Monte Carlo Methods for Sampling Classical and Quantum Particles

Jennifer Barry
Advisor: Amy Bug

Swarthmore College
Department of Physics and Astronomy
2007

Thesis Overview

"Monte Carlo" (MC) methods are nondeterministic algorithms for simulating various phenomena in the physical and social sciences. In this study, we evaluated several MC methods for simulating a light particle in a Lennard-Jones (LJ) fluid. We began this research in possession of a basic code that modeled the fluid and light particle in a canonical ensemble. It used Metropolis sampling to equilibrate the fluid and Path Integral Monte Carlo (PIMC) to sample the light particle. This method has been used successfully to calculate positronium lifetimes in spherical and cylindrical pores, but, like all computer simulations, it has its limitations. The canonical ensemble limits us to a fixed number of fluid atoms, making it difficult to simulate a multi-phase system at coexistence, and Metropolis sampling allows us to simulate only one temperature at a time.

Our first major modification to this program was to introduce an option to allow simulation of the fluid within a grand canonical ensemble. Using the Widom Test Particle Insertion Method, we were able to show that these grand canonical simulations agreed with the original canonical simulations for average quantities of density and chemical potential. We then used the grand canonical ensemble simulation to find a point on the phase coexistence curve for the fluid.

Metropolis sampling is inefficient at phase coexistence because the simulation is slow to overcome the free energy barriers. Therefore, we next implemented a method that modifies Metropolis sampling by using a weighting function to help the system explore the whole energy space. We were able to show that, given the optimal weighting function, this "Multicanonical sampling" allows the system to reach every point in the space with equal probability, sampling evenly even in systems with high free energy barriers.

The usefulness of Multicanonical sampling is limited by a fundamental ignorance of this optimal weighting function. To combat this difficulty, we implemented Wang-Landau sampling, which constructs the density of energy states as the simulation proceeds. The results of a Wang-Landau simulation can therefore be used to find ensemble averages at any temperature. We were able to show that one Wang-Landau simulation for the fluid could generate literature results that would require many different Metropolis simulations.

We also implemented Wang-Landau for a light particle in a simple harmonic potential and obtained results matching analytical calculations. The analytical calculation we used was specific to the Path Integral approach, showing that our work agrees with the previous work in this field. In particular, we found excellent agreement with the computational and analytical work of Vorontsov-Velaminov and Lyubartsev [30].

We then extended these techniques to the combined system, simulating helium as a light particle in an argon LJ fluid. A comparison of classical Metropolis sampling, PIMC, and Quantum Wang-Landau showed agreement between PIMC and Quantum Wang-Landau and classical sampling at high temperatures, but only between Wang-Landau and PIMC at low temperatures. From this we conclude that helium must be modeled as a quantum particle at low temperatures in this environment and that our different techniques are consistent.

Finally, we report on efforts to benchmark the PIMC and Quantum Wang-Landau algorithms. We found that, for any individual temperature, Quantum Wang-Landau is 100 to 500 times slower than PIMC. Keeping in mind that we did not use a system that would be too inefficient for PIMC and that Wang-Landau gives information about almost any temperature, while PIMC only gives information about one, these are not unpromising results. Clearly, the next step is to choose a system that is time-consuming for a standard PIMC algorithm to see if the Quantum Wang-Landau benchmark is even more favorable in comparison.

Contents

1 Introduction

"Monte Carlo" (MC) is a collection of computational methods that simulate complex statistical behaviors. Radiation damage, the stock market, phase transitions in materials, and many other such problems have all been the subjects of MC simulations [16].

Here we describe different sampling methods for a light particle in a fluid. In this context "light" means that the particle's state can only be accurately described with quantum mechanics. Specifically, for this project, our model of the light particle will correspond to a helium atom in a system of heavier atoms. Our goal was to allow efficient sampling of this system under as many different conditions as possible.

We model both the light particle and an environment, a Lennard-Jones (LJ) fluid. The LJ fluid has been the focus of a number of numerical studies [[15], [29], [34], [35]], as it is a realistic model of atomic fluids. Many groups, including our own, have written codes to simulate a LJ fluid in a canonical ensemble using a basic Metropolis sampling algorithm [[6], [10]]. However, this type of simulation is slow to converge under certain conditions, such as a high-density fluid or a fluid at a phase coexistence point. In this thesis, we study other methods of simulating a LJ fluid in an attempt to circumvent these problems.

Physicists need simulation techniques that are consistent with not just classical, but also quantum mechanical behavior. For example, we might want to use a simulation to explore such quantum properties as tunneling and light particle lifetimes that cannot be modeled accurately using classical methods. The problem of simulating a quantum particle using MC methods has been widely studied [[22], [23], [24], [26], [30]]. In previous work, this lab has used the Path Integral Monte Carlo (PIMC) method, which treats each quantum particle as a "polymer" made up of a number of "beads" [[6], [7], [8], [10], [17], [18], [36]]. Using this method, modeling "one" particle actually requires modeling anywhere from five to two thousand beads. Because the basic accept/reject MC algorithm (described in Section 3.3.1) is somewhat computationally inefficient, many programs use a modified version of this algorithm, such as threading or staging [[9], [26]]. Our lab has a history of modeling positronium (the bound state of a positron and electron) in a fluid using PIMC with the threading technique and has used these simulations to successfully calculate the lifetime of positronium in spherical and cylindrical pores [[6], [10], [36]]. However, even this algorithm is inefficient in certain systems. For example it may take prohibitively many time steps to converge in a system with a light particle made up of a large number of beads, a dense fluid and/or in a large pore. Thus, in my thesis work, we explore other sampling techniques that are designed to overcome these limitations and allow for a simulation that can work at a range of temperatures, fluid densities and, perhaps, pore sizes.

In the course of developing our research codes, we have written programs that allow us to simulate the fluid and the light particle, separately or together, using a variety of techniques, including Metropolis, Multicanonical, and Wang-Landau sampling. This ability to simulate the classical and quantum parts of our system individually allows us to use our codes, not only to study a light particle in a fluid, but also to study certain properties of the fluid or light particle alone. For example, in this thesis we begin to use our techniques to study a fluid at a phase coexistence point. This work could be expanded in the future to trace out the fluid coexistence curve for a LJ fluid or even more interestingly, isotherms for such a fluid absorbed into pores in a model of a porous solid. In addition, in this thesis we test our techniques by simulating a quantum particle in a simple harmonic potential. Were we to change that potential to the appropriate cubic or quartic form, we might be able to use the new techniques to study tunneling. Whether for the classical or the light particle or both, our algorithms (which are all "modern" in the sense that they have been developed in the last decade or two) allow the system to explore regions that are thermodynamically important, but might not be visible in a reasonable amount of computation time with more conventional MC methods.

The most prevalent application of this work is to the study of positronium (the bound state of a positron and an electron) in a fluid. Although our light particle is currently being modeled as helium, we could turn it into positronium by decreasing the mass of the particle, increasing the number of beads and using

the appropriate interaction potential with the fluid to give us a simulation similar to that of Miller and Reese [[22], [23]]. We could then use the simulation to investigate such properties as bubble formation and positronium lifetimes in a fluid [[21], [23]]. Knowledge of positron or positronium lifetimes is important in many fields, including industry and medicine. PET (Positron Emission Tomography) scans, for example, which are used to detect brain tumors, rely on the annihilation of positrons. We could also use positron lifetimes to explore defects in nickel [28]. If we filled nickel with a gas, the lifetime of a positron in a void in the metal would be correlated with the size and shape of the cavity and we could use this technique to explore the integrity of the nickel.

Previously in this field there has been concentration on sampling techniques for a LJ fluid alone. Wilding pioneered the technique of combining histogram reweighting and Multicanonical sampling to trace the phase coexistence curve [35]. Landau and Wang introduced what is now known as the Wang-Landau method as a technique for constructing the density of states for an Ising model [32]. In 2002, Shell et al [27] extended this technique to continuous systems and tested it on a LJ fluid.

Less work has been done on extending more modern MC sampling techniques to quantum particles. Vorontsov-Velyaminov and Lyubartsev [30] used the Wang-Landau technique for a quantum particle in a simple harmonic potential. There appears to have been little investigation into more complicated potentials. To date, no one has tried to use these techniques for sampling a system comprised of a quantum particle and a LJ fluid as we do in this thesis.

This thesis is arranged in five more sections. We discuss the theory of basic MC sampling and present an overview of the necessary quantum mechanics and thermodynamics in the Theory section. We then treat specific MC sampling algorithms for both the classical and the quantum particles in the Methods section. In the Results and Discussion section, we first present results for the fluid and light particle individually, including showing agreement with analytical or literature values where appropriate. We then explain our results from the combined system and show benchmarks to evaluate the computational efficiency of our algorithms.

For the Theory and Methods portions of this thesis, we are presenting computational algorithms that may be considered "common knowledge" to a computational physicist. As a result, we do not give specific citations for every derivation, but, rather, at the end of each subsection refer the reader to a list of references.

2 Theory

We begin this section with an overview of the theory behind MC algorithms and then present a short review of the necessary thermodynamics and quantum mechanics. In Section 3 we will tie these together and give algorithms for classical and quantum Monte Carlo.

2.1 Monte Carlo Sampling Algorithms

All of the simulations we discuss in this thesis can be said to use Monte Carlo sampling algorithms. Monte Carlo algorithms are nondeterministic methods for simulating various phenomena and they are widely applied. For example, there is a rich history behind using Monte Carlo algorithms to calculate the value of π [2]. The method presented below [1], whimsically known as "integration by darts," is intended to introduce the reader to the concept of using probabilistic techniques to construct an estimate of a result we may not be able to obtain analytically.

Consider a unit circle inscribed in a square as shown in Figure 2.1. To calculate π, we "throw darts" at this square and count the number that fall within the unit circle. In other (more mathematical) words, we randomly generate τ pairs of numbers where each member of the pair is chosen randomly to fall between

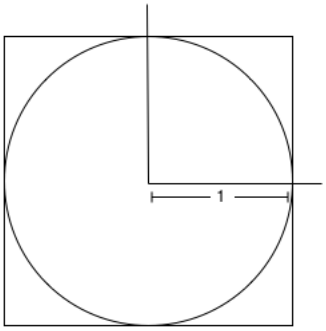


Figure 2.1: We count the number of darts that "hit" within the first quadrant of the unit circle and use this value to estimate π.

0 and 1, and count the number τ_{hit} that are less than a distance of one away from the origin. Clearly, if we repeat this process long enough

$$\pi \approx 4 \times Area\ of\ circle\ in\ first\ quadrant = \frac{4\tau_{hit}}{\tau}. \tag{1}$$

Viola! A Monte Carlo method for computing the value of π. Of course, this is not the most efficient method for calculating π (even after ten million repetitions the answer is only exact to two decimal places [1]), but it is a MC method.

In general, however, we need not depend on sampling in such a blind fashion. In many calculations, including those we are going to be using in this thesis, we are interested in functions that are negligible in all but a small fraction of the total space. Basically, instead of sampling randomly we would like to sample according to some probability distribution. Focusing on only the "important" regions of the space (if known) is a way of reducing the variance of an estimation technique; it is known as "importance sampling."

Notice that in calculating π above, we were calculating an average, of sorts. Now consider a general average of the form:

$$< A >= \frac{\int A(\mathbf{x})\rho(\mathbf{x})d\mathbf{x}}{\int \rho(\mathbf{x})d\mathbf{x}}. \tag{2}$$

The probability of finding the system around $\mathbf{x}$ is given by the probability density

$$W(\mathbf{x}) = \frac{\rho(\mathbf{x})}{\int \rho(\mathbf{x})d\mathbf{x}}. \tag{3}$$

Now if we were somehow able to generate configurations $\{x_i\}$ according to W (we will address exactly how we do that in a moment), we would have a distribution such that we are sampling much more often in those areas of the space in which the function is likely to be non-negligible. If we sample a total of n points, the average is:

$$< A >= \lim_{n\to\infty} \frac{1}{n} \sum_{i}^{n} A(\mathbf{x_i}) \tag{4}$$

Notice that we were able to calculate $< A >$ *without* explicitly calculating $\int \rho(\mathbf{x})d\mathbf{x}$. This will be very important later.

Now we come to the question of *how* we generate points according to W. We first assume the system can be set up as an "irreducible Markov chain." A Markov chain is a sequence of trials that fulfills two conditions: (1) the outcome of each trial belongs to a finite set of possible outcomes (the "state space"), and (2) the outcome of each trial depends *only* on the outcome of the trial immediately preceding it [1].

A Markov chain is considered "irreducible" (or "ergodic") if every state in the chain can eventually be reached from every other state. If we have a system that satisfies these requirements, we can use an "accept/reject" method known as a Metropolis method to calculate points distributed according to W. Suppose we start with the system in state o and randomly generate the next possible state n. We need a criterion that determines whether we transition to ("accept") n or stay in o (thereby "rejecting" n). After a large number of trials, we want the probability of finding the system in state n to be proportional to $W(n)$.

Let the transition probability from o to n be denoted by $\pi(o \to n)$. Once we reach an equilibrium distribution, we do not want $\pi(o \to n)$ to destroy that distribution. Therefore, in equilibrium, the average rate of transitions into n must equal the average rate of transitions out of n (the balance condition). This "balance" condition is sufficient, but most working algorithms rely on the stronger "detailed balance" condition [13] and thus we use that here. We let the number of moves from n to o be equal to the number of moves from o to n:

$$W(o)\pi(o \to n) = W(n)\pi(n \to o). \tag{5}$$

Now we need to construct $\pi(o \to n)$. We let

$$\pi(o \to n) = \alpha(o \to n) \times acc(o \to n) \tag{6}$$

where $acc(o \to n)$ is the probability that we accept a trial move from o to n and α can be chosen to be any function such that $\alpha(o \to n) = \alpha(n \to o)$ (when writing these formulas in matrix form, α is a symmetric matrix). Since α is symmetric, we can substitute equation (6) for π in (5) and cancel the α terms:

$$W(o)acc(o \to n) = W(n)acc(n \to o). \tag{7}$$

Therefore:

$$\frac{acc(o \to n)}{acc(n \to o)} = \frac{W(n)}{W(o)}. \tag{8}$$

While infinitely many choices of $acc(o \to n)$ satisfy equation (8), the choice made for the Metropolis algorithm is [13]:

$$acc(o \to n) = \begin{cases} \frac{W(n)}{W(o)} & W(n) < W(o) \\ 1 & W(n) \geq W(o). \end{cases} \tag{9}$$

To decide whether to accept or reject a trial move, we first compute $\frac{W(n)}{W(o)}$. If this is greater than or equal to one, we automatically accept the move. Otherwise, we generate a random number between 0 and 1. If this number is less than $\frac{W(n)}{W(o)}$, we accept the move, if not, we reject it. This is the basic Metropolis sampling algorithm and will form the foundation for many of the sampling techniques we use.

References for this section: Allen and Tildesley Ch. 4 [1], Frenkel and Smit Ch. 3 [13], Gould et. al Ch. 11 and 16 [14], Kalos and Whitlock [16]

2.2 Random Number Generator Tests

From the last section it is clear that random numbers are extremely important to Monte Carlo simulations and thus we need a good pseudo-random number generator. The question is then, of course, what do we mean by "good"? The definition of a "good" generator is going to depend on the application, but for the purposes of this research, there are a number of tests we can use to determine the properties of a random number generator. If our generator passes these tests, described below, we can be fairly sure that we are getting an adequately random distribution of numbers.

2.2.1 Periodicity Test

The first test we do is a test for the period of the random number generator. If the period is shorter than the length of our run, the generator could potentially cause problems. This test is easy to implement; we simply plot a random walk from the origin using the random number generator. When we reach the period of the random number generator, the plot will start to repeat itself.

2.2.2 Uniformity Test

It is also of vital importance to us that the random number generator generate all machine representable numbers between 0 and 1 with equal probability. We can test for uniformity simply by creating a histogram of how often numbers fall within a certain interval. If the histogram is flat, the generator is uniform. That brings up the question of how flat is "flat," which leads us to our next test.

2.2.3 Chi-Square Test

The Chi-Square Test tells us whether the distribution of numbers into the histogram bins in the "Uniformity Test" is consistent with the laws of statistics. If we distribute the numbers into M bins, y_i is the number in each bin and E_i is the expected number in each bin, then the chi-square statistic is

$$\chi^2 = \sum_{i=1}^{M} \frac{(y_i - E_i)^2}{y_i}. \tag{10}$$

In general each of the terms in Equation (10) should be of order 1. Thus to pass this test, χ^2 should be of the same order as M. Clearly, a smaller χ^2 is better, but a χ^2 of zero or very nearly zero is actually probably indicative of a short period. In other words, we are seeing such a low χ^2 because, if we are generating N random numbers, N is either very close to a multiple of the period of the generator or (more likely) N is much larger than the period of the generator.

2.2.4 Hidden Correlations Test

The last property we check for is hidden-correlations. We do not want k successive random numbers to be correlated in any way. The easiest way to check for these "hidden correlations" is simply to plot x_{i+k} against x_i. If there is an obvious pattern to the plot, there is probably something wrong with the random number generator.

References for this section: Gould et al Ch. 7 [14]

2.3 Thermodynamics Review

In this section, we briefly review the two thermodynamic ensembles we are interested in simulating in this thesis.

2.3.1 Canonical Ensemble

We begin with the canonical ensemble. Suppose we want the average of an observable A and, for the moment, that we can describe the system using classical mechanics (we will address quantum mechanics later in this thesis). The partition function for a canonical ensemble of N particles is

$$Q = \frac{1}{\Lambda^{3N} N!} \int \exp\left(\frac{-H(\mathbf{r}^N, \mathbf{p}^N)}{k_B T}\right) d\mathbf{r}^N d\mathbf{p}^N, \tag{11}$$

where Λ is the thermal de Broglie wavelength $\sqrt{\frac{h^2}{2\pi m k_B T}}$, $H(\mathbf{r^N}, \mathbf{p^N})$ is the Hamiltonian of the system, k_B is the Boltzmann constant, and T is the temperature. The average value of an observable in the system $< A >$ is then

$$< A >= \frac{\int A(\mathbf{r}^N, \mathbf{p}^N) \exp(-\beta H(\mathbf{r}^N, \mathbf{p}^N)) d\mathbf{r}^N d\mathbf{p}^N}{\int \exp(-\beta H(\mathbf{r}^N, \mathbf{p}^N)) d\mathbf{r}^N d\mathbf{p}^N} \tag{12}$$

where $\beta = 1/(k_B T)$. Notice that this average is in a form to which we could easily apply the Metropolis algorithm. From equation (12) the probability density of finding configuration $\mathbf{r}^N$ and momenta $\mathbf{p}^N$ for the canonical ensemble is clearly

$$W_{N,V,T}(\mathbf{r}^N, \mathbf{p}^N) = \frac{\exp(-\beta H(\mathbf{r}^N, \mathbf{p}^N))}{\int \exp(-\beta H(\mathbf{r}^N, \mathbf{p}^N)) d\mathbf{r}^N d\mathbf{p}^N}. \tag{13}$$

As a point of interest, notice that if we can split the Hamiltonian into two pieces, a kinetic energy piece $\kappa(\mathbf{p}^N)$ that depends only on momentum and a potential $U(\mathbf{r}^N)$ that depends only on position:

$$H(\mathbf{r}^N, \mathbf{p}^N) = \kappa(\mathbf{p}^N) + U(\mathbf{r}^N) \tag{14}$$

our probability distribution and partition function will likewise "split":

$$W_{N,V,T}(\mathbf{r}^N, \mathbf{p}^N) = \left(\frac{\exp(-\beta\kappa(\mathbf{p}^N))}{\int \exp(-\beta\kappa(\mathbf{p}^N))d\mathbf{p}^N}\right)\left(\frac{\exp(-\beta U(\mathbf{r}^N))}{\int \exp(-\beta U(\mathbf{r}^N))d\mathbf{r}^N)}\right) \tag{15}$$

and

$$Q = \frac{1}{\Lambda^{3N}N!}\left(\int \exp\left(\frac{-\kappa(\mathbf{p}^N)}{k_BT}\right)d\mathbf{p}^N\right)\left(\int \exp\left(\frac{-U(\mathbf{r}^N)}{k_BT}\right)d\mathbf{r}^N\right). \tag{16}$$

If we have an ideal gas ($U = 0$) then the potential energy parts of Q and W vanish. Therefore, if we are modeling a fluid as a system of particles with the kinetic energy of an ideal gas, but also some "excess" potential that depends only on position (pair interactions for example), we can write the probability distribution and partition function for that system as

$$W_{N,V,T}(\mathbf{r}^N, \mathbf{p}^N) = W_{ideal}W_{excess} \tag{17}$$

and

$$Q = Q_{ideal}Q_{excess}. \tag{18}$$

where W_{ideal} is the probability distribution for an ideal gas and Q_{ideal} is the partition function for an ideal gas. Since both of these quantities can be solved for analytically, that leaves us with only the task of finding the "excess" quantities. We will discuss in the Methods section how we use the Metropolis algorithm to do this.

References for this section: Allen Tildesley Ch. 4 [1], Frenkel and Smit Ch. 3 [13], McQuarrie Ch. 3 [20]

2.3.2 Grand Canonical Ensemble

In a simulation of the grand-canonical ensemble, we hold chemical potential μ, volume V, and temperature T fixed, but allow the number of particles N to change. The chemical potential is the derivative of the Helmholtz free energy F with respect to particle number when we hold V and T constant:

$$\mu = \left(\frac{dF}{dN}\right)_{V,T}. \tag{19}$$

Recall that Helmholtz free energy is defined as

$$F = U - TS \tag{20}$$

where U is the internal energy of the system, T is the temperature and S is the entropy. Basically, μ is the amount by which the free energy will change if we add another particle to the system while holding the volume and temperature constant. In other words, the chemical potential is how much a system "wants" more particles. Note that if we have a fluid in phase coexistence, the chemical potential must be the same for both phases. Were it not, one phase would "want" particles more and the particles would diffuse to that phase until the chemical potentials equilized. Thus equal chemical potentials are a stability criterion for phase coexistence.

The complete grand canonical partition function, given a hamiltonian $H(\mathbf{r}^N, \mathbf{p}^N)$ is [20]

$$Q_{\mu,V,T} = \sum_{N=0}^{\infty} \frac{1}{N!\Lambda^{3N}} \exp(\beta\mu N) \int \exp(-\beta H(\mathbf{r}^N, \mathbf{p}^N))d\mathbf{r}^N d\mathbf{p}^N. \tag{21}$$

where N indexes the number of particles. In our simulation N will vary in a manner described in Section 3.2.2. We will not derive equation (21) here, but we will give an idea of how it might be derived by deriving a simpler (and more applicable to this thesis) case.

For this derivation, we first assume that when adding and removing particles, we are actually just exchanging particles with a reservoir of an ideal gas. We assume that in our subsystem of interest we have particles with the kinetic energy of an ideal gas interacting via a potential $U(\mathbf{r}^N)$ that is not dependent on momentum.

First, just consider a system of N such identical atoms. The partition function is

$$Q(N,V,T) = \frac{1}{\Lambda^{3N}N!}\int_0^L \cdots \int_0^L \exp(-\beta U(\mathbf{r}^N) d\mathbf{r}^N. \tag{22}$$

If we let $\mathbf{s}$ represent $\mathbf{r}$ scaled by L equation (22) becomes

$$Q(N,V,T) = \frac{V^N}{\Lambda^{3N}N!}\int_0^1 \cdots \int_0^1 \exp(-\beta U(\mathbf{s}^N; L) d\mathbf{s}^N \tag{23}$$

where $U(\mathbf{s}^N; L)$ indicates that U depends on the real rather than the scaled distances. The Helmholtz free energy is

$$\begin{aligned} F(N,V,T) &= -k_B T \ln Q \\ &= -k_B T \ln\left(\frac{V^N}{\Lambda^{3N}N!}\right) - k_B T \ln\left(\int \exp(-\beta U(\mathbf{s}^N; L)) d\mathbf{s}^N\right) \\ &= F^{id}(N,V,T) + F^{ex}(N,V,T). \end{aligned} \tag{24}$$

As shown in the last line of equation (24), the free energy can be written as the free energy of an ideal gas plus an excess free energy. Thus assume the system is separated by a piston from an ideal gas reservoir. The total volume of the system plus the reservoir is held at V_0 and there are a total of M particles (see Figure 2.2). In other words, we have $M - N$ ideal gas molecules in a volume of $V_0 - V$. We take the product of the partition function of our system and that of the ideal gas to find the total partition function of the ideal gas plus the reservoir

$$Q(N,M,V,V_0,T) = \frac{V^N (V_0 - V)^{M-N}}{\Lambda^{3M} N!(M-N)!}\int d\mathbf{s}^{M-N} \int \exp(-\beta U(\mathbf{s}^N; L)) d\mathbf{s}^N. \tag{25}$$

Note that the factor $\int d\mathbf{s}^{M-N}$ is equal to unity and that we have assumed the thermal wavelength of the ideal gas is also equal to Λ.

Now we allow the two systems to exchange particles. Basically, we assume that the particles are identical, but inside the volume V they interact with one another and outside they do not. Thus a transfer from a coordinate outside to a coordinate inside changes the potential energy from $U(\mathbf{s}^N; L)$ to $U(\mathbf{s}^{N+1}; L)$. The partition function taking into account all possible distributions of the M particles over the two volumes is

$$Q(M,V,V_0,T) = \sum_{N=0}^{M} Q(N,M,V,V_0,T) = \sum_{N=0}^{M} \frac{V^N (V')^{M-N}}{N!(M-N)!\Lambda^{3N}\Lambda^{3(M-N)}} \int \exp(-\beta U(\mathbf{s}^N; L)) d\mathbf{s}^N. \tag{26}$$

Using the definition of μ from equation (19) and that of F^{id} from equation (24), we find that for an ideal gas

$$\mu^{id} = k_B T \ln(\Lambda^3 \rho). \tag{27}$$

We define

$$e^{C_M} = \Lambda^{3M}\frac{M^M}{V_0^M} = \Lambda^{3M}\rho^M = e^{\beta\mu M}. \tag{28}$$

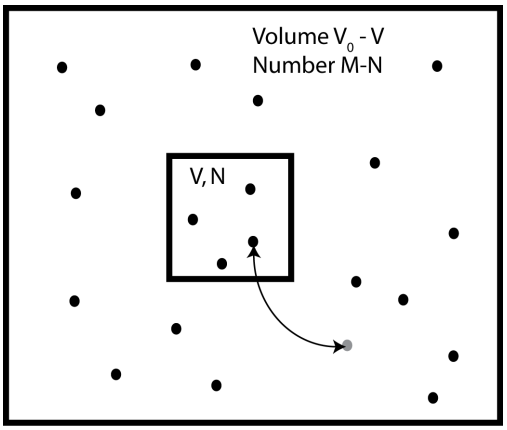


Figure 2.2: We model the grand canonical ensemble as applying to an "interacting" gas of N particles (where N is variable) in a volume V that can exchange particles with an ideal gas. The total volume of the ideal plus interacting gas is V_0 and the total number of particles is M.

Using e^{C_M}, we can rewrite Q as

$$Q(M,V,V_0,T) = e^{-C_M} \sum_N \frac{V^N (V_0 - V)^{M-N}/V_0^M}{(N!(M-N)!/M^M)(\Lambda^{3N}\Lambda^{3(M-N)}/\Lambda^{3M})} \int \exp(-\beta U(\mathbf{s}^N; L)) d\mathbf{s}^N. \quad (29)$$

Thus if we take the limit that the ideal gas is much larger than the interacting one ($M \to \infty, V' \to \infty, (M/V') \to \rho$), we come up with an expression for $Q(\mu, V, T)$

$$\begin{aligned} Q(M,V,V_0,T) &= e^{-C_M} \sum_N \frac{V^N V_0^{M-N}/V_0^M}{N!(M^{M-N}/M^M)} \int \exp(-\beta U(\mathbf{s}^N; L)) d\mathbf{s}^N & (30)\\ &= e^{-C_M} \sum_N \frac{V^N M^N \Lambda^{3N}}{N! V_0^N \Lambda^{3N}} \int \exp(-\beta U(\mathbf{s}^N; L)) d\mathbf{s}^N & (31)\\ &= e^{-C_M} \sum_N \frac{V^N}{N!\Lambda^{3N}} e^{\beta\mu N} \int \exp(-\beta U(\mathbf{s}^N; L)) d\mathbf{s}^N & (32)\\ &= Q(\mu, V, T) e^{-C_M} & (33) \end{aligned}$$

The only worrisome part of this derivation is the inclusion of the extra e^{-C_M} term. The argument of the exponent represents the free energy of the total system (V_0 in its entirety). If we always consider a system where M is extremely large (as we are doing), this term does not affect the physics in any way as it will be the same for any system we are studying and we drop it. Thus the partition function we use for this system is

$$Q(\mu, V, T) = \sum_{N=0}^{\infty} \frac{\exp(\beta\mu N) V^N}{\Lambda^{3N} N!} \int \exp(-\beta U(\mathbf{s}^N; L)) d\mathbf{s}^N. \quad (34)$$

From this partition function, we can calculate the probability density for having N particles in the configuration $\mathbf{s}^N$:

$$W_{\mu,V,T}(\mathbf{s}^N, N) \propto \frac{\exp(\beta\mu N) V^N}{\Lambda^{3N} N!} \exp(-\beta U(\mathbf{s}^N; L)). \quad (35)$$

From this probability distribution, the average value of an observable in this system is

$$< A >= \frac{\sum_N \int A(\mathbf{r}^N) W_{\mu,V,T}(\mathbf{r}^N, N) d\mathbf{r}^N}{\sum_N \int W_{\mu,V,T}(\mathbf{r}^N, N) d\mathbf{r}^N}. \quad (36)$$

Another quantity useful to introduce here is the activity z defined as

$$z = \frac{\exp(\beta\mu)}{\Lambda^3}. \tag{37}$$

In chemistry and physics, the activity can tell us something about the partial pressure of a gas. At low pressure, the activity is equal to the ratio of the partial pressure of the gas to the standard pressure. Thus in some ways activity is a measure of how "close" our gas is to an ideal gas. In practical terms, using z instead of μ makes the mathematics slightly simpler and allows us to use units in which $\Lambda = 1$. In terms of z the probability density is

$$W_{\mu,V,T}(\mathbf{s}^N, N) = \frac{(zV)^N}{N!}\exp(-\beta U(\mathbf{s}^N)). \tag{38}$$

Notice that there is no dependence on Λ in this expression.

To calculate z without involving Λ look again at equation (24). Since $\mu = dF/dN$, it is clear that at large N, we can write the chemical potential as

$$\mu = -k_B T \ln\left(\frac{Q_{N+1}}{Q_N}\right). \tag{39}$$

Substituting for Q and solving gives

$$\begin{aligned} \mu &= -k_B T \ln\left(\frac{V/\Lambda^3}{N+1}\right) - k_B T \ln\left(\frac{\int \exp(-\beta U(\mathbf{s}^{N+1}))d\mathbf{s}^N}{\int \exp(-\beta U(\mathbf{s}^N))d\mathbf{s}^N}\right) \\ &= \mu_{ideal} + \mu_{excess} \end{aligned} \tag{40}$$

Using this expression for μ in equation (37) we find

$$\begin{aligned} z &= \exp(\beta\mu_{excess})\frac{\exp\left(\ln\left[\frac{(N+1)\Lambda^3}{V}\right]\right)}{\Lambda^3} \\ &= \frac{(N+1)\exp(\beta\mu_{excess})}{V} \end{aligned} \tag{41}$$

Thus z has no dependence on the thermal de Broglie wavelength and we can calculate it either from knowledge of the excess chemical potential and the volume of the system or from the excess chemical potential and the ideal chemical potential expressed in a units system where $\Lambda = 1$.

We will explain how to use Metropolis sampling with these two ensembles in the Methods section.

References for this section: Allen and Tildesley Ch. 4 [1], Frenkel and Smit Ch. 5 [13], McQuarrie Ch. 3 [20]

2.4 Quantum Mechanics Review

Here we present a brief overview of a few topics in quantum mechanics that are relevant to our simulations.

In quantum mechanics, the expected value $< \hat{A} >$ of an observable $\hat{A}$ is given as [5]

$$< \hat{A} >= Tr(\hat{W}\hat{A}) \tag{42}$$

where $\hat{W}$ is the density operator for the system. Recall that the density operator $\hat{W}$ is defined as a positive, self-adjoint operator. Here we will usually use the coordinate representation of the density operator:

$$W(\mathbf{r}, \mathbf{r}', \beta) =< \mathbf{r}|\hat{W}|\mathbf{r}' >= \sum_n \psi_n^*(\mathbf{r})\hat{W}\psi_n(\mathbf{r}') \tag{43}$$

where the ψ_n are a complete set of quantum states.

For the canonical ensemble (and we always simulate our light particle in a canonical ensemble), the quantum mechanical density operator is

$$\hat{W} = \exp(-\beta \hat{H}) \tag{44}$$

where $\hat{H}$ is the Hamiltonian operator for the system.

