Path Integral Approaches and Graphics Processing Unit Tools for Quantum Molecular Dynamics Simulations

by

Stephen Joel Constable

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Science
in
Chemistry

Waterloo, Ontario, Canada, 2012

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

This thesis details both the technical and theoretical aspects of performing path integrals through classical Molecular Dynamics (MD) simulations. In particular, Graphics Processing Unit (GPU) computing is used to augment the Path Integral Molecular Dynamics (PIMD) portion of the widely available Molecular Modelling Tool Kit (MMTK) library. This same PIMD code is also extended in a different direction: a novel method for nuclear ground state property prediction is introduced that closely mimics existing code in functional form.

In order to add GPU computing capabilities to the existing MMTK codebase, the open source Open Molecular Mechanics (OpenMM) library was used. OpenMM provides high performance implementations of a variety of commonly used MD algorithms, with the goal of supporting current and future specialized hardware. Due to the object oriented nature of both codes, and the use of SI units in each, the development process was rather painless. The integration of OpenMM with MMTK is seamless, and arbitrary systems are supported without the user even needing to know that GPU acceleration is being used. The hybrid OpenMM-MMTK code is benchmarked against the vanilla MMTK code in terms of speed and accuracy, and the results show that GPU computing is the obvious choice for PIMD simulations.

Starting with a desire to apply the highly efficient Path Integral Langevin Equation (PILE) thermostat to the Path Integral Ground State (PIGS) problem, a new hybrid PILE-PIGS, or LE-PIGS, method was developed. This thesis describes the theoretical justification for this method, including the introduction of a modified normal mode representation based on the Discrete Cosine Transform (DCT). It is shown that in DCT space, the equations of motion of a PIGS system are virtually identical to the equations of motion of a PIMD system in Fourier space. This leads to direct reuse of existing PILE code in MMTK, and options to extend this ground state problem to OpenMM for the purpose of GPU acceleration. The method is applied to a series of model systems, and in each case convergence to the exact ground state energy is observed.

A number of avenues for further research are revealed. Most obviously, the OpenMM PIMD code uses the PILE internally so it can easily be modified to run LE-PIGS, which would create the first GPU-accelerated PIGS method. This would allow for the study of the ground states of large molecules, such as proteins, at high accuracy, in short time. Extensions to the OpenMM PIGS code are suggested that would increase the speed of the code on Nvidia GPUs. Also, the problem of finding the optimal estimate of the ground state wave function for use in LE-PIGS is discussed, and some ideas are presented on how to address this issue. Overall, future outlooks for this project are bright.

Acknowledgements

I would like to acknowledge my supervisor, Dr. Pierre-Nicholas Roy, for all the ideas that he has put into this thesis. Furthermore, he wrote the matrix multiplication code that was used to calculate good parameters for my LE-PIGS code. He also pushed me to compete at conferences and scholarship applications, and for that I am glad. Most importantly, he made himself available to me and the rest of our group on a daily basis, and often acted to inspire us with our own work.

I would like to thank my fellow members of the Roy research group, specifically Chris Ing and Matt Schmidt. Chris was the pioneer of path integrals in MMTK, and was an invaluable source of knowledge when I was getting started with my project, and later when I was implementing my LE-PIGS method. Matt worked to test my code on real systems to make sure it was working as expected, and I often found myself bouncing ideas off of him. I would also like to thank Konrad Hinsen, the primary author of MMTK, for his advice regarding the implementation of the OpenMM-MMTK hybrid code.

The credit for the schematic graphics of the flow of data on a motherboard goes to the very talented Steven Curtis.

Finally, the early editions of this thesis were proofread by a panel of my peers: many thanks to Richard Simms, Jasper Huang, Julian Martin, and Steven Curtis for their efforts.

Dedication

To my brother, Liam. May he be inspired to pursue a career in science one day.

Table of Contents

List of Figures

List of Abbreviations

MD	Molecular Dynamics
PIMD	Path Integral Molecular Dynamics
PIMC	Path Integral Monte Carlo
PIGS-MD	Path Integral Ground State Molecular Dynamics
RPMD	Ring Polymer Molecular Dynamics
PILE	Path Integral Langevin Equation
WNLE	White Noise Langevin Equation
MMTK	Molecular Modelling Toolkit
DFT	Discrete Fourier Transform
FFT	Fast Fourier Transform
DCT	Discrete Cosine Transform
HPC	High Performance Computing
DMC	Diffusion Monte Carlo
GPU	Graphics Processing Unit
CMD	Centroid Molecular Dynamics
PIGS	Path Integral Ground State
AMBER	Assisted Model Building with Energy Refinement
QTST	Quantum Transition State Theory
CUDA	Compute Unified Device Architecture
FFTW	Fastest Fourier Transform in the West
LE-PIGS	Langevin Equation Path Integral Ground State

Chapter 1

Introduction

Molecular Dynamics (MD) is a formal and computational tool that is used to solve problems in classical statistical mechanics. The approach can also be extended to account for quantum mechanical effects. The objective of the present work is to contribute to MD methodology in the area of software development, and on formal and methodological fronts. The software development contribution aspect aims at bringing the power of Graphical Processing Unit (GPU) computing to the Molecular Modelling Toolkits (MMTK), an open source toolkit for modelling and simulations. Another novel contribution is the development of a general approach to obtain exact, within statistical error, ground state properties of complex molecular systems. In order to understand how MD is used to compute observable properties, a derivation from classical statistical mechanics theory is first

required. Such essential elements are presented in the next section.

1.1 Fundamental Concepts of Statistical Mechanics

Classical statistical mechanics is distinguished from quantum statistical mechanics by the use of the continuous variables position and momentum to calculate properties[1], whereas quantum statistical mechanics considers averages of discrete quantum states. In classical particle mechanics, the total energy of a one-dimensional system with one particle of mass m feeling an external potential V is given, in Cartesian coordinates, by the Hamiltonian, H,

$$H(p,q) = \frac{p^2}{2m} + V(q) \ , \tag{1.1}$$

which is a function of the particle's position, q, and momentum, p. Classically, the momentum of a particle takes the form

$$p = mv = m\dot{q} = m\frac{\mathrm{d}q}{\mathrm{d}t} \ , \tag{1.2}$$

where v is the velocity. The first term of the Hamiltonian is simply the classical expression for the kinetic energy, K:

$$\frac{p^2}{2m} = \frac{m^2v^2}{2m} = \frac{1}{2}mv^2 = K \ . \tag{1.3}$$

Thus, the Hamiltonian is an expression for the total energy, E, of a classical system.

Consider a system consisting of many independent copies of the hypothetical system described by Equation (1.1) at some energy E: specifically N copies such that $N \to \infty$. Such a collection of systems is referred to as a *statistical ensemble*, and is useful in describing the bulk properties of molecular systems. With such a large collection of independent systems, it is not feasible to describe the ensemble in terms of the variables p and q because there are in principle infinitely many such variables. However, it is relatively easy to deal with averages of properties, such as the energy, over the entire ensemble. An ensemble average of some property A is denoted $\langle A \rangle$. In order to explicitly calculate the value of $\langle A \rangle$, one must invoke the fundamental postulate of statistical mechanics: in an ensemble of systems there is no bias as to which *state* a system will occupy (i.e. all accessible states are populated evenly). Within the context of classical statistical mechanics, the term 'state' refers to a unique combination of p and q, or more simply, a snapshot of the system. Such an average takes the following form:

$$\langle A \rangle = \frac{1}{\Omega} \int \mathrm{d}p \int \mathrm{d}q A \; \delta(H(p,q) - E) \; , \tag{1.4}$$

where δ is the dirac delta function, E is the energy of the ensemble members, and Ω is some normalization constant. The integral over p and q is called the integral over *phase*

space, which is spanned by every state of the system. The form of Ω may be calculated explicitly by noting it is expected that $\langle 1 \rangle$ should equal 1. By Equation (1.4):

$$\frac{1}{\Omega} \int \mathrm{d}p \int \mathrm{d}q \; \delta(H(p,q) - E) = 1 \; , \tag{1.5}$$

or, equivalently,

$$\Omega(E) \propto \int \mathrm{d}p \int \mathrm{d}q \; \delta(H(p,q) - E) \; . \tag{1.6}$$

The normalizing constant Ω has a special importance in statistical mechanics, and is referred to as the *density of states*. It is so named because it acts to count the number of times that a unique value of p and q, and therefore a unique state of the system, is observed within the ensemble. The proper density of states is normalized by some constant, but exact knowledge of this factor is not required because the same constant is present in both the numerator and denominator of Equation (1.4).

1.2 Molecular Dynamics

In general, for a molecular system, is is not feasible to directly solve for the average of A. Instead, it is possible to invoke the ergodic hypothesis: Given sufficient time, the trajectory of an ensemble member will eventually visit all of its available states. This means that

a time average over a trajectory can be directly equated with an ensemble average, or formally

$$\langle A \rangle = \lim_{T \to \infty} \frac{1}{T} \sum_{t=0}^{t=T} A(p(t), q(t)) \tag{1.7}$$

where T is the total time that the trajectory is observed. This hypothesis directly leads to the molecular dynamics method: given an ensemble member, numerically solve Hamilton's equations of motion [1], which dictate that

$$\frac{\mathrm{d}p(t)}{\mathrm{d}t} = -\frac{\partial H}{\partial q} \ , \tag{1.8}$$

$$\frac{\mathrm{d}q(t)}{\mathrm{d}t} = \frac{\partial H}{\partial p} \ . \tag{1.9}$$

This time evolution of the system according to Hamilton's Equations constitutes a trajectory. It is important to note that due to conservation of energy,

$$H(p(t), q(t)) = H(p(0), q(0)) = E \tag{1.10}$$

where E is the total energy of the system. Thus, when considering a classical system governed by Hamilton's Equations of motion, the form of the Hamiltonian and a single configuration of the system is sufficient to describe the trajectory of the system. After some amount of time, the average of some property A over the course of the trajectory

will converge to the ensemble average value. This technique is easily applicable to any system where the potential energy function is defined and differentiable everywhere in the accessible region of phase space. This requirement arises owing to the fact that, in solving Hamilton's equations of motion numerically, the gradient of the potential energy function is used to generate the forces on each particle. An algorithm that solves these equations numerically is called an *integrator*. One popular integrator is the Velocity Verlet integrator [2], given by the following set of equations:

$$v(t + \frac{1}{2}\Delta t) = v(t) + \frac{1}{2m}\nabla V(x(t))\Delta t \tag{1.11}$$

$$x(t + \Delta t) = x(t) + v(t + \frac{1}{2}\Delta t)\Delta t \tag{1.12}$$

$$v(t + \Delta t) = v(t + \frac{1}{2}\Delta t) + \frac{1}{2m}\nabla V(x(t + \Delta t))\Delta t \tag{1.13}$$

In this manner, the entire trajectory of a system can be elucidated from just an initial configuration of p and q, and knowledge of the gradient of V. Assuming the potential energy surface being explored is sufficiently smooth, this trajectory can reasonably be expected to visit all states contributing to the entire ensemble, and thus the time average will be equal to the ensemble average. The parameter Δt is called the time step, and it defines the smallest unit of time in the simulation. Ideally, Δt is chosen such that it is small enough that the dynamics of the system are sampled accurately, but large enough

that the simulation does not become prohibitively long in wall clock time. Specifically, Δt too large can lead to a collision event where two atoms that would normally repel each other are propagated into close distance of each other. The next time that ∇V is evaluated, the force will be extremely high and the atoms will be thrust apart at high velocity. This high velocity then increases the likelihood of a second collision event, and eventually the kinetic energy of the system diverges to infinity.

With the trajectory of a system in hand, it is possible to evaluate the time dependence of properties. One common approach to measuring properties as a function of time is through the use of *correlation functions*. The autocorrelation function, C_{AA} is an ensemble average that measures the time progression of a property A relative to its own starting value:

$$C_{AA}(t) = \langle A(0)A(t)\rangle \tag{1.14}$$

where it is understood that $A(0)$, the starting value, is averaged over all possible values. By invoking the ergodic hypothesis, it can be shown that

$$\langle A(0)A(t)\rangle = \frac{1}{t_{\max}} \sum_{t_0=1}^{t_{\max}} A(t_0)A(t_0+t) \ . \tag{1.15}$$

While it is obvious that this average can be computed directly, a more nuanced form can also be achieved. By appending a string of $t_{\max}$ zeros to the end of the data, $C_{AA}(t)$ can

be rapidly computed from the Fourier transform of $A(t)$ [3]. The Fourier transform can be computed rapidly on a computer using the FFT method, with $O(n \log n)$ time. The algorithm to calculate $C_{AA}(t)$ using the FFT are as follows:

1. Append an extra $t_{\max}$ data to prevent spurious correlations from arising, giving $2t_{\max}$ data in total.

2. Calculate $A(\nu)$ as the Fourier transform of $A(t)$ using the FFT.

3. Calculate $A^*(\nu)A(\nu)$ by squaring all values.

4. Calculate the unnormalized correlation function C'_{AA} by performing the inverse FFT on $A^*(\nu)A(\nu)$.

5. Normalize C'_{AA} with $(t_{\max} - t)^{-1}$ to yield C_{AA}.

The correlation time, τ_0 of property A is the time it takes for C_{AA} to decay to the equilibrium value, $\langle A \rangle^2$. The correlation time is important because measurements of A between $t = 0$ and $t = \tau_0$ are not statistically independent. The standard error on $\langle A \rangle$ is

$$\text{st. err.}(A) = \frac{\sigma_A^2}{\sqrt{N_A}} \tag{1.16}$$

where σ_A is the standard deviation of A and N_A is the number of statistically independent observations of A. Thus, the correlation time of A is required in order to accurately gauge

the measurement error in $\langle A \rangle$. Some dynamic properties can also be computed from C_{AA}, including diffusion coefficients and IR spectra.

