The ReaxFF reactive force-field: Development, applications and future directions. *npj Computational Materials*, 2(September 2015), 2016. ISSN 20573960. doi: 10.1038/npjcompumats.2015.11. URL <http://dx.doi.org/10.1038/npjcompumats.2015.11>.
- [37] Michael F. Russo and Adri C.T. van Duin. Atomistic-scale simulations of chemical reactions: Bridging from quantum chemistry to engineering. *Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms*, 269(14):1549–1554, 7 2011. ISSN 0168583X. doi: 10.1016/j.nimb.2010.12.053. URL <https://linkinghub.elsevier.com/retrieve/pii/S0168583X10009869>.
- [38] F.H. Stillinger and T.A. Weber. Computer simulation of local order in condensed phases of silicon. *Physical Review B*, 31(8):5262, 1985. URL [http://prb.aps.org/abstract/PRB/v31/i8/p5262\\_1](http://prb.aps.org/abstract/PRB/v31/i8/p5262_1).
- [39] H BEKKER, HJC BERENDSEN, EJ DIJKSTRA, S ACHTEROP, R VONDRUMEN, D VANDERSPOEL, A SIJBERS, H Keegstra, and MKR RENARDUS. Gromacs - a parallel computer for molecular-dynamics simulations. In RA DeGroot and J Nadrchal, editors, *PHYSICS COMPUTING '92*, pages 252–256. World Scientific Publishing, 1993. ISBN 981-02-1245-3. 4th International Conference on Computational Physics (PC 92) ; Conference date: 24-08-1992 Through 28-08-1992.
- [40] James C Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D Skeel, Laxmikant Kalé, and Klaus Schulten. Scalable molecular dynamics with NAMD. *Journal of computational chemistry*, 26(16):1781–802, 12 2005. ISSN 0192-8651. doi: 10.1002/jcc.20289. URL <http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=2486339&tool=pmcentrez&rendertype=abstract>.
- [41] Kefang Liu, Shuguang Tan, Sheng Niu, Jia Wang, Lili Wu, Huan Sun, Yanfang Zhang, Xiaoqian Pan, Xiao Qu, Pei Du, Yumin Meng, Yunfei Jia, Qian Chen, Chuxia Deng, Jinghua Yan, Hong-Wei Wang, Qihui Wang, Jianxun Qi, and George Fu Gao. Cross-species recognition of SARS-CoV-2 to bat ACE2. *Proceedings of the National Academy of Sciences*, 118(1):e2020216118, 1 2021. ISSN 0027-8424. doi: 10.1073/pnas.2020216118. URL <http://www.pnas.org/lookup/doi/10.1073/pnas.2020216118>.
- [42] Cameron Abrams and Giovanni Bussi. Enhanced Sampling in Molecular Dynamics Using Metadynamics, Replica-Exchange, and Temperature-Acceleration. *Entropy*, 16(1):163–199, 12 2013. ISSN 1099-4300. doi: 10.3390/e16010163. URL <http://www.mdpi.com/1099-4300/16/1/163>.
- [43] Ilario G. Tironi and Wilfred F. Van Gunsteren. A molecular dynamics simulation study of chloroform. *Molecular Physics*, 83(2):381–403, 10 1994. ISSN 0026-8976. doi: 10.1080/00268979400101331. URL <http://www.tandfonline.com/doi/abs/10.1080/00268979400101331>.

- [44] Charles H. Bennett. Efficient estimation of free energy differences from Monte Carlo data. *Journal of Computational Physics*, 22(2):245–268, 1976. ISSN 10902716. doi: 10.1016/0021-9991(76)90078-4.
- [45] Shankar Kumar, John M Rosenberg, Djamel Bouzida, Robert H Swendsen, and Peter A Kollman. THE weighted histogram analysis method for free-energy calculations on biomolecules. I. The method. *Journal of Computational Chemistry*, 13(8):1011–1021, 1992. ISSN 1096-987X. doi: 10.1002/jcc.540130812. URL <http://dx.doi.org/10.1002/jcc.540130812>.
- [46] Giovanni Bussi and Alessandro Laio. Using metadynamics to explore complex free-energy landscapes. *Nature Reviews Physics*, 2(4):200–212, 2020. ISSN 25225820. doi: 10.1038/s42254-020-0153-0. URL <http://dx.doi.org/10.1038/s42254-020-0153-0>.
- [47] Jeffrey Comer, James C. Gumbart, Jérôme Hénin, Tony Lelièvre, Andrew Pohorille, and Christophe Chipot. The adaptive biasing force method: Everything you always wanted to know but were afraid to ask. *Journal of Physical Chemistry B*, 119(3):1129–1151, 2015. ISSN 15205207. doi: 10.1021/jp506633n.
- [48] Alessandro Laio and Michele Parrinello. Escaping free-energy minima. *Proceedings of the National Academy of Sciences of the United States of America*, 99(20):12562–6, 10 2002. ISSN 0027-8424. doi: 10.1073/pnas.202427399. URL <http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=130499&tool=pmcentrez&rendertype=abstract>.
- [49] Alessandro Barducci, Giovanni Bussi, and Michele Parrinello. Well-tempered metadynamics: A smoothly converging and tunable free-energy method. *Physical Review Letters*, 100(2):1–4, 2008. ISSN 00319007. doi: 10.1103/PhysRevLett.100.020603.