In this thesis, we use a discretization of the path intregral form of the density operator for our numerical calculations. We begin with the partition function. In quantum statistical mechanics, rather than having the trace of the density operator equal to unity as is often the case, the trace is instead interpreted as the partition function of the system. To obtain the trace of the density operator (and hence the partition function), we set $\mathbf{r}$ equal to $\mathbf{r}'$ in equation (43) and integrate over the coordinate

$$Q_{NVT} = \int \sum_n \psi_n^*(\mathbf{r}) \hat{W} \psi_n(\mathbf{r}) d\mathbf{r}. \tag{45}$$

This is not an easy partition function with which to work since, except for specific, special cases of $\hat{H}$, we have no easy way of doing the integral in equation (45). We attempt to put it into a more tractable form for approximation by breaking the exponent into P parts:

$$Q_{1VT} = \int < \mathbf{r_1} | e^{\frac{-\beta \hat{H}}{P}} ... e^{\frac{-\beta \hat{H}}{P}} ... e^{\frac{-\beta \hat{H}}{P}} | \mathbf{r_1} > d\mathbf{r}. \tag{46}$$

Note that equation (46) is written just for $N = 1$ particles. Generalization is possible, but only the $N = 1$ case is relevant to this thesis so we will concentrate on that. If we insert the identity in the form $\int |\mathbf{r}><\mathbf{r}| d\mathbf{r}$ between each exponent in equation (46), our expression becomes

$$Q_{1VT} = \int < \mathbf{r_1} | e^{\frac{-\beta \hat{H}}{P}} | \mathbf{r_2} >< \mathbf{r_2} | e^{\frac{-\beta \hat{H}}{P}} | \mathbf{r_3} > ... < \mathbf{r_P} | e^{\frac{-\beta \hat{H}}{P}} | \mathbf{r_1} > d\mathbf{r_1} d\mathbf{r_2} ... d\mathbf{r_P} \tag{47}$$

which we can simplify to

$$Q_{1VT} = \int W(\mathbf{r_1}, \mathbf{r_2}, \beta/P) W(\mathbf{r_2}, \mathbf{r_3}, \beta/P) ... W(\mathbf{r_P}, \mathbf{r_1}, \beta/P) d\mathbf{r_1} d\mathbf{r_2} ... d\mathbf{r_P}. \tag{48}$$

At first glance this expression may seem anything but "more tractable." However, we now have P different density matrices all corresponding to lower β (ie, higher temperature) than the original. At sufficiently large P, and therefore high temperature, we can use the approximation[1]

$$W(\mathbf{r_a}, \mathbf{r_b}, \beta/P) \approx W_{free}(\mathbf{r_a}, \mathbf{r_b}, \beta/P) \exp\left(-\frac{\beta}{2P} (U(\mathbf{r_a}) + U(\mathbf{r_b})) \right) \tag{49}$$

where $U(\mathbf{r_a})$ is the classical potential energy and $\hat{W}_{free}$ is coordinate representation of the density matrix for a free particle

$$W_{free}(\mathbf{r_a}, \mathbf{r_b}, \beta/P) = \left(\frac{Pm}{2\pi\beta\hbar^2} \right)^{\frac{3}{2}} \exp\left(-\frac{Pm}{2\beta\hbar^2} r_{ab}^2 \right). \tag{50}$$

[1] A short derivation of this approximation [25]. We begin with the identity:

$$e^{-\tau(A+B)} = e^{-\tau U/2} e^{-\tau K} e^{-\tau U/2} + O(\tau^3).$$

Then

$$W(\mathbf{r}, \mathbf{r}'; \tau) = < \mathbf{r} | e^{-\tau(U+K)} | \mathbf{r}' > = < \mathbf{r} | e^{-\tau K} | \mathbf{r}' > e^{-\tau/2(U(\mathbf{r}) + U(\mathbf{r}'))}.$$

The first term is just $W_{free}(\mathbf{r}, \mathbf{r}'; \tau)$.

where $r_{ab}^2 = |\mathbf{r_a} - \mathbf{r_b}|$. We can now write the full expression for Q_{1VT} as

$$Q_{1VT} = \left(\frac{Pm}{2\pi\beta\hbar^2}\right)^{\frac{3P}{2}} \int \exp\left(-\frac{Pm}{2\beta\hbar^2}(r_{12}^2 + r_{23}^3 + ... + r_{P1}^2)\right) \exp\left(-\frac{\beta}{P}(U(\mathbf{r_1}) + U(\mathbf{r_2}) + ... + U(\mathbf{r_P}))\right) d\mathbf{r_1}...d\mathbf{r_P}. \tag{51}$$

To make this slightly clearer, we rewrite Q_{1VT} as

$$Q_{1VT} = \int \exp(-\beta V(\mathbf{r}))d\mathbf{r_1}...d\mathbf{r_P} \tag{52}$$

and W in the coordinate representation as

$$W(\mathbf{r}) = \exp(-\beta V(\mathbf{r})) \tag{53}$$

where V is given by

$$V(\mathbf{r}) = \frac{1}{P}U(\mathbf{r}) + \kappa(\mathbf{r}). \tag{54}$$

Here $U(\mathbf{r})$ is the classical potential and $\kappa(\mathbf{r})$ can be written as

$$\kappa(\mathbf{r}) = \left(\frac{Pm}{2\beta^2\hbar^2}\right)(|\mathbf{r}_{12}|^2 + |\mathbf{r}_{23}|^2 + ... + |\mathbf{r}_{P1}|^2) \tag{55}$$

$$= \left(\frac{Pm}{2\beta^2\hbar^2}\right)\sum_{a=1}^{P}|\mathbf{r}_a - \mathbf{r}_{a+1}|^2. \tag{56}$$

where $\mathbf{r}_{P+1} = \mathbf{r}_1$. Notice that equation (56) appears similar to the harmonic oscillator potential; this is a point we will make great use of in the Methods section when we fully explain how to use Monte Carlo methods to equilibrate quantum particles.

References for this section: Allen and Tildesley Ch. 10 [1], Boccio Ch. 6 [5]

We now have all the pieces we need to develop sampling algorithms for our system. In the next section we put these pieces together to describe our various simulation techniques.

3 Methods

In this section we discuss the application of the theory to actual sampling techniques. We give details of our fluid simulation and of the different sampling methods we have employed.

3.1 Fluid Simulation

We simulate an argon fluid of temperature T and volume V. We assume a kinetic energy whose thermodynamic average value is $\frac{3}{2}Nk_BT$ and a pair-pair potential, known as the Lennard-Jones potential

$$U_{LJ}(r) = 4\epsilon\left(\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right). \tag{57}$$

A diagram of this potential is shown in Figure 3.1. Clearly at large r, this potential is proportional to r^{-6}; at this range all we are modeling is the attractive dipole-induced dipole (London dispersion) force between the molecules[2]. The exact form of the non-assymptotic behavior, including the repulsion term,

[2]An easy way to see that the dipole-induced dipole potential goes as r^{-6} is that for a dipole $E \propto r^{-3}$. Then

$$< U >= \frac{1}{2}\int_{all\_space} E^2 d\tau \propto r^{-6}.$$

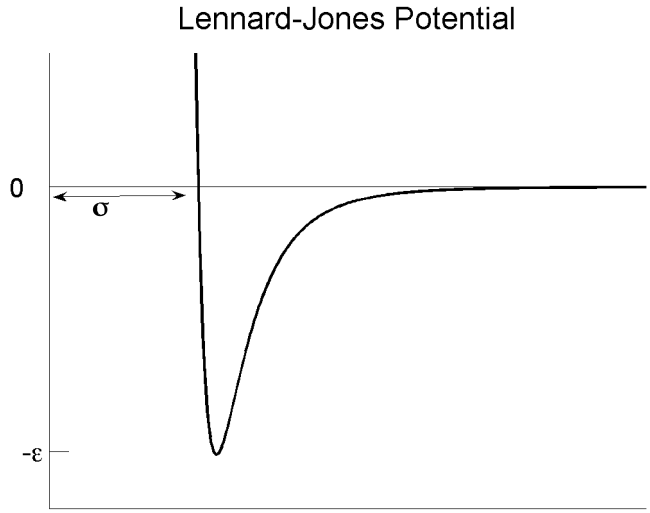


Figure 3.1: The Lennard-Jones potential. The parameters ϵ and σ are marked on the diagram.

however, is empirical. The parameters ϵ and σ are specific to the fluid(s) being modeled, where ϵ is the depth of the well and σ is the hard sphere diameter (also known as the van der Waals Radius). For argon, $\epsilon = 3.794 \cdot 10^{-4}$ au and $\sigma = 6.433$ au. We use both atomic units and reduced units in this thesis; for a full discussion, see Appendix A. For the parameters used during simulation, see the modules *FLInfo.f95* and *LPInfo.f95* in Appendices B.1 and B.2 respectively.

In order to speed the simulation and to work sensibly with a periodic boundary condition, we also use a cutoff of $r_{cut} = 2.5\sigma$, giving us a potential energy for the entire fluid of

$$U_{fluid} = \sum_{i=1}^{N} \sum_{j>i}^{N} 4\epsilon \begin{cases} \left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6} & r_{ij} \leq r_{cut} \\ 0 & r_{ij} > r_{cut} \end{cases} \tag{58}$$

If we can approximate the structure of the fluid for $r > r_{cut}$, we can correct for this cutoff later by adding a correction u^{tail} to the calculated energy. In this thesis, for u^{tail} we use the simplest possible approximation, given, for each individual particle in the fluid, by

$$u^{tail} = \frac{1}{2} 4\pi\rho \int_{r_{cut}}^{\infty} r^2 U(r) dr \tag{59}$$
$$= \frac{8}{3}\pi\epsilon\sigma^3\rho \left(\frac{1}{3} \left(\frac{\sigma}{r_{cut}} \right)^9 - \left(\frac{\sigma}{r_{cut}} \right)^3 \right)$$

where ρ is the number density of the fluid.

We can also calculate the pressure of the fluid during the simulation by using the virial. The pressure P of the fluid is given by

$$P = \frac{\rho}{\beta} + \frac{vir}{V} \tag{60}$$

where $\beta = 1/k_B T$ and vir is the virial defined by

$$vir = \frac{1}{3} \sum_{i} \sum_{j>i} \mathbf{f}(\mathbf{r}_{ij}) \cdot \mathbf{r}_{ij}. \tag{61}$$

Here

$$\mathbf{f}(\mathbf{r}_{ij}) = -\frac{dU}{d\mathbf{r}_{ij}}. \tag{62}$$

The tail correction to the pressure is [13]

$$P^{tail} = \frac{16}{3}\pi\rho^2\epsilon\sigma^3\left(\frac{2}{3}\left(\frac{\sigma}{r_{cut}}\right)^9 - \left(\frac{\sigma}{r_{cut}}\right)^3\right). \tag{63}$$

Thus equations (60)-(63) give us a way of sampling the pressure during the simulation.

We simulate the fluid as a bulk and hence we use periodic boundary conditions. In other words, we take a cubic cell and assume that the cell repeats itself infinitely. However, in computing the potential, we use the minimum image convention [1] that we only consider the potential from the closest "image" of a given atom. Since we are using a truncated potential, it is clear that so long as our cutoff radius is less than half the box size, the minimum image convention is a consistent method for obtaining the properties of a bulk.

In the next sections we discuss different methods for sampling the fluid.

References for this section: Allen and Tildesley Ch. 4 [1], Frenkel and Smit Ch. 3 [13]

3.2 Techniques for Sampling Classical Particles

We use a wide variety of techniques to sample our fluid. In all of them, we treat the fluid using classical statistical mechanics (we will discuss sampling for quantum particles in Section 3.3).

3.2.1 Metropolis Canonical Monte Carlo

Recall from the Theory section that Metropolis Sampling is an algorithm for sampling according to a probability distribution. We derived the probability distributions for our thermodynamic ensembles of interest in the Theory section as well; all that remains is to write down the specific sampling algorithms for each.

Recall from equation (13) that the probability density in the canonical ensemble is

$$W_{N,V,T}(\mathbf{r}^N, \mathbf{p}^N) = \frac{\exp(-\beta H(\mathbf{r}^N, \mathbf{p}^N))}{\int \exp(-\beta H(\mathbf{r}^N, \mathbf{p}^N)) d\mathbf{r}^N d\mathbf{p}^N}.$$

As explained in Section 3.1, we can split the Hamiltonian for our fluid into two pieces, a kinetic energy $\kappa = \sum_N p_i^2/(2m)$ and the Lennard-Jones Potential energy $U(\mathbf{r}^N)$:

$$H(\mathbf{r}^N, \mathbf{p}^N) = \kappa + U(\mathbf{r}^N) \tag{64}$$

We know that κ integrates to a known constant and thus do not need to calculate it in the simulation. Therefore, we can use equation (15) to write the probability distribution function for this system:

$$W_{N,V,T}(\mathbf{r}^N) \propto \exp(-\beta U(\mathbf{r}^N)). \tag{65}$$

Now $W_{N,V,T}(\mathbf{r}^N)$ is of a form to allow us to use the Metropolis algorithm to equilibrate the fluid. The Metropolis canonical Monte Carlo (CMC) algorithm that we use is:

For each particle in the system:

1. Calculate the potential energy of the system U
2. Give the particle a random displacement $\mathbf{r}' = \mathbf{r} + \Delta$
3. Calculate the new energy of the system U'
4. Accept the move with probability:

$$acc(\mathbf{r} \to \mathbf{r}') = \min(1, \exp(-\beta(U' - U))) \tag{66}$$

This procedure has been shown to satisfy balance (although not detailed balance) [13]. Our Fortran 95 implementation of it can be found in Appendix B.1, in the *FLFluidFncs.f95* module on pages 11-13.

References for this section: Allen Tildesley Ch. 4 [1], Frenkel and Smit Ch. 3 [13], Gould et. al Ch. 11 and 16 [14], McQuarrie Ch. 3 [20]

3.2.2 Metropolis Grand Canonical Monte Carlo

From the Theory section (equation (38)) the probability density of the grand canonical ensemble (with only pair interactions) is

$$W_{\mu,V,T}(\mathbf{s}^N, N) = \frac{(zV)^N}{N!} \exp(-\beta U(\mathbf{s}^N)).$$

The exact Metropolis algorithm for this ensemble is the similar to that for CMC except that this time we can choose to move, create, or destroy a particle. In this simulation, we do each of these with equal probability (ie, we move a particle 1/3 of the time, create a particle 1/3 of the time, and destroy a particle 1/3 of the time). In our code in Appendix B.1, we make this choice on line 81 of the *FLFLuidFncs.f95* module in the *flMCStep* subroutine on page 12.

For moving a particle the ratio of distributions is

$$\frac{W(\mathbf{s'}^N, N)}{W(\mathbf{s}^N, N)} = \exp(-\beta(U(\mathbf{s'}^N; L) - U(\mathbf{s}^N; L)),$$

and thus $acc(\mathbf{s} \to \mathbf{s}')$ is the same as for the canonical ensemble

$$acc(\mathbf{s} \to \mathbf{s}') = \min(1, \exp(-\beta(U(\mathbf{s}'; L) - U(\mathbf{s}; L)))). \tag{67}$$

To create a particle:

$$\frac{W(\mathbf{s}^{N+1}, N+1)}{W(\mathbf{s}^N, N)} = \frac{zV}{N+1} \exp(-\beta(U(\mathbf{s}^{N+1}; L) - U(\mathbf{s}^N; L))),$$

and to destroy:

$$\frac{W(\mathbf{s}^{N-1}, N-1)}{W(\mathbf{s}^N, N)} = \frac{N}{zV} \exp(-\beta(U(\mathbf{s}^{N-1}; L) - U(\mathbf{s}^N; L))).$$

The corresponding acceptance probabilities are:

$$acc(N \to N+1) = \min\left(1, \frac{zV}{(N+1)} \exp(\beta(-U(\mathbf{s}^{N+1}; L) + U(\mathbf{s}^N; L))\right) \tag{68}$$

$$acc(N \to N-1) = \min\left(1, \frac{N}{zV} \exp(-\beta(U(\mathbf{s}^{N-1}; L) - U(\mathbf{s}^N; L))\right). \tag{69}$$

We now have all of the theory required to implement GCMC. For our implementation of this GCMC algorithm see Appendix B.1, pages 11-13.

The Metropolis method is a useful simulation technique, but, like all such techniques, it has its weaknesses. For our purposes, the main limitation of the Metropolis algorithm is its inefficiency in exploring spaces with high free energy barriers. Thus we now discuss modifications of, and alternatives to, the Metropolis method, beginning with the Multicanonical method.

References for this section: Allen and Tildesley Ch. 4 [1], Frenkel and Smit Ch. 5 [13], Gould et. al Ch. 11 and 16 [14], McQuarrie Ch. 3 [20]

3.2.3 Multicanonical Method

The Multicanonical method is a modification of the Metropolis method that, for example, when used on a fluid at a phase coexistence boundary, artificially enhances the probability that the system will be found in a mixed state. When using this method we sample, not from a distribution with our usual Hamiltonian, but rather from a modified distribution with effective Hamiltonian [35]

$$\tilde{H}(\mathbf{r}^N, \mathbf{p}^N) = H(\mathbf{r}^N, \mathbf{p}^N) + \eta(N). \tag{70}$$

Here $\eta(N)$ is a preweighting function chosen such that the associated probability distribution

$$\tilde{W}(N|V, \beta, \mu, \eta(N)) = \frac{1}{\tilde{Q}} \int e^{-\beta \tilde{H}(\mathbf{r}^N, \mathbf{p}^N)} d\mathbf{r}^N d\mathbf{p}^N. \tag{71}$$

is a constant. Here V is volume, β is $\frac{1}{kT}$ where T is temperature and k is Boltzmann's constant, and μ is chemical potential. Note that $\tilde{Q}$ is the partition function for the modified distribution using $\tilde{H}$. Clearly the choice $\eta(N) = \ln W(N)$ gives a constant $\tilde{W}$, where $W(N)$ is the original unmodified probability distribution function appropriate for the ensemble

$$W(N|V, \beta, \mu, \eta(N)) = \frac{1}{Q} \int e^{-\beta H(\mathbf{r}^N, \mathbf{p}^N)} d\mathbf{r}^N d\mathbf{p}^N. \tag{72}$$

However, calculating this optimal $\eta(N)$ must in principle be as difficult as solving the original problem since it is simply the logarithm of the function we are trying to find. There are many methods for making sensible choices for $\eta(n)$; we will discuss one such method in section 3.2.4. We introduce this weighting function in our code as the variable *eta* in the subroutine *flMCStep* in the *FLFluidFncs.f95* module on pages 11-13 of Appendix B.1.

Once we have found a good choice for $\eta(N)$ we need a method of regaining $W(N)$ from data for $\tilde{W}(N)$. Given that we know $\eta(N)$ the conversion is actually very simple:

$$W(N|V, \beta, \mu) = e^{\eta(n)} \tilde{W}(N|\beta, \mu, \eta(N)). \tag{73}$$

Simulating a fluid in a grand canonical ensemble using this algorithm is similar to Metropolis GCMC with a modified Hamiltonian $\tilde{H}$. Using this modified Hamiltonian, the acceptance probabilities of creation and destruction are modified to

$$a\tilde{c}c(N \to N+1) = \min(1, \exp(\eta(N) - \eta(N+1)) \frac{zV}{N+1} \exp(\beta(-U(\mathbf{s}^{N+1}; L) + U(\mathbf{s}^N; L))) \tag{74}$$

$$a\tilde{c}c(N \to N-1) = \min(1, \exp(\eta(N) - \eta(N-1)) \frac{N}{zV} \exp(-\beta(U(\mathbf{s}^{N-1}; L) - U(\mathbf{s}^N; L))). \tag{75}$$

In a 2001 paper Wilding used this method to calculate a number of points on the phase coexistence curve for a fluid [35], effectively demonstrating that the Multicanonical method helps surmount the phase transition barrier. We follow his lead in testing this algorithm in section 4.3.2.

References for this section: Berg and Neuhaus [4], Wilding [35]

3.2.4 Histogram Reweighting

Histogram reweighting is a technique that allows us to use histograms with one set of model parameters to estimate histograms with other values of those parameters. In this case, our parameters are β (temperature) and μ. Consider a histogram taken at parameters β_0 and μ_0. The probability density is [35]

$$W(N, U|V, \beta_0, \mu_0) = \frac{1}{Q_0} \int \delta(U - U(\mathbf{r}^N)) e^{-\beta_0(U(\mathbf{r}) - \mu_0 N)} d\mathbf{r}^N. \tag{76}$$

To find the probability density at β_1 and μ_1 we can simply reweight the histogram

$$W(N,U|V,\beta_1,\mu_1) = \frac{Q_0}{Q_1}\exp(-(\beta_1(U-N\mu_1)-\beta_0(E-N\mu_0))p(N,E|V,\beta_0,\mu_0). \tag{77}$$

This allows us then to extrapolate from known data to obtain histograms at different temperatures and chemical potentials.

One use for this technique, proposed by Wilding [35], is to combine histogram reweighting and Multicanonical simulations. In this method, we reweight a histogram from one run into the preweighting function for a Multicanonical simulation at a different temperature and chemical potential. Although this sounds at first like an ideal solution, the process is slower and more cumbersome than it first appears. By the time we have data on which we can do histrogram reweighting, we have lost enough precision to make reweighting to anything but a β_1 near to our original β_0 difficult. For a full explanation of this process and its limitations, see section 4.3.2.

References for this section: Ferrenberg and Swensden [11], [12], Wilding [35]

3.2.5 Classical Wang-Landau Method

The Wang-Landau Method is an alternative to using Metropolis importance sampling. This algorithm solves for the density of states $g(E)$ where $g(E)$ is defined such that the probability that a system has energy E is

$$P(E,\beta) = \frac{g(E)e^{-\beta E}}{Q}. \tag{78}$$

An algorithm designed to compute $g(E)$ is a different approach from the Metropolis algorithm. Note that were we able to find just $g(E)$ we would not have to run simulations at more than one temperature; we could simply vary the value of β in our post-analysis. Wang and Landau [[31], [32]] propose a method for finding $g(E)$ that we outline here.

This method for determining $g(E)$ is through a random walk in state-space. At every state visited, the energy is noted and the appropriate bin in an energy histogram $H(E)$ is marked. However, since unlikely states rarely get visited in an unweighted random walk, we need to weight the random walk in some way that allows all the states to be visited with approximately equal probability [[31], [32]]. The easiest way to do this is simply to weight by $\frac{1}{g(E)}$, giving a flat histogram for the energy distribution $H(E)$. Of course, we do not know $g(E)$ so we must construct it "on the fly" during the simulation.

We begin with an arbitrary state and an initial guess at $g(E)$ (in general we set $g(E) = 1$ for all E). We then do a Monte Carlo move and compute the energy before the move E_0 and after the move E_1 and accept the move with probability

$$acc(E_0 \to E_1) = \min\left(1, \frac{g(E_0)}{g(E_1)}\right). \tag{79}$$

We update $g(E)$ by multiplying the accepted energy E_{acc} (E_0 if the move is rejected and E_1 if the move is accepted) by a modification factor $f > 1$:

$$g(E_{acc}) = fg(E_{acc}), \tag{80}$$

and increment our histogram. Generally we begin with an initial value for f of $f_0 = e$. We continue until we have produced a "flat" histogram. We define "flat" to mean that the minimum value of the histogram divided by the mean value of the histogram is greater than some set constant (we use 0.93 for most runs). Once we have a flat histogram we reset the histogram by setting $H(E) = 0$ for all E and reduce f with the rule $f_{n+1} = \sqrt{f_n}$. We stop the simulation when f falls below a preset value. Note that in practice we work with the logarithms of $g(E)$ and f rather than their full values to avoid losses of precision. The code for this is in Appendix B.1. The Wang-Landau move itself is part of the code in *flMCStep* in the

FLFLuidFncs.f95 module on pages 11-13 with the histogram and density of states being updated by the function *updateWL* on page 14. Checking to see if the histogram is flat and incrementing accordingly is done in *FLMain.f95* on page 2. We use the function *binE* in *FLFluidFncs.f95* on page 11 to put the energy into the correct histogram bin.

Unlike Metropolis sampling, the data from a Wang-Landau run is not useful without some post-processing analysis because all of our data needs to be reweighted by that factor of $e^{-\beta E}$. Fortunately, this is a quick and simple computation. Suppose we have data for an observable x. We need two pieces of data for each step i: the value of x at that step x_i and the energy E_i at that step. We then construct the two dimensional histogram $H(E, x)$. The average value of x, $< x >$, for a particular value of β can be found from this histogram using

$$< x >= \frac{1}{\sum_i \sum_j g(E_j) e^{-\beta E_j} H(E_j, x_i)} \sum_i x_i \left(\sum_j g(E_j) e^{-\beta E_j} H(E_j, x_i) \right) \tag{81}$$

An example of this reweighting can be found in Appendix B.4 on line 40 of the Matlab function *reweight.m*.

References for this section: Gould et al Ch. 15 [14], Wang and Landau [31], [32]

3.2.6 Entropic Sampling

In many papers one will find references to "entropic sampling" so, although we did not implement this sampling technique, we give a brief mention of it here. This technique, first proposed by Jooyoung Lee in 1993 [19], is similar to Wang-Landau sampling. As with Wang-Landau sampling, we construct $S(E) = \ln(g(E))$ and we sample in the same manner

$$acc(E_0 \to E_1) = \min\left(1, \frac{\exp(S(E_0))}{\exp(S(E_1))}\right). \tag{82}$$

However, for this method we do not construct $S(E)$ during the simulation. Instead, we run one short simulation with $S_0(E) = 0$ for all E and construct the histogram $H_0(E)$. We then make a new estimate $S_1(E)$ for $S(E)$ as

$$S_1(E) = \begin{cases} S_0(E) & H_0(E) = 0 \\ S_0(E) + \ln(H_0(E)) & \text{else} \end{cases} \tag{83}$$

For our next run we can use this as the estimate of $S(E)$, which, in turn, generates a better guess at $S(E)$. Thus this algorithm is thus a cross between Wang-Landau and Multicanonical sampling.

References for this section: Lee [19]

3.3 Methods for Sampling Quantum Particles

3.3.1 Path Integral Monte Carlo (PIMC)

Recall from the Theory section (equation (53)) that we can treat a quantum particle as a chain made up of P pseudo-particles (or "beads"). If we use this approximation, we can write the probability density for quantum particles in a canonical ensemble as we did in equation (53).

$$W(\mathbf{r}) = \exp(-\beta V(\mathbf{r}))$$

where V is given by equation (54)

$$V(\mathbf{r}) = \frac{1}{P} U(\mathbf{r}) + \kappa(\mathbf{r}).$$

Here $U(\mathbf{r})$ is the classical potential and $\kappa(\mathbf{r})$ can be written for a single quantum particle as shown in equation (56)

$$\kappa(\mathbf{r}) = \left(\frac{Pm}{2\beta^2\hbar^2}\right)\sum_{a=1}^{P}|\mathbf{r}_a - \mathbf{r}_{a+1}|^2.$$

As we mentioned in the Theory section, this equation is reminiscent of a harmonic oscillator potential. In fact, in our simulation we make use of that analogy, visualizing the single quantum particle as a "polymer" of P beads in which neighboring beads are held together with a spring. The full chain contributes its harmonic potential and each bead also contributes $\frac{1}{P}U$ to the total energy of the system. Thus, we can write down a Monte Carlo algorithm for sampling a single quantum particle in a potential U that is very similar in concept to our CMC algorithm:

For each bead:

1. Calculate the energy of the system $E = \kappa + \frac{1}{P}U$
2. Give a bead a random displacement $\mathbf{r}' = \mathbf{r} + \Delta$
3. Calculate the new energy of the system $E' = \kappa' + \frac{1}{P}U'$
4. Accept the move with probability:

$$acc(\mathbf{r} \to \mathbf{r}') = \min(1, \exp(-\beta(E' - E))) \tag{84}$$

Our implementation of PIMC is the function *move* in the module *LPFncs.f95* on page 40 of Appendix B.2.

References for this section: Allen and Tildesley Ch. 10 [1], Boccio Ch. 6 [5], Coker et al [9], Cronin [10], Larrimore [18], Miller and Reese [22], Miller et al [23], Wolfson [36]

3.3.2 PIMC with Threading

When using PIMC, a large number of moves are rejected because the "springs" between the beads are usually very "stiff" thanks to the "kinetic" part of the energy. One method of fixing this, proposed by Pollock and Ceperley [24], is to use "threading." Threading involves directly sampling from the kinetic energy propagator rather than accepting and rejecting moves based on kinetic energy.

Consider the kinetic part of the density matrix W_κ. By direct substitution, we can write $W_\kappa(\mathbf{r}, \mathbf{r}')$ as

$$W_\kappa(\mathbf{r}, \mathbf{r}') = \int W_\kappa(\mathbf{r}, \mathbf{r}'; \beta - \beta')W_\kappa(\mathbf{r}'', \mathbf{r}'; \beta')d\mathbf{r}''. \tag{85}$$

If we iterate this process P times, we find

$$W_\kappa(\mathbf{r}, \mathbf{r}') = \int ... \int W_\kappa(\mathbf{r}, \mathbf{r_1}; \tau)W_\kappa(\mathbf{r_1}, \mathbf{r_2}; \tau)...W_\kappa(\mathbf{r_{P-1}}, \mathbf{r}'; \tau)d\mathbf{r_1}...d\mathbf{r_{P-1}}. \tag{86}$$

Now we can write the integrand of equation (86) as

$$W(\mathbf{r}, \mathbf{r}'; \beta)\left(\frac{W(\mathbf{r}, \mathbf{r_1}; \tau)W(\mathbf{r_1}, \mathbf{r}'; \beta - \tau)}{W(\mathbf{r}, \mathbf{r}'; \beta)}\right)\left(\frac{W(\mathbf{r_1}, \mathbf{r_2}; \tau)W(\mathbf{r_2}, \mathbf{r}'; \beta - 2\tau)}{W(\mathbf{r_1}, \mathbf{r}'; \beta - \tau)}\right)$$
$$... \left(\frac{W(\mathbf{r_{P-2}}, \mathbf{r_{P-1}}; \tau)W(\mathbf{r_{P-1}}, \mathbf{r}'; \beta - \tau)}{W(\mathbf{r_{P-2}}, \mathbf{r}'; 2\tau)}\right). \tag{87}$$

where $\tau = \beta/P$. Note that the extra terms all cancel. Now the nth term brackets in equation (87) is the properly normalized probability that, given endpoints $\mathbf{r_n}$ and $\mathbf{r}'$, the next internal point chosen will be

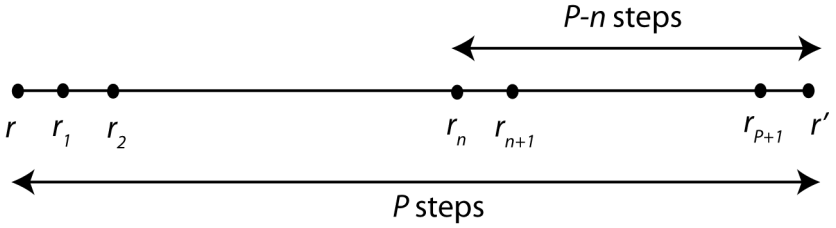


Figure 3.2: The "chain" of possible next states. There are a total of P steps in the chain.

at $\mathbf{r_{n+1}}$. To lowest order in τ these terms are Gaussians:

$$\begin{aligned} f(\mathbf{r_{n+1}},\tau) &= \frac{W(\mathbf{r_n},\mathbf{r_{n+1}};\tau)W(\mathbf{r_{n+1}},\mathbf{r}';\beta-(n+1)\tau)}{W(\mathbf{r_n},\mathbf{r}';\beta-n\tau)} \\ &= \frac{1}{\sqrt{2\pi\sigma_{n+1}^2}}\exp\left(\frac{-(\mathbf{r_{n+1}}-\bar{\mathbf{r}}_{\mathbf{n+1}})^{\mathbf{2}}}{2\sigma_{n+1}^2}\right) \end{aligned} \tag{88}$$

Now we just need expressions for $\bar{\mathbf{r}}_{\mathbf{n+1}}$ and σ_{n+1}. Clearly, the average value for $\bar{\mathbf{r}}_{\mathbf{n+1}}$ should be one step along the chain from $\mathbf{r_n}$ to $\mathbf{r}'$. The chain has a total of P steps with $P-n$ steps between $\mathbf{r_n}$ and $\mathbf{r}'$ (see Figure 3.2). Thus

$$\bar{\mathbf{r}}_{\mathbf{n+1}} = \mathbf{r_n} + \frac{1}{P-n}(\mathbf{r}'-\mathbf{r_n}). \tag{89}$$

Rearranging equation (89) and multiplying by τ/τ

$$\bar{\mathbf{r}}_{\mathbf{n+1}} = \frac{\beta-(n+1)\tau}{\beta-n\tau}\mathbf{r_n} + \frac{\tau}{\beta-n\tau}\mathbf{r}'. \tag{90}$$

We can find σ_{n+1} by normalizing f. We find

$$\sigma_{n+1}^2 = \tau\Lambda^2\left(\frac{\beta-(n+1)\tau}{\beta-n\tau}\right) \tag{91}$$

Pollock and Ceperley [24] show this algebraically.

Thus we can use equation (88) to approximate the acceptance probability for a single bead move as a Gaussian. Then, instead of doing trial moves and explicitly calculating "kinetic" energy, we just pick trial moves from the appropriate Gaussian distribution. To speed up the process, we actually move a number of sequential beads (a "thread") at one time. This guarantees that we will have the correct distribution of moves under the kinetic energy propagator. We then accept and reject moves explicitly based only on the potential energy U. Our implemenation of threading is the *moveThread* function of the module *LPFncs.f95* on pages 40-41 of Appendix B.2. We pick trial moves from the correct Gaussian distribution in the function *tryboth* on page 44 and calculate potential energy in the function *lpflPot* on page 39.

References for this section: Cronin [10], Pollock and Ceperly [24], Wolfson [36]

3.3.3 Quantum Wang-Landau

We can also adapt the Wang-Landau method to simulate a quantum particle. However, now we have β both in the numerator and denominator of the exponent in the integrand of our partition function:

$$Q = \int \exp\left(-\frac{\kappa(\mathbf{r}^N,\mathbf{p}^N)}{\beta} - \beta U(\mathbf{r}^N)\right) d\mathbf{r}^N d\mathbf{p}^N. \tag{92}$$

Vorontsov-Velyaminov and Lyubartsev [30] point out that this difference in β dependence does not prevent us from using Wang-Landau, but does require that we sample a two-dimensional density of states

$g(E_1, E_2)$, where E_1 corresponds to kinetic energy, and E_2 corresponds to potential energy. Rewriting the partition function in terms of E_1, E_2 and $g(E_1, E_2)$ gives

$$Q = \int \exp\left(-\frac{E_1}{\beta} - \beta E_2\right) g(E_1, E_2) dE_1 dE_2 \tag{93}$$

Thus, because g is a function of two variables, we must construct a two-dimensional histogram when carrying out the Wang-Landau method in order to find observables. Reweighting the simulation data to find the average value of an observable is similar to the classical case, but now we need to store the kinetic and potential energies separately so that they can be properly reweighted at the end of the simulation. When we actually reweight an observable we now need a three-dimensional histogram consisting of kinetic energy, potential energy, and the observable. See the function *reweightSHO.m* in Appendix B.4 for an example of how this is implemented. Otherwise, however, we do not need to modify our algorithm for the Wang-Landau method from the classical case. A Wang-Landau move is implemented in the function *moveWL* on page 43 of Appendix B.2 in the module *LPFncs.f95*. We calculate energies in this same module using the functions *kinEnergy*, *kin1* and *potEnergy* on pages 38 and 44 and bin them using the functions *binKin* and *binPot* in the *LPGenFncs.f95* module on page 46.

References for this section: Gould et al Ch. 15 [14], Vorontsov-Velyaminov and Lyubartsev [30], Wang and Landau [31], [32]

3.4 Methods for Determining Chemical Potential

For simulations in a grand canonical ensemble we need to input a chemical potential and therefore we require a method for determining the chemical potential of a system. Here we discuss algorithms that can be used to find the chemical potential for a simulation.