Overall, MD is a powerful and general method that has been used to study a diverse array of physical systems: gases, liquids, solids, clusters, interfaces, and proteins are all examples of systems that have been studied with MD [4].

1.3 The Canonical Ensemble

Molecular dynamics as described above samples the microcanonical or NVE ensemble; the ensemble retains a constant number of particles (N), volume (V), and energy (E). For many applications, it is desired to work within the canonical (or NVT) ensemble, the main difference being that instead of conservation of energy in individual ensemble members, the temperature (T), or equivalently the kinetic energy, is conserved on average. This implies a slightly different ensemble average, since each member of the ensemble may be found in a different energy state. The obvious question follows: if energy is allowed to vary, in which energy states will the ensemble be found? To answer this question, the idea of a *heat bath* is introduced; the heat bath acts to exchange energy with the system such that a constant temperature is maintained (NVT), but the total energy of the system plus heat bath remains constant (NVE) [5].

Consider a system S coupled to a large external heat bath S', which together form a large system S^* whose energy is conserved. By the fundamental postulate of statistical mechanics, the system can be found in any accessible energy state m with equal a priori probability. The state m may vary in energy, so the energy corresponding to state m will be labelled E_m. The total energies of S^* and S' are denoted E^* and E' respectively, and it is obvious that the relation $E^* = E_m + E'$ holds universally. To answer the question of which states (i.e. values of m) the system S will be found in, it is assumed that the probability of observing state m is proportional to the number of states available to the heat bath, or formally:

$$p_m = C'\Omega'(E') \tag{1.17}$$

where p_m is the probability of S to be found in state m, Ω' is the density of states of the heat bath, and C' is some normalization constant. Taking the logarithm of both sides and acknowledging that the energy E' is simply the difference between the total energy E^* and the system energy E_m gives:

$$\ln(p_m) = \ln(C') + \ln(\Omega'(E^* - E_m)) \ . \tag{1.18}$$

Note that the total energy E^* is much greater than the system energy E_m, so a Taylor expansion in E^* around E_m may be used to describe the behaviour of the second term in

Equation (1.18). This approximation is useful even with only the first two terms of the expansion [6] [7]:

$$\ln(\Omega'(E^* - E_m)) = \sum_{k=0}^{\infty} \frac{(E' - E^*)^k}{k!} \frac{\mathrm{d}^k \ln(\Omega'(E^*))}{\mathrm{d}E'^k} \approx \ln(\Omega'(E^*)) - \frac{\mathrm{d}}{\mathrm{d}E'} \ln(\Omega'(E^*))E_m \ . \tag{1.19}$$

The term $\frac{\mathrm{d}}{\mathrm{d}E'} \ln(\Omega'(E^*))$ is given the symbol β and is referred to as the thermodynamic beta or the inverse temperature. Substituting β into equation (1.19) yields

$$\ln(p_m) = \ln(C') + \ln(\Omega'(E^*)) - \beta E_m \tag{1.20}$$

which can be exponentiated to give

$$p_m = C'\Omega'(E^*)e^{-\beta E_m} \ . \tag{1.21}$$

E^* is constant, so the two constants may be combined to give $C = C'\Omega'(E^*)$, and then the final result for p_m is obtained:

$$p_m = Ce^{-\beta E_m} \ . \tag{1.22}$$

With the equation for p_m in hand, it is possible to formulate an average of a property

A in the NVT ensemble:

$$\langle A \rangle = C \int \mathrm{d}p \int \mathrm{d}q e^{-\beta H(p,q)} A(p,q) \tag{1.23}$$

noting similarly to the NVE ensemble that $\langle 1 \rangle = 1$, the explicit form of C can be calculated:

$$\langle 1 \rangle = C \int \mathrm{d}p \int \mathrm{d}q e^{-\beta H(p,q)} = 1 \tag{1.24}$$

$$C = \frac{1}{Z} \propto \frac{1}{\int \mathrm{d}p \int \mathrm{d}q e^{-\beta H(p,q)}} \tag{1.25}$$

where Z is shorthand for an expression proportional to $\int \mathrm{d}p \int \mathrm{d}q e^{-\beta H(p,q)}$, called the *canonical partition function*. The role of Z in the NVT ensemble is analogous to the role of Ω in the NVE ensemble; it serves to normalize probabilities and encodes all the thermodynamic data for the system.

Some attention should be paid to the explicit form of β. Consider the formula for the entropy, S of a system [5]:

$$S = k_B \ln(\Omega) \ . \tag{1.26}$$

An infintesimal change in S is therefore

$$\mathrm{d}S = k_B \mathrm{d} \ln(\Omega) \ . \tag{1.27}$$

Recalling that the definition of β is

$$\beta = \frac{\mathrm{d}\ln(\Omega)}{\mathrm{d}E} \ , \tag{1.28}$$

the formulæ for $\mathrm{d}S$ and β can be equated via $\mathrm{d}\ln(\Omega)$:

$$\beta = \frac{1}{k_B}\frac{\mathrm{d}S}{\mathrm{d}E} \ . \tag{1.29}$$

Thermodynamically, it is known that

$$\frac{\mathrm{d}S}{\mathrm{d}E} = \frac{1}{T} \ , \tag{1.30}$$

and therefore the explicit form of β is

$$\beta = \frac{1}{k_B T} \ . \tag{1.31}$$

Thus, an alternative way to write Z is

$$Z \propto \int \mathrm{d}p \int \mathrm{d}q e^{\frac{-H(p,q)}{k_B T}} \tag{1.32}$$

The NVT ensemble more accurately reflects common experimental conditions, and

therefore NVT ensemble averages will more faithfully reproduce many experimentally observed phenomena. Unfortunately, the previously described MD method is inherently restricted to sample the NVE ensemble. It is therefore desirable to incorporate modifications to the MD method so that the NVT ensemble may be sampled. In general, any such modification is referred to as a *thermostat*, because it acts to maintain a constant temperature throughout the simulation.

One basic approach to thermostatting an MD simulation is the Andersen Thermostat [8]: at random time intervals, simulate a collision with a heat bath. The time interval at which to adjust the velocities is called the collision frequency, and is chosen from the Poisson Distribution $P(\nu) = \nu e^{-\nu t}$. During a collision event, the velocities of each particle in the system are adjusted by randomly sampling from the Maxwell-Boltzmann distribution at temperature T. Between collisions, the simulation is progressed via the regular NVE integrator. While this thermostat gives a correct, NVT ensemble average, the trajectory becomes completely unphysical due to the random reassignment step.

The Nosé-Hoover Thermostat [9] offers an improvement over the Andersen Thermostat. The Nosé-Hoover thermostat attempts to model the interactions between the system S and the heat bath S' in a simplified manner by introducing an additional degree of freedom to the system that acts to constrain the kinetic energy of the system to the desired temperature. The Nosé-Hoover approach can be repeated multiple times to yield a Nosé-

Hoover Chain thermostat [10], which is generally considered to be the gold standard in thermostatting Path Integral Molecular Dynamics (PIMD) simulations, described in the next section [11].

While the Nosé-Hoover Chain thermostat is effective, it is conceptually difficult to implement due to the additional degrees of freedom required. A slightly simplified thermostat is the Langevin equation [12] [13], which models the additional degrees of freedom in a stochastic manner. Specifically, a friction term γ is added for each particle that describes the drag experienced by the presence of heat bath particles, and a random number ξ is added to simulate high-energy collisions with the bath. Formally the force on particle i, F_i, becomes

$$F_i(t) = \nabla V(q_i(t)) - \gamma_i p_i(t) + \sqrt{\frac{2\gamma_i m}{\beta}}\xi(t) \ . \tag{1.33}$$

The friction parameters γ_i controls per-particle coupling to the heat bath such that lighter particles do not become overdamped and heavier particles do not become underdamped. The magnitude of the random force also scales with γ_i, while the temperature is set with β. The random number ξ is assumed to have a mean of zero and obey a delta correlation function,

$$\langle \xi(0)\xi(t) \rangle = \delta(t) \ , \tag{1.34}$$

so that there is no correlation between measurements of $\xi(t)$. The relative simplicity of

the Langevin equation compared to the Nosé-Hoover Chain thermostat, combined with its statistical advantages over the Andersen thermostat, make it a popular thermostat choice in modern MD simulations [13].

1.4 Path Integral Molecular Dynamics

Classical MD can describe many systems with good accuracy. However, sometimes the effects of the underlying quantum structure of the system cannot be ignored, and classical MD breaks down. This is especially true in the low temperature limit, where the thermal wavelength of each particle becomes large [14] and exchange effects become important. Additionally, classical MD fails to accurately describe tunneling effects in the transfer of a proton from acid to base [15]. In order to capture these effects, the path integral approach of Feynnman is adopted [16]. It was pointed out that MD and Monte Carlo [17] [18] algorithms could be extended to sample quantum effects via path integrals through a classical-quantum isomorphism [19] [20]; specifically, each particle is replaced with a cyclical polymer of particles feeling a modified potential.

In electronic structure theory, the Born-Oppenheimer approximation is often employed [21]. This approximation asserts that the many body wavefunction of a molecule can be

separated into a product of electronic and nuclear wavefunctions:

$$|\Psi\rangle = |\psi_{\text{elec.}}\rangle|\psi_{\text{nuc.}}\rangle \ . \tag{1.35}$$

Electronic structure theorists then focus on deriving the form of $|\psi_{\text{elec.}}\rangle$ through various techniques. One might reasonably expect that at equilibrium, path integrals could be used to sample averages of properties of both the electronic and nuclear wavefunctions. Unfortunately, imposing the required symmetry constraints on the electronic wavefunction (i.e. the Pauli exclusion principle) leads to the summation of rapidly oscillating positive and negative regions of $|\psi_{\text{elec.}}\rangle$, resulting in what is known in numerical analysis as *catastrophic cancellation*. The integral is algebraically exact, but attempts to numerically evaluate it diverge from the expected result. Physicists have termed this effect the *fermion sign problem* [22]. For this reason, PIMD is effectively restricted to bosonic systems, such as even-numbered nuclei in an effective potential. The following derivation is heavily based on work by Tuckerman [23].

Quantum mechanically, the nuclear portion of the system can be thought of as a state vector, $|\Psi\rangle$, which encodes all of the nuclear properties of the system. In turn, this state vector can be represented in an arbitrary, possibly infinite, basis set of vectors $\{|\phi_i\rangle\}$. Thus,

without loss of generality:

$$|\Psi\rangle = \sum_{i=0}^{\infty} c_i |\phi_i\rangle \ . \tag{1.36}$$

Consider a statistical ensemble consisting of Z such systems. An ensemble average of property A must consist of the weighted average over each ensemble member's expectation values of the quantum mechanical operator $\hat{A}$:

$$\langle A\rangle = \frac{1}{Z}\sum_{k=1}^{Z} \langle \Psi^{(k)}|\hat{A}|\Psi^{(k)}\rangle \ . \tag{1.37}$$

By representing each $|\Psi^{(k)}\rangle$ in its ϕ basis, it is obvious that

$$\langle A\rangle = \sum_{i,j}\left(\frac{1}{Z}\sum_{k=1}^{Z} c_i^{(k)} c_j^{(k)}\right)\langle \phi_i|\hat{A}|\phi_j\rangle \ . \tag{1.38}$$

From Equation (1.38), it is evident that $\langle A\rangle$ can be compactly expressed in terms of the trace of a matrix:

$$\langle A\rangle = \sum_{i,j} \rho_{ij} A_{ij} = \mathrm{Tr}(\rho A) \ , \tag{1.39}$$

where $A_{ij} = \langle \phi_i|\hat{A}|\phi_j\rangle$ and ρ, called the density matrix, is given by

$$\rho_{ij} = \frac{1}{Z}\sum_{k=1}^{Z} c_i^{(k)} c_j^{(k)} \ . \tag{1.40}$$

The density matrix can also be expressed as a density operator ρ, having $\rho_{ij} = \langle\phi_i|\rho|\phi_j\rangle$, explicitly given by:

$$\rho = \frac{1}{Z}\sum_{k=1}^{Z}|\Psi^{(k)}\rangle\langle\Psi^{(k)}| \ , \tag{1.41}$$

implying that ρ is the ensemble average of projection operators.

The quantum time evolution operator can be introduced to investigate the time evolution properties of ρ. Specifically,

$$|\Psi(t)\rangle = e^{-i\hat{H}t/\hbar}|\Psi(0)\rangle \ , \tag{1.42}$$

and therefore

$$\rho(t) = \frac{1}{Z}\sum_{k=1}^{Z} e^{-i\hat{H}t/\hbar}|\Psi^{(k)}(0)\rangle\langle\Psi^{(k)}(0)|e^{i\hat{H}t/\hbar} \tag{1.43}$$

$$= e^{-i\hat{H}t/\hbar}\rho e^{i\hat{H}t/\hbar} \ . \tag{1.44}$$

By taking the partial derivative of Equation (1.44) with respect to time, it can be shown that

$$\frac{\partial\rho(t)}{\partial t} = -\frac{i}{\hbar}(\hat{H}\rho(t) - \rho(t)\hat{H}) = -\frac{i}{\hbar}\left[\hat{H}, \rho(t)\right] \ . \tag{1.45}$$

Note that at equilibrium, there can be no change in the density with respect to time

$\left(\frac{\partial \rho(t)}{\partial t} = 0\right)$, which implies that $\hat{H}$ and ρ commute at equilibrium. This relationship can be used to show that ρ and $\hat{H}$ can be diagonalized together, and that ρ can be expressed as a function of $\hat{H}$. This means that

$$\rho = f(\hat{H}) = \sum_i f(E_i)|E_i\rangle\langle E_i| \ . \tag{1.46}$$

The particular choice of $f(E_i)$ implies a particular choice of ensemble. For the canonical ensemble, $f(E_i) = e^{-\beta E_i}/Z$, with Z as the quantum canonical partition function

$$Z = \sum_i e^{-\beta E_i}|E_i\rangle\langle E_i| = \mathrm{Tr}\left(e^{-\beta \hat{H}}\right) \tag{1.47}$$

and therefore the quantum mechanical canonical average of property A is given by

$$\langle A\rangle = \frac{\mathrm{Tr}(e^{-\beta \hat{H}}\hat{A})}{\mathrm{Tr}(e^{-\beta \hat{H}})} \tag{1.48}$$

In order to evaluate the quantum mechanical canonical average of a property A using

MD, consider the representation of Z in a position basis,

$$Z = \mathrm{Tr}(e^{-\beta H}) = \int \mathrm{d}x \langle x|e^{-\beta \hat{H}}|x\rangle \tag{1.49}$$

$$= \int \mathrm{d}x \langle x|e^{-\beta(\hat{K}+\hat{V})}|x\rangle \tag{1.50}$$

In general, it is not possible to evaluate $\langle x|e^{-\beta \hat{H}}|x\rangle$ directly because $\hat{K}$ and $\hat{V}$ do not commute. However, Trotter splitting [24] may be employed in order to separate the exponential term. A symmetric second order splitting gives:

$$Z = \lim_{P\to\infty} \int \mathrm{d}x \langle x| \left(e^{\frac{-\beta}{2P}\hat{V}} e^{\frac{-\beta}{P}\hat{K}} e^{\frac{-\beta}{2P}\hat{V}} \right)^P |x\rangle \ . \tag{1.51}$$

The operator $e^{\frac{-\beta}{2P}\hat{V}} e^{\frac{-\beta}{P}\hat{K}} e^{\frac{-\beta}{2P}\hat{V}}$ will be represented compactly by Φ. This factorized form can be further simplified by introducing a complete set of states $|x\rangle\langle x|$ as the identity I

$$I = \int \mathrm{d}x |x\rangle\langle x| \tag{1.52}$$

between each evaluation of Φ:

$$Z_P = \int \mathrm{d}x \langle x|(\Phi)^P|x\rangle \tag{1.53}$$

$$= \int \mathrm{d}x \langle x|(I\Phi)^P|x\rangle \tag{1.54}$$

$$= \int \mathrm{d}x_1 \ldots \mathrm{d}x_P \langle x_1|\Phi|x_2\rangle\langle x_2|\Phi|x_3\rangle \ldots \langle x_P|\Phi|x_1\rangle \tag{1.55}$$

$$= \int \mathrm{d}x_1 \ldots \mathrm{d}x_P \left(\prod_{i=1}^{P} \langle x_i|\Phi|x_{i+1}\rangle\right) \ , \tag{1.56}$$

where $x_{P+1} \equiv x_1$, as required by the conditions of the trace. Now, the expression for $\langle x_i|\Phi|x_{i+1}\rangle$ may be simplified further:

$$\langle x_i|\Phi|x_{i+1}\rangle = \langle x_i|e^{\frac{-\beta}{2P}\hat{V}} e^{\frac{-\beta}{P}\hat{K}} e^{\frac{-\beta}{2P}\hat{V}}|x_{i+1}\rangle \tag{1.57}$$

$$= e^{\frac{-\beta}{2P}V(x_i)} \langle x_i|e^{\frac{-\beta}{P}\hat{K}}|x_{i+1}\rangle e^{\frac{-\beta}{2P}V(x_{i+1})} \ . \tag{1.58}$$

In order to ascertain the value of $\langle x_i|e^{\frac{-\beta}{P}\hat{K}}|x_{i+1}\rangle$ another identity is used, this time in momentum space:

$$\langle x_i|e^{\frac{-\beta}{P}\hat{K}}|x_{i+1}\rangle = \int \mathrm{d}p \langle x_i|p\rangle\langle p|e^{\frac{-\beta}{P}\hat{K}}|x_{i+1}\rangle \ . \tag{1.59}$$

The (Hermitian) $e^{\frac{-\beta}{P}\hat{K}}$ operator can then act on its eigenstate to the left, yielding

$$\langle x_i|e^{\frac{-\beta}{P}\hat{K}}|x_{i+1}\rangle = \int \mathrm{d}p e^{\frac{-\beta p^2}{2mP}} \langle x_i|p\rangle\langle p|x_{i+1}\rangle \ . \tag{1.60}$$

By acknowledging that $\langle x|p\rangle = \frac{1}{\sqrt{2\pi\hbar}} e^{ipx/\hbar}$, the integral can be solved analytically:

$$\langle x_i|e^{\frac{-\beta}{P}\hat{K}}|x_{i+1}\rangle = \frac{1}{2\pi\hbar}\int \mathrm{d}p e^{\frac{-\beta p^2}{2mP}} e^{ip(x_i - x_{i+1})/\hbar} \tag{1.61}$$

$$= \sqrt{\frac{mP}{2\pi\beta\hbar^2}} e^{-\frac{mP}{2\beta\hbar^2}(x_i - x_{i+1})^2} \ . \tag{1.62}$$

The total expression for the elements of Φ is therefore

$$\langle x_i|\Phi|x_{i+1}\rangle = \sqrt{\frac{mP}{2\pi\beta\hbar^2}} e^{\frac{-\beta}{2P}V(x_i)} e^{-\frac{mP}{2\beta\hbar^2}(x_i - x_{i+1})^2} e^{\frac{-\beta}{2P}V(x_{i+1})} \ , \tag{1.63}$$

and by invoking the circularity of the trace, we arrive at the following expression for the partition function:

$$Z_P = \sqrt{\frac{mP}{2\pi\beta\hbar^2}} \int \mathrm{d}x_1 \ldots \mathrm{d}x_P \exp\left\{\sum_{i=1}^{P}\left[\frac{mP}{2\beta\hbar^2}\left(x_i - x_{i+1}\right)^2 + \frac{\beta}{P}V(x_i)\right]\right\} \ . \tag{1.64}$$

This expression is known as the *discretized path integral* form of the partition function, and it is exact in the limit of $P \to \infty$. The use of MD to perform path integration is referred to as Path Integral Molecular Dynamics (PIMD).

With the expression for Z_P given in Equation (1.64), consider the expectation value of property A as given in Equation (1.48). The expectation value within a discrete path

integral formalism is therefore

$$\langle A\rangle_P = \frac{1}{Z_P}\sqrt{\frac{mP}{2\pi\beta\hbar^2}}\int \mathrm{d}x_1 \ldots \mathrm{d}x_P \; A(x_1)\exp\left\{\sum_{i=1}^{P}\left[\frac{mP}{2\beta\hbar^2}\left(x_i - x_{i+1}\right)^2 + \frac{\beta}{P}V(x_i)\right]\right\} \; . \tag{1.65}$$

However, this integrand is unchanged under the cyclic relabelling of x_1 to x_2, x_2 to x_3, etc. Therefore, $\langle A\rangle$ can also be described as

$$\langle A\rangle_P = \frac{1}{Z_P}\sqrt{\frac{mP}{2\pi\beta\hbar^2}}\int \mathrm{d}x_1 \ldots \mathrm{d}x_P \frac{1}{P}\sum_{i=1}^{P} A(x_i)\exp\left\{\sum_{i=1}^{P}\left[\frac{mP}{2\beta\hbar^2}\left(x_i - x_{i+1}\right)^2 + \frac{\beta}{P}V(x_i)\right]\right\} \; . \tag{1.66}$$

The notation $\langle A\rangle_P$ has been adopted to represent the discretized average of A, with the exact value of $\langle A\rangle$ being achieved in the limit of $P \to \infty$. Note that the above expression is valid for a property A for which the corresponding operator is diagonal in the position representation. A closed path integral cannot be employed when dealing with operators that are not diagonal in the position representation, such as momentum.

In order to compute the value of $\langle A\rangle_P$, consider an angular frequency ω_P given by

$$\omega_P = \frac{\sqrt{P}}{\beta\hbar} \tag{1.67}$$

and an effective potential

$$V_{\text{eff}} = \sum_{i=1}^{P} \left[m\omega_P^2 \left(x_i - x_{i+1}\right)^2 + \frac{\beta}{P} V(x_i) \right] \tag{1.68}$$

Substituting this value into the expression for Z_P, it is found that

$$Z_P = \sqrt{\frac{mP}{2\pi\beta\hbar^2}} \int \mathrm{d}x_1 \ldots \mathrm{d}x_P \; e^{-\beta V_{\text{eff}}} \; , \tag{1.69}$$

which is of a similar form to the q component of the integrand of the classical ensemble average for a cyclical polymer of P identical particles. These fictional particles are commonly referred to as *beads*. A fictional momentum integral is introduced in order to complete the analogy:

$$\bar{Z}_P(p,q) = C\sqrt{\frac{mP}{2\pi\beta\hbar^2}} \int \mathrm{d}\bar{p}_1 \ldots \mathrm{d}\bar{p}_P \int \mathrm{d}q_1 \ldots \mathrm{d}q_P \exp\left\{ -\beta \left[\sum_{i=1}^{P} \frac{\bar{p}_i{}^2}{2\bar{m}} + V_{\text{eff}}(q_1 \ldots q_P) \right] \right\} \; . \tag{1.70}$$

Note that the momentum integral is completely decoupled from the position integral, and so $\bar{Z}_P$ can be normalized by dividing by the analytical result:

$$C = \left[\prod_{i=1}^{P} \left(\frac{2\pi\bar{m}}{\beta} \right)^{\frac{P}{2}} \right]^{-1} \; . \tag{1.71}$$

The integral in Equation (1.70) is exactly that of a classical partition function describing the canonical ensemble average of a cyclic polymer of P beads feeling a modified potential, and therefore MD can be employed to obtain averages of properties in this system. The fictional momenta $\bar{p}$ are related to a fictional mass $\bar{m}$ which is arbitrary and may be chosen to increase sampling efficiency of the MD simulation [25]. Averages of properties arising from the integration of Equation (1.70) must be taken carefully; specifically the momenta are completely fictitious and so the average total energy of the system $\langle H \rangle$ is meaningless. Instead, property estimators based on the form of Equation (1.66) must be employed to derive estimates from trajectories derived from this system. Consider the statistical mechanics expression for the average energy:

$$\langle E \rangle = \frac{\partial}{\partial \beta} \ln(Z) = \frac{1}{Z} \frac{\partial Z}{\partial \beta} \ . \tag{1.72}$$

By considering the partition function given in Equation (1.64) and taking the derivative, the following energy estimator is derived

$$E_P = \left\langle \frac{P}{2\beta} - \sum_{i=1}^{P} \left[\frac{mP}{2\beta^2 \hbar^2} (q_{i+1} - q_i)^2 - \frac{1}{P} V(q_i) \right] \right\rangle \ , \tag{1.73}$$

which is referred to as the primitive energy estimator. Other energy estimators exist, but will not be discussed here [26].

1.5 Computational Considerations

The integration of the equations of motion implied by Equation (1.70) poses some technical challenges. Specifically, the ω_P^2 term increases linearly with P, analogous to stiffening of the bonds of the ring polymer. This stiffening leads to inefficient phase space sampling (i.e. the system becomes non-ergodic)[27], and requires a small Δt in the integrator to capture the very rapid oscillations. Thus, any attempts to increase P beyond a certain threshold will be stymied. In order to overcome this issue, a change of variables to normal modes is introduced. The rationale for this choice is that it allows for analytic propagation of the inter-bead spring terms.