3.4.1 Widom Test Particle Insertion Method

The Widom Test Particle Insertion Method is a simple way of calculating the chemical potential during a simulation. We start from the thermodynamic definition of chemical potential

$$\mu = \left(\frac{\partial F}{\partial N}\right)_{VT} \tag{94}$$

where F is the free energy of the system. Recall from equation (41) that we can write μ as

$$\begin{aligned} \mu &= -k_B T \ln\left(\frac{V}{\Lambda^3(N+1)}\right) - k_B T \ln\left(\frac{\int \exp(-\beta U(\mathbf{s}^{N+1}; L)) d\mathbf{s}^{N+1}}{\int \exp(-\beta U(\mathbf{s}^N; L)) d\mathbf{s}^N}\right) \\ &= \mu_{id} + \mu_{ex}, \end{aligned}$$

separating μ into an ideal gas component (which we can solve for analytically) and an "excess" component. Now let $\Delta U = U(\mathbf{s}^{N+1}; L) - U(\mathbf{s}^N; L)$. Then we can write the excess chemical potential as

$$\mu_{ex} = -k_B T \ln\left(\int < \exp(-\beta \Delta U) >_N d\mathbf{s}^{N+1}\right) \tag{95}$$

where $< \cdots >_N$ is the canonical ensemble average over the N-particle system. Therefore, we now have μ_{ex} in a form we can compute using Monte Carlo sampling. To carry out this calculation, we create an N-particle system and then compute the change in energy that would result if we added another the particle to the system in a random location. Note that we do not actually add the particle to the simulation; the particle number always remains fixed at N. The average of $\exp(-\beta \Delta U)$ over all random locations gives the value for the integral in equation (95). Note that this method is efficient and reasonably accurate, but only suited to low density simulations. Our implementation of the Widom Method can be found in the function *widomTest* on page 15 of Appendix B.1 as part of the *FLFluidFncs.f95* module.

References for this section: Frenkel and Smit Ch. 7 [13], Widom [33]

3.4.2 Bennet Method

The Bennet Method, or the Overlapping Distribution Method, is another method for determining chemical potential that, unlike the Widom Method, does not fail at high density. However, it is less efficient as it requires that two different simulations be run.

Consider two N-particle systems, system 0 and system 1, with partition functions Q_0 and Q_1. For convenience's sake, we assume these systems have the same volume. The difference in free energy between these systems is given by

$$\begin{aligned} \Delta F &= -k_B T \ln\left(\frac{Q_1}{Q_0}\right) \\ &= -k_B T \ln\left(\frac{\int \exp(-\beta U_1(\mathbf{s}^N; L)) d\mathbf{s}^N}{\int \exp(-\beta U_0(\mathbf{s}^N; L)) d\mathbf{s}^N}\right). \end{aligned} \tag{96}$$

Consider the energy difference $\Delta U = U_1(\mathbf{s}^N; L) - U_0(\mathbf{s}^N; L)$ between the two systems. The probability density function for ΔU when sampling system 1 is given by

$$W_1(\Delta U) = \frac{\int \exp(-\beta U_1)\delta(U_1 - U_0 - \Delta U) d\mathbf{s}^N}{q_1} \tag{97}$$

where $Q_1 = \int \exp(-\beta U_1(\mathbf{s}^N)) d\mathbf{s}^N$. Using properties of the delta function we can rewrite W_1 in the following manner:

$$\begin{aligned} W_1(\Delta U) &= \frac{\int \exp(-\beta(U_0 + \Delta U))\delta(U_1 - U_0 - \Delta U) d\mathbf{s}^N}{Q_1} \\ &= \frac{Q_0}{Q_1} \exp(-\beta \Delta U) \frac{\int \exp(-\beta U_0)\delta(U_1 - U_0 - \Delta U) d\mathbf{s}^N}{Q_0} \\ &= \frac{Q_0}{Q_1} \exp(-\beta \Delta U) W_0(\Delta U) \end{aligned} \tag{98}$$

where W_0 is the probability of finding ΔU when sampling from system 0. Now $\Delta F = -k_B T \ln\left(\frac{Q_1}{Q_0}\right)$ and therefore

$$\ln(W_1(\Delta U)) = \beta(\Delta F - \Delta U) + \ln(W_0(\Delta U)). \tag{99}$$

We define the quantities f_0 and f_1

$$f_0(\Delta U) = \ln(W_0(\Delta U)) - \frac{\beta \Delta U}{2} \tag{100}$$

$$f_1(\Delta U) = \ln(W_1(\Delta U)) + \frac{\beta \Delta U}{2} \tag{101}$$

such that

$$f_1(\Delta U) = f_0(\Delta U) + \beta \Delta F. \tag{102}$$

If we have two systems, we can find ΔF by fitting a polynomial in ΔU to the data for f_1 and f_0 and looking at the difference between them. This difference should be a constant equal to $\beta \Delta F$.

For the particular problem of calculating chemical potential, we can consider system 1 to have $N + 1$ particles and system 0 to have N particles. From equation (95) it is clear that the difference in free energy between these systems is μ_{ex}. Thus

$$\beta \mu_{ex} = f_1(\Delta U) - f_0(\Delta U). \tag{103}$$

In practice, we run two simulations, one with N particles and one with $N + 1$ particles. Similar to the Widom Method, in the $N+1$ particle system, we record the change in potential energy were we to *destroy*

one of the particles. Conversely, in the N particle system we record is the change in potential energy were we to *create* a particle. While it might seem that this could all be done in the same system (by recording the energy change both when we remove and when we create a particle), it turns out at that at a relatively low number of particles (hundreds), the exact size of the system can make a difference and we really do need two simulations, one with N particles and one with $N + 1$.

References for this section: Bennet [3], Frenkel and Smit Ch. 7 [13]

Other sampling methods for classical and light particles can be found in the literature. We have given a broad overview of the topic and have discussed enough possible sampling methods to explain the significance of our results.

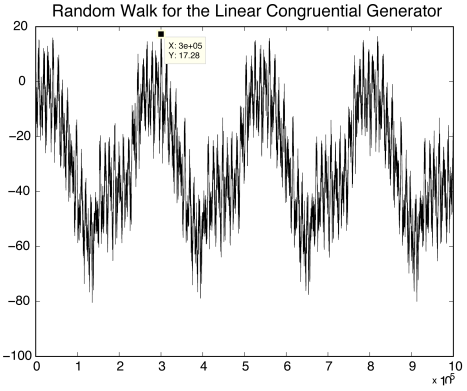
4 Results and Discussion

4.1 Random Number Generator Tests

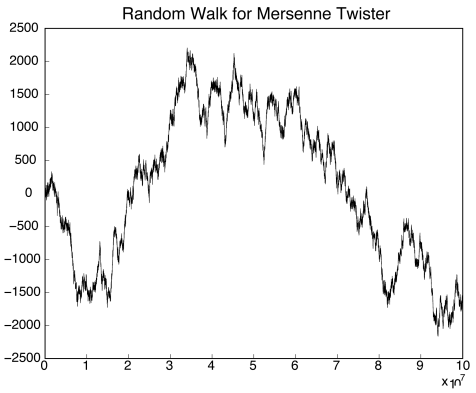
Firstly, we discuss our results from the random number generator tests. These are not research results, but a working random number generator is a prerequisite to all of the results that follow in the next sections. We use a Mersenne Twister (MT) generator in our simulations, but for comparison, we here present tests of both the MT generator and a linear congruential (LC) generator with reshuffling, the algorithms for which can be found in Appendix B.3.

4.1.1 Periodicity

We ran random walks using the MT and LC generators. The two plots are shown in Figures 4.1(a) and 4.1(b). The period of the linear congruential generator is very clear; it looks to be approximately



4.1(a) Periodicity test for linear congruential random number generator.



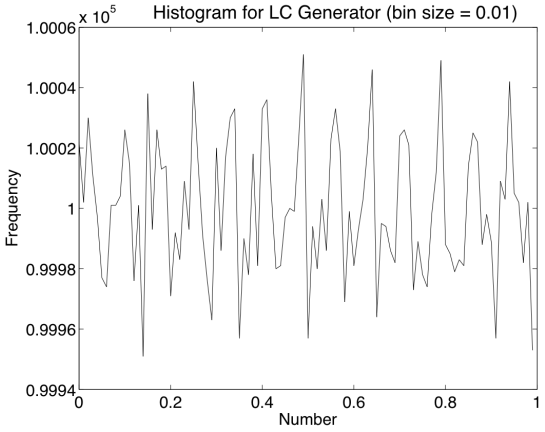
4.1(b) Periodicity test for Mersenne Twister generator.

Figure 4.1: Periodicity Tests

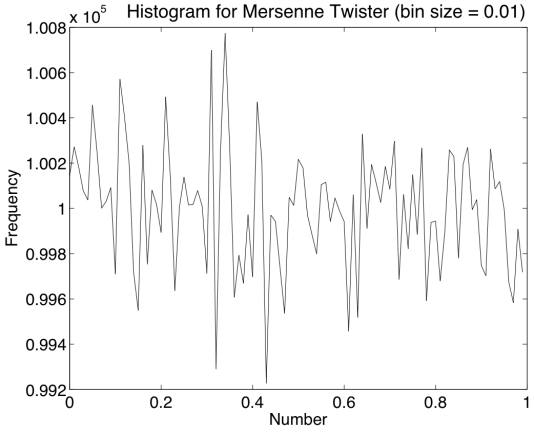
250,000 steps. However, despite running the Mersenne Twister for a hundred million steps, we are unable to identify the length of the period. Since most of our runs are on the order of a million steps and approximately one billion random numbers, it is clear that the Mersenne Twister is better suited to our needs.

4.1.2 Uniformity

The histograms for the linear congruential generator and the Mersenne Twister generator for ten million random numbers are given in Figures 4.2(a) and 4.2(b). These both appear somewhat "flat," and we can confirm that appearance if we define the "flatness" of a histogram as the minimum value of the histogram



4.2(a) Uniformity Test for linear congruential random number generator.



4.2(b) Uniformity Test for Mersenne Twister generator.

Figure 4.2: Uniformity Tests

divided by the mean value. With that definition, the linear congruential generator has a flatness of 0.9995 and the Mersenne Twister has a flatness of 0.9923.

4.1.3 Chi-Square Test

We compute χ^2 for the linear congruential and the Mersenne Twister generator using the histogram created when we tested for uniformity. For the Mersenne Twister $\chi^2 = 76.8599$ and for the linear congruential generator $\chi^2 = 0.5263$. Since, in this case, $M = 100$, the Mersenne Twister generator passes this test. However, the linear congruential generator has an incredibly tiny χ^2 confirming our hypothesis that we are running close to a multiple of the period. However, since the period is so small, this is hard to avoid.

4.1.4 Hidden Correlations

Neither generator appears to have any hidden correlations. The plots are not included because they are simply filled-in unit squares.

For all of the results presented in the remainder of this section, we have used the MT generator.

4.2 CMC Simulation Results

4.2.1 Comparison of Average Potential Energy

We begin with a simple comparison of our original Metropolis CMC simulation with literature values. We ran the simulation with 864 particles at a reduced density of 0.45 and a temperature of 1.764 (parameters picked to match [29]) for 15000 time steps starting from an ordered configuration and recorded the average potential energy per particle at each time step. The resulting graph of particle energy at each time step is shown in Figure 4.3. The tail correction (equation (59)) has been included in the graph.

We find an average energy of 2.91 ± 0.03. For these same parameters, Verlet [29] found an average potential of -2.89, which is within the error of our result.

We also must ask ourselves how we know that our simulation has run long enough. We first must make sure the system is equilibrated. Here, we do this by plotting the running average and considering the system "equilibrated" when the running average is always within 5% of the time average (this run took

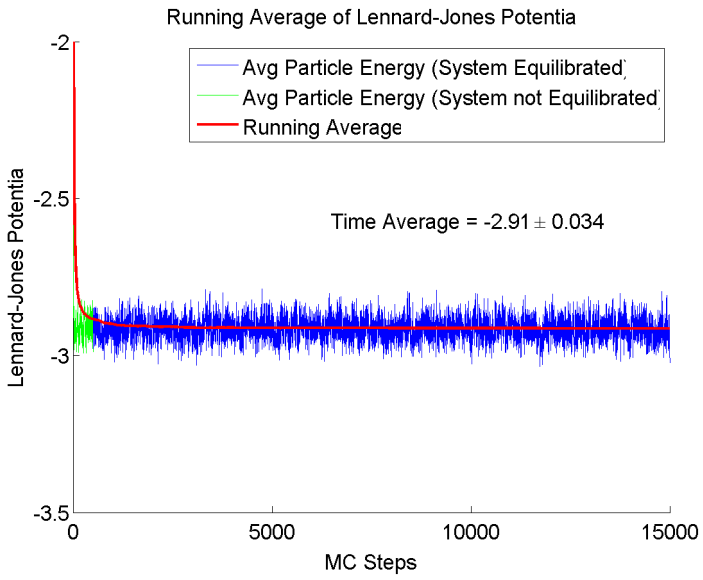


Figure 4.3: Average potential energy for our Metropolis CMC simulation.

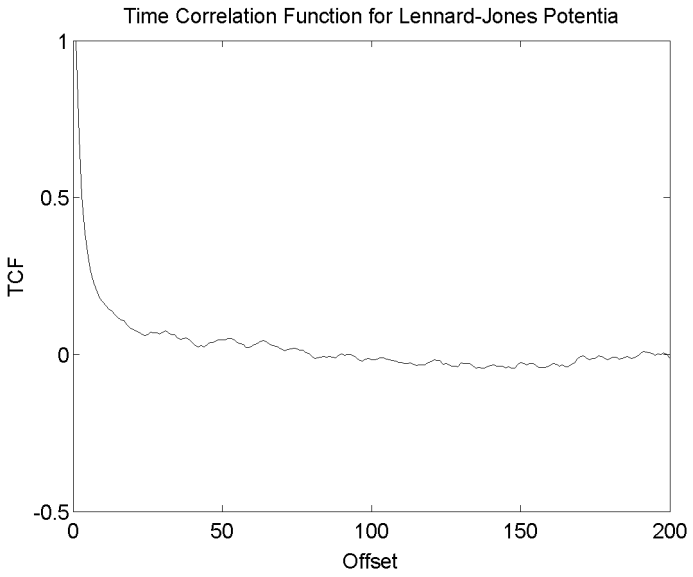


Figure 4.4: Time correlation function for a CMC run.

approximately 500 steps to equilibrate). Once we have an equilibrated system, we need to know how long we should run the system afterwards to allow it to fully explore the state space. We do this in two steps.

We first compute the normalized time correlation function of the data, which is shown in Figure 4.4. From this, we estimate how many time steps are "correlated" by finding the first point where the time correlation function drops below 0.01. For the potential energy for this fluid simulation, we find that energies separated by at least 80 times steps are uncorrelated.

We then arrange our data into "blocks" of large numbers of data points in an attempt to find how long we need to run a simulation (see the function *blocking.m* in Appendix B.4). Basically, we want blocks for which the error between the means of the blocks is of the same order as the error of the means within

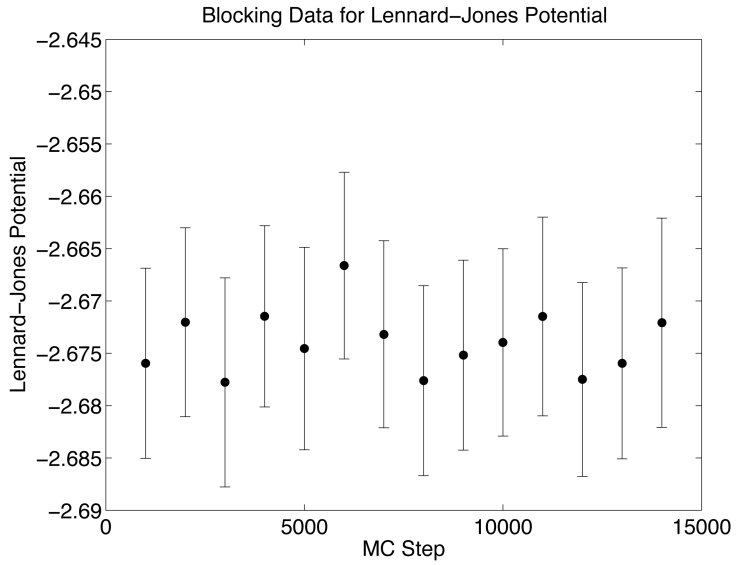


Figure 4.5: Blocks for a CMC run. The error bars show the error within each block.

the blocks. To compute the error of the mean within a block, we divide the standard deviation of a block by the square root of the number of *uncorrelated* data points in the block. Figure 4.5 shows data blocks of 1000 time steps (corresponding to 1000/80 = 12.5 uncorrelated points). Here the average error within one block is 0.0092 while the standard deviation of the block means is 0.0031. These are "on the same order" and we conclude that the system needs approximately 1000 steps to explore the full space.

4.2.2 Calculation of Pressure

As a matter of practical interest, as well as a test of our fluid simulation, we can compute the pressure at different densities. Using a temperature of $T^* = 2$ and $N = 500$ particles, we ran simulations at 9 different densities and computed the pressure using equation (60) and adding in the correction given in equation (63). We ran the simulations for 2000 time steps, allowing 500 time steps for the system to equilibrate. A comparison of our data with that of the molecular dynamics simulations from [15] is given in Figure 4.6

Once again, our simulation data is acceptably close to data given in the literature, showing that the basic CMC simulation is sound. Further, we can say something about the physics of the situation. At low densities, the fluid should behave as an ideal gas and P/ρ should be equal to $\frac{3}{2}kT$. Looking at Figure 4.6, we do see this linear relationship in this region. As the density increases, pressure scales as $\rho/\beta + A\rho^2$ where A is a constant. To see this relationship, consider equation (60). The virial is basically a measure of the number of interactions. The number of interactions per particle should scale linearly with density and the virial is N times this quantity. Thus the term vir/V scales as a function of density squared, and we can see this relationship in Figure 4.6.

4.3 GCMC Simulation Results

4.3.1 Widom Test Particle Insertion

The first test of our GCMC simulation involved a comparison to runs of our CMC simulation. We ran several CMC simulations at different densities and, as a run at a given density proceeded, used the Widom method to determine the corresponding chemical potential, μ. The simulations were done for a total of 2000 steps using a temperature of $T^* = 2$. For the CMC simulation, we used $N = 500$ Lennard-Jones

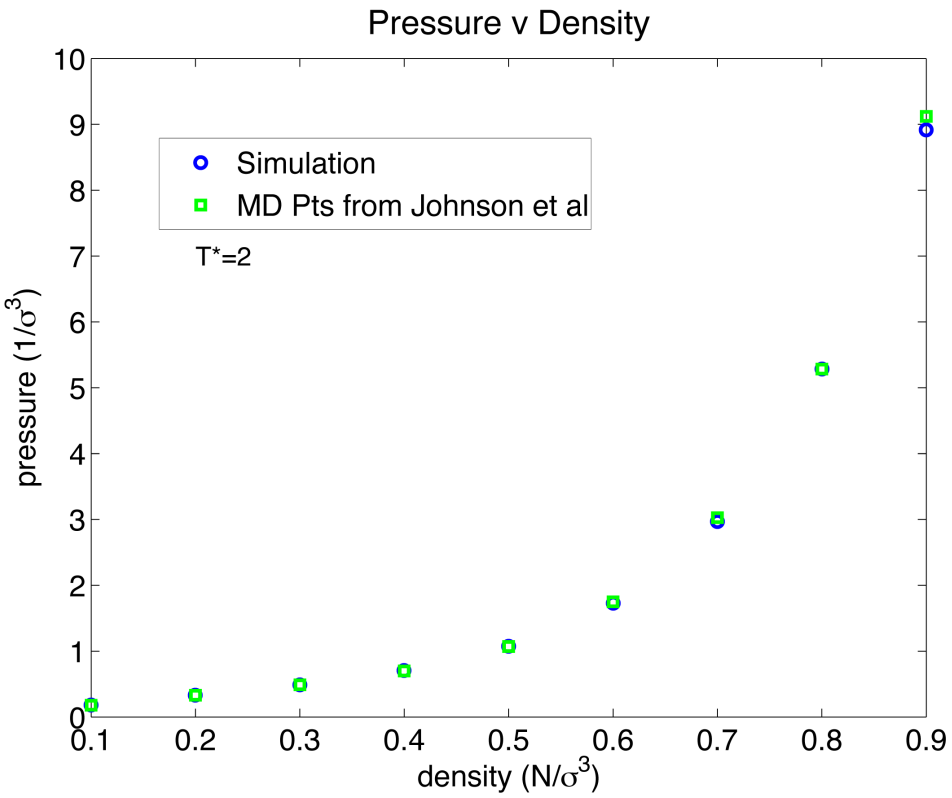


Figure 4.6: A comparison of pressure data between our simulation and literature.

particles. We then ran independent GCMC simulations at the chemical potentials calculated from the CMC simulations, and recorded the average density of the GCMC runs. The data are shown in Table 1. Graphical presentation of the data shown in Table 1 is given in Figure 4.7.

CMC Density ($\frac{N}{\sigma^3}$)	GCMC Density ($\frac{N}{\sigma^3}$)	Error (%)
0.1	0.0974	2.6
0.15	0.1514	0.93
0.2	0.2070	3.5
0.25	0.2700	8.0
0.3	0.3029	1.0
0.35	0.3529	0.83
0.4	0.3926	1.7
0.45	0.4681	4.0
0.5	0.4909	1.8

Table 1: The average density of GCMC simulations (middle column) run at chemical potentials found from CMC simulations (density left column).

We see from Table 1 and Figure 4.7 that the density obtained from the GCMC simulation is close to the original density of the CMC simulation. This is a good indication that both the grand-canconical sampling and the Widom method are working correctly, and give a consistent picture of the fluid in equilibrium.

As another test, we can use the Widom method during the GCMC simulation to calculate chemical potential as the run proceeds. If the Widom Method is working properly, it should calculate the same

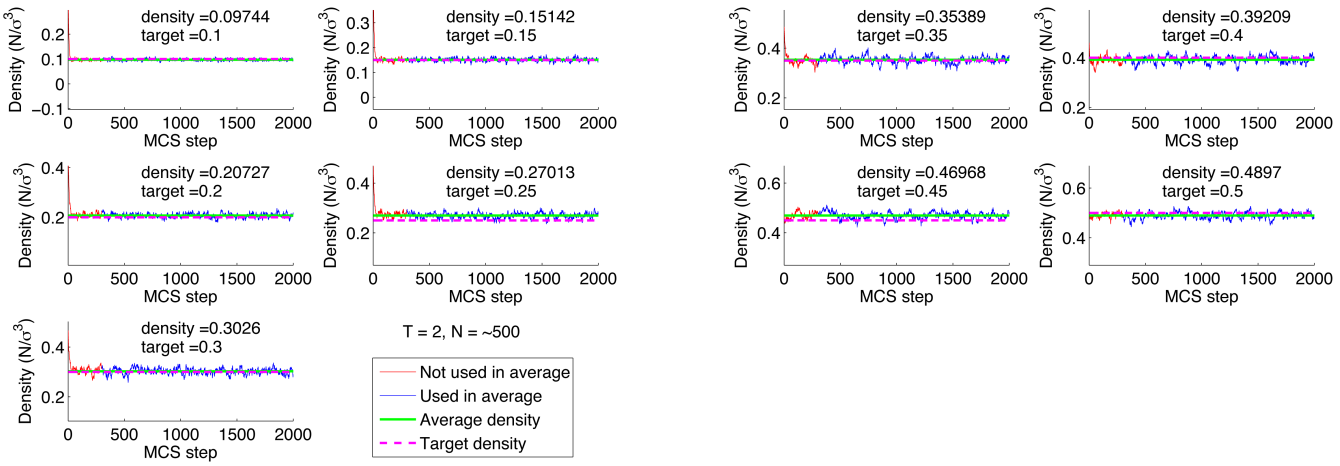


Figure 4.7: Density from GCMC simulations compared with density from CMC simulations. Here "target" density refers to the density from the CMC simulations that we hope is matched by the GCMC simulation.

chemical potential as originally input. For these simulations, we actually use the activity $z = \frac{\exp(\beta\mu)}{\Lambda^3}$ where Λ is the thermal deBroglie wavelength (see the code in Appendix B.1). The results of this test are given in Table 2 and Figure 4.8.

Input Activity ($\frac{1}{\sigma^3}$)	Widom Test Activity ($\frac{1}{\sigma^3}$)	Error (%)
0.0820	0.0827	0.85
0.1184	0.1193	0.76
0.1541	0.1575	2.2
0.1956	0.1769	9.6
0.2178	0.2090	4.0
0.2585	0.2707	4.7
0.3002	0.2768	7.8
0.4211	0.4258	1.1
0.4608	0.4162	9.7

Table 2: The activity calculated by the Widom method (middle column) and the original inputted activity (left column). The agreement between these two columns is an indication that our Widom method and GCMC simulation are performing correctly.

Once again, we have evidence that our simulation, and in particular our implementation of the Widom method, is sound. The GCMC simulations give back almost exactly the same activity as was originally input. However, we do see a tendency for the error in activity to increase at higher density. This tendency owes to the issues with the Widom method at higher densities discussed in section 3.4.1. Should one need a method more accurate at high densities one could implement the Bennet method described in 3.4.2.

4.3.2 Multicanonical Results

In order to get interesting data for the Multicanonical runs, we first had to find a point near the liquid-gas phase transition. For the truncated Lennard-Jones potential for argon, the critical point is at $\beta = 0.842$ and $z = 0.0388$ [35]. We chose a temperature just a little below the critical temperature and adjusted the value of μ until we found the value of μ that would place us on the coexistence curve. This "adjustment" of μ was an iterative process. We started with the value of μ almost at the critical point and did a

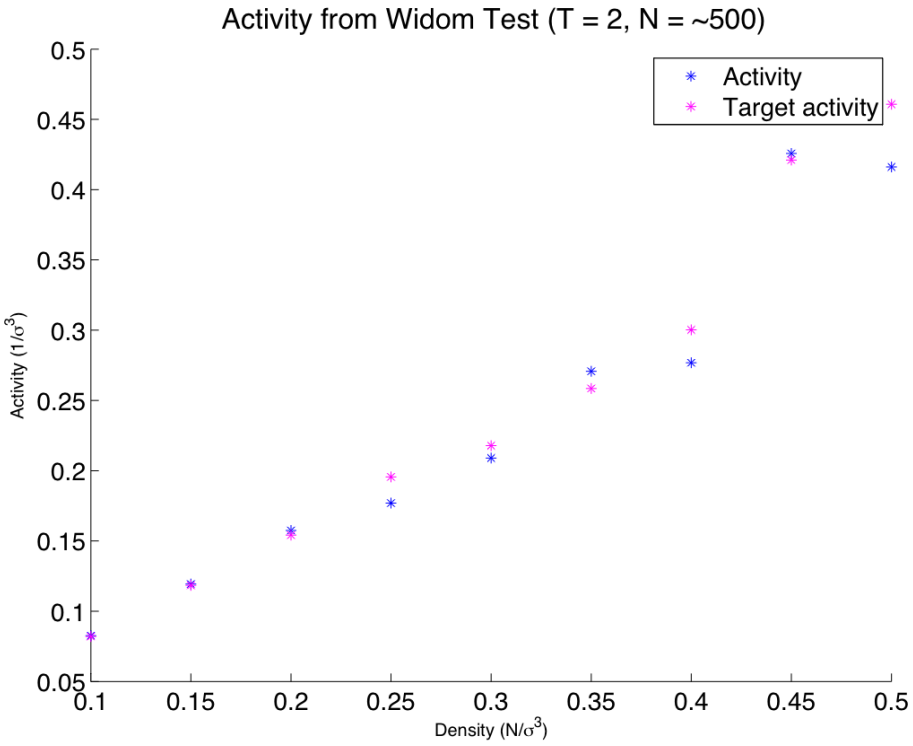


Figure 4.8: Activity as calculated by Widom Test vs the input ("target") activity.

run. When that run finished, we took the resulting histogram (which did not show phase coexistence) and used histogram reweighting as in equation (77) until we found a value of μ that produced a density histogram with two peaks of roughly the same height. We then used this value of μ in simulation and repeated the process until we found a value for μ where the simulation produced a density histogram with two peaks of approximately the same height with the same area underneath. The input parameters of $z = 0.0597103$ (this precision is needed) and $\beta = 0.855$ eventually produced the density plot shown in Figure 4.9. Note that nothing yet discussed has actually involved Multicanonical sampling; we were still just attempting to find a first guess for a preweighting function.

Just observing the data in Figure 4.9 by eye, we can conclude that the system appears to be spending approximately equal time in the liquid and gas phases. We can do a better analysis, however, computing the histogram shown in Figure 4.10.

As a last step in implementing the Multicanonical technique, we used a smoothed version of the histogram shown in Figure 4.10 as a preweighting function for a Multicanonical simulation run with the same parameters ($\beta = 0.855$ and $z = 0.0597103$). A summary of that run is shown in Figure 4.11.

With the exception of the large left-hand spike in the raw data histogram shown in the bottom-left of Figure 4.11, we obtain a flat histogram of raw densities. The "spike" is caused because in the original (unweighted) simulation, the simulation was never able to explore that area. The "perfect" histogram for these parameters would have slightly-larger-than-zero values in this region, but our simulation never explored that space at all. Note that we will almost always have this "edge" to our data. This is the region that Multicanonical sampling is "pushed" into where it really should not be. Looking closely at Figure 4.11, we see that there is also a jagged edge on the high density side. However, this edge is damped down because the energy is higher in this region and the $e^{-\beta H}$ term in equation (71) forces the height of the spike down. However, the spikes do not affect the data in the area of interest (they are, in fact, there because they are *not* in the area of interest); the re-biased histogram in the bottom-right of Figure

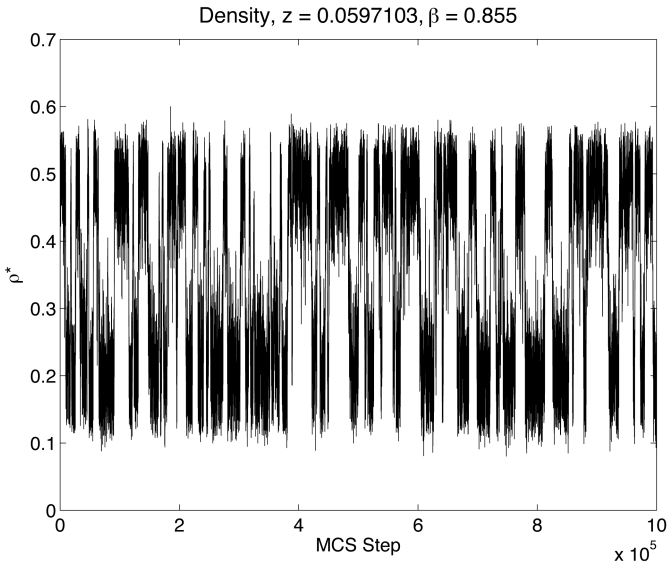


Figure 4.9: Reduced density over the GCMC run

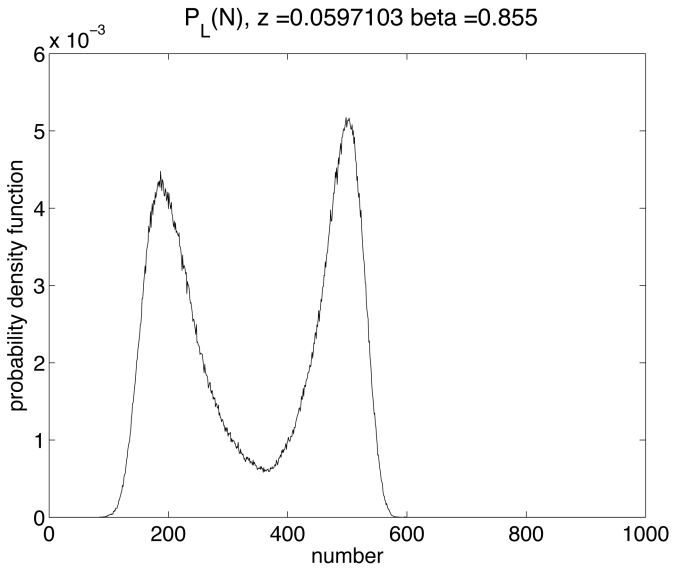


Figure 4.10: Density histogram

4.11 is similar in form to the original simulation. We conclude that the Multicanonical simulations of the fluid are giving accurate results.

4.4 Wang-Landau Results

4.4.1 Classical Wang-Landau Results

The Multicanonical method is not as useful as it first appears. Simply finding that first point to create figure 4.10 took a number of simulations because we had to iteratively calculate μ. Should we wish to explore a system at a different temperature, we would have to reweight the histogram to a different temperature and chemical potential before we could use it as our preweighting function. However, this reweighting in temperature does not give a completely accurate histogram; the reweighting only varies

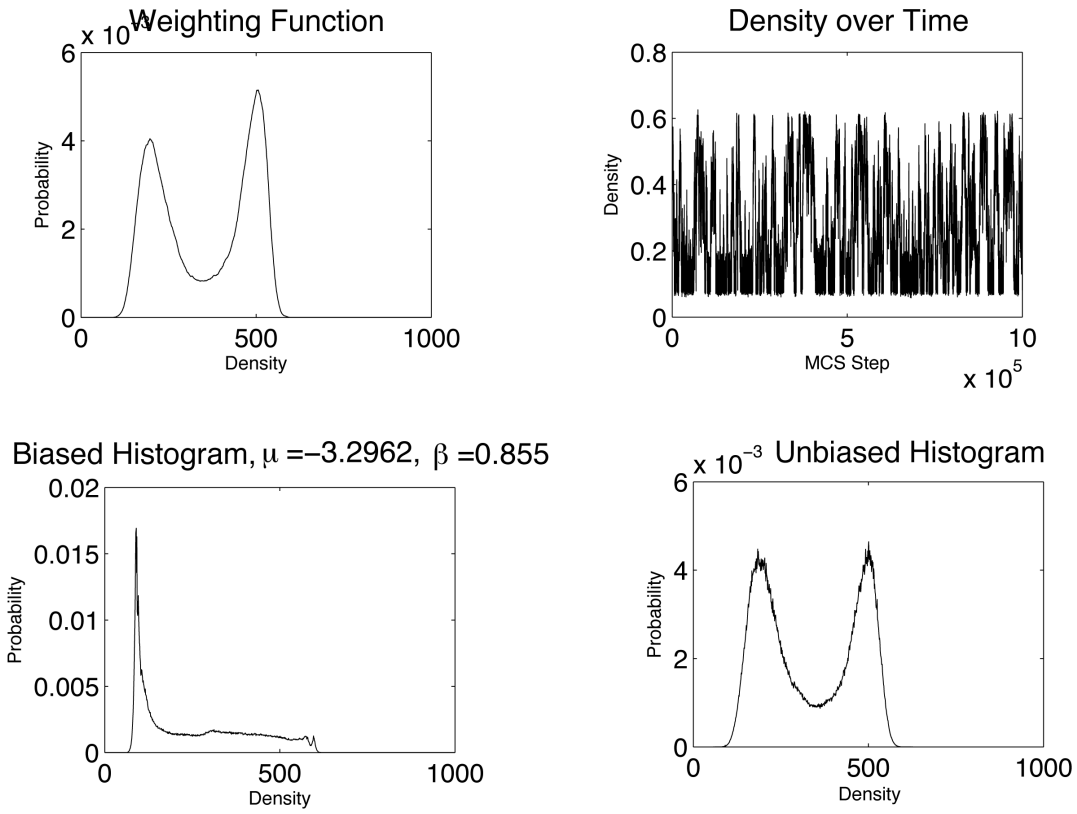


Figure 4.11: Top-left: The preweighting function. Top-right: Raw densities over the simulation. Bottom-left: The density histogram of the raw density data before unweighting it with the preweighting function. Bottom-right: The density histogram after unweighting with the preweighting function.

the heights of the peaks without moving them further apart. In fact, if we look at the mathematics of the situation, reweighting *cannot* move the peaks further apart. The top of the peak is a maximum and thus the first derivative with respect to number of particles is zero. Therefore, there is no set "direction" for the peaks to move with changing temperature. Thus changes in temperature have to be very small so that the fluid and gas densities are approximately the same in order to have this method work at all. Trying to use Multicanonical sampling and histogram reweighting to trace a coexistence curve or find a stable point low on the curve can be done [35], but it is a long and tedious process. Adding a light particle to the system would triple the number of preweighting functions required (since we need one for kinetic and potential energy of the light particle) and, in all probability, make simulation at more than a few temperatures extremely difficult.

Thus we consider another sampling method that does not require a preweighting function or a temperature of simulation. The Wang-Landau method takes longer than any single Multicanonical simulation, but gives information about all temperatures and thus does not require the iterative methods of multicaonical sampling.

Our first step upon implementing Wang-Landau sampling for the argon fluid was to try to compare our results with those in the literature. We ran one Wang-Landau simulation for a canonical ensemble ($\rho^* = 0.1$, $N = 100$) and used the resulting density of states to calculate the average potential energies at numerous different temperatures. For this system, we could pick "all E" by looking at our data from previous Metropolis runs, such as that shown in Figure 4.3. From these data, we chose a lower bound of -4 reduced units per particle and an upper bound of 0 (recall that we are bounded above by zero). The density of states as it evolved over the run is shown in Figure 4.12. In this figure, an "iteration" refers

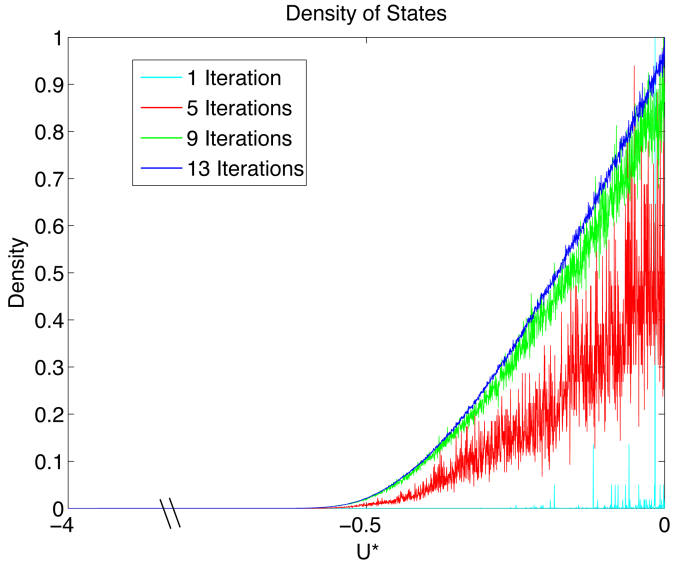


Figure 4.12: Evolution of density of states for a Wang-Landau simulation.

to a point in the run where we have a flat histogram and are about to decrease our f value in Equation (80). In the calculations and results presented below, we used $g(E)$ after 13 iterations.

From equation (78), the potential energy $U(\beta)$ at a particular value for β is given by

$$U(\beta) = \int_{allE} \frac{1}{Q} g(E) e^{-\beta E} dE. \tag{104}$$

Since $g(E)$ is necessarily discrete in a numerical simulation, we can evaluate this integral as a sum in post-processing (see *reweight.m* in Appendix B.4). Doing so for a range of temperatures gives average potential energy as a function of temperature. Making use of the data in Johnson, et al [15] for comparison purposes once again, we get the results shown in Figure 4.13. Most of the results we compare to are from Molecular Dynamics (MD) simulations, but we also included the few Monte Carlo simulation data points.

This is a very impressive result. *One* Wang-Landau simulation gives data that would, for this particular figure, take 12 MD or Metropolis sampling MC simulations to reproduce.

Looking at the figure, we do see that Wang-Landau appears to be less accurate at low temperatures. This is for a variety of reasons. First of all, the simulations from [15] to which we are comparing have slightly different parameters, which could cause some minor differences in results. The MD simulations utilize another algorithm entirely and even the MC simulations use a different cutoff radius than we do. We guess that the greatest inaccuracy occurs, however, because, of necessity, the Wang-Landau simulation has a lower and upper bound set on the energies it can explore. We set up the historgram for the density of states *before* running the simulation. Thus we, as the programmer, artificially set bounds on how low and how high the energy in the simulation can reach by the bounds we set on the density of states. In this case, they were $U^* = 0$ and $U^* = -4$. Clearly, $U^* = -4$ was not quite low enough and the system might, if we had let it, have spent time in a lower energy state. It is true that there is an absolute lowest energy for this configuration; however, if we allowed the simulation to explore all the way down to that energy, the simulation would take much longer to run without generating much more useful data. If we were to run a simulation where we knew we wanted high temperature results, we would use a lower lower bound on energy (and probably a lower upper bound as well to keep the simulation from running too long).

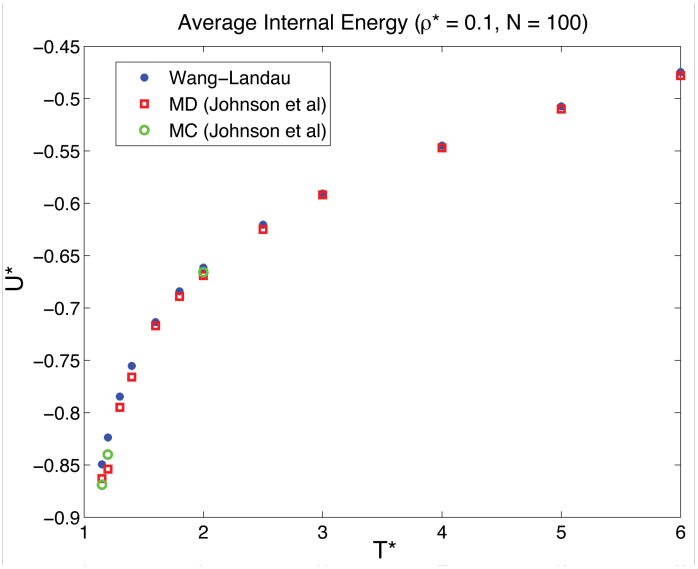


Figure 4.13: Average internal energy as obtained from a Wang-Landau simulation at different temperatures.

4.4.2 Quantum Wang-Landau Simulation Results

Our eventual goal is to use Wang-Landau to simulate a light particle in a fluid. Thus, we first implemented Wang-Landau PIMC for a light particle in a one-dimensional, simple harmonic oscillator (SHO) potential. Interestingly, the SHO has an analytical solution not just for the "exact" case of $P = \infty$ but even at a finite number of beads P [30], though this expression can be difficult to evaluate as P increases. We tested our implementation of Wang-Landau on a system with 8 beads. For this system, the analytical solution for average potential energy $< U >$ is

$$< U >_{P=8} = \frac{\beta}{4P^2} \frac{32C^7 - 48C^5 + 20C^3 - 2C}{4C^8 - 8C^6 + 5C^4 - C^2} \tag{105}$$

where $C = 1 + \frac{1}{2}\left(\frac{\beta}{P}\right)^2$. This equation is derived in [30]. Qualitatively we can see that the behavior is correct. At large β, $< U >$ goes as $1/\beta$. The low temperature limit is therefore 0 and not 0.5 as it would be for the $P = \infty$ case. At low β, the expression goes to the equipartition theoretical limit[3] $\frac{1}{2}kT$.

As with our testing of the classical simulation, we ran one Wang-Landau simulation and used that to calculate potential energy at a variety of different values of β. The two-dimensional density of states is shown as a surface plot in Figure 4.14. One potential issue we address here is the that the density of states appears to peak where we cut off kinetic energy and that if we allowed higher kinetic energies, the peak would increase in height. This is true, but it really only effects low values of β where the term $\exp(-E_1/\beta)$ is far from one and therefore a significant factor in the partition function. As a result, so long as we are interested in higher values for β, we can cut off the energy where we do without having it affect the solution too greatly. Note also that the density of states is not discrete, although harmonic

[3]
$$< U >_{\beta\to 0} = \frac{\beta}{4P^2} \frac{32\left(1 + 7/2\left(\frac{\beta^2}{P^2}\right)\right) - 48\left(1 + 5/2\left(\frac{\beta^2}{P^2}\right)\right) + 20\left(1 + 3/2\left(\frac{\beta^2}{P^2}\right)\right) - 2\left(1 + \left(\frac{\beta^2}{P^2}\right)\right)}{4\left(1 + 4\left(\frac{\beta^2}{P^2}\right)\right) - 8\left(1 + 3\left(\frac{\beta^2}{P^2}\right)\right) + 5\left(1 + 2\left(\frac{\beta^2}{P^2}\right)\right) - \left(1 + \left(\frac{\beta^2}{P^2}\right)\right)}$$
$$= \frac{1}{4\beta}\left(2 + 21\left(\frac{\beta^2}{P^2}\right)\right) \approx \frac{1}{2}kT$$

oscillator energy levels are. This is because, although the kinetic and potential energy should always add to a multiple of $\frac{1}{2}\hbar\omega$, their ratio to each other is not fixed. Thus the two dimensional density of states is, in theory, continuous.

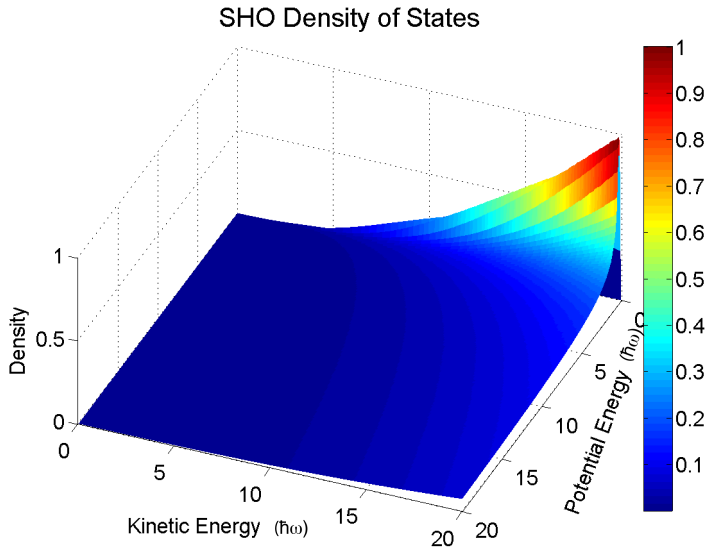


Figure 4.14: Density of states for SHO Wang-Landau simulation.

Finding average potential energy from the density of states in this case is similar to the classical case except that now we need to do the two dimensional integral (see *reweightSHO.m* in Appendix B.4 for an idea of how we might do this integral)

$$U(\beta) = \int_{allE_1}\int_{allE_2} \frac{1}{Q} E_2 g(E_1, E_2) \exp\left(-E_1/\beta - E_2\beta\right) dE_1 dE_2 \tag{106}$$

where E_1 is kinetic energy and E_2 is potential energy.

For comparison, we also ran a Metropolis sampling PIMC at $\beta = 0.5$ and reweighted the resulting histogram with different values of β. The graph comparing analytical potential energy, energy from the Wang-Landau simulation, and energy reweighted from a Metropolis simulation is shown in Figure 4.15. Even in this simple system where we might expect histogram reweighting to be quite effective, it appears that Wang-Landau does a much better job than histogram reweighting. We can make this even clearer by plotting the fractional error of the Wang-Landau and histogram reweighted runs against the analytical value. This plot is shown in Figure 4.16.

From the plot of error, we conclude that Wang-Landau is much more accurate than histogram reweighting; it almost never shows more than a 0.1 fractional error. The reweighted results of a Metropolis sampling run have a consistently larger error except right around $\beta = 0.5$, where the run actually took place. The high error in Wang-Landau at low β arises from the reason discussed above; the range over which we sampled kinetic energies was not quite broad enough to get an accurate result at this high a temperature. Breadth and good resolution are competing needs for the histogram used in the Wang-Landau method.

4.5 Light Particle in a Frozen Fluid

Finally, we put the light-particle in a non-trivial potential. In order to have a physical identity for our light particle, we simulated helium with five beads interacting with a fluid via the Lennard-Jones potential. As

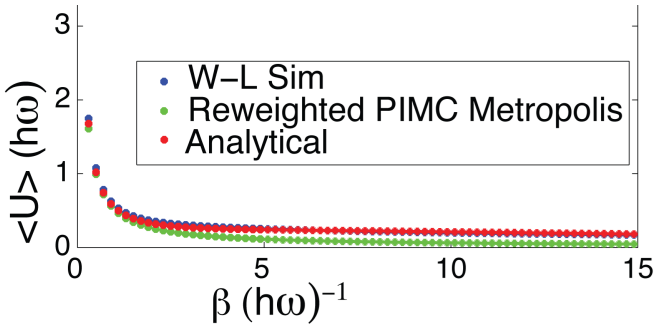


Figure 4.15: Average potential energy for Wang-Landau simulation, reweighted Metropolis sampling, and analytical solution of a simple harmonic oscillator.

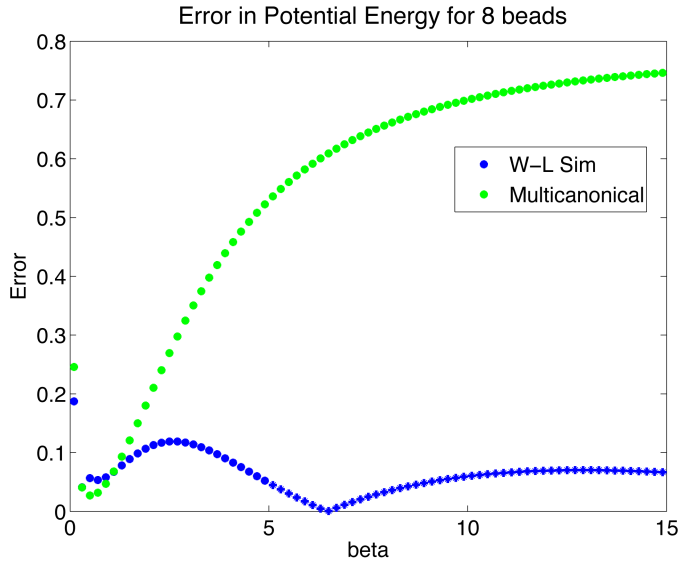


Figure 4.16: The fractional error between the analytical solution and the simulations for potential energy.

a first step, we allowed only the helium to move; we kept the fluid frozen and did both Quantum Wang-Landau and PIMC with threading sampling runs. We used the same analysis as in section 4.4.2 to find average potential energy at many different temperatures from the Wang-Landau data. This potential has no analytical solution so we ran our PIMC algorithm for a range of different temperatures and recorded the average potential energies. We also ran a classical Metropolis sampling to determine whether it was necessary to treat He as a quantum particle. The comparison between classical Metropolis, PIMC, and Quantum Wang-Landau is shown in Figure 4.17.

This plot shows agreement between the Wang-Landau and PIMC algorithms, although we see that the Wang-Landau algorithm is plagued by the same problems at high values of β as we have been seeing throughout this thesis. Once again, our potential energy setting was too high to capture the full range at these lower temperatures. To lower it, however, would require either a substantially longer running time or less accurate answers at lower β. Note that we can, in the same running time, obtain accurate answers at high β, we would just need to sacrifice accuracy at the lower values.

The agreement between classical Metropolis and PIMC is also good at low values for β and falls away at higher values. Here, however, we cannot fix this disagreement by changing the parameters of our simulation; we are finding that at low temperatures we can no longer approximate helium as a quantum

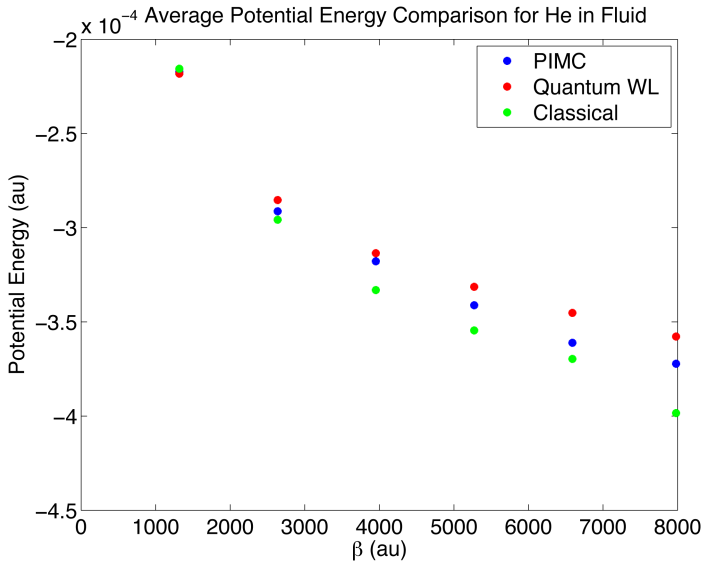


Figure 4.17: Average potential energy for Wang-Landau, PIMC, and classical Metropolis sampling for a light particle in a frozen fluid.

particle. We can see this more quantitatively by explicitly calculating de Broglie wavelengths for the particles. In order that we model the particles as classical particles, the mean separation between the particles needs to be much greater than their wavelength. For our system, an approximation of the mean separation $\bar{R}$ between particles is just

$$\bar{R} = \left(\frac{V}{N}\right)^{1/3} = \left(\frac{1}{1.127 \cdot 10^{-3}}\right)^{1/3} = 9.609 \text{ au}. \tag{107}$$

The relevant de Broglie wavelength is actually the sum of the de Broglie wavelengths of the quantum and fluid particles

$$\bar{\lambda} = \lambda_{Ar} + \lambda_{He} = \frac{h}{\bar{p}_{Ar}} + \frac{h}{\bar{p}_{He}} = \frac{h}{\sqrt{3m_{Ar}kT}} + \frac{h}{\sqrt{3m_{He}kT}} \tag{108}$$

where we have just used $\frac{1}{2m}\bar{p}^2 = \frac{3}{2}kT$ for the last step. Plugging in $\beta = 500$ au^{-1} into this, we find $\bar{\lambda} = 1.25$ au, and the condition $\bar{R} >> \bar{\lambda}$ is satisfied. If, however, we move out to $\beta = 8000$ au^{-1}, $\bar{\lambda} = 5.011$ au. This is only a factor of two smaller than $\bar{R}$ and, therefore, at this value for β, we are no longer in a region in which classical approximations apply. This is reflected in Figure 4.17 where we see that the classical and PIMC results diverge at high β.

Having established agreement between PIMC and Quantum Wang-Landau, we can also do benchmarking tests to compare the computational efficiency of the two quantum algorithms. In other words, it is hoped that Wang-Landau allows us to simulate the light particle in potentials requiring infeasible equilibration times by conventional Metropolis algorithms, which makes computation time of interest here. Wang-Landau and PIMC are roughly equivalent in terms of the amount of time per MC step. Wang-Landau takes on the order of 60 seconds for 100000 steps and PIMC takes 40 seconds. Thus here we compare MC steps.

We ran a Wang-Landau simulation, recording the density of states each time we reduced f and calculated average potential energy for each iteration of the density of states. With Wang-Landau sampling, there is no way to get a running average for potential energy. Rather we only get successively better and better estimates for $g(E)$ at discrete points when the energy histogram is flat. Our calculations for potential energy for $\beta = 5272$ au are shown as a function of MC steps in Figure 4.18.

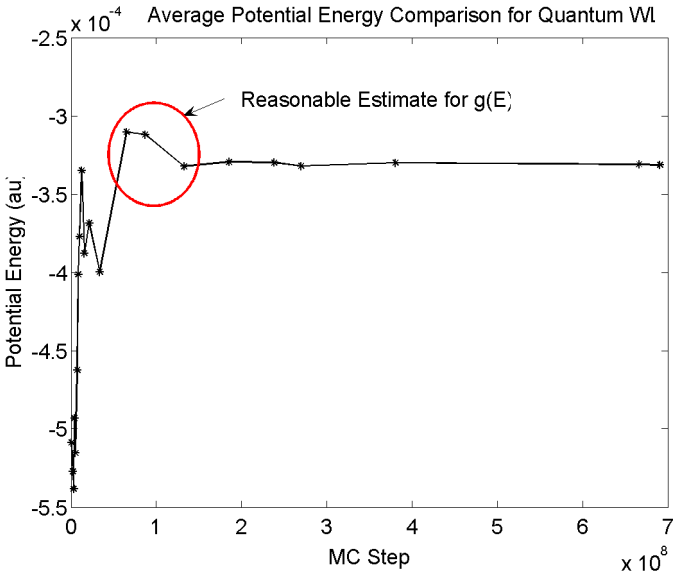


Figure 4.18: Average potential energy as a function of MC steps for a Wang-Landau simulation. Stars are the data points showing the average using successively better estimates for $g(E)$. The system appears to reach a steady average around 60 million MC steps.

To estimate how long this run takes, we use the following procedure. We consider the expected value of potential energy to be the value at the last data point in Figure 4.18. We then find the standard deviation of all the points in Figure 4.18 from this expected value and plot the number of standard deviations each point is from the expected value in Figure 4.19. The run is considered long enough when we are within one standard deviation of the expected value. This occurs at approximately 60 million MC steps.

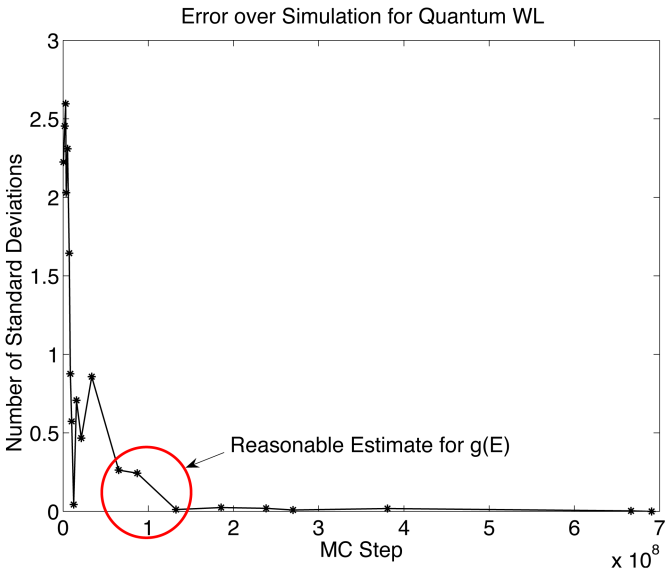


Figure 4.19: Simulation error as a function of MC steps for a Wang-Landau simulation. Stars are the data points showing the error at successively better estimates for $g(E)$. The system appears to reach a steady average around 60 million MC steps.

For PIMC sampling, we are able to calculate the energy every MC step, but just looking at that graph

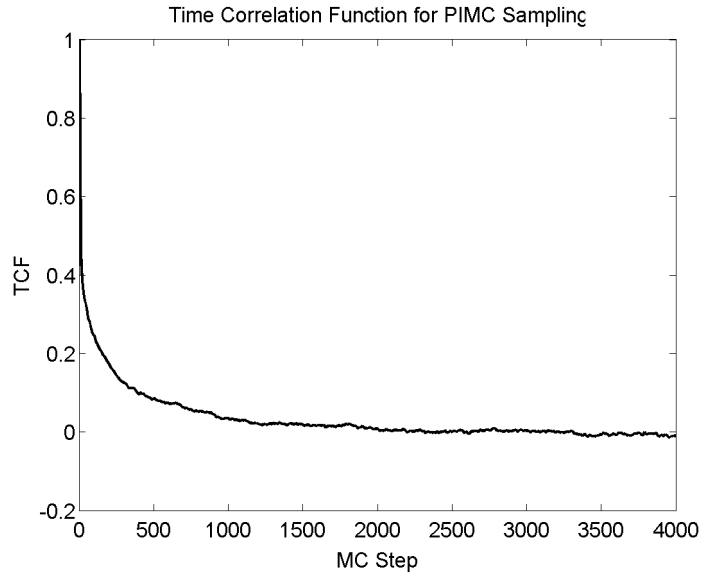


Figure 4.20: Time correlation function for a PIMC run.

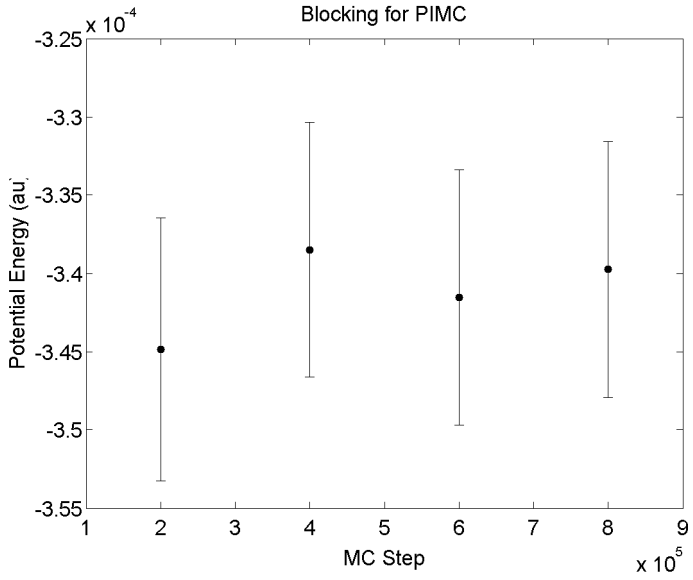


Figure 4.21: Average potential energy as a function of MC steps for a PIMC blocks of 200000 data points.

is not very enlightening. Thus we use the technique described in section 4.2.1 to estimate how long we need to run the simulation before it has fully explored the space. The time correlation function for a PIMC run is shown in Figure 4.20. From it, we found that steps at least 900 steps apart are uncorrelated. Then, using blocks of size 200000 for a 1000000 step run, we obtain the graph shown in Figure 4.21. Here the standard deviation of the means is $2.8 \cdot 10^{-6}$ and the average error per block is $8.2 \cdot 10^{-6}$. Blocks any smaller have a standard deviation of the means that is greater than the average error per block and we conclude that for this sampling technique, the simulation requires approximately 200000 MC steps to successfully explore the whole space. We used the function *blocking.m* in Appendix B.4 to do the calculations and generate figure 4.21.

Thus, for each individual temperature, Wang-Landau sampling is about two orders of magnitude slower

(by MC steps) than Metropolis sampling. However, Wang-Landau sampling is not limited to a single temperature; we could generate data for almost any temperature whose accessible range is consistent with the histogram we generate. In addition, we are using a density of $1.127 \cdot 10^{-3}$ au and only five beads for this test, making our space relatively easy to explore. Were we to increase the complexity of the simulation by increasing the density or the number of beads, we would see a large decrease in the efficiency of Metropolis sampling. Wang-Landau sampling in principle is advantageous by a system with energy barriers. The hope is that it will still perform well under conditions where conventional sampling is unacceptably slow.

5 Conclusion

We have discussed a number of MC techniques for sampling both classical and quantum particles. Beginning with basic CMC sampling for classical particles and PIMC sampling using threading for quantum particles, we have extended our simulation codes to allow Multicanonical and Wang-Landau sampling for classical particles in either a canonical or grand canonical ensemble and Quantum Wang-Landau for light particles.

We first showed that our basic CMC simulation was producing results that agreed with literature values, and, using the Widom Test Particle Insertion Method, we demonstrated that our GCMC simulations gave results consistent with this CMC code. We then used the GCMC simulation to locate a point on the argon liquid/gas coexistence curve.

In an effort to sample efficiently in a larger variety of environments, we also introduced Multicanonical sampling into our code. This allowed us to use information about the system to "push" the simulation into regions that were energetically unfavorable. However, problems in acquiring the necessary information about the system led us to try other sampling techniques.

Our last modification of the classical codes, was the Wang-Landau sampling method which allowed us to run one simulation to find average quantities at almost any temperature. We used this method to generate data about the potential energy of a fluid that agreed with literature results from MD and other MC simulations.

With these sampling algorithms working for classical particles, we began to explore algorithms for sampling quantum particles. The basic algorithms discussed in this thesis were PIMC, PIMC with threading, and Quantum Wang-Landau. Of these, threading was implemented by another researcher so our main interest was in the Quantum Wang-Landau results.

We began our research into Quantum Wang-Landau using a light particle in a simple harmonic potential. When we found that our results closely matched the analytical solution for finite bead numbers, we went on to try a more complex and interesting potential. We put the light particle into a fluid and used a Lennard-Jones potential to describe interactions between the particle and the fluid. Comparisons between Quantum Wang-Landau and PIMC with threading for this potential were favorable.

We lastly presented a benchmarking technique for Quantum Wang-Landau and PIMC with threading. We found that for a system of 5 beads in an argon fluid with a density of $1.127 \cdot 10^{-3}$ au, Quantum Wang-Landau is on the order of 100 to 500 times slower than PIMC with threading. However, these tests were done in a system that was fairly easy to equilibrate. In future work we hope to choose a system that is more time consuming for traditional PIMC algorithms and see if the Quantum Wang-Landau benchmark is more favorable in comparison.

Future work might also include allowing the fluid to move in simulation and trying to model the light particle as positronium (rather than helium). This work could also be extended to the two-chain approach pioneered by this lab in earlier work [17].

Acknowledgments

I would like to thank my thesis advisor Amy Bug for her help and support throughout the research and writing of this thesis and my second reader John Boccio for his input and second opinion. In addition, thanks to George Hang, Dan Peterson, Joe Grimm, and Michelle Tomasik for many helpful and enlightening conversations during our Wednesday lunch meetings. Funding for the summer research for this thesis was provided by Swarthmore College.

References

[1] M.P. Allen and D.J. Tildesley. *Computer Simulation of Liquids*. Clarendon Press, Oxford, 1987.

[2] Petr Beckmann. *A History of Pi*. Golem Press, Boulder, CO, 1970.

[3] C. H. Bennet. Efficient estimation of free energy differences from monte carlo data. *J. Comp. Phys.*, 22:245–268, 1976.

[4] Bernd A. Berg and Thomas Neuhaus. Multicanonical ensemble: A new approach to simulate first-order phase transitions. *Phys. Rev. Lett.*, 68(1):9–12, January 1992.

[5] John R. Boccio. *Quantum Mechanics: Mathematical Structure, Physical Structure and Applications in the Physical World*. In Preparation, 2007.

[6] A. L. R. Bug, T. W. Cronin, P. A. Sterne, and Z. S. Wolfson. Simulation of positronium: Toward more realistic models of void spaces in materials. *Radiat. Phys. Chem.*, Preprint, 2006.

[7] A. L. R. Bug, M. Muluneh, J. Waldman, and P. A. Sterne. Positronium in solids: Computer simulation of pick-off and self annihilation. *Mater. Sci. Forum*, 445-446:375–79, 2004.

[8] A. L. R. Bug and P. A. Sterne. Pimc simulation of ps annihilation: From micro to mesopores. *Phys. Rev. B*, 173:094106, 2006.

[9] D. F. Coker, B. J. Berne, and D. Thirumalai. Path integral monte carlo studies of the behavior of excess electrons in simple fluids. *J. Chem. Phys.*, 86(10):5689–5701, 1987.

[10] Timothy Cronin. Path integral monte carlo simulations of positronium in cylindrical pore spaces. Undergraduate thesis, Swarthmore College, Swarthmore, PA, 2006.

[11] Alan M. Ferrenberg and Robert H. Swendsen. New monte carlo technique for studying phase transitions. *Phys. Rev. Lett.*, 61(23):2635–2638, December 1988.

[12] Alan M. Ferrenberg and Robert H. Swendsen. Optimized monte carlo data analysis. *Phys. Rev. Lett.*, 63(21):1195–1198, September 1989.

[13] Daan Frenkel and Berend Smit. *Understanding Molecular Simulation: From Algorithms to Applications*. Academic Press, New York, 2nd edition, 2002.

[14] Harvey Gould, Jan Tobochnik, and Wolfgang Christian. *An Introduction to Computer Simulation Methods: Applications to Physical Systems*. Addison Wesley, New York, 3rd edition, 2007.

[15] J. Karl Johnson, John A. Zollweg, and Keith E. Gubbins. The lennard-jones equation of state revisited. *Mol. Phys.*, 78(3):591–618, 1993.

[16] Malvin H. Kalos and Paula A. Whitlock. *Monte Carlo Methods Volume I: Basics*. John Whiley and Sons, New York, 1986.

[17] L. Larrimore, R. N. McFarland, P. A. Sterne, and A. L. R. Bug. A two-chain path integral model of positronium. *J. Chem. Phys.*, 113:10642, 2000.

[18] Lisa Larrimore. Path integral monte carlo simulations of positronium in argon. Undergraduate thesis, Swarthmore College, Swarthmore, PA, 2002.

[19] Jooyoung Lee. New monte carlo algorithm: Entropic sampling. *Phys. Rev. Lett.*, 71(2), 1993.

[20] Donald A. McQuarrie. *Statistical Thermodynamics*. University Science Books, Mills Valley, CA, 1973.

[21] Konstantin V. Mikhin, Sergey V. Stepanov, and Vsevolod M. Byakov. Formation of the ps bubble in liquid media. *Radiat. Phys. Chem.*, 68:413–417, 2003.

[22] Bruce N. Miller and Terrence Reese. Path integral simulation of positronium. *Nucl. Instrum. Meth. B*, 192:176–179, 2002.

[23] Bruce N. Miller, Terrence L. Reese, , and Gregory Worrell. Positron lifetime distributions in fluids. *Phys. Rev. E*, 47(6):4083–4087, 1993.

[24] E. L. Pollock and D. M. Ceperley. Simulation of quantum many-body systems by path-integral methods. *Phys. Rev. B*, 30(5), 1984.

[25] E. L. Pollock and Karl J. Runge. Lectures on path-integral computations. In *Proceedings of the Workshop on Path Integral Methods in Theoretical Physics*, 1991.

[26] T. Reese and B. N. Miller. Positronium in xenon: The path-integral approach. *Phys, Rev. E*, 487(4), April 1993.