Consider the spring potential term, V_{spring}:

$$V_{\text{spring}} = \sum_{i=1}^{P} m\omega_P^2 \left(q_i - q_{i+1}\right)^2 = \sum_{i=1}^{P} \frac{mP}{\beta^2\hbar^2} \left(q_i - q_{i+1}\right)^2 \tag{1.74}$$

The second derivative or Hessian matrix of this potential is of the following form:

$$H(V_{\text{spring}}) = \begin{pmatrix} 2 & -1 & 0 & 0 & -1 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & \ddots & \ddots & \ddots \\ -1 & 0 & 0 & -1 & 2 \end{pmatrix} \tag{1.75}$$

which is known as a circulant matrix. A circulant matrix is diagonalizable by the unitary transform corresponding to the Discrete Fourier Transform (DFT) matrix of the same dimensions. The DFT matrix of size N is a Vandermonde matrix of the N^{th} root of unity:

$$\text{DFT}_N = \frac{1}{\sqrt{N}} \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{(N-1)} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{(N-1)} & \omega^{2(N-1)} & \cdots & \omega^{(N-1)(N-1)} \end{pmatrix} \tag{1.76}$$

with

$$\omega = e^{-2\pi i/N} \ . \tag{1.77}$$

The DFT matrix can be used as a unitary transformation to give $H(V_{\text{spring}})$ in new coordinates corresponding to Fourier normal modes:

$$\tilde{H}(V_{\text{spring}}) = \text{DFT}_P^\dagger H(V_{\text{spring}})\text{DFT}_P = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & f_1 & 0 & 0 & 0 \\ 0 & 0 & f_2 & 0 & 0 \\ 0 & 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & 0 & f_{P-1} \end{pmatrix} \tag{1.78}$$

where $\{f_i\}$ are a set of frequencies corresponding to the eigenvalues of $H(V_{\text{spring}})$. This transformed Hessian implies a transformed functional form of V_{spring}; in normal mode coordinates, the following form is achieved:

$$V_{\text{spring}} = \sum_{k=0}^{P-1} m\omega_k^2 \tilde{q}^2 \tag{1.79}$$

where $\tilde{q}$ represents the fourier transformed position, $\text{DFT}_P(q_1, \ldots, q_P)^\dagger$, and with

$$\omega_k = 2\omega_P \sin(k\pi/P) \ . \tag{1.80}$$

The form of Equation (1.80) is that of P independent harmonic oscillators. The equations of motion of the harmonic oscillator are known analytically in terms of sines and cosines,

so within this normal mode representation, the energies and forces resulting from the inter-bead spring terms may be evaluated analytically. Considering also the Fourier transformed momentum, $\tilde{p}$, the following relation may be used to propagate the spring terms:

$$\begin{pmatrix} \tilde{p}'_k \\ \tilde{q}'_k \end{pmatrix} = \begin{pmatrix} \cos(\omega_k \Delta t) & -m\omega_k \sin(\omega_k \Delta t) \\ (m\omega_k)^{-1} \sin(\omega_k \Delta t) & \cos(\omega_k \Delta t) \end{pmatrix} \begin{pmatrix} \tilde{p}_k \\ \tilde{q}_k \end{pmatrix} . \tag{1.81}$$

This analytic propagation does not fall victim to the ergodicity and timestep problems outlined previously. Furthermore, it is possible to apply the Langevin equation in normal mode coordinates, and the form of Equation (1.80) offers an analytical expression for a statistically optimal γ value for normal modes 1 though $P-1$. Considering the free ring polymer Hamiltonian

$$H_{P,\text{free}}(\tilde{p}, \tilde{q}) = \sum_{k=0}^{P-1} \left[\frac{\tilde{p}_k^2}{2m} + m\omega_k^2 \tilde{q}^2 \right] , \tag{1.82}$$

the autocorrelation function $\langle H_{P,\text{free}}(0) H_{P,\text{free}}(t) \rangle$ can be worked out analytically to give the following friction values:

$$\gamma_k = \begin{cases} 1/\tau_0 & \text{if } k = 0 \\ 2\omega_k & \text{if } k > 0 \end{cases} \tag{1.83}$$

with τ_0 as the correlation time of the zeroth normal mode (centroid), which cannot be determined analytically. Thus, the initial Langevin friction problem of supplying P friction

values to optimize sampling has been reduced to supplying 1 friction value, γ_c, the centroid friction value. The application of the Langevin equation thermostat to PIMD via normal mode analysis is referred to as the Path Integral Langevin Equation (PILE) [11] thermostat, and it offers an improvement in efficiency over the standard Langevin approach in PIMD, the White Noise Langevin Equation (WNLE) [28]. An integrator to solve the equations of motion arising from the PILE is as follows:

$$\tilde{p}_k(t) = \mathrm{DFT}_P \; p_j(t) \tag{1.84}$$

$$\tilde{p}_{k,f}(t) = c_{1,k}\tilde{p}_k(t) + \sqrt{\frac{mP}{\beta}} c_{2,k}\xi(t + \Delta t/2) \tag{1.85}$$

$$p_{j,f}(t) = \mathrm{DFT}_P^{\dagger} \; \tilde{p}_{k,f}(t) \tag{1.86}$$

$$p_{j,f}(t + \Delta t/2) = p_{j,f}(t) + \frac{\Delta t}{2}\nabla V(q_j(t)) \tag{1.87}$$

$$\tilde{p}_{k,f}(t + \Delta t/2) = \mathrm{DFT}_P \; p_{j,f}(t + \Delta t/2) \tag{1.88}$$

$$\tilde{q}_k(t) = \mathrm{DFT}_P \; q_j(t) \tag{1.89}$$

$$\begin{pmatrix} \tilde{p}'_{k,f}(t + \Delta t/2) \\ \\ \tilde{q}'_k(t) \end{pmatrix} = \begin{pmatrix} \cos(\omega_k \Delta t) & -m\omega_k \sin(\omega_k \Delta t) \\ \\ (m\omega_k)^{-1} \sin(\omega_k \Delta t) & \cos(\omega_k \Delta t) \end{pmatrix} \begin{pmatrix} \tilde{p}_{k,f}(t + \Delta t/2) \\ \\ \tilde{q}_k(t) \end{pmatrix} \tag{1.90}$$

$$p'_{j,f}(t + \Delta t/2) = \mathrm{DFT}_P^{\dagger} \; \tilde{p}'_{k,f}(t + \Delta t/2) \tag{1.91}$$

$$q_j(t + \Delta t)) = \mathrm{DFT}_P^{\dagger} \; \tilde{q}'_k(t) \tag{1.92}$$

$$p_{j,f}(t + \Delta t) = p'_{j,f}(t + \Delta t/2) + \frac{\Delta t}{2}\nabla V(q_j(t + \Delta t)) \tag{1.93}$$

$$\tilde{p}_{k,f}(t + \Delta t) = \mathrm{DFT}_P \; p_{j,f}(t + \Delta t) \tag{1.94}$$

$$\tilde{p}_k(t) = c_{1,k}\tilde{p}_{k,f}(t + \Delta t) + \sqrt{\frac{mP}{\beta}} c_{2,k}\xi(t + \Delta t/2) \tag{1.95}$$

$$p_j(t + \Delta t) = \mathrm{DFT}_P^{\dagger} \; \tilde{p}_k(t + \Delta t) \tag{1.96}$$

where $c_{1,k}$ and $c_{2,k}$ are mode-specific constants related to the Langevin equation:

$$c_{1,k} = e^{-(\Delta t/2)\gamma_k} \tag{1.97}$$

$$c_{2,k} = \sqrt{1 - c_{1,k}^2}\ . \tag{1.98}$$

The integration of the PILE equations of motion is not as complex as it seems, since performing the DFT is fast using FFT algorithms; the DFT can be performed in $O(n \log(n))$ time.

The PILE approach described above has recently been included in the MMTK using CPU based code [29]. The work reported here has allowed the addition of GPU computing to MMTK for both classical MD and PILE simulations. Another contribution of the present work is a generalization of the PILE concepts to ground state systems.

The remainder of this thesis is organized as follows: Chapter 2 contains a description of the addition of GPU computing capability to the MMTK code; the ground state generalization of the PILE approach is presented in Chapter 3 along with representative examples. Conclusions and future directions are finally presented in Chapter 4.

Chapter 2

MMTK & OpenMM Code

When applying the principles of MD, it is crucial that the execution of computer code be as fast as possible. With increased computer speed it is possible to decrease the numerical error of a simulation by using lower error thresholds and to decrease the statistical error by increasing the length of the trajectory, all while keeping the wall clock time fixed. Researchers in the field of High Performance Computing (HPC) seek to build computers and computer networks that can solve scientific problems as quickly as possible. Support for long MD simulations, particularly of large molecules such proteins, are popular targets of such endeavours. While many software packages for MD exist to facilitate simulation on a variety of hardware, this document focusses on the Molecular Modelling Tool Kit (MMTK) [30] Python scripting language. A wide range of chemistry related algorithms are

supported by MMTK, including MD and PIMD with arbitrary potential energy functions. One needs only to supply code for the potential as well as its derivative and MMTK can generate a trajectory using a variety of ensembles and integrators. This versatility makes MMTK a good tool for research and development of novel MD methods. Structurally, MMTK is composed of a Python interface that is exposed to the user, underpinned by C code that is used to rapidly evaluate potential energy functions and perform integration. This combination allows the use of highly optimized routines such as BLAS and LAPACK to solve MD problems, while still maintaining simplicity for the scientific user.

Arguably the most common approach to accelerating the performance of an MD simulation is through the use of *parallel computing*, which seeks to break down a large task into smaller independent subtasks that can be executed in isolation from each other. On a single processor machine parallel computing offers little advantage, but on a machine with many processors the speed of execution will scale with Amdahl's Law:

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}} \tag{2.1}$$

where P is the fraction of the code that can be executed in parallel, N is the number of processors available, and S is the speedup (i.e. the ratio of parallelized runtime to serial runtime). This equation describes a sigmoidal curve in N, with additional processors

offering a large speedup at first, and subsequently having diminishing returns. Thus, hardware manufacturers work to provide researchers with machines having more processors (greater N), while software vendors attempt to develop new parallel algorithms that provide better scaling (greater P) [31] or reduce the computation time entirely. Parallel computing is supported on MMTK with the C language energy evaluation code, but no other attempts are made at parallelism. A method of coding that greatly increases the effective N available to the system through novel use of hardware will be subsequently introduced.

2.1 GPU Computing

Graphics Processing Units (GPUs), informally called graphics cards, were originally intended to drive computer monitors so as to display data. With the growth of data-intensive visualization needs, including but not limited to the computer video game industry, GPUs began to take over computational duties normally performed by the Central Processing Unit (CPU). Due to the highly parallel nature of the task of rendering an image on a screen, GPU architecture naturally grew to include many programmable elements, called shaders. These shaders could be reprogrammed to perform arbitrary tasks, such as the evaluation of molecular mechanics forcefields, and the propagation of MD simulations [32]. With the increasing application of general purpose computation to GPUs (GPGPU pro-

gramming), GPU vendor Nvidia released a generation of GPUs with a focus on GPGPU called CUDA, along with a specialized CUDA language based on C. Nvidia "C for CUDA" is a C-like language that facilitates the production of GPU bytecode for Nvidia GPUs, allowing application developers to leverage GPU functionality without investing the time required to learn the specifics of GPU hardware and a GPU-specific shader language. The development of CUDA was followed by OpenCL [33], an open standard for heterogenous computing including GPUs from all major vendors and CPUs.

The types of problems that are most suited for a GPU approach are those which exhibit some form of parallelism. While GPUs can offer massive speed advantages for parallel executable code, traditional CPUs still reign in the single-threaded serial execution regime. In traditional CPU-based code execution, data is fetched from some long-term storage (HDD, CD-ROM, etc.) and stored in RAM. The CPU then requests data from RAM and stores it in the CPU cache, where it is operated upon. The CPU has a dedicated access channel to the RAM so this process is very rapid, and the presence of a CPU cache means that data only occasionally needs to be written back to RAM. A schematic of the flow of data for CPU execution is provided in Figure 2.2. In GPU execution, data that is initially stored in RAM must be transferred to the GPU over the motherboard's bus. This process is slow relative to CPU RAM access because the motherboard bus is low-bandwidth and shared by all peripherals. Once the data has reached the GPU, it must be stored in the

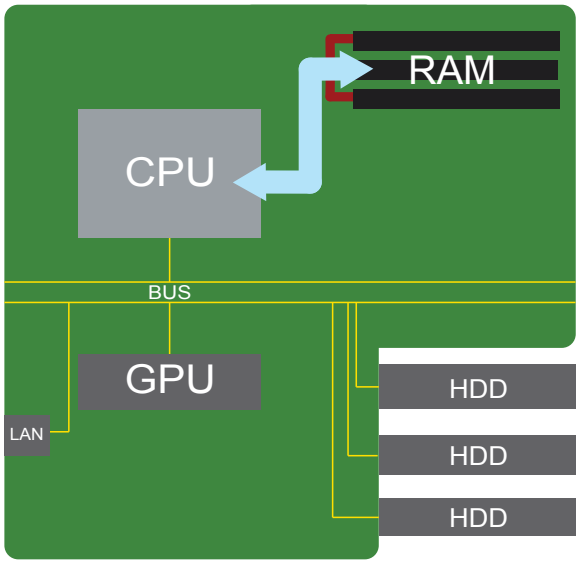

Figure 2.1: A schematic of the flow of data in CPU-oriented computing. The CPU has a dedicated, high-bandwidth connection to main memory, which can be rapidly accessed (blue arrows). Due to the high speed of the CPUs and fast memory access, this is an example of a latency oriented architecture.