[27] M. S. Shell and A. Z. Panagiotpoulos P. G. Debenedetti. Generalization of the wang-landau method for off-latice simulations. *Phys. Rev. E*, 66, 2002.

[28] B. L. Shivachev, T. Troev, and T. Yoshiie. Positron lifetime computations of defects in nickel containing hydrogen or helium. *J. Nucl. Mater.*, 306:105–111, 2006.

[29] Loup Verlet. Computer experiments on classical fluids. i. thermodynamical properties of lennard-jones molecules. *J. Phys. Rev.*, 159(1):98–103, 1967.

[30] P. N. Vorontsov-Velyaminov and A. P. Lyubartsev. Entropic sampling in the path integral monte carlo method. *J. Phys. A: Math. Gen.*, 36:685–693, 2003.

[31] Fugao Wang and D. P. Landau. Determining the density of states for classical statistical models: A random walk algorithm to produce a flat histogram. *Phys. Rev. E*, 64, October 2001.

[32] Fugao Wang and D. P. Landau. Efficient, multiple-range random walk algorithm to calculate the density of states. *Phys. Rev. Lett.*, 86(10), March 2001.

[33] B. Widom. Some topics in the theory of fluids. *J. Chem. Phys.*, 59(11), December 1963.

[34] N. B. Wilding. Critical-point and coexistence-curve properties of the lennard-jones fluid: A finite-size scaling study. *Phys. Rev. E*, 52(1), July 1995.

[35] N. B. Wilding. Computer simulation of fluid phase transitions. *Am. J. Phys.*, 69(11), November 2001.

[36] Zachary Wolfson. Path integral monte carlo simulation of positronium within a dielectric spherical cavity. Undergraduate thesis, Swarthmore College, Swarthmore, PA, 2006.

Glossary

Activity	In thermodynamics, the quantity $\exp(\beta\mu)/\Lambda^3$ where μ is the chemical potential, $\beta = 1/(kT)$ and Λ is the thermal de Broglie wavelength. It gives information about the partial pressure of a gas.	13, 30
Bennet Method	A method used to find the chemical potential of a simulation that is computationally more intensive than the Widom Method, but works better at higher density.	25, 31
Canonical Ensemble	A thermodynamic ensemble in which we fix volume, temperature and number of particles.	9, 17, 27–29
Chemical Potential	In thermodynamics, the derivative of free energy with respect to particle number. Equal chemical potentials are a stability criterion for phase coexistence.	10, 24–26
Complete	A set of states (or vectors) $\{\psi\}$ is complete if every other state in the system can be written as a linear combination of the $\psi_\mathbf{i}$.	13
Entropic Sampling	A cross between Wang-Landau and Multicanonical sampling that constructs the density of energy states during simulation.	21
Ergodic	See Irreducible	7
Grand Canonical Ensemble	A thermodynamics ensemble in which we fix volume, temperature and chemical potential.	10–13, 18, 29–31
Importance Sampling	A method of sampling where focus on areas we consider more "important".	7, 8
Irreducible	A system is "irreducible" (or "ergodic") if every state can eventually be reached from every other state.	7
Light/Quantum Particle	A particle whose dynamics can only be accurately modeled using quantum mechanics.	5

Markov Chain	A sequence of trials that fulfills two conditions: (1) the outcome of each trial belongs to a finite set of possible outcomes (the "state space), and (2) the outcome of each trial depends *only* on the outcome of the trial immediately preceding it.	7
Metropolis Sampling	An accept/reject method of importance sampling.	7, 17, 18
Monte Carlo (MC) Algorithms	Nondeterministic algorithms used for simulating various phenomena.	6–8
Multicanonical Sampling	An modification of Metropolis sampling where we sample from a modified distribution to help the system explore energetically unfavorable states.	18, 31, 32
Overlapping Distribution Method	See Bennet Method	25
Path Integral Monte Carlo (PIMC)	An MC algorithm for simulating quantum particles by visualizing them as a polymer of beads connected by "springs".	21–23, 35, 37, 39, 41
Positive Operator	All eigenvalues of the operator are positive.	13
Self-Adjoint Operator	The operator and the hermitian conjugate of the operator are equal.	13
Threading	A modification of PIMC that increases the speed of the algorithm.	22, 37, 39, 41
Wang-Landau Sampling	An alternative method to Metropolis sampling where we construct the density of energy states during simulation.	20, 21, 23, 33–35, 37, 39,
Widom Method	A method used to find the chemical potential of a simulation.	24, 29

A Units

We use two sets of units predominantly in this thesis: reduced units and atomic units. For the fluid simulations with no light particle, we use reduced units, as this allows the quantities in which we are interested (namely, $\beta = 1/(kT)$, energy, density, and position) to all be of order unity. Reduced units for a Lennard-Jones fluid have distance in terms of the hard sphere diameter σ and energy in terms of the well depth ϵ. This gives us densities that vary between 0 and 1. In addition, we consider "reduced temperature" to actually be the quantity $T^* = k_B T$, where k_B is Boltzmann's constant. For argon, the conversions from standard units to reduced units are

$$\rho^* = \frac{\rho}{\sigma^3} = \frac{\rho}{(3.405 \cdot 10^{-10}\ \mathrm{m})^3} = \left(2.53 \cdot 10^{28}\ \mathrm{m}^{-3}\right)\rho$$

$$\beta = \frac{\epsilon}{k_B T} = \frac{119.8\ \mathrm{K}}{T}$$

$$U^* = \frac{U}{\epsilon} = (6.05 \cdot 10^{20}\ \mathrm{J}^{-1})U$$

Notice that quantities, with the exception of β, expressed in reduced units are usually denoted with a * after the variable name.

When we introduce the light particle, reduced units no longer make sense and we use atomic units instead. In atomic units, we write mass in terms of the mass of an electron, angular momentum in terms of $\hbar$, and charge in terms of the charge on the electron. The conversions from standard units are

$$m_{au} = \frac{m}{m_e} = (1.10 \cdot 10^{30}\ \text{au (mass)}\cdot\mathrm{kg}^{-1})m$$

$$l_{au} = \frac{l}{\hbar} = (9.48 \cdot 10^{33}\ \text{au (angular momentum)}\cdot\mathrm{J}^{-1}\cdot\mathrm{s}^{-1})l$$

$$q_{au} = \frac{q}{q_e} = (6.24 \cdot 10^{18}\ \text{au (charge)}\cdot\mathrm{C}^{-1})q$$

We also need the conversions between atomic units and reduced units for density, energy, and temperature. For argon, these conversions are

$$\rho^* = \sigma_{au}^3 \rho_{au} = 6.433^3 \rho_{au} = 266.23\rho_{au}$$

$$\beta^* = \epsilon_{au}\beta_{au} = 3.79 \cdot 10^{-4}\beta_{au}$$

$$U^* = \frac{U_{au}}{\epsilon_{au}} = 2635.87 U_{au}$$

B Source Code[1]

B.1 Fluid Codes

PROGRAM FLMAIN

```
c Adapted from Ar_cylinder.f90 by Jenny Barry, Summer 2006

c~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~c
c Modules (in alphabetical order):
c       FLFluidFncs.f95 (~350 lines) - functions and subroutines dealing with a fluid
c                                   interacting via the Lennard-Jones potential
c       FLGenFuncs (~200 lines) - functions less specific to this code but still
c                                   involving physics (calculating average
c                                   energy for example)
c       FLInfo.f95 (~150 lines) - all global variables and constants
c       FLMain.f95 (~100 lines minus comments) - main method
c       FLInitEnd (~400 lines) - functions and subroutines required to initialize
c                                   and/or end the program
c       FLUtil.f95 (~100 lines) - functions and subroutines of general use (doubling
c                                   the size of an array for example)
c       Mac_types.f90 (~10 lines) - Types definitions

cINPUT:
c       COMMAND FILE:
c       The prompt requires a loop file that contains a list of data file names.
c       This allows the program to run many data runs at once
c       Example file: ./example/flloop.in
c
c       FLUID DATA FILE:
c       This file has information about the fluid and the options for simulation
c       Examples:     ./example/basic/flbasic.in
c                     ./example/cyl/flcyl.in
c                     ./example/grand/flgrand.in
c                     ./example/multi/flmulti.in
c
c       INIT DATA FILE
c       Line 1: starting step size
c       Rest of file: positions of atoms (note that it is important to get the number of atoms right
c               in the main input file as well)
c       Example: ./example/init.in
c
c       MULTI DATA FILE (when running a MULTICANONICAL simulation):
c       Line 1: number of bins in the eta weighting function
c       Line 2: the file name with the data for the eta weighting function
c       Example: ./example/multi/multi.in
c
c       ETA DATA FILE (when running a MULTICANONICAL simulation):
c       Contains the eta function when using multicanonical weighting
c       Example: ./example/multi/eta.in
c
cOUTPUT (9 files):
c       flposn: contains the x, y, z coordinates of each fluid atom at every WRITEDATA timesteps
c               the first line for each time step is the number of particles, the current stepsize,
c               and a throwaway number (to make the file easy to read into a data analysis program)
c       density: contains density, number of atoms at every WRITEDATA timesteps
c       energy: contains average energy at every WRITEDATA timesteps
c       restart: contains fluid positions and other data necessary to restart the simulation from the
c               point it ended. formatted correctly so that it can be used as an input file to start
c               a new simulation from the last position of this simulation
c       initfl: the initial position of the fluid
c       widom: contains number of particles, dE, exp(-beta*dE) where dE is from widom particle insertion
c       denstate: contains the density of states after EACH flat histogram (Wang-Landau ONLY)
c       histe: contains the energy histogram after EACH flat histogram (Wang-Landau ONLY)
c       pressure: contains pressure data
```

[1] We used *lgrind* to convert from source code to latex.

```
c
c     TO CHANGE THESE FILES OR ADD ANOTHER, LOOK AT THE SYNTAX IN FLInfo.f95. To add a file,
c     increase NUMOUT, add another file descriptor to OUTFILES and add another name to OUTNAMES.

      use types
      use FLInfo
      use FLInitEnd
      use FLFluidFncs
      use FLGenFncs
      use FLUtil                                                                    70


c     Main Program

      implicit NONE

      integer :: i, irun,j, cnt, WRITEDATA = 1
      real (dp) :: V, avE, flatness, wid !! real (dp) is defined in MacTypes
      character(fileNameLength) :: fname
                                                                                    80
      call init_genrand() !! Inits the Mersenne twister

c     Getting the command input file
      write(*,*) "Enter the name of the input file:"
      read(*,*) fname
      open(unit=CMDFILE, file=fname, status='old',action='read')
      read(CMDFILE,*); read(CMDFILE,*) loop
      read(CMDFILE,*)

      runs: do j=1,loop                                                             90
c     loop over data runs
c           read in data file
            read(CMDFILE,*) fname
            open(unit=FLFILE, file=fname,status='old',action='read')

c           initialize program (in FLInitEnd)
            call FLInit()

c           calculate the volume
            if (isCyl) then                                                         100
                  V = pi*rcav2*hcav
            else
                  V = rcav**3
            endif

c           initial values
            numWid = 0
            irun = 0
            fmsize = fmSTART
                                                                                    110
c           main loop
            MC_passes: do  !! if W-L we need a while loop so this cannot be a for loop

                  if (irun.GT.npass-1.AND.(.NOT.isWL)) then
                        exit
                  else if (isWL.AND.irun > 0.AND.minval(histE)/(sum(histE)/(1.0*size(histE))) > FLAT) then
c                 flat histogram!
                        fwl = sqrt(fwl) !! decrease f
                        print *, "flat histogram!, f = ",fwl
                        irun = 0                                                    120
                        write(HISTENG, *) size(histE)
                        write(GFILE, *) size(histE)
                        do i=1,size(histE)
                              write(HISTENG,*) histE(i)
                              write(GFILE,*) g(i)
                        enddo
                        call resetHist() !! in FLInitEnd
                        if (fwl < MINF) exit
```

```
                endif
                                                                                    130
c               do a move!
                call flMCStep() !! in FLFluidFncs

                if (MOD(irun,WRITEDATA) == 0) then
c               write data
                    avE = averageEnergy()
                    wid = widomTest()
                    write(WIDOMDATA,*) irun, wid, exp(-beta*wid)
                    write(AVGENG,*) avE
                    write(HISTRHO,*) (nf*sig0**3)/V, nf                             140
                    write(PLOTPOS,*) nf, fmsize, -1
                    do i=1,nf
                        write(PLOTPOS,*) xf(i,1), xf(i,2), xf(i,3)
                    enddo
                    write(PRESSURE,*) rhof, ((rhof/sig0**3)/beta+virial()/V)/eps0*sig0**3
                endif

                if (isWL.AND.mod(irun, 100)==0) then
c               print out some info (what you might want to know depends on what is running)
                    if (irun > 0) then                                              150
                        flatness = minval(histE)/(sum(histE)/(1.0*size(histE)))
                        print *, "flatness: ", flatness
                    else
                        flatness = 0
                    endif
                    print *, "te = ",totalEnergy(), "nf = ",nf, "f = ", fwl
                    if (flatness < 1e-8) then
                        cnt = 0
                        do i=1,size(histE)
                            if (histE(i) < 1) then                                  160
                                cnt = cnt+1
                            endif
                        enddo
                        print *, "There are ", cnt, " zero elements."
                    endif
                else if ((.NOT.isWL).AND.irun*100/npass > (irun-1)*100/npass) then
                    write(*,*) (irun*100/npass), "percent done"
                endif

                irun = irun + 1                                                     170
            end do MC_passes

            call cleanUp() !! in FLInitEnd
        enddo runs

        call endPro() !! in FLInitEnd

        print *, "program finished, see data files for results"

END PROGRAM FLMAIN                                                                  180
```

```
c FLInitEnd
c Contains routines used to initialize or end/clean up after FLMain
c Subroutines:
c       cleanUp, endPro, FlInit, initMulti, initWL, openFiles, packCube, packCyl,
c       startFromConfig, resetHist


MODULE FLInitEnd

use types                                                                    10
use FLInfo

CONTAINS

c~~~~~~~~~~~~~~~~~~~~~~
c cleanUp subroutine
c~~~~~~~~~~~~~~~~~~~~~~

c Cleans up after one run by deallocating variables and closing single-run files
c Deallocates:                                                               20
c       xf, xfn, eta, g, histE
c Closes:
c       INFILES, OUTFILES
c Called by:
c       Main

subroutine cleanUp()
        implicit none

        integer :: i                                                         30

c       write restart file
        write(RESTART,*) fmsize
        do i=1,nf
                write(RESTART,*) xf(i,1), xf(i,2), xf(i,3)
        enddo
        write(RESTART,*) !! a blank line is needed

c       close input files
        do i=1,size(INFILES)                                                 40
                close(INFILES(i))
        enddo

c       close output files
        do i=1,size(OUTFILES)
                close(OUTFILES(i))
        enddo

c       deallocate (or we can't reallocate for another run
        deallocate(xf, xfn)                                                  50
        if (isMulti) then
                deallocate(eta)
        endif
        if (isWL) then
                deallocate(g, histE)
        endif
end subroutine cleanUp


c~~~~~~~~~~~~~~~~~~~~~~                                                      60
c endPro subroutine
c~~~~~~~~~~~~~~~~~~~~~~

c Cleans up after entire program by closing multiple-run files
c Closes:
c       GBLINFILES
c Called by:
```

```
c       Main
c Notes:
c       does NOT call cleanUp!                                                  70

subroutine endPro()
        implicit none

        integer :: i

c       close global in files - probably just command loop
        do i=1,size(GBLINFILES)
                close(GBLINFILES(i))
        enddo                                                                   80
end subroutine endPro


c~~~~~~~~~~~~~~~~~~~~
c FLINIT SUBROUTINE
c~~~~~~~~~~~~~~~~~~~~

c Initializes the fluid atoms, based on data from FLFILE
c Initializes:
c       npass, rhof, fmass, beta, z, isGrand, isCyl, isMulti, isWL, nf, rcav, rcav2   90
c Also allocates space for xf and xfn (and eta, g, histe if appropriate) and places the fluid atoms
c Calls:
c       initMulti, initWL, openFiles, packCyl, packCube, startFromConfig, resetHist
c Called by:
c       Main

subroutine FlInit()
        implicit none

        integer :: bool, i                                                      100
        logical :: initCond

        call openFiles() !! open output files

c        basic data
        read(FLFILE,*);read(FLFILE,*) npass
        read(FLFILE,*);read(FLFILE,*) rhof
        read(FLFILE,*);read(FLFILE,*) fmass
        read(FLFILE,*);read(FLFILE,*) beta
        read(FLFILE,*);read(FLFILE,*) z                                         110
        read(FLFILE,*);read(FLFILE,*) bool !!isGrand

c       isGrand
        if (bool.EQ.0) then
                isGrand = .false.
        else
                isGrand = .true.
        endif

c       isCyl                                                                   120
        read(FLFILE,*);read(FLFILE,*) bool
        if (bool.EQ.0) then
                isCyl = .false.
        else
                isCyl = .true.
        endif

c       isMulti
        read(FLFILE,*); read(FLFILE,*) bool
        if (bool == 1) then                                                     130
                isMulti = .TRUE.
                call initMulti()
        else
                isMulti = .FALSE.
        endif
```

```
c       isWL
        read(FLFILE,*); read(FLFILE,*) bool
        if (bool == 1) then
                isWL = .TRUE.                                                    140
                call initWL()
        else
                isWL = .FALSE.
        endif

c       dimensions and number of fluid atoms
        if (isGrand) then
                read(FLFILE,*);read(FLFILE,*) rcav !! We specify rcav for gc
                if (isCyl) then
                        nf = int(rhof*pi*hcav*rcav**2)                           150
                        read(FLFILE,*) hcav
                else
                        nf = int(rhof*rcav**3.0)
                endif
        else
                read(FLFILE,*);read(FLFILE,*) nf !! We specify nf for can
                if (isCyl) then
                        rcav = (nf/(rhof*pi*hcav))**(1.0/2.0)
                        read(FLFILE,*) hcav
                else                                                             160
                        rcav = (nf/rhof)**(1.0/3.0)
                endif
        endif
        rcav2 = rcav*rcav

c       allocate stuff
        if (isGrand) then
                TOTAL = nf*2 !! We do not know exactly how much to allocate so start with 2X
        else
                TOTAL = nf+1 !! we allocate one extra in case we want to do widom test   170
        endif
        allocate(xf(TOTAL,3), stat = error)
        if (error .ne. 0) then
                write(*,*) "Unable to allocate memory for the array: xf(:,:)"
                STOP
        endif
        allocate(xfn(TOTAL,3), stat = error)
        if (error .ne. 0) then
                write(*,*) "Unable to allocate memory for the array: xfn(:,:)"
                STOP                                                             180
        endif

c       start from initial condition
        read(FLFILE,*);read(FLFILE,*) bool
        if (bool.NE.0) then
                initCond = .TRUE.
                call startFromConfig()
        else if (isCyl) then
                initCond = .FALSE.
                call packCyl()                                                   190
        else
                initCond = .FALSE.
                call packCube()
        endif

c       write initial positions to file
        do i=1,nf
                write(INITIAL,*) xf(i,1), xf(i,2), xf(i,3)
        enddo
        print *, "the requested density gives", nf, "fluid atoms", " and a cavity of size", rcav   200
end subroutine FlInit
```

```
c~~~~~~~~~~~~~~~~~~~~~~~~
c initMulti subroutine
c~~~~~~~~~~~~~~~~~~~~~~~~

c initializes program to use multicanonical sampling

subroutine initMulti()                                                  210
      implicit none

      integer :: i
      character(fileNameLength) :: fname

c     open MULTIFILE
      read(FLFILE,*) fname
      open(unit=MULTIFILE, file = fname, status='old', action='read')
      read(MULTIFILE, *); read(MULTIFILE,*) multibins
                                                                        220
c     allocate space for eta
      allocate(eta(multibins), stat = error)
      if (error .ne. 0) then
            write(*,*) "Unable to allocate memory for the array: eta(:)"
            STOP
      endif

c     open ETAFILE
      read(MULTIFILE,*); read(MULTIFILE,*) fname
      open(unit=ETAFILE, file=fname,status='old', action='read')        230
      do i=1,multibins
            read(ETAFILE,*) eta(i)
      enddo
end subroutine initMulti


c~~~~~~~~~~~~~~~~~~~~~
c initWL subroutine
c~~~~~~~~~~~~~~~~~~~~~
                                                                        240
c initializes program to use wang-landau sampling

subroutine initWL()
      implicit none
      integer :: i

      allocate(g(gbins), stat = error)
      if (error .ne. 0) then
            write(*,*) "Unable to allocate memory for the array: g(:)"
            STOP                                                        250
      endif

      allocate(histE(gbins), stat = error)
      if (error .ne. 0) then
            write(*,*) "Unable to allocate memory for the array: histE(:)"
            STOP
      endif

      call resetHist()
                                                                        260
      do i=1,gbins
            g(i) = 0.0      !! this is actually ln(g)
      enddo
      fwl = exp(1.0)
end subroutine initWL


c~~~~~~~~~~~~~~~~~~~~~~~~
c openFiles subroutine
c~~~~~~~~~~~~~~~~~~~~~~~~                                               270
```

```
c opens input files

subroutine openFiles()
      implicit none

      integer :: i
      character(fileNameLength) :: dirname

      read(FLFILE,*);read(FLFILE,*) dirname                                                    280
      do i=1,size(OUTFILES)
            open(unit=OUTFILES(i),file=(trim(dirname) // trim(OUTNAMES(i))),status='replace',action='write')
      enddo
end subroutine openFiles

c~~~~~~~~~~~~~~~~~~~~~~~
c packCube subroutine
c~~~~~~~~~~~~~~~~~~~~~~~

c Places fluid atoms uniformly in a cube (for when using periodic boundary conditions)   290
c Called by:
c       FLInit

subroutine packCube()

      integer :: i
      real (dp) :: spc, rat

      if (nf**(1.0/3.0)*sig0.GT.rcav) then
            write(*,*) "Too many atoms for this volume. Increase rcav or decrease rhof."   300
            STOP
      endif

      if (nf.LT.1) then
            write(*,*) "No atoms!"
            STOP
      endif

      spc = rcav/(nf**(1.0/3.0)) !! spacing between the atoms
      do i=1,nf                                                                            310
            rat = spc*i/rcav
            xf(i,1) = (rat - int(rat))*rcav
            xf(i,2) = (int(rat)*spc/rcav - int(int(rat)*spc/rcav))*rcav
            xf(i,3) = int(int(rat)*spc/rcav)*spc
      enddo

end subroutine packCube

c~~~~~~~~~~~~~~~~~~~~~~~
c packCyl subroutine                                                                   320
c~~~~~~~~~~~~~~~~~~~~~~~

c Places fluid atoms uniformly in a cylinder
c Called by:
c       FLInit

subroutine packCyl()

      real (dp) :: rpack, rring, zring
      integer :: nlayers, npack, nrings, l                                             330

c       now we need to place the fluid atoms in the cavity in a dense fashion
      rpack=2.0*rcav/sig0

      if (rpack.LT.1.0) then
            print *, "cannot pack any fluid atoms into a cylinder so small"
            STOP
      endif
```

```
      if (rpack.LT.2.0) then                                                    340
            print *, " we use simple axial packing"
            if (nf.GT.int(hcav/sig0)) then
                  print *, "we cannot fit all requested atoms by axial packing"
                  STOP
            endif
            nlayers=nf
            do k=1,nlayers
                  xf(k,1)=0.0
                  xf(k,2)=0.0
                  xf(k,3)=k*sig0                                                350
            enddo
      else
            npack=6
            nrings=0
            l=0
            RingIteration: do !! this blind loop iterates over multiple packing rings
                  if (l.GT.nf) exit !! will never exit initially, only after running out of fluid atoms to place
                  npack=int(pi/asin(1.0/(rpack-1.0)))
                  nrings=nrings+1
                  rring=sig0/(2.0*sin(pi/npack))                                360
                  LayerIteration: do k=1,int(hcav/sig0) !! loop over the layers of atoms
                        zring=k*sig0
                        do
                              l=l+1
                              if (l.GT.nf) exit
                              xf(l,1)=rring*cos(2.0*l*pi/npack)
                              xf(l,2)=rring*sin(2.0*l*pi/npack)
                              xf(l,3)=zring
                              if (mod(l,npack).EQ.0) exit
                        enddo                                                   370
                  enddo LayerIteration

                  rpack=1.0/sin(pi/npack)-1.0

                  if (rpack.LT.2.0.AND.ℓ.LT.nf) then
                        print *, " we use simple axial packing for the remaining atoms"
                        if ((nf-l).GT.int(hcav/sig0)) then
                              print *, "we cannot fit all remaining atoms by axial packing"
                              STOP
                        endif                                                   380
                        do k=l,nf
                              xf(k,1)=0.0
                              xf(k,2)=0.0
                              xf(k,3)=(k-l)*sig0
                        enddo
                        l=l+int(hcav/sig0)
                  endif

                  if (npack.LT.6.AND.ℓ.LT.nf) then !! abort if there are atoms let to place
c                 and we're trying to put them inside a less than 6-ring        390
                        print *, "too many atoms for too small a cavity"
                        STOP
                  endif
            enddo RingIteration
      endif
end subroutine packCyl


c~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
c startFromConfig subroutine                                                    400
c~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

c starts the simulation from an initial configuration stored in a file

subroutine startFromConfig()
      integer :: i
      character(fileNameLength) :: init
```

```
        read(FLFILE,*) init !! Name of initial configuration file
        open(unit=INITFILE, file=init, status='old', action='read')                    410
        read(INITFILE,*) fmsize
        do i=1,nf
                read(INITFILE,*) xf(i,1), xf(i,2), xf(i,3)
        enddo
end subroutine startFromConfig

c~~~~~~~~~~~~~~~~~~~~~~~~~
c resetHist subroutine
c~~~~~~~~~~~~~~~~~~~~~~~~~
                                                                                       420
c resets the energy histogram for wang-landau sampling
c Called by:
c       FLInit, Main

subroutine resetHist()
        implicit none
        integer :: i

        do i=1,gbins
                histE(i) = 0                                                           430
        enddo
end subroutine resetHist

end MODULE
```

```
c FLFluidFncs
c Contains functions specific to the fluid for use with FLMain.f95.
c Functions:
c       binE, updateWL, widomTest
c Subroutines
c       flMCStep

MODULE FLFluidFncs

use types
use FLInfo
use FLUtil
use FLGenFncs

CONTAINS

c~~~~~~~~~~~~~~~~~
c binE function
c~~~~~~~~~~~~~~~~~

c Input:
c       the energy to bin
c Returns:
c       the bin in the histogram corresponding to energy
c Called by:
c       updateWL

integer function binE(energy)
        implicit none

        real (dp), INTENT(IN) :: energy


        binE = int(energy/GRES + size(histE))

end function binE


c~~~~~~~~~~~~~~~~~~~~~~
c flMCStep subroutine
c~~~~~~~~~~~~~~~~~~~~~~

c Moves, creates, or destroys a fluid atom using the Metropolis method
c Modifies:
c       xf, xfn, fmac, fmacsum, fmacsize, nf, TOTAL
c Called by:
c       Main
c Calls:
c       calcLJ, increaseCapacity
c Notes:
c       I am not sure Wang-Landau and grand canonical work together; they have
c       not been thoroughly tested

subroutine flMCStep()
        implicit none

        integer :: i,move_index, bin1, bin2
c destroy, create, or move
        real (dp) :: dcm
        real (dp) :: fboltzf, cavfactor, flenergy, flenergyn, deltaE, rn, dstr, crt, V, nm, nmsum, wt, te

        if (isCyl) then
                V = pi*rcav2*hcav
        else
                V = rcav**3
        endif
        fmacsum = 0.0
```

```
      nmsum = 0.0

      te = totalEnergy()                                                    70

      move_index = 1
      MoveCreateDestroyLoop: do
c     This has to be a while loop because nf will change if using g.c.!
            if (move_index > nf) exit
            flenergy=0.0
            flenergyn=0.0
            deltaE = 0.0
            fmac = 0.0
            nm=0.0                                                          80
            if (isGrand) then
                  dcm = ran1(initrand)
            else
c           We always just move
                  dcm = 0
            endif
            rn = ran1(initrand) !! in FLUtil
            xfn=xf
c           Move, create destroy choices
            if (dcm < 1.0/3.0) then                                         90
c           Move this atom
                  xfn(move_index,1)=xf(move_index,1)+fmsize*(ran1(initrand)-0.5)
                  xfn(move_index,2)=xf(move_index,2)+fmsize*(ran1(initrand)-0.5)
                  xfn(move_index,3)=xf(move_index,3)+fmsize*(ran1(initrand)-0.5)

                  if (isCyl) then
c                 cylinder
c                       outer reaches of atoms electron cloud
                        cavfactor = sqrt(xfn(move_index,1)**2+xfn(move_index,2)**2)+0.5*sig0
                                                                            100
c                       periodic z-condition
                        if (xfn(move_index,3).GT.hcav) xfn(move_index,3)=xfn(move_index,3)-hcav
                        if (xfn(move_index,3).LT.0.0) xfn(move_index,3)=xfn(move_index,3)+hcav
                  else
c                 periodic boundaries
                        cavfactor = 0  !! we do not need to worry about going "outside" the cavity

c                       periodic conditions in all three directions
                        do i=1,3
                              do                                            110
                                    if ((xfn(move_index,i).LT.rcav).AND.(xfn(move_index,i).GT.0)) exit
                                    if ((xfn(move_index,i)).GT.rcav) xfn(move_index,i)=xfn(move_index,i)-rcav
                                    if ((xfn(move_index,i)).LT.0.0) xfn(move_index,i)=xfn(move_index,i)+rcav
                              end do
                        enddo
                  endif

c                 calculate old and new energies
                  do i=1,nf
                        if (i.EQ.move_index) cycle                          120
                        flenergy = flenergy+calcLJ(move_index,i,xf)
                        flenergyn = flenergyn+calcLJ(move_index,i,xfn)
                  enddo
                  deltaE = flenergyn - flenergy
                  if (isWL) then
c                 do Wang-Landau stuff
                        if (updateWL(te, (te+deltaE), cavfactor)) then
                              xf = xfn
                              fmac = 1.0
                              te = te+deltaE                                130
                        endif
                  else
c                 do Metropolis stuff
                        if (flenergyn.LT.flenergy.AND.cavfactor.LT.rcav) then
                              xf=xfn
```

```
                        fmac=1.0 !! move accepted
                    else if (cavfactor.LT.rcav) then
                        fboltzf=dexp(−beta*(flenergyn−flenergy))
                        if (rn.LT.fboltzf) then
                            xf=xfn                                          140
                            fmac=1.0 !! move accepted
                        endif
                    endif
                endif
                nm = 1.0    !! the number of times we moved a particle
            else if (dcm < 2.0/3.0) then
c           destroy the atom
                do i=1,nf
                    if (i.EQ.move_index) cycle
                    deltaE = deltaE − calcLJ(move_index,i,xf)               150
                enddo
                if (isWL) then
c               do WL stuff
                    if (updateWL(te, (te+deltaE), cavfactor).AND.nf.GT.1) then
c                   destruction accepted
                        xf(move_index, :) = xf(nf, :)
                        nf = nf−1
                        move_index = move_index−1 !! so we visit the last atom
                        te = te+deltaE
                    endif                                                   160
                else
c               do Metropolis stuff
                    if  (isMulti) then
c                   do multicanonical weighting
                        bin1 = int((1.0*multibins)*nf/V)
                        bin2 = int((1.0*multibins)*(nf−1)/V)
                        wt = exp(eta(bin1)−eta(bin2))
                    else
                        wt = 1.0 !! normal metropolis has no weight
                    endif                                                   170
                    dstr = (deltaE − 1.0/beta*log(nf/(z*V)))
                    if (wt*exp(−beta*dstr).GT.rn) then
c                   destruction accepted
                        xf(move_index, :) = xf(nf, :)
                        nf = nf−1
                        move_index = move_index−1 !! so we visit the last atom
                    endif
                endif
            else
c           create an atom                                                  180
                if (isCyl) then
c               Cylinder
                    xfn(nf+1,1) = (rcav−0.5*sig0)*(2*ran1(initrand) −1)
                    xfn(nf+1,2) = (rcav−0.5*sig0)*(2*ran1(initrand) −1)
                    xfn(nf+1,3) = hcav*ran1(initrand)
                else
c               Periodic
                    do i=1,3
                        xfn(nf+1,i) = rcav*ran1(initrand)
                    enddo                                                   190
                endif
                if ((nf+1).GE.TOTAL) call increaseCapacity() !! in FLUtil
                do i=1,nf
                    deltaE = deltaE + calcLJ(nf+1,i,xfn)
                enddo
                if (isWL) then
c               Wang-Landau stuff
                    if (updateWL(te, (te+deltaE), cavfactor)) then
                        xf = xfn
                        nf = nf+1                                           200
                        te = te+deltaE
                    endif
                else
```

```
c                       Metropolis stuff
                              if  (isMulti) then
c                             Multicanonical weighting
                                    bin1 = int((1.0*multibins)*nf/V)
                                    bin2 = int((1.0*multibins)*(nf+1)/V)
                                    wt = exp(eta(bin1)−eta(bin2))
                              else                                                    210
c                             No weighting for Metropolis
                                    wt = 1.0
                              endif
                              crt = (deltaE − 1.0/beta*log(z*V/(nf+1)))
                              if (wt*exp(−beta*crt).GT.rn) then
c                             creation accepted
                                    xf = xfn
                                    nf = nf+1
                              endif
                       endif                                                          220
                endif

c               Move rate stuff (for adjusting step size)
                fmacsum=fmacsum+fmac
                nmsum = nmsum+nm
                move_index=move_index+1
        enddo MoveCreateDestroyLoop

        fmac = fmacsum/nmsum !! Determine fraction of accepted moves (we tried nmsum moves)
                                                                                      230
c       adjust fmsize so the acceptance rate is roughly 30%
        if(fmac.GT.0.55) fmsize=fmsize*1.1
        if(fmac.LT.0.25) fmsize=fmsize/1.1
        do
                if (fmsize < rcav) exit
                fmsize = fmsize/2 !! do not let fmsize get too big
        enddo
        fmacsum = 0.0
end subroutine flMCStep
                                                                                      240

c~~~~~~~~~~~~~~~~~~~~
c updateWL Function
c~~~~~~~~~~~~~~~~~~~~

c updates the histogram and density of states for use with wang-landau sampling
c Input:
c       flenergy: energy before move
c       flenergyn: energy after move
c       cavfactor: factor to make sure we do not leave the cylinder if we are in one  250
c Returns:
c       true if move is accepted, false otherwise
c Called by:
c       flMCStep
c Calls:
c       binE

logical function updateWL(flenergy, flenergyn, cavfactor)
        implicit none
                                                                                      260
        real (dp), INTENT(IN) :: flenergy, flenergyn, cavfactor
        integer :: bin1, bin2

c       bins
        bin1 = binE(flenergy)
        bin2 = binE(flenergyn)

        if ((bin1.LE.size(histE).AND.bin1.GT.0).AND.(bin2.LE.size(histE).AND.bin2.GT.0)) then
c       we are not moving outside the histogram
                if (ran1(initrand).LT.exp(g(bin1) − g(bin2)).AND.cavfactor.LT.rcav) then  270
c               accept move
```

```
                updateWL = .TRUE.
                histE(bin2) = histE(bin2) + 1
                g(bin2) = log(fwl)+g(bin2)
            else
c           reject move
                updateWL = .FALSE.
                histE(bin1) = histE(bin1) + 1
                g(bin1) = log(fwl)+g(bin1)
            endif                                                                    280
        else if (bin1.GT.size(histE).OR.bin1.LE.0) then
c       we somehow wound up outside the histogram (if we start in a bad configuration for example)
            print *, 'An impossible configuration has been reached.'
            STOP
        else
c       the move would take us outside the histogram so we reject it
            updateWL = .FALSE.
            histE(bin1) = histE(bin1) + 1
            g(bin1) = log(fwl)+g(bin1)
        endif                                                                        290
end function updateWL


c^^^^^^^^^^^^^^^^^^^^^^
c widomTest Function
c^^^^^^^^^^^^^^^^^^^^^^

c performs a widom insertion and looks at the change in energy
c Returns:
c       the change in energy a random insertion of a particle would cause        300
c Called by:
c       Main
c Calls:
c       calcLJ

real (dp) function widomTest()
        real (dp) :: deltaE
        integer :: i

        deltaE = 0                                                               310

c       "insert" particle
        if (isCyl) then
c       cylinder
            xfn(nf+1,1) = (rcav-0.5*sig0)*(2*ran1(initrand) -1)
            xfn(nf+1,2) = (rcav-0.5*sig0)*(2*ran1(initrand) -1)
            xfn(nf+1,3) = hcav*ran1(initrand)
        else
c       periodic
            do i=1,3                                                             320
                xfn(nf+1,i) = rcav*ran1(initrand)
            enddo
        endif

c       calculate energy change
        do i=1,nf
            deltaE = deltaE + calcLJ(nf+1,i,xfn)
        enddo

        numWid = numWid+1 !! keep track of the number of tests we have done      330
        widomTest = deltaE
end function widomTest

end MODULE FLFluidFncs
```

```
c FLGenFncs
c Contains general functions (or functions that could be generalized)
c Functions:
c       averageEnergy, calcF, calcLJ, flrrel, flrrel2, totalEnergy, virial

MODULE FLGenFncs

use types
use FLInfo
use FLUtil

CONTAINS

c~~~~~~~~~~~~~~~~~~~~~~~~~
c averageEnergy Function
c~~~~~~~~~~~~~~~~~~~~~~~~~

c calculates the average energy
c Returns:
c       average energy
c Called by:
c       Main
c Calls:
c       totalEnergy

real (dp) function averageEnergy()
        implicit none

        averageEnergy = totalEnergy()/nf
end function averageEnergy


c~~~~~~~~~~~~~~~~~
c calcF Function
c~~~~~~~~~~~~~~~~~

c calculates force assuming a Lennard-Jones potential
c Parameters:
c       i, j: indices in mols of the two atoms between which to calculate the force
c       mols: an nx3 array of the positions of atoms interacting via Lennard-Jones potential
c Returns:
c       The force between mols(i) and mols(j) assuming a Lennard-Jones potential
c Called by:
c       virial

real (dp) function calcF(i, j, mols)
        implicit none

        integer, INTENT(IN) :: i, j
        real (dp), DIMENSION(:,:), INTENT(IN) :: mols
        real (dp) :: dist

        dist = flrrel(i,j,mols)  !! we need the actual distance

c       sets cutoff distance for LJ interaction to rcut
        if ((isCyl.AND.dist.LT.min(0.5*hcav,rcut)).OR.((.NOT.isCyl).AND.dist.LT.(rcut))) then
                calcF = 4.0*eps0*(12*(sig0**12/dist**13)-6*(sig0**6/dist**7))
        else
                calcF = 0.0
        endif

end function calcF


c~~~~~~~~~~~~~~~~~~
c calcLJ Function
```

```
c~~~~~~~~~~~~~~~~~~
c calculates Lennard-Jones potential between two molecules                    70
c Parameters:
c       i, j: indices in mols of the two atoms between which to calculate the Lennard-Jones potential
c       mols: an nx3 array of the positions of atoms interacting via Lennard-Jones
c Returns:
c       The Lennard-Jones potential between mols(i) and mols(j)
c Called by:
c       flMCStep


real (dp) function calcLJ(i, j, mols)                                        80
        implicit none

        integer, INTENT(IN) :: i, j
        real (dp), DIMENSION(:,:), INTENT(IN) :: mols
        real (dp) :: dist2

        dist2 = flrrel2(i,j,mols)  !! we use the squared value since that is faster to calculate

c       sets cutoff distance for LJ interaction to rcut
        if ((isCyl.AND.dist2.LT.(min((0.5*hcav)**2,rcut**2))).OR.((.NOT.isCyl).AND.dist2.LT.(rcut**2))) then   90
                calcLJ = 4.0*eps0*((sig0**2/dist2)**6-(sig0**2/dist2)**3)
        else
                calcLJ = 0.0
        endif
end function calcLJ


c~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
c flrrel and flrrel2 Functions
c~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~                                              100

c calculates the (squared - flrrel2) distance between two molecules using the minimum image convention
c Parmeters:
c       i, j: indices in mols of the two atoms between which to calculate the distance (squared)
c       mols: an nx3 array of the positions of atoms
c Returns:
c       The distance (flrrel) or distance squared (flrrel2) between mols(i) and mols(j)
c Called by:
c       calcF, calcLJ, virial
c Notes:                                                                      110
c       It is noticeably faster to not take a square root if it can be avoided

real (dp) function flrrel(i,j,mols)
        implicit none
        integer, INTENT(IN) :: i, j
        real (dp), DIMENSION(:,:), INTENT(IN) :: mols

        flrrel = sqrt(flrrel2(i,j,mols))

end function flrrel                                                           120

real (dp) function flrrel2(i,j,mols)
        implicit none

        integer, INTENT(IN) :: i, j
        real (dp), DIMENSION(:,:), INTENT(IN) :: mols
        integer :: n

        flrrel2 = 0
        if (isCyl) then                                                       130
c       cylinder
c               try distance above and below
                flrrel2= (mols(i,1)-mols(j,1))**2+(mols(i,2)-mols(j,2))**2+&
                (min(abs(mols(i,3)-mols(j,3)), abs(hcav-abs(mols(i,3) - mols(j,3)))))**2
        else
```

```
c       periodic
c               try distance above and below for all three dimensions
                do n=1,3
                        flrrel2 = flrrel2 + &
                        (min(abs(mols(i,n)−mols(j,n)), abs(rcav−abs(mols(i,n) − mols(j,n))))**2
                enddo
        endif
end function flrrel2


c~~~~~~~~~~~~~~~~~~~~~~~
c totalEnergy Function
c~~~~~~~~~~~~~~~~~~~~~~~

c calculates the total energy
c Returns:
c       total energy
c Called by:
c       averageEnergy, flMCStep
c Calls:
c       calcLJ

real (dp) function totalEnergy()
        implicit none

        integer :: i,j


        totalEnergy = 0
        do i=1,nf
                do j=i+1,nf
                        totalEnergy = totalEnergy + calcLJ(i,j,xf)
                enddo
        enddo
end function totalEnergy


c~~~~~~~~~~~~~~~~~~
c virial Function
c~~~~~~~~~~~~~~~~~~

c calculate the virial
c Returns:
c       virial
c Called by:
c       Main (to calculate pressure)
c Calls:
c       calcF, flrrel

real (dp) function virial()
        implicit none
        integer :: i,j


        virial = 0

        do i=1,nf
                do j=i+1,nf
                        virial = virial + 1.0/3.0*calcF(i,j,xf)*flrrel(i,j,xf)
                enddo
        enddo
end function virial

end MODULE FLGenFncs
```

```
c FLUtil
c Useful routines mostly involving array manipulation or random numbers
c Contains:
c       copyArray, increaseCapacity, ran1

MODULE FLUtil

use types
use FLInfo

CONTAINS

c~~~~~~~~~~~~~~~~~~~~~~~~~
c copyArray subroutine
c~~~~~~~~~~~~~~~~~~~~~~~~~

c Copies 1-D array src to dest
c Intent(in):
c       src - the array from which to copy
c Intent(out)
c       dest - the array into which to copy
c Called by:
c       increaseCapacity

subroutine copyArray(src, dest)
        implicit none

        real (dp), DIMENSION(:), INTENT(IN) :: src
        real (dp), DIMENSION(:), INTENT(OUT) :: dest
        integer :: i

        do i=1,size(src)
                dest(i) = src(i)
        enddo
end subroutine

c~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
c increaseCapacity subroutine
c~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

c Doubles the capacity of xf and xfn
c Modifies:
c       xf, xfn, TOTAL
c Called by:
c       flMCStep
c Calls:
c       copyArray

subroutine increaseCapacity()

        implicit none

        real (dp), DIMENSION(TOTAL,3) :: tmp, tmpn
        integer :: j

        do j = 1,3
                call copyArray(xf(:,j), tmp(:,j))
                call copyArray(xfn(:,j), tmpn(:,j))
        enddo

c       you must deallocate an array before you can reallocate it!
        deallocate(xf, xfn, stat = error)

        if (error.ne.0) then
                write(*,*) "Unable to deallocate xf and xfn"
                STOP
        endif
```

```
        allocate(xf(TOTAL*2,3), stat = error)
        if (error .ne. 0) then                                                  70
                write(*,*) "Unable to increase size of xf. error was ",error
                STOP
        endif

        allocate(xfn(TOTAL*2,3), stat = error)
        if (error .ne. 0) then
                write(*,*) "Unable to increase size of xfn. error was ",error
                STOP
        endif
                                                                                80
        do j = 1,3
                call copyArray(tmp(:,j), xf(:,j))
                call copyArray(tmpn(:,j), xfn(:,j))
        enddo
        TOTAL = TOTAL*2
        print *, "increased capacity to ",TOTAL

end subroutine increaseCapacity

                                                                                90

c~~~~~~~~~~~~~~~~~~
c RAN1 FUNCTION
c~~~~~~~~~~~~~~~~~~

c Generates random numbers using the Mersenne twister
c This function takes extra arguments because it was also
c a linear congruential generator
double precision function ran1(idum)
        implicit none                                                           100
        integer, INTENT(IN) :: idum
        double precision :: genrand_res53

        ran1 = genrand_res53()

end function ran1

end MODULE FLUtil
```

```
c FLInfo
c Global variables and constants for FLMain

MODULE FLInfo

use types

implicit none

c~~~~~~~~~~~~~~                                                          10
c Constants
c~~~~~~~~~~~~~~
integer, PARAMETER :: initrand = 0
integer, PARAMETER :: fileNameLength=200 !! maximum file name length
real (dp), PARAMETER :: pi = 3.1415926535 !! pi
real (dp), PARAMETER :: kB = 3.16679e-6  !! Boltzmanns constant in AU
integer :: error                         !! To keep error return values

c~~~~~~~~~~~~~~~~
c Conversions                                                             20
c~~~~~~~~~~~~~~~~
real (dp), PARAMETER :: ANGSTR2AU = 1.8893


c~~~~~~~~~
c Files
c~~~~~~~~~
integer, PARAMETER :: NUMIN=10, NUMOUT=9
integer, DIMENSION(NUMIN), PARAMETER :: INFILES = (/14, 15,16,17,18,19,20,21,22,23/)
integer, DIMENSION(NUMIN), PARAMETER :: GBLINFILES = (/33,34,35,36,37,38,39,40,41,42/)   30
integer, DIMENSION(NUMOUT), PARAMETER :: OUTFILES=(/43, 44, 45, 46, 47, 48,49,50, 51/)
c names to identify files in code
integer, PARAMETER :: CMDFILE = GBLINFILES(1), &
                      FLFILE = INFILES(1), INITFILE = INFILES(2), &
                      MULTIFILE = INFILES(3), ETAFILE = INFILES(4), &
                      PLOTPOS = OUTFILES(1), HISTRHO = OUTFILES(2), &
                      AVGENG = OUTFILES(3), RESTART=OUTFILES(4), &
                      INITIAL=OUTFILES(5), WIDOMDATA = OUTFILES(6), &
                      GFILE = OUTFILES(7), HISTENG = OUTFILES(8), &
                      PRESSURE = OUTFILES(9)                              40
c file names
CHARACTER(len=8), DIMENSION(NUMOUT), PARAMETER :: OUTNAMES = &
      (/"flposn ", &       !! The fluid position
      "density ", &        !! Density/number of atoms in fluid
      "energy ", &         !! Average energy of the fluid
      "restart ", &        !! Use to restart sim if necessary
      "initfl  ", &        !! Initial fluid positions
      "widom ", &          !! Data from Widom Test
      "denstate", &        !! ln(density of states) for WL
      "histe   ", &        !! Energy histogram for WL                     50
      "pressure"/)         !!pressure data (no correction)


c~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
c General Fluid Information
c~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
integer :: loop                                !! The number of runs to do
integer :: npass                               !! Number of MC steps to do
integer :: nf                                  !! Current number of fluid atoms (will change with GC)
integer :: TOTAL                               !! Current max number of fluid atoms (for GC)   60
real (dp), ALLOCATABLE, DIMENSION(:,:) :: xf, xfn !! Arrays to store position info
real (dp), PARAMETER :: eps0 = 1.0             !! Depth of well is 1.0 in reduced units
real (dp), PARAMETER :: sig0 = 1.0             !! Van der Waals radius is 1.0 in reduced units
real (dp), PARAMETER :: rcut = 2.5*sig0        !! Cutoff radius
real (dp) :: fmass                             !! fluid mass
real (dp) :: rhof                              !! fluid density
real (dp), PARAMETER :: fmSTART = 0.9          !! Starting step size for fluids
```

```
real (dp) :: rcav, rcav2, hcav                !! Cavity properties
real (dp) :: beta                             !! 1/kT
real (dp) :: z                                !! exp(beta*mu)/GAMMA^3                          70
                                              !! mu = chemical potential
                                              !! GAMMA = (h^2/(2*pi*fmass)*beta)^(1/2)
real (dp) :: fmac, fmacsum, fmsize=fmSTART    !! Accepted move stuff for fluids
                                              !! (real to avoid problems with integer division)

c^~~~~~~~~~~
c Options
c^~~~~~~~~~~
logical :: isGrand                            !! True to use grand canonical, false to use canonical
logical :: isCyl                              !! True to use cylinder                          80
logical :: isMulti                            !! True to use multicanonical
logical :: isWL                               !! True to use Wang-Landau

c^~~~~~~~~~~~~~
c Widom Test
c^~~~~~~~~~~~~~
real (dp) :: widom = 0                        !! fluid averages
integer :: numWid = 0                         !! number info for averages

c^~~~~~~~~~~~~~~~~~                                                                            90
c Multicanonical
c^~~~~~~~~~~~~~~~~~
real (dp), ALLOCATABLE, DIMENSION(:) :: eta !! Weighting for multicanonical
integer :: multibins                          !! for multicanonical: size of eta

c^~~~~~~~~~~~~~~
c Wang-Landau
c^~~~~~~~~~~~~~~
real (dp), ALLOCATABLE, DIMENSION(:), TARGET :: g !! g(E) for W-L
real (dp), ALLOCATABLE, DIMENSION(:), TARGET :: histE !! H(E) for W-L                          100
real (dp), PARAMETER :: GRES = 0.5            !! Resolution of g(E) and histE
real (dp), PARAMETER :: FLAT = 0.93           !! When the histogram is considered "flat"
real (dp), PARAMETER :: MINF = 1.0000001 !! Stop when fwl falls below this value
integer :: gbins = 5000                       !! g(E) info
real (dp) :: fwl                              !! g(E) = fwl*g(E)

end MODULE FLInfo
```

Makefile:

```
#dependent files must be listed AFTER those files on which they are dependent
SRCFILES = mt19937ar.f Mac_types.f90 FLInfo.f95 FLUtil.f95 FLInitEnd.f95 FLGenFncs.f95 FLFluidFncs.f95 FLMain.f95

LDFLAGS = -L/Users/jbarry/Documents/Thesis/lib -lplplot \
        -X -framework -X Carbon -YEXT_NAMES=LCS
OPTFLAGS = -O3 -cpu:g5
PROFLAGS = -profile

default: GCMain
                                                                                                    10
GCMain: $(SRCFILES)
        f90 $(SRCFILES) -o FLMain $(OPTFLAGS) $(LDFLAGS)

clean:
        rm *.mod
        rm FLMain
```

B.2 Light Particle Codes

PROGRAM LPMAIN

```
c~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
c Modules (in alphabetical order):
c       LPFncs.f95 (~700 lines) - functions and subroutines dealing with the light particle
c       LPGenFncs.f95 (~350 lines) - functions less specific to this code but still
c                                    involving physics (calculating average
c                                    energy for example)
c       LPInfo.f95 (~175 lines) - all global variables and constants
c       LPInitEnd (~700 lines) - functions and subroutines required to initialize
c                                    and/or end the program
c       LPMain.f95 (~200 lines minus comments) - main method
c       LPUtil.f95 (~125 lines) - functions and subroutines of general use (doubling
c                                    the size of an array for example)
c       Mac_types.f90 (~10 lines) - Types definitions
c
cINPUT:
c       COMMAND FILE:
c       The prompt requires a loop file that contains a list of data file names.
c       This allows the program to run many data runs at once
c       Example file: ./example/loop.in
c
c       GENERAL DATA FILE:
c       This file has general information about the simulation, such as the directory
c       to store output files and general simulation constants.
c       Examples:       ./example/metr/genInfo.in
c                       ./example/wanglandau/genInfo.in (careful! this one runs a wang-
c                       landau simulation so there's no way of making it short. once you
c                       start it, it could take up to 3 days to finish)
c
c       FLUID DATA FILE:
c       This file has information about the fluid.
c       Examples: ./example/fluid.in
c
c       INIT FLUID DATA FILE
c       Line 1: starting step size
c       Rest of file: positions of fluid atoms (note that it is important to get the number of
c       atoms right in the fluid input file as well)
c       Example: ./example/frozenfl
c
c       LIGHT PARTICLE FILE
c       This file has information about the light particle.
c       Example: ./example/lp.in
c
c
c
cOUTPUT (7 files):
c       lpposn: contains the x, y, z coordinates of each bead at every WRITEDATA timesteps
c               the first line for each time step is the number of particles, and two
c               throwaway numbers (to make the file easy to read into a data analysis program)
c       flposn: could contain the x, y, z coordinates of each fluid atom at every WRITEDATA timesteps
c               the first line for each time step is the number of particles, the current stepsize,
c               and a throwaway number (to make the file easy to read into a data analysis program)
c               currently, not usually written to since we are not moving the fluid
c       density: could contain density, number of atoms at every WRITEDATA timesteps
c               currently, not usually written to since we are not moving the fluid
c       energy: contains average potential and kinetic energy at every WRITEDATA timesteps
c       denstate: contains the density of states after EACH flat histogram, written as
c               two-dimensional (Wang-Landau ONLY)
c       histe: contains the energy histogram after EACH flat histogram written as
c               two-dimensional (Wang-Landau ONLY)
c       accept: data about the acceptance rate and number of beads moved
c
c       TO CHANGE THESE FILES OR ADD ANOTHER, LOOK AT THE SYNTAX IN LPInfo.f95. To add a file,
c       increase NUMOUT, add another file descriptor to OUTFILES and add another name to OUTNAMES.
```

```
      use types
      use LPInfo
      use LPInitEnd
      use LPFncs                                                          70


c     Main Program

      implicit none

      integer :: i, loops, irun,j, cnt, counter, WRITEDATAWL = 5000, WRITEDATAM = 10
      real (dp) :: V, flatness
      character(fileNameLength) :: fname
                                                                          80
      call init_genrand(initrand) !! Inits the Mersenne twister
      call openCommandFile() !! In LPInitEnd

      runs: do loops=1,loop

c           reset variables
            numWid = 0
            irun = 0
            acsum = 0
            fmsize = fmSTART                                              90
            cmmove = cmSTART
            LPmove = LPSTART

c           initialize everything with the input files
            read(CMDFILE, *); read(CMDFILE,*) fname
            open(unit=GENFILE, file=fname,status='old', action='read')
            call GenInit()
            read(CMDFILE,*) fname
            open(unit=FLFILE,file=fname,status='old',action='read')
            call FLInit()                                                 100
            read(CMDFILE,*) fname
            open(unit=LPFILE, file=fname,status='old',action='read')
            call LPInit()

c           calculate the volume
            if (isCyl) then
                  V = pi*rcav2*hcav
            else
                  V = rcav**3
            endif                                                         110

c           main loop
            MC_passes: do  !! this needs to be a while loop for WL

                  if (irun.GT.npass-1.AND.(.NOT.isWL)) then
                        exit
                  else if (isWL.AND.irun > 0.AND.(mod(irun,CHECK)==0).AND.minval(histE)/(sum(histE)/(1.0*gbins1*gbins2)) >
c                 flat histogram
                        write(HISTENG, *) gbins1, gbins2
                        write(GFILE, *) irun, fwl                         120
                        do i=1,gbins1
                              write(HISTENG,*) histE(i,:)
                              write(GFILE,*) g(i,:)
                        enddo
                        call resetHist() !! in LPInitEnd
                        irun = 0
                        fwl = sqrt(fwl) !! decrease f
                        print *, "flat histogram!, f = ",fwl
                        if (fwl < MINF) exit
                        if (fwl < REDF) then                              130
                              FLAT = FLAT - REDFPER*FLAT
                              print *, "FLAT = ", FLAT
                        endif
```

```
                endif

c               move
                if (nb == 1) then
c               classical particle
                        call moveOne() !! in LPFncs
                else if (isWL) then                                          140
c               Wang-Landau
                        call moveWL(irun) !! in LPFncs
                        call updateCM() !! in LPGenFncs
                else if (isThread) then
c               Threading
                        call moveThread(irun) !! in LPFncs
                        call updateCM()     !! in LPGenFncs
                else
c               Metropolis
                        call move(irun) !! in LPFncs                         150
                        call updateCM() !! in LPGenFncs
                endif

c               print some data to a file
                if (isWL.AND.(mod(irun, WRITEDATAWL)==0)) then
c               WL data
                        write(LPPOS, *) nb,irun,fwl
                        do i=1,nb
                                write(LPPOS,*) x(i,1,1),x(i,1,2),x(i,1,3)
                        enddo                                                160

c                       we don't let the files get too big
                        counter = counter + 1
                        write(AVGENG, *) kinEnergy(), potEnergy(), irun, fwl
                        if (counter > 10000000) then
                                print *, 'exceeded maximum lines in file. . .exiting'
                                exit
                        endif
                else if (.NOT.isWL.AND.mod(irun,WRITEDATAM)==0) then
c               Metropolis data                                              170
                        write(LPPOS, *) irun,-1,-1
                        do i=1,nb
                                write(LPPOS,*) x(i,1,1),x(i,1,2),x(i,1,3)
                        enddo

                        write(AVGENG,*) kinEnergy(), potEnergy(), irun, mb

                endif

c               print to screen                                              180
                if (isWL.AND.mod(irun, 5000)==0) then
c               WL stuff
                        if (irun > 0) then
                                flatness = minval(histE)/(sum(histE)/(1.0*gbins1*gbins2))
                                print *, "flatness: ", flatness
                        else
                                flatness = 0
                        endif
                        print *, "pe = ",potEnergy(), "ke = ", kinEnergy(), "nf = ",nf, "f = ", fwl
c                       calculate how many zero elements if flatness = 0     190
                        if (flatness < 1e-8) then
                                cnt = 0
                                do i=1,gbins1
                                        do j=1,gbins2
                                                if (histE(i,j) < 1) then
                                                        cnt = cnt+1
                                                endif
                                        enddo
                                enddo
                                print *, "There are ", cnt, " zero elements."  200
                        endif
```

```
                else if (.NOT.isWL.AND.int(irun*100/npass) > int((irun-1)*100/npass)) then
c                   Metropolis
                        write(*,*) (irun*100/npass), "percent done"
                endif

c                   using accepted move ratios to determine cm step size and
c                   how many beads we use
                    acsum = acsum + ac                                                  210
                    accmsum = accmsum + accm
                    if (.NOT.isWL.AND.mod(irun,nevalu) == nevalu-1) then
                         ac = acsum/(nevalu-1)          ! Determine fraction of accepted moves
                                                          ! single bead moves only occur nevalu-1 times

c                       adjust mb so the acceptance rate is roughly 50%
                        if(ac > 0.5) mb=mb+1
                        if(ac < 0.5) mb=mb-1
                        if(mb < 1) mb=1
                        if(mb > nb) mb=nb                                               220
                        acsum = 0
                        write(ACCFILE, *) irun, ac, mb
                    endif
c                   Every (nevalu^2)-th move, we look at accmsum, and see how many moves are being accepted.
c                   If the rate is too low or two high, we adjust the size of the cm step.
                    if (.NOT.isWL.AND.mod(irun,nevalu**2) == nevalu**2-1) then
                        accm = accmsum/nevalu          !! Determine fraction of accepted moves
c                       adjust cmmove so the acceptance rate is roughly 50%
                        if(accm.GT.0.6.AND.cmmove.LT.rcav/4.0) cmmove=cmmove*1.2
                        if(accm.GT.0.6.AND.cmmove.GE.rcav/4.0) cmmove=rcav/4.0          230
                        if(accm.LT.0.4) cmmove=cmmove*0.8
                        accmsum = 0

                    endif

                    irun = irun + 1

                end do MC_passes

                call cleanUp() !! LPInitEnd                                             240
            enddo runs
            call endPro()
            print *, "program finished, see data files for results"

END PROGRAM LPMAIN
```

```
c InitFncs
c Contains routines used to initialize and/or end the program
c Subroutines:
c       cleanUp, endPro, FlInit, GenInit, initBeads,
c       initMulti, initWL, LPInit openCommandFile,
c       openFiles, packCube, packCyl, placeBeads,
c       resetHist, startFromConfig


MODULE LPInitEnd                                                        10

use types
use LPInfo
use LPFncs
use LPUtil

CONTAINS

c~~~~~~~~~~~~~~~~~~~~~
c cleanUp subroutine                                                    20
c~~~~~~~~~~~~~~~~~~~~~

c Cleans up after one run by deallocating variables and closing single-run files
c Deallocates:
c       x, xn, x_changed, xf, xfn, eta, g, histE
c Closes:
c       INFILES, OUTFILES
c Called by:
c       Main
                                                                        30
subroutine cleanUp()
        implicit none

        integer :: i

        do i=1,size(INFILES)
                close(INFILES(i))
        enddo

        do i=1,size(OUTFILES)                                           40
                close(OUTFILES(i))
        enddo

        deallocate(x, xn, xf, xfn, x_changed)
        if (isMulti) then
                deallocate(eta)
        endif
        if (isWL) then
                deallocate(g, histE)
        endif                                                           50
end subroutine cleanUp

c~~~~~~~~~~~~~~~~~~~~~
c endPro subroutine
c~~~~~~~~~~~~~~~~~~~~~

c Cleans up after entire program by closing multiple-run files
c Closes:
c       GBLINFILES
c Called by:                                                            60
c       Main
c Notes:
c       does NOT call cleanUp!

subroutine endPro()
        implicit none
```

```
        integer :: i

        do i=1,size(GBLINFILES)                                                    70
                close(GBLINFILES(i))
        enddo

end subroutine endPro


c~~~~~~~~~~~~~~~~~~~~~
c FLINIT SUBROUTINE
c~~~~~~~~~~~~~~~~~~~~~
                                                                                   80
c Initializes the fluid atoms based on data from FLFILE
c Initializes:
c       rhof, rcav, hcav, nf
c Allocates space for:
c       xf, xfn
c Calls:
c       packCyl, packCube, startFromConfig
c Called by:
c       Main
                                                                                   90
subroutine FlInit()
        implicit none

        logical :: initCond
        integer :: bool

        read(FLFILE,*);read(FLFILE,*) rhof

c       dimensions and number of fluid atoms
c       for now, this should always be canonical and periodic, but we leave this in here  100
c       to make updating the code for the other options reasonably easy
        if (isGrand) then
                read(FLFILE,*);read(FLFILE,*) rcav !! We specify rcav for gc
                if (isCyl) then
                        nf = int(rhof*pi*hcav*rcav**2)
                        read(FLFILE,*) hcav
                else
                        nf = int(rhof*rcav**3.0)
                endif
        else                                                                       110
                read(FLFILE,*);read(FLFILE,*) nf !! We specify nf for can
                if (isCyl) then
                        rcav = (nf/(rhof*pi*hcav))**(1.0/2.0)
                        read(FLFILE,*) hcav
                else
                        rcav = (nf/rhof)**(1.0/3.0)
                endif
        endif
        rcav2 = rcav*rcav
                                                                                   120
        if (isGrand) then
                TOTAL = nf*2
        else
                TOTAL = nf+1 !! we allocate one extra in case we want to do widom test
        endif

        allocate(xf(TOTAL,3), stat = error)
        if (error .ne. 0) then
                write(*,*) "Unable to allocate memory for the array: xf(:,:)"
                STOP                                                               130
        endif

        allocate(xfn(TOTAL,3), stat = error)
        if (error .ne. 0) then
                write(*,*) "Unable to allocate memory for the array: xfn(:,:)"
```

```
            STOP
        endif

c       at the moment, we should always be starting from an initial condition,
c       but the day may come when we are not...                                  140
        read(FLFILE,*);read(FLFILE,*) bool !! start from initial condition
        if (bool.NE.0) then
            initCond = .TRUE.
            call startFromConfig()
        else if (isCyl) then
            initCond = .FALSE.
            call packCyl()
        else
            initCond = .FALSE.
            call packCube()                                                      150
        endif

        print *, "the requested density gives", nf, "fluid atoms", " and a cavity of size", rcav

end subroutine FlInit


c~~~~~~~~~~~~~~~~~~~~~~
c GENINIT SUBROUTINE
c~~~~~~~~~~~~~~~~~~~~~~                                                          160

c Initializes the general variables for the simulation and opens files
c based on data in GENFILE
c Initializes:
c       beta, hbar, isWL
c Calls:
c       openFiles
c Called by:
c       Main
                                                                                 170
subroutine GenInit()
        integer :: bool

        call openFiles()

        read(GENFILE,*);read(GENFILE,*) beta
        read(GENFILE,*);read(GENFILE,*) hbar
        read(GENFILE,*);read(GENFILE,*) bool !! isWL

c       isWL                                                                     180
        if (bool == 1) then
            isWL = .TRUE.
            isThread = .FALSE. !! cannot do threading and WL together
            call initWL()
        else
            isWL = .FALSE.
        endif
        read(GENFILE,*);read(GENFILE,*) bool !! isThread
        if (bool == 1 .AND. .NOT.isWL) then
            isThread = .TRUE.                                                    190
        else
            isThread = .FALSE.
        endif

end subroutine GenInit


c~~~~~~~~~~~~~~~~~~~~~~~~
c initBeads subroutine
c~~~~~~~~~~~~~~~~~~~~~~~~                                                        200

c Initializes positronium beads from a file
c Modifies:
```

```
c       x, xn
c Called by:
c       LPInit

subroutine initBeads()
        implicit none
                                                                                210
        integer :: ierror
        integer :: ibeadcount ! number of beads in reading file
        integer :: ic         ! charge (electron = 1 or positron = 2)
        integer :: ib         ! bead
        character(fileNameLength) :: fname

        ibeadcount = 0
        read(LPFILE, *) fname
        open(unit=LPINITFILE, file = fname, status = 'old', action = 'read', iostat = ierror)
        if(ierror /= 0) then                                                    220
                write(*,*) 'An error occured opening init file for bead'
                STOP
        end if
        charge3: do ic = 1, 2
                bead3: do ib = 1, nb
                        read(LPINITFILE,*,iostat = ierror) x(ib,ic,1), x(ib,ic,2), x(ib,ic,3)
                        ibeadcount = ibeadcount+1
                        if (ierror /= 0) STOP
                end do bead3
        end do charge3                                                          230

        !Close off the chain
        x(nb+1,:,:) = x(1,:,:)

        if(ibeadcount /= 2*nb) then
                write(*,*) 'too few or too many beads in bead init file'
                STOP
        endif

        if (isCyl) then                                                         240
                x(:,:,3)=x(:,:,3)+0.5*hcav !! put the positronium halfway up the cylinder (in the middle)
        endif

        xn = x

end subroutine initBeads

c~~~~~~~~~~~~~~~~~~~~~~~~
c initMulti subroutine
c~~~~~~~~~~~~~~~~~~~~~~~~                                                       250

c Initializes the program to use multicanonical sampling
c Allocates space for:
c       eta
c Modifies:
c       eta, multibins
c Called by:
c       nothing at the moment, but might be useful to
c       reintroduce multicanonical sampling
                                                                                260
subroutine initMulti()
        implicit none

        integer :: i
        character(fileNameLength) :: fname

        read(FLFILE,*); read(FLFILE,*) fname
        open(unit=MULTIFILE, file = fname, status='old', action='read')
        read(MULTIFILE, *); read(MULTIFILE,*) multibins
                                                                                270
        allocate(eta(multibins), stat = error)
```

```
        if (error .ne. 0) then
                write(*,*) "Unable to allocate memory for the array: eta(:)"
                STOP
        endif
        read(MULTIFILE,*); read(MULTIFILE,*) fname

        open(unit=ETAFILE, file=fname,status='old', action='read')
        do i=1,multibins
                read(ETAFILE,*) eta(i)                                          280
        enddo
end subroutine initMulti

c~~~~~~~~~~~~~~~~~~~~
c initWL subroutine
c~~~~~~~~~~~~~~~~~~~~

c Initializes the program to use Wang-Landau sampling
c Allocates space for:
c       g, histE                                                                290
c Called by:
c       GenInit

subroutine initWL()
        implicit none

        allocate(g(gbins1, gbins2), stat = error)
        if (error .ne. 0) then
                write(*,*) "Unable to allocate memory for the array: g(:)"
                STOP                                                            300
        endif

        allocate(histE(gbins1, gbins2), stat = error)
        if (error .ne. 0) then
                write(*,*) "Unable to allocate memory for the array: histE(:)"
                STOP
        endif

        call resetHist()
                                                                                310
        g = 0.0

        fwl = exp(1.0)
end subroutine initWL

c~~~~~~~~~~~~~~~~~~~~
c LPInit subroutine
c~~~~~~~~~~~~~~~~~~~~

c Initializes light particle beads                                              320
c Initializes:
c       nb, mb, numcharge, npass, amass
c Allocates space for:
c       x, xn, x_changed
c Calls:
c       initBeads, placeBeads, CreateTable
c Called by:
c       Main

                                                                                330
subroutine LPInit()
        implicit none
        integer :: bool

        read(LPFILE,*);read(LPFILE,*) nb !! # beads for the particle
        read(LPFILE,*);read(LPFILE,*) mb !! # beads moved per staging pass
        read(LPFILE,*);read(LPFILE,*) npass !! # staging passes
        read(LPFILE,*);read(LPFILE,*) amass !! mass of a single quantum particle
        read(LPFILE,*);read(LPFILE,*) bool  !! start from config
```

```
                                                                        340
      allocate(x(nb+1,2,3), stat = error)
      if (error .ne. 0) then
            write(*,*) "Unable to allocate memory for the array: x(:,:,:)"
            stop
      endif

      allocate(xn(nb+1,2,3), stat = error)
      if (error .ne. 0) then
            write(*,*) "Unable to allocate memory for the array: xn(:,:,:)"
            stop                                                        350
      endif

      allocate(x_changed(nb+1,2), stat = error)
      if (error .ne. 0) then
            write(*,*) "Unable to allocate memory for the array: x_changed(:,:)"
            stop
      endif

c     The deB wavelength is a useful bit of trivia, so we calculate it out here.
      wave = sqrt(beta*hbar*hbar/amass)                                 360

      x_changed(:,:) = .TRUE.

      if (bool == 1) then
c     start from init file
            call initBeads()
      else
c     start in random place
            call placeBeads()
      endif                                                             370

end subroutine LPInit

c~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
c openCommandFile subroutine
c~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

c Prompts the user for and opens the command file
c Called by:
c     Main                                                              380

subroutine openCommandFile()
      implicit none

      character(fileNameLength) :: fname

      write(*,*) "Enter the name of the input file:"
      read(*,*) fname
      open(unit=CMDFILE, file=fname, status='old',action='read')
      read(CMDFILE,*); read(CMDFILE,*) loop                             390
      read(CMDFILE,*)
end subroutine openCommandFile


c~~~~~~~~~~~~~~~~~~~~~~~~~
c openFiles subroutine
c~~~~~~~~~~~~~~~~~~~~~~~~~

c opens input files
c Called by:                                                            400
c     genInit

subroutine openFiles()
      implicit none

      integer :: i
      character(fileNameLength) :: dirname
```

```
        read(GENFILE,*);read(GENFILE,*) dirname
        do i=1,size(OUTFILES)                                                    410
                open(unit=OUTFILES(i),file=(trim(dirname) // trim(OUTNAMES(i))),status='replace',action='write')
        enddo
end subroutine openFiles


c~~~~~~~~~~~~~~~~~~~~~~~
c packCube subroutine
c~~~~~~~~~~~~~~~~~~~~~~~

c Places fluid atoms uniformly in a cube                                         420
c Modifies:
c       xf
c Called by:
c       FLInit

subroutine packCube()

        integer :: i
        real (dp) :: spc, rat
                                                                                 430
        if (nf**(1.0/3.0)*sig0.GT.rcav) then
                write(*,*) "Too many atoms for this volume. Increase rcav or decrease rhof."
                STOP
        endif

        if (nf.LT.1) then
                write(*,*) "No atoms!"
                STOP
        endif
                                                                                 440
        spc = rcav/(nf**(1.0/3.0)) !! spacing between the atoms

        do i=1,nf
                rat = spc*i/rcav
                xf(i,1) = (rat - int(rat))*rcav
                xf(i,2) = (int(rat)*spc/rcav - int(int(rat)*spc/rcav))*rcav
                xf(i,3) = int(int(rat)*spc/rcav)*spc
        enddo

end subroutine packCube                                                          450

c~~~~~~~~~~~~~~~~~~~~~~~
c packCyl subroutine
c~~~~~~~~~~~~~~~~~~~~~~~

c Places fluid atoms uniformly in a cylinder
c Modifies:
c       xf
c Called by:
c       FLInit                                                                   460

subroutine packCyl()

        real (dp) :: rpack, rring, zring
        integer :: nlayers, npack, nrings, l

c       now we need to place the fluid atoms in the cavity in a dense fashion
        rpack=2.0*rcav/sig0

        if (rpack.LT.1.0) then                                                   470
                print *, "cannot pack any fluid atoms into a cylinder so small"
                STOP
        endif

        if (rpack.LT.2.0) then
```

```
            print *, " we use simple axial packing"
            if (nf.GT.int(hcav/sig0)) then
                  print *, "we cannot fit all requested atoms by axial packing"
                  STOP
            endif                                                              480
            nlayers=nf
            do k=1,nlayers
                  xf(k,1)=0.0
                  xf(k,2)=0.0
                  xf(k,3)=k*sig0
            enddo
      else
            npack=6
            nrings=0
            l=0                                                                490
            RingIteration: do !! this blind loop iterates over multiple packing rings
                  if (l.GT.nf) exit !! will never exit initially, only after running out of fluid atoms to place
                  npack=int(pi/asin(1.0/(rpack-1.0)))
                  print *, "we pack in a ring of", npack, "fluid atoms"
                  nrings=nrings+1
                  rring=sig0/(2.0*sin(pi/npack))
                  print *,"we pack at radius", rring
                  LayerIteration: do k=1,int(hcav/sig0) !! loop over the layers of atoms
                        zring=k*sig0
                        do                                                     500
                              l=l+1
                              if (l.GT.nf) exit
                              xf(l,1)=rring*cos(2.0*l*pi/npack)
                              xf(l,2)=rring*sin(2.0*l*pi/npack)
                              xf(l,3)=zring
                              if (mod(l,npack).EQ.0) exit
                        enddo
                  enddo LayerIteration

                  rpack=1.0/sin(pi/npack)-1.0                                  510

                  if (rpack.LT.2.0.AND.ℓ.LT.nf) then
                        print *, " we use simple axial packing for the remaining atoms"
                        if ((nf-l).GT.int(hcav/sig0)) then
                              print *, "we cannot fit all remaining atoms by axial packing"
                              STOP
                        endif
                        do k=l,nf
                              xf(k,1)=0.0
                              xf(k,2)=0.0                                      520
                              xf(k,3)=(k-l)*sig0
                        enddo
                        l=l+int(hcav/sig0)
                  endif

                  if (npack.LT.6.AND.ℓ.LT.nf) then
c                 abort if there are atoms let to place
c                 and we're trying to put them inside a less than 6-ring
                        print *, "too many atoms for too small a cavity"
                        STOP                                                   530
                  endif
            enddo RingIteration
      endif
end subroutine packCyl

c~~~~~~~~~~~~~~~~~~~~~~~~~~
c subroutine placeBeads
c~~~~~~~~~~~~~~~~~~~~~~~~~~

c Places beads in a Gaussian distribution                                      540
c Modifies:
c       x, xn
c Calls:
```

```
c       gauss
c Called by:
c       LPInit

subroutine placeBeads()
        implicit none
        integer :: is = 1       ! (rightnow, a dummy) variable for gaussian RNG                 550
        integer :: id           ! counter for dimension
        integer :: ic           ! charge (electron = 1 or positron = 2)
        integer :: ib           ! bead
        real (dp) :: xsum,xshift
        double precision :: gg
        logical :: goodPos

        if (isCyl) then
                gg = min(wave*wave,rcav*rcav/12.0)
        else                                                                                    560
                gg = min(wave*wave, rcav/2.0)
        endif
        ! start gg smaller if you wish
        if (isWL) gg = 0.2

        dim: do id = 1,3
                xsum = 0.0
                charge1: do ic = 1,2!numcharge
                        bead1: do ib = 1, nb
                                x(ib,ic, id) = gauss(gg,is)                                     570
                                xsum=xsum+x(ib,ic,id)
                        end do bead1
                        x(nb+1,ic,id) = x(1,ic,id)
                end do charge1
                xsum = xsum/2.0/float(nb)
                xshift = xc(id)-xsum
                charge2: do ic = 1,2
                        bead2: do ib = 1,nb+1
                                x(ib,ic,id)=x(ib,ic,id)+xshift
                        end do bead2                                                            580
                end do charge2
        end do dim
        x_changed(:,:) = .TRUE.
        if (isCyl) then
                x(:,:,3)=x(:,:,3)+0.5*hcav !! put the positronium halfway up the cylinder (in the middle)
        else
                if (.NOT. isWL) then
                        x(:,:,3) = x(:,:,3)+0.5*rcav
                else
c                       find a place where they aren't too close to the fluid                   590
                        goodPos = .FALSE.
                        do
                                if (goodPos) exit
                                goodPos = .TRUE.
                                if (((potEnergy()<CENTER2.OR.potEnergy()>(1.0*gbins2)*GRES2+CENTER2))) then
                                        goodPos = .FALSE.
                                        x(:,:,3) = x(:,:,3)+0.5*ran1(initrand)
                                        x_changed=.TRUE.
                                        do ib = 1, nb
                                                do id = 1,3                                     600
                                                        do ic = 1,numcharge
                                                                if (x(ib,ic,id).GT.rcav) then
                                                                        print *, 'No good initial position'
                                                                        STOP
                                                                endif
                                                        enddo
                                                enddo
                                        enddo
                                endif
                        enddo                                                                   610
                endif
```

```
        endif
        xn = x
end subroutine placeBeads

c~~~~~~~~~~~~~~~~~~~~~~~~
c resetHist subroutine
c~~~~~~~~~~~~~~~~~~~~~~~~

c Sets histE to all zeroes                                                    620
c Modifies:
c       histE
c Called by:
c       initWL, Main


subroutine resetHist()
        implicit none

        histE = 0.0                                                           630

end subroutine resetHist

c~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
c startFromConfig subroutine
c~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

c Initializes the fluid atoms from a file
c Modifies:
c       xf                                                                    640

subroutine startFromConfig()
        integer :: i
        character(fileNameLength) :: init

        read(FLFILE,*) init !! Name of initial configuration file
        print *, 'reading initial fluid positions from ', init
        open(unit=INITFILE, file=init, status='old', action='read')
        read(INITFILE,*) fmsize
        fmsize= fmsize*REDUCED2AU                                             650
        do i=1,nf
                read(INITFILE,*) xf(i,1), xf(i,2), xf(i,3)
        enddo
        xf(:,:) = xf(:,:)*REDUCED2AU
        print *, 'The energy of the starting configuration is ', totalFluidEnergy()
end subroutine startFromConfig

end MODULE LPInitEnd
```

```
c LPFncs
c Functions and subroutines for light particle
c Functions:
c       kinEnergy, kin1, lpflPot, potEnergy
c Subroutines:
c       move, move_cm, moveOne, moveWL, tryboth

MODULE LPFncs

use types                                                              10
use LPInfo
use LPUtil
use LPGenFncs

CONTAINS


c~~~~~~~~~~~~~~~~~~~~~~
c kinEnergy Function
c~~~~~~~~~~~~~~~~~~~~~~                                                20

c Returns:
c       current kinetic energy of the system

real (dp) function kinEnergy()
        implicit none

        integer :: i,j

        kinEnergy = 0.0                                                30
        do i=1,nb
                j = i+1
                if (i==nb) j = 1
                kinEnergy = kinEnergy + kin1(i,x,j,x)
        enddo
end function kinEnergy

c~~~~~~~~~~~~~~~~~
c kin1 Function
c~~~~~~~~~~~~~~~~~                                                     40

c Calculates the kinetic energy between two beads in chains x1 and x2
c Parameters:
c       i: index of bead in x1
c       x1: first chain
c       j: index of bead in x2
c       x2: second chain (usually same as x1)
c Returns:
c       the kinetic energy between x1(i,1,:) and x2(j,1,:)
                                                                       50
real (dp) function kin1(i,x1, j, x2)
        implicit none

        integer, INTENT(IN) :: i, j
        real (dp), dimension(nb+1,2,3), intent(in) :: x1, x2

        kin1 = 0.5*amass*nb/hbar**2*(mag2(x1(i,1,:), x2(j,1,:)))
end function kin1

c~~~~~~~~~~~~~~~~~~~                                                   60
c lpflPot Function
c~~~~~~~~~~~~~~~~~~~

c Calculates the potential between the fluid and light particle
c We use the truncated Lennard-Jones potential between argon and helium with
c Parameters:
c       LP - light particle to calculate potential for, 0 to use all particles
```

```
c       fl - fluid particle to calculate potential for, 0 to use all particles
c       xbead - bead positions
c       xfluid - fluid positions                                                    70
c       checkChanged - true if we only calculate potentials for beads that have been
c               moved (ie, x_changed = true), false to calculate regardless of x_changed
c Returns:
c       the potential between LP and fl
c Notes: the input vseudoar1(0,0,x,xf,.false.) will calculate the full potential energy
c       of the system

real (dp) function lpflPot(LP, fl, xbead, xfluid, checkChanged)
        implicit none
                                                                                    80
        real (dp), dimension(nb+1,2,3), intent(in) :: xbead
        real (dp), dimension(nf,3), intent(in) :: xfluid
        integer, INTENT(IN) :: fl, LP