GPU's Video RAM (VRAM) in order to be operated on by the GPU's processors. Once the data has been loaded into VRAM, execution by the GPU processors is performed rapidly; a modern GPU may have upwards of 3000 processor cores operating at once. The processed data is then transferred back over the bus into RAM where it can be accessed by the CPU for analysis. A schematic of the flow of data in GPU code execution is provided in Figure 2.2.

To rapidly implement GPU features in the existing MMTK codebase, the Open Molecular Modelling (OpenMM) [34] library was employed. The goal of the OpenMM project is

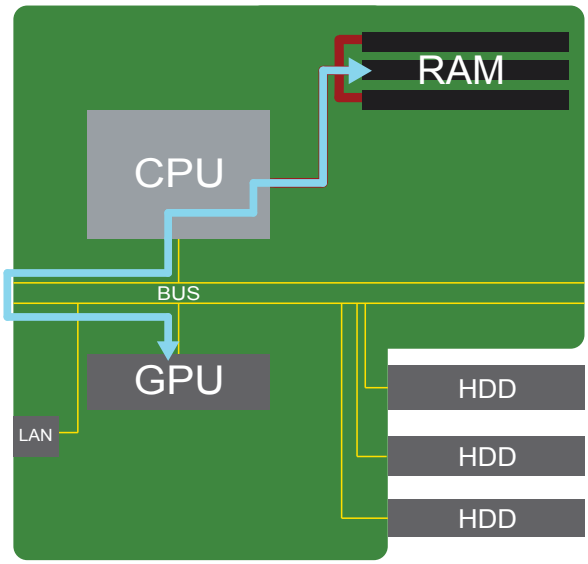

Figure 2.2: A schematic of the flow of data in GPU-oriented computing. The transfer bandwidth between the CPU and the GPU is limited, because data must traverse the motherboard bus (blue arrows). Once the data has reached the GPU it can be rapidly processed, so this is an example of a throughput oriented architecture.

to provide a universal interface to common molecular modelling routines (such as common force fields and integrators) abstracted away from an underlying high performance implementation. In general, OpenMM is geared towards providing a computational backend to developers of third-party chemistry programs, and not to end users. Currently, OpenMM supports a CPU-based *Reference* implementation, considered the gold standard for numerical accuracy and code readability, a cross-platform *OpenCL* implementation that supports a variety of GPUs and CPUs, and a *CUDA* implementation that is specific to Nvidia GPUs but provides maximum performance. In OpenMM parlance, these individual implementations are called `Platform`s. In order to run a simulation, a `System` object is created that describes the geometry of the subject system, and `Force` objects are created to describe the forces acting on the system. Also required is an `Integrator` object describing the type of integrator to be utilized. The OpenMM runtime then selects the fastest available `Platform` on which to run the simulation, and a `Context` is returned that represents the implementation of the requested `System` on that `Platform`. Stepping through the simulation via the `Integrator.step()` method then modifies the content of the `Context`, which can be queried for information in the form of a `State` object, containing information such as positions, velocities, and energies.

2.2 Software Architecture

The MMTK-OpenMM software is implemented via the MMTK Python to C language bridge and the OpenMM C to C++/OpenCL/CUDA bridge. Essentially, the low-level C code responsible for evaluating the potential and performing integration has all been replaced with calls to the OpenMM C API. The MMTK software is a subclass of `Dynamics.Integrator`, which allows the user to treat it like any other MMTK integrator. The Python level code is responsible for interrogating the contents of `Universe.energyEvaluatorParameters()` to tease out the various components of the user's selected force field. These force field parameters are gathered up and sent into the lower level C code, where the equivalent OpenMM `Force` objects are constructed. The Python code also records the number of path integral beads requested, and calculates the maximum number of integration steps that can be skipped between observations of the system from user supplied keywords. These parameters are passed into C integrator code, where an OpenMM `System` class is prepared that mimics the MMTK `Universe` description. This includes extracting information about periodic boundary conditions. During integration, OpenMM's `step` routine is called repeatedly in order to propagate the system in time, and `getState` is used to extract desired information (coordinates, velocities, energies) from OpenMM and report them via the standard MMTK `PyTrajectory_Output` method. Thus, the input is

entirely native MMTK code, and the output is presented as though MMTK code was run. The presence of OpenMM is entirely transparent, the only observed effect is the speedup of the simulation.

There are a few noteworthy shortcomings of the implementation. Firstly, because the overhead of transferring data to and from the GPU is considerable, the functionality of the code is restricted to the set of features supported by OpenMM. For example: MMTK supports removing centre of mass rotation from a simulation, but because OpenMM does not, this feature is silently ignored. Furthermore, the code is designed to support the AMBER force field and its derivatives; any custom forcefields must provide their own OpenMM implementation, which requires modification to the source code. Lastly, PIMD is currently not supported under CUDA which is the fastest available `Platform` in OpenMM. In order to facilitate the inclusion of PIMD support via OpenMM, custom code was written to create C language bindings to the `RPMDIntegrator` class that do not exist in OpenMM proper. A full listing of code is included in Appendix A.

2.3 Results (Timings and Accuracy)

To benchmark the new code, the radial distribution function for q-SPC/Fw water [35] was calculated in MMTK by a variety of codes. Briefly, the radial distribution function $g(r)$

[36] measures the probability of finding some atom A a given distance from another atom B. It is computed by recording all interparticle distances between A and B over the course of a trajectory and binning them. Classically, there is a single $g(r)$ defined for a system, but in PIMD there is both a centroid and quantum $g(r)$. The centroid g(r) is computed by first calculating the centroid of each path and treating the centroid as a classical particle, while the quantum $g(r)$ is computed by averaging the $g(r)$ for each bead.

For the classical distribution function, the default MMTK `LangevinIntegrator` was used, running with both 1 thread and 12 threads. The same distribution was prepared using the OpenMM-MMTK code with the Reference, OpenCL, and CUDA platforms. The relative speeds are plotted in Figure 2.3, and the results in Figures 2.4, 2.5, and 2.6.

The centroid and quantum g(r) were prepared in MMTK using the `PILangevinNormal ModeIntegrator` and in MMTK-OpenMM with the Reference and OpenCL platforms. Both codes are based around the PILE integrator. The relative speeds are plotted in Figure 2.7, and the results in Figures 2.8, 2.9, and 2.10.

Due to the extremely long runtimes at higher numbers of beads, no additional $g(r)$ data is included, but an estimate of the runtimes for 16 beads is provided in Figure 2.11

Reference values for the $g(r)$ are included in Figure 2.12.

From the results of the runtime benchmarking provided in Figures 2.3, 2.7, and 2.11, it

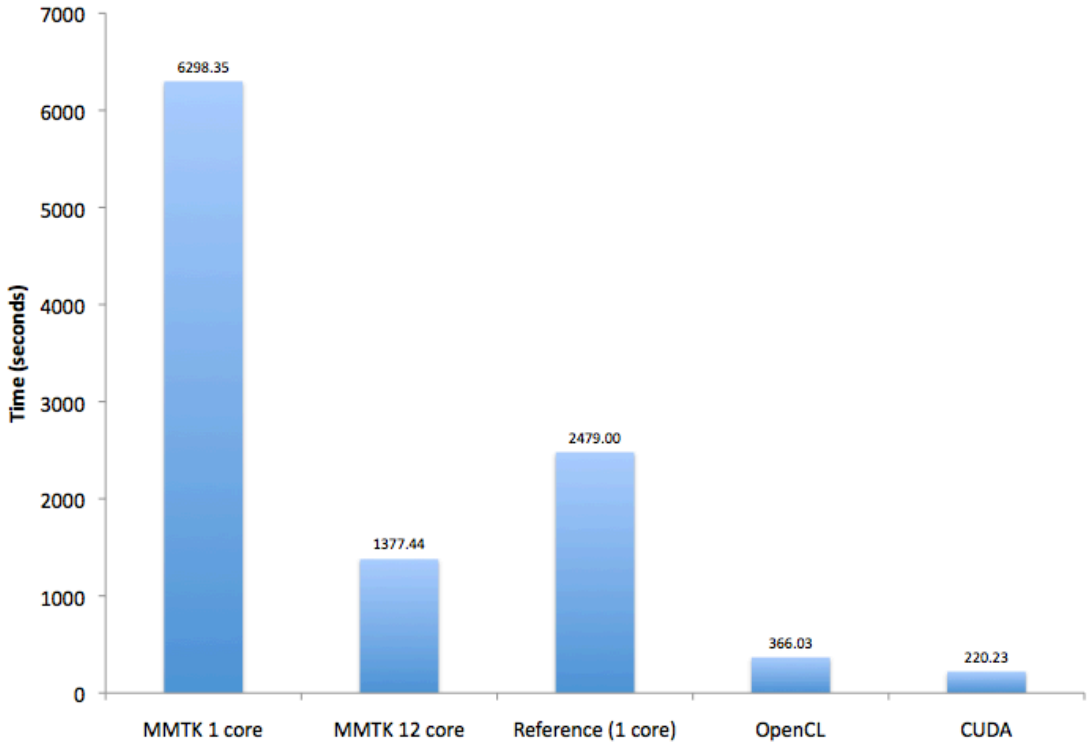

Figure 2.3: The absolute runtimes of a classical MD simulation using multiple codebases. The system consisted of 69 q-SPC/Fw waters in a periodic box measuring 12 x 12 x 12 angstroms. The total simulation time was 50ps, the timestep was 0.1fs, and values were reported every 0.1ps.

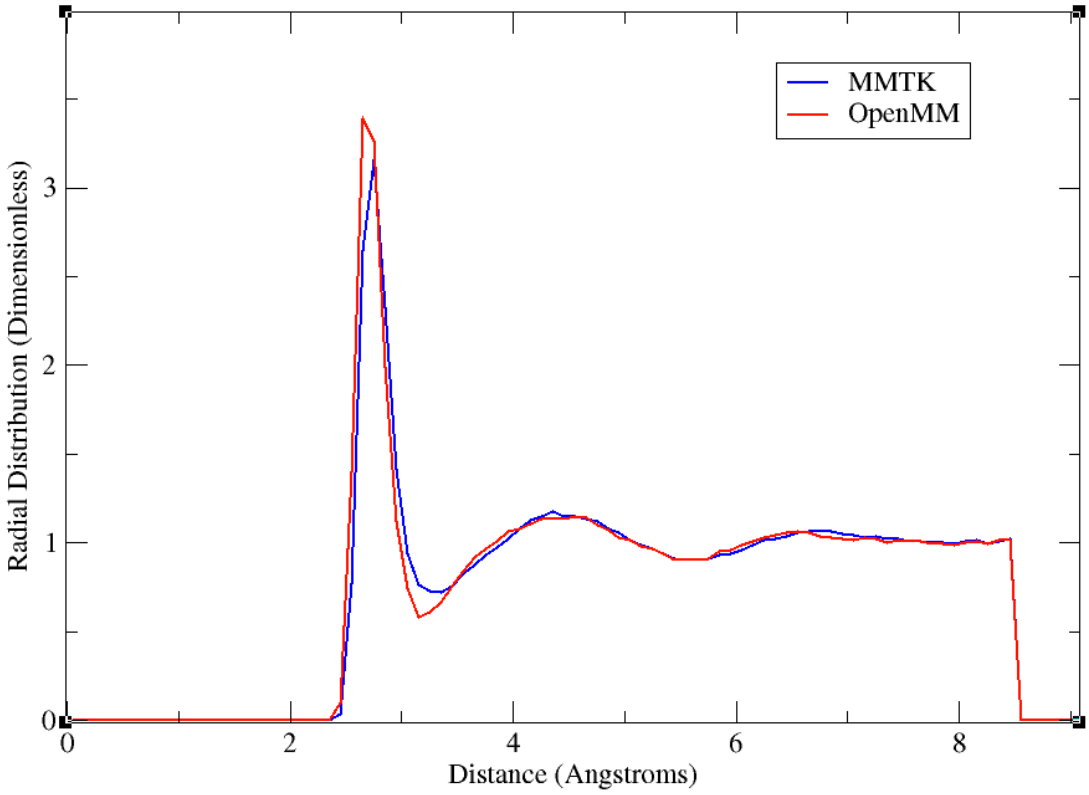

Figure 2.4: The values of the O-O $g(r)$ for q-SPC/Fw water in the classical (one bead) limit, as predicted by the different codebases.

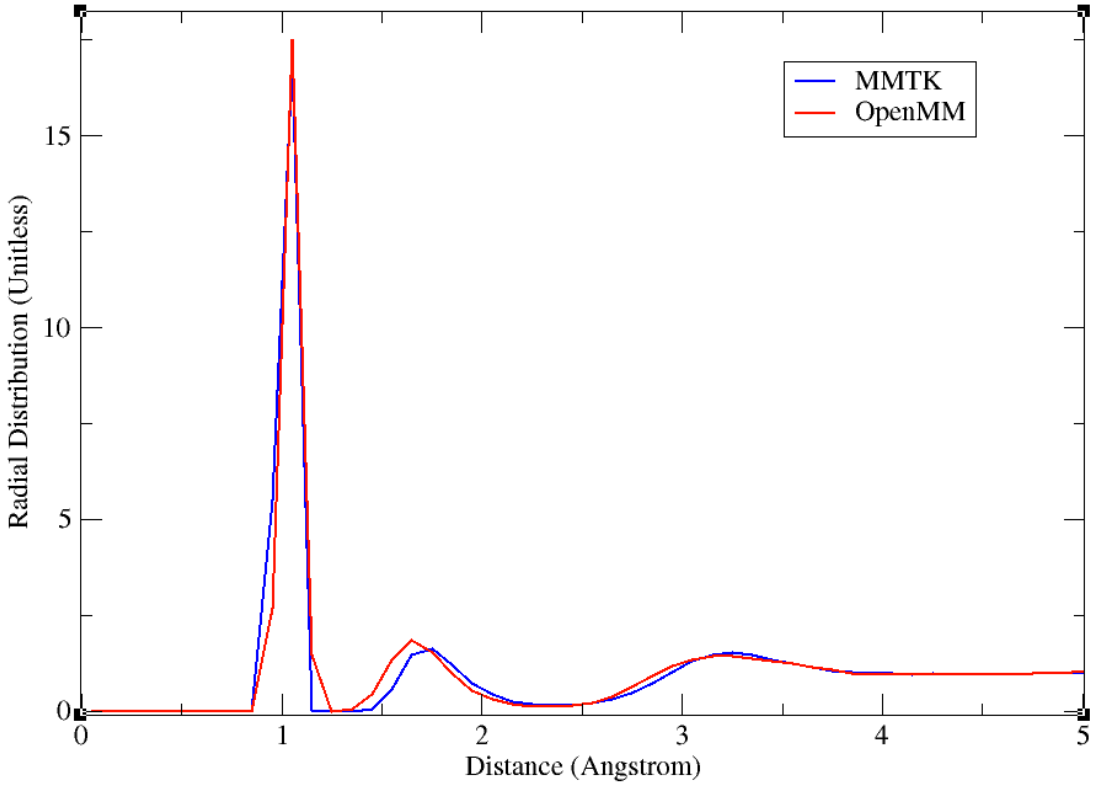

Figure 2.5: The values of the O-H $g(r)$ for q-SPC/Fw water in the classical (one bead) limit, as predicted by the different codebases.

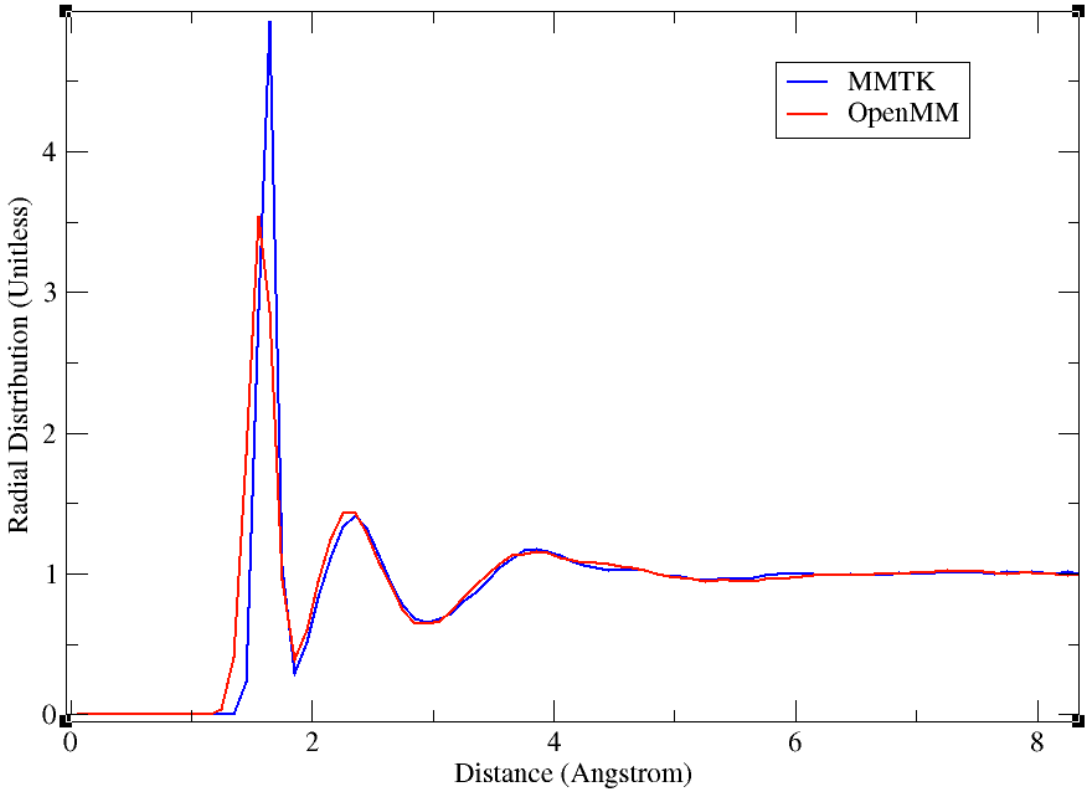

Figure 2.6: The values of the H-H $g(r)$ for q-SPC/Fw water in the classical (one bead) limit, as predicted by the different codebases.

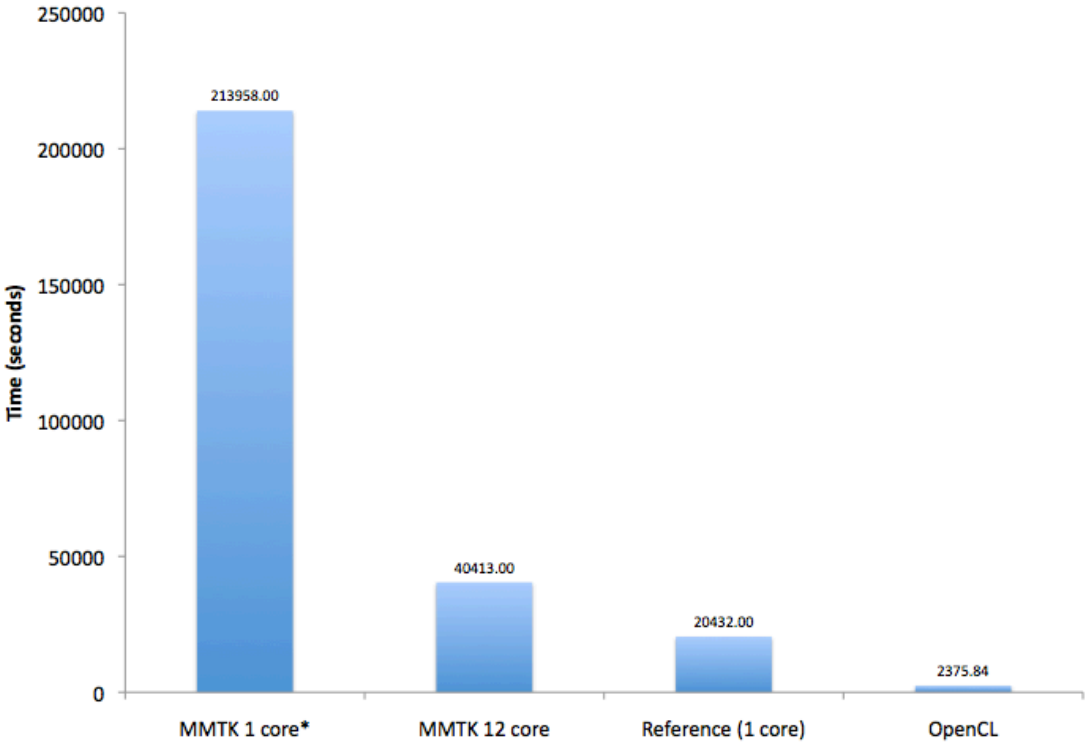

Figure 2.7: The absolute runtimes of a PIMD simulation of 8 beads using multiple codebases. The system consisted of 69 q-SPC/Fw waters in a periodic box measuring 12 x 12 x 12 angstroms. The total simulation time was 50ps, the timestep was 0.1fs, and values were reported every 0.1ps. The results for single core MMTK were extrapolated from a 10ps simulation due to the extremely slow runtime.

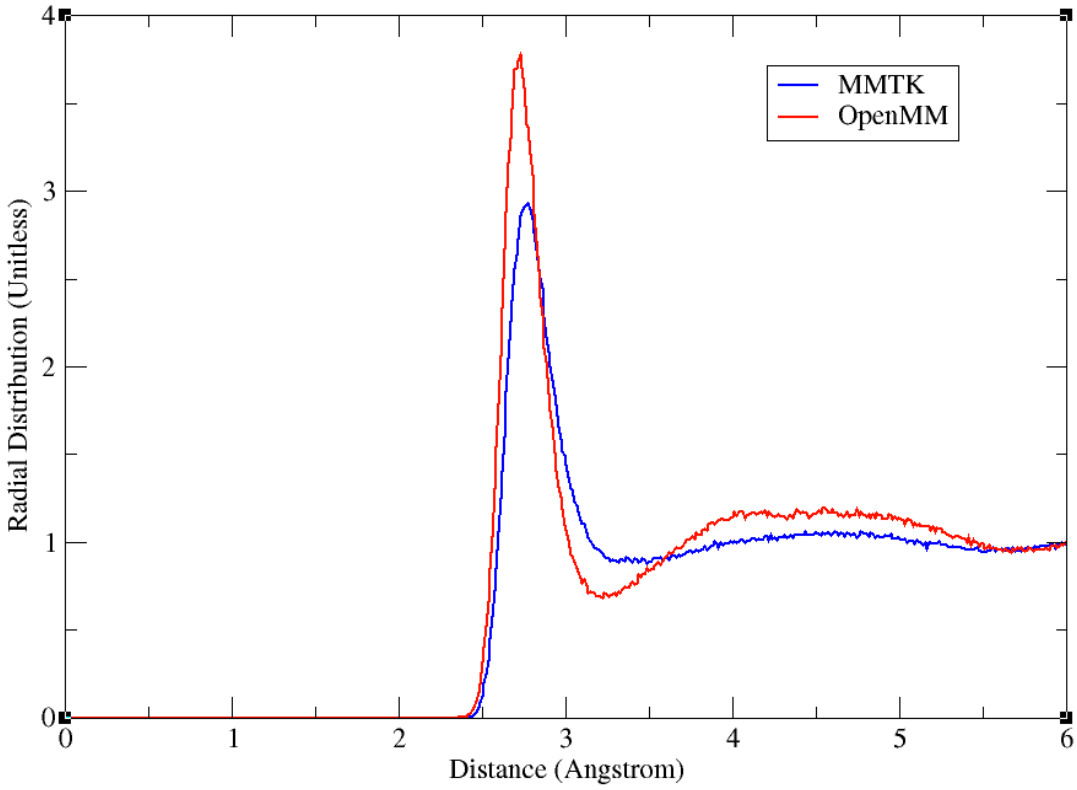

Figure 2.8: The values of the O-O $g(r)$ for q-SPC/Fw water with 8 beads, as predicted by the different codebases.

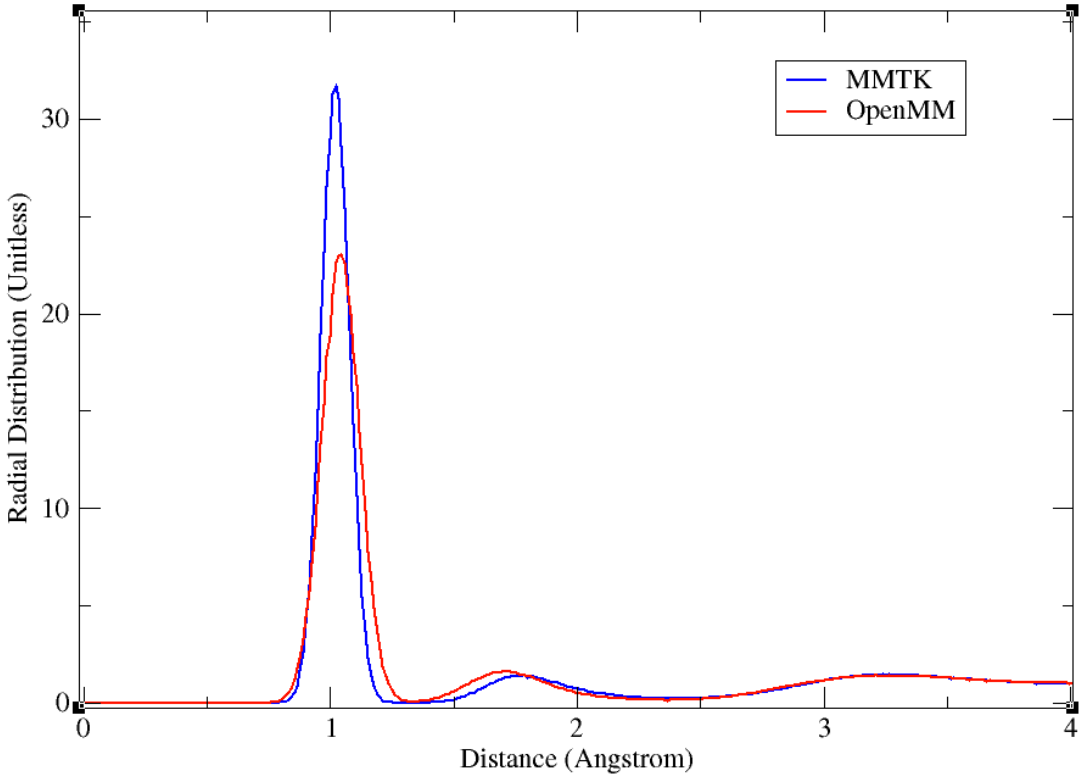

Figure 2.9: The values of the O-H $g(r)$ for q-SPC/Fw water with 8 beads, as predicted by the different codebases.