        logical, intent(in) :: checkChanged
        integer :: i, j
        real (dp) :: dist2

        lpflPot = 0.0
        if (LP < 1 .AND. fl < 1) then
c       calculate potential of whole system                                         90
                do i = 1,nb
                        if (.NOT.checkChanged.OR.(checkChanged.AND.x_changed(i,1))) then
                                do j=1,nf
                                        dist2 = reldist2(xbead(i,1,:), xfluid(j,:)) !!in GenFncs
                                        if (dist2.LT.(rcutLp**2)) then
                                                lpflPot = lpflPot + 4.0*epsLp*((sigLp**2/dist2)**6-(sigLp**2/dist2)**3)
                                        endif
                                enddo
                        endif
                enddo                                                               100
        else if (LP < 1) then
c       calculate potential of all beads with one fluid atom
                do i=1,nb
                        if (.NOT.checkChanged.OR.(checkChanged.AND.x_changed(i,1))) then
                                dist2 = reldist2(xbead(i,1,:), xfluid(fl,:))
                                if (dist2.LT.(rcutLp**2)) then
                                        lpflPot = lpflPot + 4.0*epsLp*((sigLp**2/dist2)**6-(sigLp**2/dist2)**3)
                                endif
                        endif
                enddo                                                               110
        else if (fl < 1) then
c       calculate potential of all fluid atoms with one fluid bead
                if (.NOT.checkChanged.OR.(checkChanged.AND.x_changed(i,1))) then
                        do j=1,nf
                                dist2 = reldist2(xbead(LP,1,:), xfluid(j,:))
                                if (dist2.LT.(rcutLp**2)) then
                                        lpflPot = lpflPot + 4.0*epsLp*((sigLp**2/dist2)**6-(sigLp**2/dist2)**3)
                                endif
                        enddo
                endif                                                               120
        else
c       calculate potential between one fluid atom and one bead
                if (.NOT.checkChanged.OR.(checkChanged.AND.x_changed(i,1))) then
                        dist2 = reldist2(xbead(LP,1,:), xfluid(fl,:))
                        if (dist2.LT.(rcutLp**2)) then
                                lpflPot = 4.0*epsLp*((sigLp**2/dist2)**6-(sigLp**2/dist2)**3)
                        endif
                endif
        endif
                                                                                    130
end function lpflPot

c~~~~~~~~~~~~~~~~~~
c move subroutine
c~~~~~~~~~~~~~~~~~~
```

```
c Moves beads using basic PIMC
c Parameters:
c       irun - the step number of the loop
c Modifies:                                                              140
c       x, xn
c Calls:
c       kinEnergy, kin1, lpflPot, ran1
c Called by:
c       Main

subroutine move(irun)

        implicit none
        integer, INTENT(IN) :: irun                                      150
        integer :: ib, id, ibm, ibp
        real (dp) :: ke, kenew, pe, penew

c       initial energies
        ke = kinEnergy()/(beta**2)
        pe = potEnergy()

        do ib=1,nb
c       we always attempt to move all the beads one by one
                ibp = ib+1                                               160
                ibm = ib-1
                if (ib == nb) ibp = 1
                if (ib == 1) ibm = nb
                do id = 1,3
                        xn(ib,1,id) = x(ib,1,id) + LPmove*(ran1(initrand) - 0.5)
                enddo

c               calculate energies
                kenew = ke - kin1(ib,x,ibp,x)/(beta**2) - kin1(ib,x,ibm,x)/(beta**2) + kin1(ib,xn, ibp, xn)/(beta**2) &
                        + kin1(ib,xn, ibm,xn)/(beta**2)                  170
                penew = pe - 1.0/float(nb)*lpflPot(ib,0,x,xf,.false.) + 1.0/float(nb)*lpflPot(ib,0,xn,xf, .false.)
                if (dexp(-beta*(kenew+penew-(ke+pe))) > ran1(initrand)) then
                        ac=1.0d0
                        x = xn;
                        pe = penew
                        ke = kenew
                        x = xn
                else
                        xn =x
                endif                                                    180
        enddo

end subroutine move




c~~~~~~~~~~~~~~~~~~~~~~~~
c moveThread subroutine
c~~~~~~~~~~~~~~~~~~~~~~~~                                                190

c Moves beads or the center of mass using a threading algorithm
c Parameters:
c       irun - the step number of the loop
c Modifies:
c       x, xn, xc, x_changed, ac, accm
c Calls:
c       move_cm, tryboth, lpflPot
c Called by:
c       Main                                                             200

subroutine moveThread(irun)
```

```
      implicit none
      integer, INTENT(IN) :: irun
      integer :: i, j
      real (dp) :: vsum, vsumnew, vchange, de, det
      real (dp) :: effBeta       !! beta / number of beads

      effBeta = beta / float(nb)                                          210



c     do a staging move on the beads (re-pick from a gaussian distribution)
c     do electron and positron moves serially

      if (mod(irun,NCMMOVE).eq.0) then
            call move_cm(xn) !! move center of mass
      else
            call tryboth(xn) !! move according to threading               220
      end if

c     calculate potentials
      vsum=lpflPot(0, 0, x, xf, .true.)
      vsumnew=lpflPot(0, 0, xn, xf, .true.)
      vchange = (vsumnew - vsum)*effBeta

      det = -vchange
      de = dlog(ran1(initrand) + 1.0d-10)
      ac = 0.0d0                                                          230
      accm = 0.0d0

      if (det.GT.de) then
c     the move was accepted
            if (mod(irun,NCMMOVE).eq.0) then
                  accm = 1.0d0
            else
                  ac=1.0d0
            endif
                                                                          240
            do i = 1,nb
                  do j = 1,numcharge
                        if(x_changed(i,j)) then
                              x(i,j,:) = xn(i,j,:)
                              if(i.EQ.1) x(nb+1,j,:) = xn(nb+1,j,:)
                        endif
                  enddo
            enddo
      else
c     the move was rejected                                               250
            do i = 1,nb
                  do j = 1,numcharge
                        if(x_changed(i,j)) then
                              xn(i,j,:) = x(i,j,:)
                              if(i.EQ.1) xn(nb+1,j,:) = x(nb+1,j,:)
                        endif
                  enddo
            enddo
      endif
                                                                          260
end subroutine moveThread


c~~~~~~~~~~~~~~~~~~~~~
c move_cm subroutine
c~~~~~~~~~~~~~~~~~~~~~

c moves the center of mass of the two chains
c Parameters:
c     xnew - the old center of mass                                       270
c Pass-by-reference
```

```
c       xnew - will contain the new center of mass
c Modifies:
c       x_changed
c Called by:
c       move

subroutine move_cm(xnew)
        implicit none
        integer :: ic, ib                                                    280
        real (dp), DIMENSION(3) :: d
        real (dp), intent(INOUT), DIMENSION(:,:,:) :: xnew

        d(1) = cmmove*(ran1(initrand) - 0.5)
        d(2) = cmmove*(ran1(initrand) - 0.5)
        d(3) = cmmove*(ran1(initrand) - 0.5)
        do ic = 1,numcharge
                do ib = 1,nb+1
                        xnew(ib,ic,:) = xnew(ib,ic,:) + d
                enddo                                                        290
        enddo
        x_changed(:,:) = .true.

end subroutine move_cm


c~~~~~~~~~~~~~~~~~~~~~
c moveOne subroutine
c~~~~~~~~~~~~~~~~~~~~~
                                                                             300
c Moves a classical particle ("one bead")
c Modifies:
c       x, xn, x_changed, xEnergyOld, xEnergyNew,
c Calls:
c       move_cm, tryboth, LookUpTable, vfun1, vspeudoar
c Called by:
c       Main

subroutine moveOne()
        implicit none                                                        310

        integer :: i
        real (dp) :: vold, vnew, vchange

        do i=1,3
                xn(1,1,i) = x(1,1,i)+LPmove*(ran1(initrand)-0.5)
        enddo

c       calculate potentials
        vold = lpflPot(0,0,x,xf,.false.)                                     320
        vnew = lpflPot(0,0,xn,xf,.false.)

        vchange = vnew - vold
        if (dexp(-beta*vchange) > ran1(initrand)) then
c       accept move
                x = xn
        else
c       reject move
                xn = x
        endif                                                                330

end subroutine moveOne

c~~~~~~~~~
c moveWL
c~~~~~~~~~

c Does a Wang-Landau move
c INPUT:
```

```
c       irun - the current loop index (for printing purposes, not used in calculation)                    340
c Modifies:
c       x, xn, g, histE
c Calls:
c       kinEnergy, potEnergy, binPot, binKin
c Called by:
c       Main
cNotes: Assumes one chain only

subroutine moveWL(irun)
        implicit none                                                                                     350
        integer, intent(in) :: irun
        real (dp) :: ke, pe, kenew, penew
        integer :: ib, id, ibp, ibm, binp, bink, binpnew, binknew

        ac = 0

c       initial energies
        ke = kinEnergy()
        pe = potEnergy()
                                                                                                          360
        do ib=1,nb
c       we always attempt to move all the beads one by one
                ibp = ib+1
                ibm = ib-1
                if (ib == nb) ibp = 1
                if (ib == 1) ibm = nb
                do id = 1,3
                        xn(ib,1,id) = x(ib,1,id) + LPmove*(ran1(initrand) - 0.5)
                enddo
                                                                                                          370
c               calculate energies
                kenew = ke - kin1(ib,x,ibp,x) - kin1(ib,x,ibm,x) + kin1(ib,xn, ibp, xn) + kin1(ib,xn, ibm,xn)
                penew = pe - 1.0/float(nb)*lpflPot(ib,0,x,xf,.false.) + 1.0/float(nb)*lpflPot(ib,0,xn,xf, .false.)

c               bin energies (binning functions in LPGenFncs)
                binp = binPot(pe)
                bink = binKin(ke)
                binpnew = binPot(penew)
                binknew = binKin(kenew)
                if ((bink.LE.gbins1.AND.bink.GT.0).AND.(binp.LE.gbins2.AND.binp.GT.0) &                   380
                        .AND.(binknew.LE.gbins1.AND.binknew.GT.0).AND.(binpnew.LE.gbins2.AND.binpnew.GT.0)) then
c               everything is within the histograms
                        if (ran1(initrand).LT.exp(g(bink,binp)-g(binknew,binpnew))) then
c                       move accepted!
                                pe = penew
                                ke = kenew
                                x = xn
                                histE(binknew,binpnew) = histE(binknew,binpnew)+1
                                g(binknew,binpnew) = log(fwl) + g(binknew, binpnew)
                        else                                                                              390
c                       move rejected!
                                xn =x
                                histE(bink,binp) = histE(bink,binp)+1
                                g(bink,binp) = g(bink,binp)+log(fwl)
                        endif
                else if (.NOT.((bink.LE.gbins1.AND.bink.GT.0).AND.(binp.LE.gbins2.AND.binp.GT.0))) then
c                       we are outside our histogram... maybe a bad initial configuration?
                        print *, 'An impossible configuration has been reached with pe = ', pe
                        print *, ' ke = ',ke, ' tf = ', totalFluidEnergy(), 'bink = ', bink, 'binp = ', binp
                        STOP                                                                              400
                else
c               moving may not be energetically wrong, but it would put us outside the
c               histogram => we reject the move
                        xn = x
                        histE(bink,binp) = histE(bink,binp)+1
                        g(bink,binp) = g(bink,binp)+log(fwl)
                endif
```

```
      enddo

end subroutine moveWL                                                    410

c~~~~~~~~~~~~
c potEnergy
c~~~~~~~~~~~~

c Returns:
c       current potential energy

real (dp) function potEnergy()
      implicit none                                                      420

      potEnergy = totalFluidEnergy() + 1.0/float(nb)*lpflPot(0,0,x,xf, .false.)

end function potEnergy

c~~~~~~~~~~~~~~~~~~~~
c tryboth subroutine
c~~~~~~~~~~~~~~~~~~~~

c Makes trial moves of the beads                                         430
c Parameters:
c       xnew - old positions
c Pass-by-reference
c       xnew - will contain new positions
c Modifies:
c       x_changed
c Called by:
c       move
c Calls:
c       gauss                                                            440

subroutine tryboth(xnew)
      implicit none

      integer :: is = 1 !! (rightnow, a dummy) variable for gaussian RNG
      integer :: ic, id, ib, i, j
      real (dp), intent(INOUT), DIMENSION(:,:,:) :: xnew
      double precision :: const, g

      const=2.0d0*wave*wave/dfloat(nb)                                   450

      x_changed(:,:) = .FALSE.

c     pick new bead positions according to gaussian distn
c     id is the axis direction
      charge: do ic=1,numcharge
            dim: do id=1,3
c                 we go from the j bead to the j+mb bead
c                 (j is selected at random)
                  j=int(nb*ran1(initrand))+1                             460
                  beads: do i=1,mb
                        ib=j+mb-i+1
c                       account for periodicity in the chain
                        if (ib .GT. nb) ib = ib-nb
c                       the gaussian width depends on which bead we are at
                        g=const*dfloat(mb-i+1)/dfloat(mb-i+2)
                        xnew(ib,ic,id)= (xnew(ib+1,ic,id)*(mb-i+1)+xnew(j,ic,id))/float(mb-i+2) &
                        + gauss(g,is)

c                       flag the fact that this bead has been moved      470
                        x_changed(ib,ic) = .TRUE.

c                       close the chain if we have moved the 1st bead.
                        if(ib .EQ. 1)then
                              xnew(nb+1,ic,id)=xnew(1,ic,id)
```

```
                    endif
                enddo beads
            enddo dim
        enddo charge
end subroutine tryboth                                                    480

end MODULE LPFncs
```

```
c LPGenFncs:
c Contains functions relating to physics but not necessarily only applicable to this program
c Functions:
c       averageEnergy, binKin, binPot, calcF, calcLJ, flrrel, flrrel2, reldist, reldist2, virial
c Subroutines:
c       updateCM

MODULE LPGenFncs

use types                                                                    10
use LPInfo
use LPUtil

CONTAINS


c~~~~~~~~~~~~~~~~~~~~~~~~~
c averageEnergy function
c~~~~~~~~~~~~~~~~~~~~~~~~~
                                                                             20
c Returns:
c       the average energy of the fluid
c Called by:
c       Main
c Calls:
c       totalEnergy

real (dp) function averageFluidEnergy()
        implicit none
                                                                             30
        averageFluidEnergy = totalFluidEnergy()/nf
end function averageFluidEnergy

c~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
c binKin and binPot Functions
c~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

c bins (kinetic/potential) energy for Wang-Landau
c Parameters:
c       energy - energy to bin                                               40
c Returns
c       bin number corresponding to input energy

integer function binKin(energy)
        real (dp), INTENT(IN) :: energy

        binKin = int((energy-CENTER1)/GRES1+1)
end function binKin

integer function binPot(energy)                                              50
        real (dp), INTENT(IN) :: energy

        binPot = int((energy-CENTER2)/GRES2+1)
end function binPot

c~~~~~~~~~~~~~~~~~
c calcF Function
c~~~~~~~~~~~~~~~~~

c Parameters:                                                                60
c       i, j: indices in mols of the two atoms between which to calculate the
c               Lennard-Jones force for ARGON (not Ar-He)
c       mols: an nx3 array of the positions of atoms interacting via Lennard-Jones
c Returns:
c       The Lennard-Jones force between mols(i) and mols(j)
c Called by:
c       virial
```

```
real (dp) function calcF(i, j, mols)
        implicit none                                                                    70

        integer, INTENT(IN) :: i, j
        real (dp), DIMENSION(:,:), INTENT(IN) :: mols
        real (dp) :: dist

        dist = flrrel(i,j,mols)

        if ((isCyl.AND.dist.LT.min(0.5*hcav,rcut)).OR.((.NOT.isCyl).AND.dist.LT.(rcut))) then
                calcF = 4.0*eps0*(12*(sig0**12/dist**13)-6*(sig0**6/dist**7))
        else                                                                             80
                calcF = 0.0
        endif

end function calcF


c~~~~~~~~~~~~~~~~~~
c calcLJ Function
c~~~~~~~~~~~~~~~~~~
                                                                                         90
c Parameters:
c       i, j: indices in mols of the two atoms between which to calculate
c               the Lennard-Jones potential for ARGON (not Ar-He)
c       mols: an nx3 array of the positions of atoms interacting via Lennard-Jones
c Returns:
c       The Lennard-Jones potential between mols(i) and mols(j)
c Called by:
c       totalEnergy

                                                                                         100
real (dp) function calcLJ(i, j, mols)
        implicit none

        integer, INTENT(IN) :: i, j
        real (dp), DIMENSION(:,:), INTENT(IN) :: mols
        real (dp) :: dist2

        dist2 = flrrel2(i,j,mols)

                                                                                         110
c       sets cutoff distance for LJ interaction to rcut
        if ((isCyl.AND.dist2.LT.(min((0.5*hcav)**2,rcut**2))).OR.((.NOT.isCyl).AND.dist2.LT.(rcut**2))) then
                calcLJ = 4.0*eps0*((sig0**2/dist2)**6-(sig0**2/dist2)**3)
        else
                calcLJ = 0.0
        endif


end function calcLJ
                                                                                         120

c~~~~~~~~~~~~~~~~~~
c flrrel function
c~~~~~~~~~~~~~~~~~~

c Parameters:
c       i, j: indices in mols of the two atoms between which to calculate the relative distance
c       mols: an nx3 array of the positions of atoms
c Returns:
c       the relative distance between mols(i) and mols(j) taking into account image positions   130
c Calls:
c       flrrel2
c Called by:
c       calcF, virial
c Notes:
```

```
c       Avoid calling this if you can just call flrrel2 since it's faster to avoid taking square roots

real (dp) function flrrel(i,j,mols)
        implicit none
        integer, INTENT(IN) :: i, j                                                    140
        real (dp), DIMENSION(:,:), INTENT(IN) :: mols

        flrrel = sqrt(flrrel2(i,j,mols))

end function flrrel

c~~~~~~~~~~~~~~~~~~
c flrrel2 function
c~~~~~~~~~~~~~~~~~~
                                                                                       150
c Parameters:
c       i, j: indices in mols of the two atoms between which to calculate the relative distance
c       mols: an nx3 array of the positions of atoms
c Returns:
c       the square of the relative distance between mols(i) and mols(j) taking
c               into account image positions (fluid-fluid and fluid-bead but NOT bead-bead)
c Called by:
c       calcF, calcLJ, reldist2

real (dp) function flrrel2(i,j,mols)                                                   160
        implicit none

        integer, INTENT(IN) :: i, j
        real (dp), DIMENSION(:,:), INTENT(IN) :: mols

        integer :: n

        flrrel2 = 0

        if (isCyl) then                                                                170
                flrrel2= (mols(i,1)-mols(j,1))**2+(mols(i,2)-mols(j,2))**2+&
                (min(abs(mols(i,3)-mols(j,3)), abs(hcav-abs(mols(i,3) - mols(j,3)))))**2
        else

                do n=1,3
                        flrrel2 = flrrel2 + &
                        (min(abs(mols(i,n)-mols(j,n)), abs(rcav-abs(mols(i,n) - mols(j,n)))))**2
                enddo
        endif
                                                                                       180

end function flrrel2

c~~~~~~~~~~~~~~~~~~
c reldist function
c~~~~~~~~~~~~~~~~~~

c Parameters:
c       v1, v2 - three dimesional vectors between which to find the relative distance taking into
c                       account any periodic boundry conditions                        190
c Returns:
c       the relative distance between v1 and v2 taking into account any periodic boundry
c               conditions (use for fluid-bead, fluid-fluid, NOT bead-bead)
c Calls:
c       reldist2

real (dp) function reldist(v1, v2)
        implicit none

        real (dp), DIMENSION(3) :: v1, v2                                              200

        reldist = sqrt(reldist2(v1,v2))
end function reldist
```

```
c~~~~~~~~~~~~~~~~~~~
c reldist2 function
c~~~~~~~~~~~~~~~~~~~

c Parameters:
c       v1, v2 - three dimesional vectors between which to find the relative distance taking into
c                       account any periodic boundry conditions
c Returns:
c       the square of the relative distance between v1 and v2 taking into account
c               any periodic boundry conditions (use for fluid-bead, fluid-fluid, NOT bead-bead)
c Called by:
c       reldist, lpflPot
c Calls:
c       flrrel2

real (dp) function reldist2(v1, v2)
        implicit none
        integer i

        real (dp), DIMENSION(3), INTENT(IN) :: v1, v2
        real (dp), DIMENSION(2,3) :: mols


        mols(1, :) = v1
        mols(2, :) = v2
        if (isCyl) then
                mols(1, 3) = mols(1, 3) - hcav*ANINT(mols(1,3)/hcav)
                mols(2, 3) = mols(2, 3) - hcav*ANINT(mols(2,3)/hcav)
        else
                do i=1,3
                        mols(1, i) = mols(1, i) - rcav*ANINT(mols(1,i)/rcav)
                        mols(2, i) = mols(2, i) - rcav*ANINT(mols(2,i)/rcav)
                enddo
        endif
        reldist2 = flrrel2(1,2,mols)
end function reldist2


c~~~~~~~~~~~~~~~~~~~~~~~~
c mag and mag2 Functions
c~~~~~~~~~~~~~~~~~~~~~~~~

c Returns:
c       the distance (mag) or the squared distance (mag2) between two
c               vectors
c Called by:
c       kin1
c Notes:
c       Call mag2 if possible since it's faster to avoid taking square roots

real (dp) function mag(v1, v2)
        implicit none

        real(dp), DIMENSION(2), INTENT(IN)::v1, v2

        mag = sqrt(mag2(v1,v2))
end function mag

real (dp) function mag2(v1, v2)
        implicit none

        real(dp), DIMENSION(2), INTENT(IN)::v1, v2

        mag2 = (v1(1) - v2(1))**2 + (v1(2)-v2(2))**2 + (v1(3)-v2(3))**2
end function mag2
```

```
c~~~~~~~~~~~~~~~~~~~~~~
c totalEnergy function
c~~~~~~~~~~~~~~~~~~~~~~

c Returns:
c       the total energy of the system
c Calls:
c       calcLJ
c Called by:                                                                    280
c       Main, potEnergy

real (dp) function totalFluidEnergy()
        implicit none

        integer :: i,j


        totalFluidEnergy = 0
        do i=1,nf                                                               290
                do j=i+1,nf
                        totalFluidEnergy = totalFluidEnergy + calcLJ(i,j,xf)
                enddo
        enddo
end function totalFluidEnergy

c~~~~~~~~~~~~~~~~~~~~~~
c updateCM subroutine
c~~~~~~~~~~~~~~~~~~~~~~
                                                                                300
c Updates the center of mass of the beads after a move
c Modifies:
c       xc

subroutine updateCM()
        implicit none
        integer :: i,j
        real (dp), DIMENSION(3) :: xc1, xc2

        xc1(:)=0.0                                                              310
        xc2(:)=0.0
        xc(:)=0.0

        do i=1,3
                do j=1,nb
                        xc1(i) = xc1(i)+x(j,1,i)/float(nb)
                        if (.NOT.isWL) xc2(i) = xc2(i)+x(j,2,i)/float(nb)
                enddo
        enddo
                                                                                320
        if (isWL) then
                xc = xc1
        else
                xc = (xc1+xc2)/2.0
         endif

end subroutine updateCM


c~~~~~~~~~~~~~~~~~~                                                             330
c virial function
c~~~~~~~~~~~~~~~~~~

c Returns:
c       the virial of the system
c Called by:
c       Main
c Calls:
c       calcF, flrrel
```

```
                                                                                                    340
real (dp) function virial()
        implicit none
        integer :: i,j


        virial = 0

        do i=1,nf
                do j=i+1,nf
                        virial = virial + 1.0/3.0*calcF(i,j,xf)*flrrel(i,j,xf)                      350
                enddo
        enddo
end function virial


end MODULE LPGenFncs
```

```
c LPUtil
c Useful routines mostly involving array manipulation or random numbers
c Contains:
c       copyArray, gauss, increaseCapacity, ran1

MODULE LPUtil

use types
use LPInfo

CONTAINS

c~~~~~~~~~~~~~~~~~~~~~~
c copyArray subroutine
c~~~~~~~~~~~~~~~~~~~~~~

c Copies 1-D array src to dest
c Intent(in):
c       src - the array from which to copy
c Intent(out)
c       dest - the array into which to copy
c Called by:
c       increaseCapacity

subroutine copyArray(src, dest)
        implicit none

        real (dp), DIMENSION(:), INTENT(IN) :: src
        real (dp), DIMENSION(:), INTENT(OUT) :: dest
        integer :: i

        do i=1,size(src)
                dest(i) = src(i)
        enddo
end subroutine


c~~~~~~~~~~~~~~~~
c GAUSS FUNCTION
c~~~~~~~~~~~~~~~~

c Generates random numbers with a gaussian width
c Called by:
c       placeBeads, tryBoth

double precision function gauss(g,ix)
        implicit double precision (a-h, o-z)
        double precision :: rr, ss
        integer :: ix

        rr = (-dlog(ran1(initrand)+1.0d-10)*g ) ** 0.5
        ss = 6.283185307d0*ran1(initrand)
        gauss = rr*dcos(ss)

end function gauss

c~~~~~~~~~~~~~~~~~~~~~~~~~~~~
c increaseCapacity subroutine
c~~~~~~~~~~~~~~~~~~~~~~~~~~~~

c Doubles the capacity of xf and xfn (just GCMain!)
c Modifies:
c       xf, xfn, TOTAL
c Calls:
c       copyArray

subroutine increaseCapacity()
```

```
        implicit none
                                                                        70
        real (dp), DIMENSION(TOTAL,3) :: tmp, tmpn
        integer :: j

        do j = 1,3
                call copyArray(xf(:,j), tmp(:,j))
                call copyArray(xfn(:,j), tmpn(:,j))
        enddo

c       you must deallocate an array before you can reallocate it!
        deallocate(xf, xfn, stat = error)                               80

        if (error.ne.0) then
                write(*,*) "Unable to deallocate xf and xfn"
                STOP
        endif

        allocate(xf(TOTAL*2,3), stat = error)
        if (error .ne. 0) then
                write(*,*) "Unable to increase size of xf. error was ",error
                STOP                                                    90
        endif

        allocate(xfn(TOTAL*2,3), stat = error)
        if (error .ne. 0) then
                write(*,*) "Unable to increase size of xfn. error was ",error
                STOP
        endif

        do j = 1,3
                call copyArray(tmp(:,j), xf(:,j))                       100
                call copyArray(tmpn(:,j), xfn(:,j))
        enddo
        TOTAL = TOTAL*2
        print *, "increased capacity to ",TOTAL

end subroutine increaseCapacity

c~~~~~~~~~~~~~~~~
c RAN1 FUNCTION
c~~~~~~~~~~~~~~~~                                                       110

c Generates random numbers via a multiple linear congruential method
c and a table (it takes an input because it was once a linear congruential
c generator)
double precision function ran1(idum)
        implicit none

        integer, intent(IN) :: idum
        double precision genrand_res53
                                                                        120
        ran1 = genrand_res53()

end function ran1

end MODULE LPUtil
```

```
c LPInfo
c Global variables and constants

MODULE LPInfo

use types

implicit none

c~~~~~~~~~~~~                                                                  10
c Constants
c~~~~~~~~~~~~
integer, PARAMETER :: fileNameLength=200
real (dp), PARAMETER :: pi = 3.1415926535
real (dp), PARAMETER :: kB = 3.16679e-6  !! Boltzmanns constant in AU
real (dp), PARAMETER :: sigAR = 3.405*1.8893 !! sig in au for Ar-Ar
real (dp), PARAMETER :: epsAR = 119.8*kB !! eps in au for Ar-Ar
real (dp), PARAMETER :: sigLp = 6.4236      !! sig in au for He-Ar (should be 6.4236)
real (dp), PARAMETER :: epsLp = 9.5181e-5 !! eps in au for He-Ar
real (dp), PARAMETER :: rcutLp = 2.5*sigLp !! cutoff distance for He-Ar LJ Pot      20
integer :: error                                !! To keep error return values

c~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
c Conversions (conversions from and to reduced units assume we are using Ar)
c~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
c note that these may or may not be used in the code, but they're good to have around
real (dp), PARAMETER :: ANGSTR2AU = 1.8893 !! Angstrom to atomic units (distance)
real (dp), PARAMETER :: ANGSTR2REDUCED = 1.0/sigAR !! Angstroms to reduced units
real (dp), PARAMETER :: AU2REDUCED=1.0/(sigAR) !! Atomic units (distance) to reduced units
real (dp), PARAMETER :: KELVIN2AUENG = kB !! Temperature in Kelvin to atomic energy units      30
real (dp), PARAMETER :: AUENG2REDUCED = 1.0/(epsAR) !! Atomic units (energy) to reduced units/Users/jbarry/Documents/Th
                                        !! (we multiply by kB to convert eps to atomic)
c Going the other way...
real (dp), PARAMETER :: AU2ANGSTR = 1.0/ANGSTR2AU, REDUCED2ANGSTR = 1.0/ANGSTR2REDUCED, &
                        REDUCED2AU = 1.0/AU2REDUCED, AUENG2KELVIN = 1.0/KELVIN2AUENG, &
                        REDUCED2AUENG = 1.0/AUENG2REDUCED


c~~~~~~~~
c Files                                                                         40
c~~~~~~~~
integer, PARAMETER :: NUMIN=10, NUMOUT=7
integer, DIMENSION(NUMIN), PARAMETER :: INFILES = (/14, 15,16,17,18,19,20,21,22,23/)
integer, DIMENSION(NUMIN), PARAMETER :: GBLINFILES = (/33,34,35,36,37,38,39,40,41,42/)
integer, DIMENSION(NUMOUT), PARAMETER :: OUTFILES=(/43, 44, 45, 46, 47, 48, 49/)
c names to identify files
integer, PARAMETER :: CMDFILE = GBLINFILES(1), &
                      GENFILE = INFILES(1), &
                      FLFILE = INFILES(2), INITFILE = INFILES(7), &
                      MULTIFILE = INFILES(3), ETAFILE = INFILES(4), &                50
                      LPFILE = INFILES(5), LPINITFILE = INFILES(6), &
                      LPPOS = OUTFILES(1), FLPOS = OUTFILES(2), &
                      HISTRHO=OUTFILES(3), AVGENG = OUTFILES(4), &
                      GFILE = OUTFILES(5), HISTENG = OUTFILES(6), &
                      ACCFILE = OUTFILES(7)
c file names
CHARACTER(len=8), DIMENSION(NUMOUT), PARAMETER :: OUTNAMES = &
      (/"lpposn ", &       !! The light particle beads position
      "flposn ", &         !! The fluid position
      "density ", &        !! Density/number of atoms in fluid                     60
      "energy ", &         !! Average energy of the fluid
      "denstate", &        !! ln(density of states) for WL
      "histe   ", &        !! Energy histogram for WL
      "accept "/)          !! Data about acceptance rates

c~~~~~~~~~~~~~~~~~~~~~~
c General Information
```

```
c~~~~~~~~~~~~~~~~~~~~~~
integer :: loop                          !! The number of runs to do
integer :: npass                         !! Number of MC steps to do                    70
real (dp) :: beta                        !! 1/kT
real (dp) :: z                           !! exp(beta*mu)/GAMMA^3
                                         !! mu = chemical potential
                                         !! GAMMA = (h^2/(2*pi*fmass)*beta)^(1/2)
real (dp) :: hbar
real (dp) :: rcav, rcav2, hcav           !! Cylinder properties

c~~~~~~~~~~~~~~~~~~~~~~
c random number info
c~~~~~~~~~~~~~~~~~~~~~~                                                            80
integer, PARAMETER :: initrand = 27651   !! init for Mersenne Twister

c~~~~~~~~~~
c options
c~~~~~~~~~~
c for the moment, some of these need to be false because the options do not
c work yet. when they do work, these can become part of the input
logical :: isGrand = .false.             !! True to use grand canonical, false to use canonical
logical :: isCyl = .false.               !! True to use cylinder
logical :: isMulti = .false.             !! True to use multicanonical                  90
logical :: isWL                          !! True to use Wang-Landau (part of input)
logical :: isThread                      !! True to use threading (part of input)

c~~~~~~~~~~~~~~~~~~~~~
c Fluid Information
c~~~~~~~~~~~~~~~~~~~~~
integer, PARAMETER :: WIDOMNUM = 1       !! Number of times to do widom test per step
real (dp), PARAMETER :: fmSTART = 0.9    !! Starting step size

integer :: nf                            !! Current number of fluid atoms (will change with GC)  100
integer :: fmnum                         !! Every fmnum steps execute a fluid move
integer :: TOTAL                         !! Current max number of fluid atoms (for GC)
integer :: numWid = 0                    !! number info for averages
integer :: multibins                     !! for multicanonical: size of eta

real (dp), ALLOCATABLE, DIMENSION(:,:) :: xf, xfn !! Arrays to store position info
real (dp), ALLOCATABLE, DIMENSION(:) :: eta !! Weighting for multicanonical
real (dp), PARAMETER :: eps0 = epsAR     !! we are assuming for now that the fluid is always Ar
real (dp), PARAMETER :: sig0 = sigAR     !! we are assuming for now that the fluid is always Ar
real (dp), PARAMETER :: rcut = 2.5*sig0  !! cutoff radius                               110
real (dp) :: rhof                        !! fluid density
real (dp) :: widom = 0                   !! fluid averages
real (dp) :: fmac, fmacsum, fmsize=fmSTART !! Accepted move stuff for fluids
                                         !! (real to avoid problems with integer division)

c~~~~~~~~~~~~~~~~~~~~~~~~~
c Wang-Landau Information
c~~~~~~~~~~~~~~~~~~~~~~~~~
real (dp), PARAMETER :: GRES1 = 400000/20.0 !! Resolution for "kinetic"
real(dp), PARAMETER :: GRES2 = 0.003/50.0 !! Resolution for potential                  120
real (dp) :: FLAT = 0.93                 !! When the histogram is considered "flat"
real (dp), PARAMETER :: REDFPER = 0      !! Reduce FLAT by this percent of current value
real (dp), PARAMETER :: REDF = 1.05      !! Start reducing FLAT when fwl is below this value
real (dp), PARAMETER :: MINF = 1.0000001 !! Stop when fwl falls below this value
real (dp), PARAMETER :: CENTER1 = 0      !! subtract this value for kinetic energy
real (dp), PARAMETER :: CENTER2 = -0.0359 !! subtract this value for potential energy
integer, PARAMETER :: gbins1 = int(400000/GRES1+0.5) !! number of bins for "kinetic"

integer, PARAMETER :: gbins2 = int(0.003/GRES2+0.5) !! number of bins for potential
                                                                                   130
integer, PARAMETER :: CHECK = 5000       !! check for flatness every CHECK steps

real (dp), ALLOCATABLE, DIMENSION(:,:) :: g !! g(E) for W-L
real (dp), ALLOCATABLE, DIMENSION(:,:) :: histE !! H(E) for W-L
real (dp) :: fwl                         !! g(E) = fwl*g(E)
```

```
c~~~~~~~~~~~~~~~~~~~~~~~~~~~~
c Light Particle Information
c~~~~~~~~~~~~~~~~~~~~~~~~~~~~
integer, PARAMETER :: nevalu = 100          !! Update acceptance rate after nevalu steps
integer, PARAMETER :: NCMMOVE = 5000 !! Move cm after ncmmove steps
real (dp), PARAMETER :: cmSTART = 0.2   !! Starting stepsize for cm moves
real (dp), PARAMETER :: LPSTART = 0.2   !! Starting stepsize for positron moves
integer, PARAMETER :: RECENTER = 50   !! Recenter every this many moves
integer, PARAMETER :: numcharge = 1       !! one chain or two

integer :: nb                                   !! Number of beads
integer :: mb                                   !! Number of beads moved per pass
integer :: nequil                               !! Number of equilibration steps

logical, ALLOCATABLE, DIMENSION(:,:) :: x_changed !! True if the bead was moved

real (dp), ALLOCATABLE, DIMENSION(:,:,:) :: x, xn !! Arrays to store position info

real (dp), DIMENSION(3) :: xc                   !! Center of mass coordinates
real (dp) :: amass                              !! Mass of particle
real (dp) :: wave                               !! deB wavelength
real (dp) :: ac, acsum, LPmove                  !! Accepted move stuff for positron
real (dp) :: accm, accmsum, cmmove = cmSTART !! Accepted move stuff for center of mass

end MODULE LPInfo
```