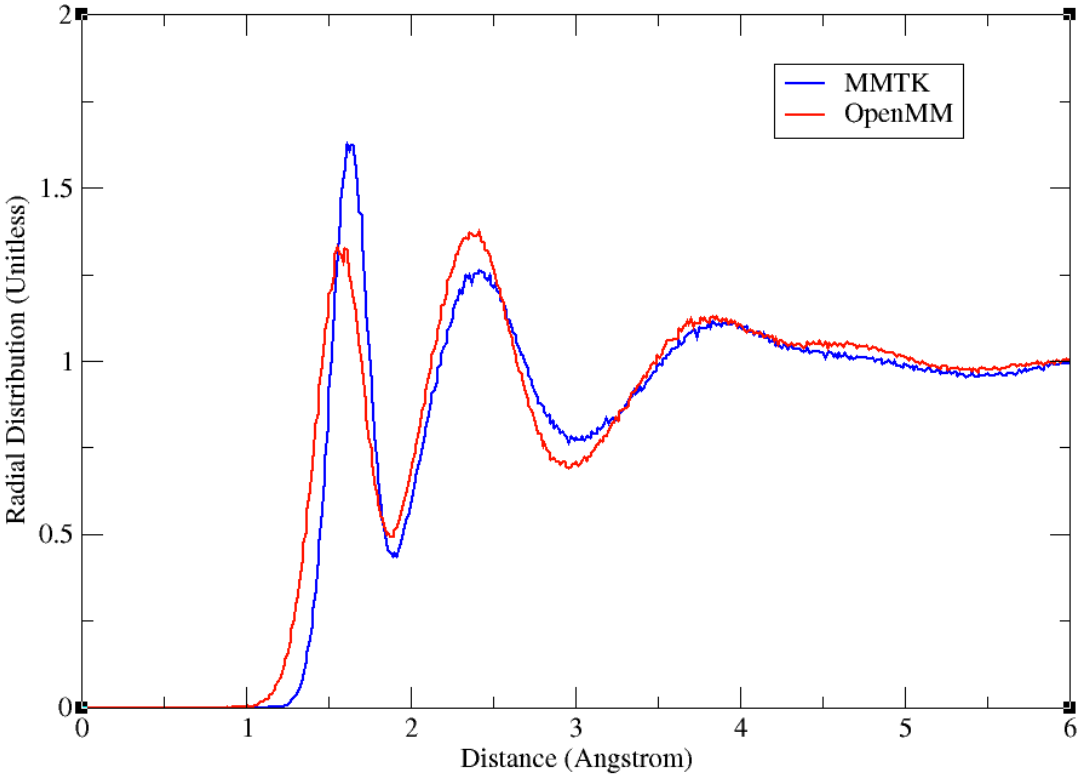

Figure 2.10: The values of the H-H $g(r)$ for q-SPC/Fw water with 8 beads, as predicted by the different codebases.

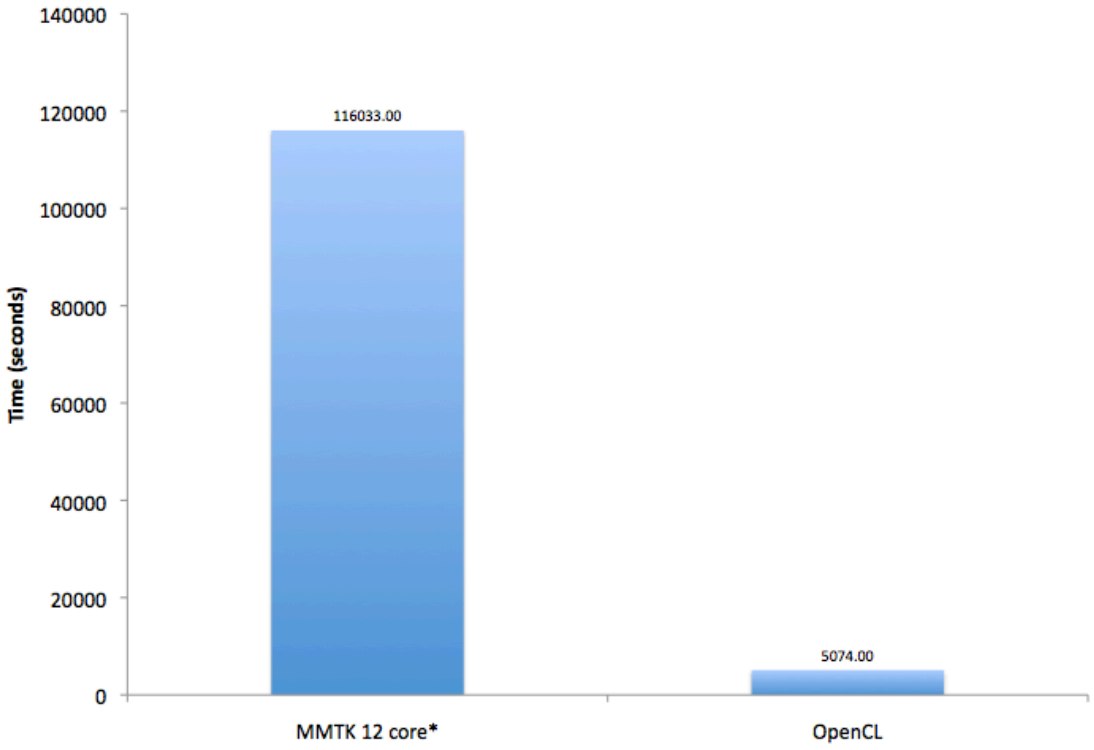

Figure 2.11: The absolute runtimes of a PIMD simulation of 16 beads using multiple codebases. The system consisted of 69 q-SPC/Fw waters in a periodic box measuring 12 x 12 x 12 angstroms. For OpenCL, the total simulation time was 50ps, and for MMTK the 50ps runtime was estimated from at 12ps run. The timestep was 0.1fs, and values were reported every 0.1ps.

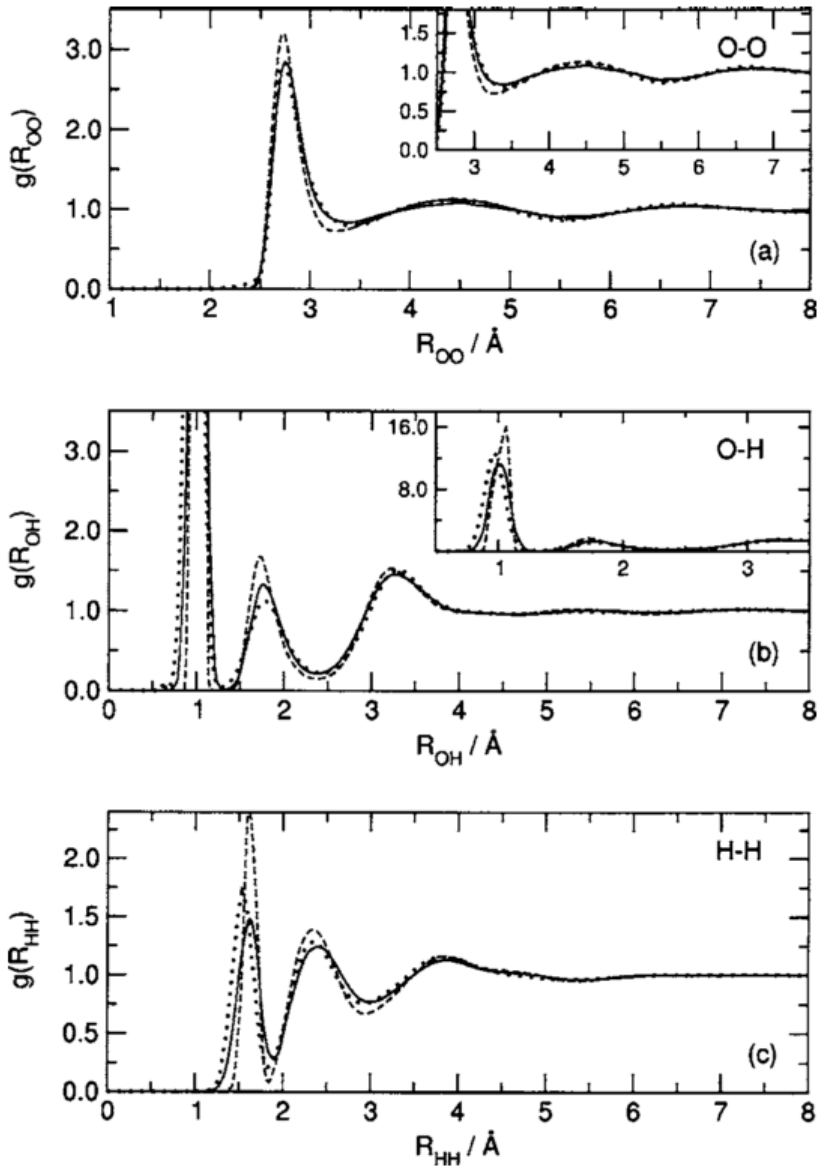

Figure 2.12: Expected curves of $g(r)$ of q-SPC/Fw water for O-O (a), O-H (b), and H-H (c) distributions. Inset figures are at different magnifications. In all cases, the solid line is PIMD, the dashed line is classical MD, and the dotted line is experimental data. Figure from [35].

is obvious that the inclusion of GPU support in MMTK affords a massive gain in computational power. At larger values of P even fully multithreaded CPU code pales in comparison to the performance offered by the OpenCL GPU code. Overall, the $g(r)$ values measured are similar but there are some slight differences, especially in the case of PIMD. This could perhaps be attributed to slight implementation differences, for example, the way that long-range force cutoffs are handled. Alternatively, the OpenMM GPU code can only run in single precision, so numeric effects could be responsible for the discrepancy. It should be noted that the underlying OpenMM PIMD code is beta quality, the release quality code has yet to be tested. Further research is required to ascertain the source of this error.

Chapter 3

LE-PIGS theory

In this section, the Path Integral Ground State (PIGS) [37] MD method for determination of the nuclear ground state density is introduced. It is demonstrated that through a Discrete Cosine Transform (DCT) rather than the DFT, the PIGS MD problem can be reformulated in terms of normal modes. These normal modes can then be propagated using the PILE to circumvent inherent ergodicity problems in the high-P limit. This novel combination of the PILE and PIGS method is referred to as the Langevin Equation Path Integral Ground State (LE-PIGS) method.

3.1 Nuclear Ground States

One system of interest is ^{4}He [38] [39] [40] [41]. This isotope of Helium is known to form an exotic phase of matter at extremely low temperatures, known as a superfluid. Superfluid ^{4}He displays a variety of unique properties, such as zero viscosity flow and the ability to flow against gravity out of an unsealed container via interactions with the container walls.

The superfluid phase is thought to consist partially of a Bose-Einstein condensate of ^{4}He. Bosons do not obey the Pauli exclusion principle; many particles are allowed to exist in the same quantum state. A Bose-Einstein condensate is a collections of bosons that have all settled into the same quantum state: the ground state. Clearly, to understand the behaviour of superfluid ^{4}He requires knowledge of its ground state. In order to probe the nuclear ground state properties of a bosonic system, consider the following state vector:

$$\Phi(x) = \lim_{\beta\to\infty} \int \mathrm{d}x' \langle x|e^{-\frac{\beta}{2}\hat{H}}|x'\rangle \psi_T(x') \tag{3.1}$$

for some trial wavefunction $|\psi_T\rangle$. The complete set of energy eigenstates of the system,

$$\sum_{n=0}^{\infty} |n\rangle\langle n| = 1 \ , \tag{3.2}$$

can be inserted to yield

$$\Phi(x) = \lim_{\beta\to\infty} \int \mathrm{d}x' \langle x| \sum_{n=0}^{\infty} e^{-\frac{\beta}{2}E_n} |n\rangle\langle n|x'\rangle \psi_T(x') \ , \tag{3.3}$$

where the relation

$$e^{-\frac{\beta}{2}\hat{H}}|n\rangle = e^{-\frac{\beta}{2}E_n}|n\rangle \tag{3.4}$$

has been used. Consider the term

$$\lim_{\beta\to\infty} \sum_{n=0}^{\infty} e^{-\frac{\beta}{2}E_n} \ . \tag{3.5}$$

Equation (3.5) can be written in terms of a Chebyshev distance:

$$D^{\beta} = \lim_{\beta\to\infty} \left(\sum_{n=0}^{\infty} \left[e^{-\frac{E_n}{2}} \right]^{\beta} \right)^{\frac{1}{\beta}\beta} \tag{3.6}$$

which reduces to

$$D^{\beta} = \max \left\{ e^{-\frac{E_n}{2}} \right\}^{\beta} \ . \tag{3.7}$$

The ground state energy E_0 is by definition the smallest value of E that can be measured, and therefore

$$D^{\beta} = \left(e^{-\frac{E_0}{2}} \right)^{\beta} = e^{-\frac{\beta}{2}E_0} \ . \tag{3.8}$$