Makefile:

```
#dependent files must be listed AFTER those files on which they are dependent
SRCFILES = mt19937ar.f Mac_types.f90 LPInfo.f95 LPUtil.f95 LPGenFncs.f95 LPFncs.f95 LPInitEnd.f95 LPMain.f95

LDFLAGS = -L/Users/jbarry/Documents/Thesis/lib -lplplot \
        -X -framework -X Carbon -YEXT_NAMES=LCS

OPTFLAGS = -O3 -cpu:g5
PROFLAGS = -profile -P

default: LPMain                                                                    10

LPMain: $(SRCFILES)
        f90 $(SRCFILES) -o LPMain $(OPTFLAGS) $(LDFLAGS)

clean:
        rm *.mod
        rm LPMain
```

B.3 General Codes

The code below defines types for use on a Macintosh computer

```
Module types
c Useful types
c Author: J. E. Pask
c       Dept. of Physics
c       University of California
c       Davis, California, 1/97
implicit none
private
public dp, hp

integer, parameter :: dp=kind(0.d0), &  !! double precision
                      hp=selected_real_kind(15) !! high precision
End Module types
```

The code below was used in testing random number generators. It includes the code for the linear congruential generator written by Tim Cronin [10] and calls the Mersenne generator, the algorithm for which is given in the module after this.

```
c testRan
c a simple program designed to test to random number generators

PROGRAM testRan

implicit none

integer :: i, j
integer, parameter :: ndim = 10
character(1), dimension(ndim) :: fc = ('0','1','2','3','4','5','6','7','8','9')
character(25) :: fname
double precision :: x, genrand_res53
integer, parameter :: limit = 1e7


x = 0

call init_genrand(27651)      !! init Mersenne twister

open(unit = 90, file = 'random1', status = 'replace', action = 'write')
open(unit = 91, file = 'randommt', status = 'replace', action = 'write')

do i=1,limit
        write(90,*) ran1(1)
        write(91,*) genrand_res53()
enddo


CONTAINS

c~~~~~~~~~~~~~~~~
 RAN1 FUNCTION
c~~~~~~~~~~~~~~~~

c Generates random numbers via a multiple linear congruential method
c and a table
double precision function ran1(idum)
        implicit none
        double precision :: r(97)
        integer, intent(IN) :: idum
        save
        integer, parameter :: M1=259200,IA1=7141,IC1=54773
        real, parameter :: RM1=1.0d0/M1
        integer, parameter :: M2=134456,IA2=8121,IC2=28411
        real, parameter :: RM2=1.0d0/M2
        integer, parameter :: M3=243000,IA3=4561,IC3=51349
        integer :: IX1, IX2, IX3, jjj
        integer :: iff=0

        if (idum < 0 .or. iff == 0) then
                iff = 1
                IX1 = mod(IC1-idum,M1)
                IX1 = mod(IA1*IX1+IC1,M1)
                IX2 = mod(IX1,M2)
                IX1 = mod(IA1*IX1+IC1,M1)
                IX3 = mod(IX1,M3)
                do jjj = 1,97
                        IX1 = mod(IA1*IX1+IC1,M1)
                        IX2 = mod(IA2*IX2+IC2,M2)
                        r(jjj) = (dfloat(IX1)+dfloat(IX2)*RM2)*RM1
                enddo
        endif
```

```
        IX1 = mod(IA1*IX1+IC1,M1)
        IX2 = mod(IA2*IX2+IC2,M2)
        IX3 = mod(IA3*IX3+IC3,M3)
        jjj = 1+(97*IX3)/M3
        if (jjj > 97 .or. jjj < 1) PAUSE
        ran1 = r(jjj)
        r(jjj) = (dfloat(IX1)+dfloat(IX2)*RM2)*RM1
end function ran1                                                    70

end PROGRAM testRan
```

The code below is code we did not write for the Mersenne twister random number generator.

```
c
c  A C-program for MT19937, with initialization improved 2002/1/26.
c  Coded by Takuji Nishimura and Makoto Matsumoto.
c
c  Before using, initialize the state by using init_genrand(seed)
c  or init_by_array(init_key, key_length).
c
c  Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
c  All rights reserved.
c  Copyright (C) 2005, Mutsuo Saito,                                    10
c  All rights reserved.
c
c  Redistribution and use in source and binary forms, with or without
c  modification, are permitted provided that the following conditions
c  are met:
c
c    1. Redistributions of source code must retain the above copyright
c       notice, this list of conditions and the following disclaimer.
c
c    2. Redistributions in binary form must reproduce the above copyright   20
c       notice, this list of conditions and the following disclaimer in the
c       documentation and/or other materials provided with the distribution.
c
c    3. The names of its contributors may not be used to endorse or promote
c       products derived from this software without specific prior written
c       permission.
c
c  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
c  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
c  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR   30
c  A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
c  CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
c  EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
c  PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
c  PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
c  LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
c  NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
c  SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
c
c                                                                       40
c  Any feedback is very welcome.
c  http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
c  email: m-mat
c
c-----------------------------------------------------------------
c  FORTRAN77 translation by Tsuyoshi TADA. (2005/12/19)
c
c     ----- initialize routines -----
c  subroutine init_genrand(seed): initialize with a seed
c  subroutine init_by_array(init_key,key_length): initialize by an array   50
c
c     ----- generate functions -----
c  integer function genrand_int32(): signed 32-bit integer
c  integer function genrand_int31(): unsigned 31-bit integer
c  double precision function genrand_real1(): [0,1] with 32-bit resolution
c  double precision function genrand_real2(): [0,1) with 32-bit resolution
c  double precision function genrand_real3(): (0,1) with 32-bit resolution
c  double precision function genrand_res53(): (0,1) with 53-bit resolution
c
c  This program uses the following non-standard intrinsics.             60
c    ishft(i,n): If n>0, shifts bits in i by n positions to left.
c                If n<0, shifts bits in i by n positions to right.
c    iand (i,j): Performs logical AND on corresponding bits of i and j.
c    ior  (i,j): Performs inclusive OR on corresponding bits of i and j.
```

```
c     ieor (i,j): Performs exclusive OR on corresponding bits of i and j.
c
c-----------------------------------
c     initialize mt(0:N-1) with a seed
c-----------------------------------
      subroutine init_genrand(s)                                          70
      integer s
      integer N
      integer DONE
      integer ALLBIT_MASK
      parameter (N=624)
      parameter (DONE=123456789)
      integer mti,initialized
      integer mt(0:N-1)
      common /mt_state1/ mti,initialized
      common /mt_state2/ mt                                               80
      common /mt_mask1/ ALLBIT_MASK
c
      call mt_initln
      mt(0)=iand(s,ALLBIT_MASK)
      do 100 mti=1,N-1
        mt(mti)=1812433253*
     &          ieor(mt(mti-1),ishft(mt(mti-1),-30))+mti
        mt(mti)=iand(mt(mti),ALLBIT_MASK)
  100 continue
      initialized=DONE                                                    90
c
      return
      end
c-----------------------------------
c     initialize by an array with array-length
c     init_key is the array for initializing keys
c     key_length is its length
c-----------------------------------
      subroutine init_by_array(init_key,key_length)
      integer init_key(0:*)                                               100
      integer key_length
      integer N
      integer ALLBIT_MASK
      integer TOPBIT_MASK
      parameter (N=624)
      integer i,j,k
      integer mt(0:N-1)
      common /mt_state2/ mt
      common /mt_mask1/ ALLBIT_MASK
      common /mt_mask2/ TOPBIT_MASK                                       110
c
      call init_genrand(19650218)
      i=1
      j=0
      do 100 k=max(N,key_length),1,-1
        mt(i)=ieor(mt(i),ieor(mt(i-1),ishft(mt(i-1),-30))*1664525)
     &          +init_key(j)+j
        mt(i)=iand(mt(i),ALLBIT_MASK)
        i=i+1
        j=j+1                                                             120
        if(i.ge.N)then
          mt(0)=mt(N-1)
          i=1
        endif
        if(j.ge.key_length)then
          j=0
        endif
  100 continue
      do 200 k=N-1,1,-1
        mt(i)=ieor(mt(i),ieor(mt(i-1),ishft(mt(i-1),-30))*1566083941)-i    130
        mt(i)=iand(mt(i),ALLBIT_MASK)
        i=i+1
```

```
        if(i.ge.N)then
          mt(0)=mt(N-1)
          i=1
        endif
  200 continue
      mt(0)=TOPBIT_MASK
c
      return                                                                    140
      end
c-------------------------------------
c     generates a random number on [0,0xffffffff]-interval
c-------------------------------------
      function genrand_int32()
      integer genrand_int32
      integer N,M
      integer DONE
      integer UPPER_MASK,LOWER_MASK,MATRIX_A
      integer T1_MASK,T2_MASK                                                   150
      parameter (N=624)
      parameter (M=397)
      parameter (DONE=123456789)
      integer mti,initialized
      integer mt(0:N-1)
      integer y,kk
      integer mag01(0:1)
      common /mt_state1/ mti,initialized
      common /mt_state2/ mt
      common /mt_mask3/ UPPER_MASK,LOWER_MASK,MATRIX_A,T1_MASK,T2_MASK          160
      common /mt_mag01/ mag01
c
      if(initialized.ne.DONE)then
        call init_genrand(21641)
      endif
c
      if(mti.ge.N)then
        do 100 kk=0,N-M-1
          y=ior(iand(mt(kk),UPPER_MASK),iand(mt(kk+1),LOWER_MASK))
          mt(kk)=ieor(ieor(mt(kk+M),ishft(y,-1)),mag01(iand(y,1)))              170
  100   continue
        do 200 kk=N-M,N-1-1
          y=ior(iand(mt(kk),UPPER_MASK),iand(mt(kk+1),LOWER_MASK))
          mt(kk)=ieor(ieor(mt(kk+(M-N)),ishft(y,-1)),mag01(iand(y,1)))
  200   continue
        y=ior(iand(mt(N-1),UPPER_MASK),iand(mt(0),LOWER_MASK))
        mt(kk)=ieor(ieor(mt(M-1),ishft(y,-1)),mag01(iand(y,1)))
        mti=0
      endif
c                                                                               180
      y=mt(mti)
      mti=mti+1
c
      y=ieor(y,ishft(y,-11))
      y=ieor(y,iand(ishft(y,7),T1_MASK))
      y=ieor(y,iand(ishft(y,15),T2_MASK))
      y=ieor(y,ishft(y,-18))
c
      genrand_int32=y
      return                                                                    190
      end
c-------------------------------------
c     generates a random number on [0,0x7fffffff]-interval
c-------------------------------------
      function genrand_int31()
      integer genrand_int31
      integer genrand_int32
      genrand_int31=int(ishft(genrand_int32(),-1))
      return
      end                                                                       200
```

```
c-----------------------------------
c     generates a random number on [0,1]-real-interval
c-----------------------------------
      function genrand_real1()
      double precision genrand_real1,r
      integer genrand_int32
      r=dble(genrand_int32())
      if(r.lt.0.d0)r=r+2.d0**32
      genrand_real1=r/4294967295.d0
      return                                                          210
      end
c-----------------------------------
c     generates a random number on [0,1)-real-interval
c-----------------------------------
      function genrand_real2()
      double precision genrand_real2,r
      integer genrand_int32
      r=dble(genrand_int32())
      if(r.lt.0.d0)r=r+2.d0**32
      genrand_real2=r/4294967296.d0                                   220
      return
      end
c-----------------------------------
c     generates a random number on (0,1)-real-interval
c-----------------------------------
      function genrand_real3()
      double precision genrand_real3,r
      integer genrand_int32
      r=dble(genrand_int32())
      if(r.lt.0.d0)r=r+2.d0**32                                       230
      genrand_real3=(r+0.5d0)/4294967296.d0
      return
      end
c-----------------------------------
c     generates a random number on [0,1) with 53-bit resolution
c-----------------------------------
      function genrand_res53()
      double precision genrand_res53
      integer genrand_int32
      double precision a,b                                            240
      a=dble(ishft(genrand_int32(),-5))
      b=dble(ishft(genrand_int32(),-6))
      if(a.lt.0.d0)a=a+2.d0**32
      if(b.lt.0.d0)b=b+2.d0**32
      genrand_res53=(a*67108864.d0+b)/9007199254740992.d0
      return
      end
c-----------------------------------
c     initialize large number (over 32-bit constant number)
c-----------------------------------                                   250
      subroutine mt_initln
      integer ALLBIT_MASK
      integer TOPBIT_MASK
      integer UPPER_MASK,LOWER_MASK,MATRIX_A,T1_MASK,T2_MASK
      integer mag01(0:1)
      common /mt_mask1/ ALLBIT_MASK
      common /mt_mask2/ TOPBIT_MASK
      common /mt_mask3/ UPPER_MASK,LOWER_MASK,MATRIX_A,T1_MASK,T2_MASK
      common /mt_mag01/ mag01
CC    TOPBIT_MASK = Z'80000000'                                       260
CC    ALLBIT_MASK = Z'ffffffff'
CC    UPPER_MASK = Z'80000000'
CC    LOWER_MASK = Z'7fffffff'
CC    MATRIX_A = Z'9908b0df'
CC    T1_MASK = Z'9d2c5680'
CC    T2_MASK = Z'efc60000'
      TOPBIT_MASK=1073741824
      TOPBIT_MASK=ishft(TOPBIT_MASK,1)
```

```
    ALLBIT_MASK=2147483647
    ALLBIT_MASK=ior(ALLBIT_MASK,TOPBIT_MASK)                                    270
    UPPER_MASK=TOPBIT_MASK
    LOWER_MASK=2147483647
    MATRIX_A=419999967
    MATRIX_A=ior(MATRIX_A,TOPBIT_MASK)
    T1_MASK=489444992
    T1_MASK=ior(T1_MASK,TOPBIT_MASK)
    T2_MASK=1875247104
    T2_MASK=ior(T2_MASK,TOPBIT_MASK)
    mag01(0)=0
    mag01(1)=MATRIX_A                                                           280
    return
    end
```

B.4 Post Processing Codes

This appendix contains codes used in post processing. All post processing was done in Matlab.

```
function [ave] = reweight(filename, numg, E)
%reweight: Matlab post-processing function
%Uses data from a Wang Landau simulation to calculate
%potential energy at a number of different temperatures

%Parameters:
%filename: the file with the density of states
%numg: the number of bins in the density of states
%E: a vector with the energy range in it (NOT the
%  energies recorded in simulation where each point
%  in the vector corresponds to a bin in the density of
%  states.  For example, for the constants given here
%  and in my thesis E = -400:0.04:-0.04

num = 100 %number of particles
          %this value is necessary because we record the
          %total energy of the simulation, but we only want
          %energy per particle. so in the end, we divide by
          %particle number
rho = 0.1 %fluid density
rc = 2.5  %cutoff radius (for calculating correction)
fsize = 14 %font size (graph)
tsize = 16 %title size (graph)

%values from johnson et al (for comparison)
jmd = [-0.863, -0.854, -0.795, -0.766, -0.717, -0.689, -0.669, -0.625, -0.592, -0.547, -0.510, -0.478]
jmc = [-0.869, -0.84, -0.6657]
tmc = [1.15,1.2,2.0]

gg = load(filename);
ind = find(gg == numg, 2, 'last') % we keep all of the densities of states as
                                  % we progress through the simulation,
                                  % but the last one will be the most
                                  % accurate
g = gg(ind(1)+1:ind(2)-1);
g = exp(g'-max(g)); %we store ln(g(E))
t= [1.15,1.2,1.3,1.4,1.6,1.8,2.0,2.5,3.0:1:6.0];   %t = 1/beta
for i=1:length(t)
    %actual reweigting!!
    ave(i) = sum(E.*g.*exp(-1.0/t(i)*(E-min(E))))/(sum(g.*exp(-1.0/t(i)*(E-min(E)))))/num;
end

%add in correction
ave = ave + 8.0/3.0*pi*rho*(1.0/3.0*(1/rc)^9 - (1/rc)^3);

%make a nice plot
figure
plot(t, ave, '*')
hold on
plot(t, jmd, 'rs')
plot(tmc, jmc, 'go')
legend('Wang-Landau', 'MD (Johnson et al)', 'MC (Johnson et al)')
set(gca, 'fontsize', fsize)
xlabel('T*', 'fontsize', fsize)
ylabel('U*', 'fontsize', fsize)
title('Average Internal Energy (\rho* = 0.1, N = 100)', 'fontsize', tsize)
```

```
function [xhist] = reweightSHO(x, H1, H2, g, beta)
%reweight: Matlab post-processing function
%Uses data from a Wang-Landau simulation in an SHO
%potential to calculate position at a number of
%different temperatures


%Parameters:
%x: the list of x values from the simulation
%H1: the list of kinetic energy values from
%    the simulation.
%H2: the list of potential energy values from
%  the simulation
%g: the density of states
%beta: beta value we are interested in
%NOTES:
%  The entries in x, H1, and H2 all need to correspond
%  ie, x(3) needs to be from the same timestep of the
%  simulation as H1(3) and H2(3)
%
%  The g file from the simulations is usually ln(g)
%  Don't forget to exponentiate it!

%Returns:
%  xhist: the probability density function for position
%  to calculate average position do x.*xhist/sum(xhist)
%  where x is the range of position values

%set these constants if your range changes!!
interval = 0.08     %assuming a square histogram, the bin size
engrng1 = 20        %kinetic energy range
engrng2 = 20        %potential energy range
nbins = engrng1/interval
xrng = 13           %position range

hist3 = zeros(nbins, nbins, nbins);
xhist = zeros(1,nbins);
h1hist = zeros(1,nbins);
h2hist = zeros(1,nbins);
enghist = zeros(nbins,nbins);

for i=1:length(x)
    indx = ceil((x(i) + xrng/2.0)*nbins/xrng);
    indh1 = ceil(H1(i)*nbins/engrng1);
    indh2 = ceil(H2(i)*nbins/engrng2);
    hist3(indx, indh1, indh2) = hist3(indx, indh1, indh2) + 1;
end

eng1 = 0:interval:engrng1-interval;
eng2 = 0:interval:engrng2-interval;

for i=1:nbins
    for j= 1:nbins
        %%reweighting!
        hist3(:,i,j) = hist3(:,i,j)*exp(-eng1(i)/beta - beta*eng2(j))*g(i, j);
    end
end

xhist = sum(sum(hist3,2),3);
enghist = reshape(sum(hist3,1),nbins,nbins);
h1hist = sum(enghist,2);
h2hist = sum(enghist,1);
avh1 = sum(eng1'.*h1hist)/sum(h1hist)
avh2 = sum(eng2.*h2hist)/sum(h2hist)
```

```
function [avgs, sds, x] = blocking(data, block, nstep)
%blocking: Matlab post-processing function
%Calculates the number of correlated steps
%for data and uses this to plot blocks with
%error bars

%Parameters:
%data: data to block
%block: size of a block
%nstep: the number of actual steps each data
%  point represents
%NOTES
%  block should be indices in data, not actual steps
%  for example, if we have 1000 MC steps and data contains
%  energy, say, for every 10 steps and we want blocks of 1000
%  MC steps, block = 10 because we only have data every 10 steps
%  (and nstep = 10)

%Returns:
%avgs: block averages
%sds: block standard deviations
%x: x-coordinate corresponding to each average (useful for plotting)

c = xcov(data, 'coeff'); %normalized tcf (xcov is a built-in matlab function)
c = c(ceil(end/2):end); %does the correlation going both directions, but we
                        %only need one
width = 500/nstep;      %to find how far we have to go to get noncorrelating steps
                        %we look for where the average of c over a
                        %window of width*2 drops below 0.05

%find non-correlated step number
i = width+1;
q = 1;
FL = -3.533120381468512E-002 %for the frozen fluid, this is the fluid energy
                             %it is always a constant so we just
                             %subtract it out to make the graphs easier
                             %to look at

while(q > 0.05)
    q = mean(c(i-width:i+width));
    i = i+1;
end
noncorr = i %how far apart steps need to be before they're noncorrelated

for i=block:block:length(data)
    avgs((i)/block) = sum(data(i-block+1:i))/(block)-FL;
    sds(i/block) = std(data(i-block+1:i));
    x(i/block) = i*nstep;
end

n = block/noncorr
sds = sds/sqrt(n); % error of the means

%make a plot
figure
errorbar(x,avgs, sds,'.k', 'MarkerSize', 20)
```