By this reasoning, Equation (3.3) simplifies to

$$\Phi(x) = \int \mathrm{d}x' \langle x|e^{-\frac{\beta}{2}E_0}|n_0\rangle\langle n_0|x'\rangle\psi_T(x') \ . \tag{3.9}$$

Terms that do not depend on x' are constant with respect to integration, can can be factored out of the integrand:

$$\Phi(x) = e^{-\frac{\beta}{2}E_0}\langle x|n_0\rangle \int \mathrm{d}x' \langle n_0|x'\rangle\psi_T(x') \tag{3.10}$$

$$\Phi(x) = e^{-\frac{\beta}{2}E_0} n_0(x) \int \mathrm{d}x' n_0^\dagger(x')\psi_T(x') \tag{3.11}$$

$$\Phi(x) = e^{-\frac{\beta}{2}E_0} n_0(x)\langle n_0|\psi_T\rangle \tag{3.12}$$

$$\Phi(x) = ce^{-\frac{\beta}{2}E_0} n_0(x) \ . \tag{3.13}$$

As long as the overlap of the chosen trial function and the ground state, $\langle n_0|\Psi_T\rangle$, is nonzero, $\Phi(x)$ will always be proportional to the exact energy eigenfunction ground state. An expression for the average of some property A in the ground state is therefore:

$$\langle A\rangle_0 = \lim_{\beta\to\infty} \frac{\langle\Phi|\hat{A}|\Phi\rangle}{\langle\Phi|\Phi\rangle} = \lim_{\beta\to\infty} \frac{1}{Z_0}\langle\Phi|\hat{A}|\Phi\rangle \ . \tag{3.14}$$

The number Z_0 is a pseudo partition function given by

$$Z_0 = \int \mathrm{d}x \int \mathrm{d}x' \psi_T(x) \langle x | e^{-\beta \hat{H}} | x' \rangle \psi_T(x') \ . \tag{3.15}$$

Observant readers will note that this expression is similar to the partition function in PIMD, and can be solved via Trotter factorization [42]. While higher-order factorizations are possible [43], a second order factorization is employed here:

$$Z_0 = \lim_{P \to \infty} \int \mathrm{d}x_1 \ldots \int \mathrm{d}x_{P-1} \int \mathrm{d}x' \psi_T(x_1) \prod_{i=1}^{P-1} \left[\langle x_i | \rho | x_{i+1} \rangle \right] \langle x_{P-1} | \rho | x' \rangle \psi_T(x') \ , \tag{3.16}$$

with

$$\rho = e^{\frac{-\beta}{2(P-1)} \hat{V}} e^{\frac{-\beta}{(P-1)} \hat{K}} e^{\frac{-\beta}{2(P-1)} \hat{V}} \ . \tag{3.17}$$

The notation of splitting $P-1$ times is merely a convenient notation, since the $\langle x_{P-1} | \rho | x' \rangle$ term contributes an extra density matrix there will be P terms total. Therefore, x' will be relabelled as x_P. Furthermore, $\tau = \beta/(P-1)$ will be introduced as a shorthand notation. The matrix elements can be evaluated analytically as in PIMD to yield

$$Z_0 = \lim_{P \to \infty} \int \mathrm{d}x_1 \ldots \int \mathrm{d}x_P \psi_T(x_1) \exp \left\{ -\tau \left[\sum_{i=1}^{P-1} [m \omega_{P-1}^2 (x_i - x_{i+1})^2] + \sum_{i=1}^{P} [c_i V(x_i)] \right] \right\} \psi_T(x_P) \tag{3.18}$$

where $\omega_{P-1} = 1/\tau\hbar$ and c_i is a scaling factor that acts on the potential:

$$c_i = \begin{cases} \frac{1}{2} & \text{if } i = 1 \text{ or } i = P \\ 1 & \text{otherwise.} \end{cases} \tag{3.19}$$

Note that for a ground state (nodeless) trial wavefunction it is always true that

$$\psi_T(x) = \exp\{\ln[\psi_T(x)]\} \ . \tag{3.20}$$

Now a new potential, V_i' can be introduce to simplify Equation (3.18):

$$V_i'(x_i) = \begin{cases} \frac{1}{2}V(x_i) - \frac{1}{\tau}\ln[\psi_T(x_i)] & \text{if } i = 1 \text{ or } i = P \\ V(x_i) & \text{otherwise.} \end{cases} \tag{3.21}$$

Substituting V_i' into Equation (3.18) gives

$$Z_0 = \lim_{P\to\infty} \int \mathrm{d}x_1 \ldots \int \mathrm{d}x_P \exp\left\{-\tau\left[\sum_{i=1}^{P-1}[m\omega_{P-1}^2(x_i - x_{i+1})^2] + \sum_{i=1}^{P}[V_i'(x_i)]\right]\right\} \tag{3.22}$$

which can be sampled in a variety of ways. Popular approaches include Diffusion Monte Carlo (DMC) [44], Green's Function Monte Carlo (GFMC) [45] [46], and Path Integral Ground State Monte Carlo (PIGSMC) [37] [47] [48] [49]. Equation (3.22) can also be

sampled via MD by introducing fictitious momenta, analogous to PIMD. In PIMD there is a circular polymer of beads, but Equation (3.22) describes an open chain polymer with an extra force on the first and last beads. Furthermore, this integral is not invariant under cyclical renaming of $\{x_i\}$, so properties cannot be taken as an average over all beads as in PIMD. Consider the expanded expression for $\langle A\rangle_0$:

$$\langle A\rangle_0 = \lim_{\beta\to\infty} \frac{1}{Z_0} \int \mathrm{d}x \int \mathrm{d}x' \psi_T(x) \langle x| e^{-\frac{\beta}{2}\hat{H}} \hat{A} e^{-\frac{\beta}{2}\hat{H}} |x'\rangle \psi_T(x') \ . \tag{3.23}$$

Each of the two density operators may be Trotter factorized $(P-1)/2$ times to give

$$\langle A\rangle_0 = \lim_{\beta\to\infty} \lim_{P\to\infty} \frac{1}{Z_0} \int \mathrm{d}x \int \mathrm{d}x' \psi_T(x) \langle x| \rho_{\frac{1}{2}}^{\frac{(P-1)}{2}} \hat{A} \rho_{\frac{1}{2}}^{\frac{(P-1)}{2}} |x'\rangle \psi_T(x') \tag{3.24}$$

with

$$\rho_{\frac{1}{2}} = e^{\frac{-2\beta}{4(P-1)}\hat{V}} e^{\frac{-2\beta}{2(P-1)}\hat{K}} e^{\frac{-2\beta}{4(P-1)}\hat{V}} = \rho \ . \tag{3.25}$$

A total of $P-1$ sets of states may be inserted to give

$$\langle A\rangle_0 = \lim_{\beta,P\to\infty} \frac{1}{Z_0} \int \mathrm{d}x_1 \ldots \int \mathrm{d}x_P \psi_T(x_1) \langle x_1| \ldots |\rho| x_{\frac{P+1}{2}}\rangle \hat{A} \langle x_{\frac{P+1}{2}}| \rho| \ldots |x_P\rangle \psi_T(x_P) \ , \tag{3.26}$$

where x' has been relabelled to x_P. Note that this formula requires the existence of $x_{\frac{P+1}{2}}$,

i.e. P must be odd. The matrix elements can be solved to yield:

$$\langle A\rangle_0 = \lim_{\beta,P\to\infty} \frac{1}{Z_0} \int \mathrm{d}x_1 \ldots \int \mathrm{d}x_P A(x_{\frac{P+1}{2}}) \exp\left\{ -\tau \left[\sum_{i=1}^{P-1} [m\omega_{P-1}^2 (x_i - x_{i+1})^2] + \sum_{i=1}^{P} [V_i'(x_i)] \right] \right\}, \tag{3.27}$$

and so averages of properties in the ground state are measured at the middle bead. This is referred to as the *primitive estimator* of A in PIGS. Note that for $A = 1$ the expression for the partition function Z_0 is recovered since $e^{\frac{-\beta}{2}\hat{H}} e^{\frac{-\beta}{2}\hat{H}} = e^{-\beta\hat{H}}$, and so $\langle 1\rangle = 1$ as expected.

For properties that commute with the density operator ρ, such as $\hat{H}$, it is possible to derive a *mixed estimator*. This estimator involves moving the position of $\hat{H}$ in the integrand all the way to the right-hand side, so that it acts on ψ_T:

$$\langle E\rangle_0 = \lim_{\beta,P\to\infty} \frac{1}{Z_0} \int \mathrm{d}x_1 \int \mathrm{d}x_P \psi_T(x_1) \prod_{i=1}^{P-1} \langle x_i|\rho|x_{i+1}\rangle \hat{H}\psi_T(x_P) \ . \tag{3.28}$$

This can be compactly represented as:

$$\langle E_0\rangle_{\text{mixed}} = \left\langle \frac{\hat{H}\psi_T(x_P)}{\psi_T(x_P)} \right\rangle \ . \tag{3.29}$$

which, since $\hat{H}$ is Hermitian, is symmetrically equivalent to

$$\langle E_0\rangle_{\text{mixed}} = \left\langle \frac{\hat{H}\psi_T(x_1)}{\psi_T(x_1)} \right\rangle \ . \tag{3.30}$$

In practice, the average of the x_P and x_1 values can be used in order to provide enhanced sampling.

3.2 Derivation of LE-PIGS Equations

The total Hamiltonian for a classical system that will sample the path integral of Equation (3.27) is given thusly:

$$H(\bar{p}, q) = \sum_{i=1}^{P} \frac{\bar{p}_i^2}{2\bar{m}} + \sum_{i=1}^{P-1} [m\omega_{P-1}^2 (x_i - x_{i+1})^2] + \sum_{i=1}^{P} [V_i'(x_i)] \ , \tag{3.31}$$

which is sampled at the reduced inverse temperature $\tau = \beta/(P-1)$. The $m\omega_{P-1}^2$ prefactor of the inter-bead spring terms suffers from similar scaling issues as the analogous term in PIMD; an increase of P toward the limit results in a stiffening of the spring terms and non-ergodic behaviour. Consider the PIGS inter-bead spring term:

$$V_{\text{spring}}' = \sum_{i=1}^{P-1} \frac{1}{2} m\omega_{P-1}^2 (x_i - x_{i+1})^2 \ . \tag{3.32}$$

Unlike PIMD, there is no circularity condition. The Hessian matrix of this potential is

$$H(V'_{\text{spring}}) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.33}$$

A matrix of this form can be diagonalized by the Discrete Cosine Transform (DCT) [50] [51] matrix. The DCT has the following form:

$$\text{DCT}_N = C_k \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \cos\left(\frac{\pi(1/2)}{N}\right) & \cos\left(\frac{\pi(3/2)}{N}\right) & \dots & \cos\left(\frac{\pi(i-1/2)}{N}\right) \\ 1 & \cos\left(\frac{\pi(1/2)2}{N}\right) & \cos\left(\frac{\pi(3/2)2}{N}\right) & \dots & \cos\left(\frac{\pi(i-1/2)2}{N}\right) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \cos\left(\frac{\pi(1/2)k}{N}\right) & \cos\left(\frac{\pi(3/2)k}{N}\right) & \dots & \cos\left(\frac{\pi(i-1/2)k}{N}\right) \end{pmatrix}, \tag{3.34}$$

The normalization constant C_k ensures that the transform is unitary:

$$C_k = \begin{cases} \sqrt{\frac{1}{N}} & \text{if } k = 0 \\ \sqrt{\frac{2}{N}} & \text{otherwise.} \end{cases} \tag{3.35}$$

In these modified normal mode coordinates, the Hessian takes on a simplified form

$$H'(V'_{\text{spring}}) = \text{DCT}_P^\dagger H(V'_{\text{spring}})\text{DCT}_P = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & f_1 & 0 & 0 & 0 \\ 0 & 0 & f_2 & 0 & 0 \\ 0 & 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & 0 & f_{P-1} \end{pmatrix}, \tag{3.36}$$

analogous to the modified potential in PIMD. The potential V'_{spring} can then be restated in terms of the normal mode positions, $\{\tilde{x}_k\}$:

$$V'_{\text{spring}}(\tilde{x}) = \sum_{k=0}^{P-1} \frac{1}{2} m\omega_k^2 (\tilde{x}_k)^2 \ , \tag{3.37}$$

with $\omega_k = 2\omega_{P-1}\sin(k\pi/(2P))$This modified potential leads to a transformed Hamiltonian in normal mode space,

$$\tilde{H}(\bar{p}, q) = \sum_{i=1}^{P} \frac{\bar{p}_i^2}{2\bar{m}} + \sum_{k=0}^{P-1} \frac{1}{2} m\omega_k^2 \tilde{q}_k^2 + \sum_{i=1}^{P} V'_i(q_i) \ . \tag{3.38}$$

Note that the form Equation (3.38) is *identical* to the starting point in the derivation of the PILE thermostat. This implies that already existing PILE integrator code can be used to perform PIGS, since the only differences are the use of the DCT instead of DFT in

defining $\tilde{x}_k$ and the different values of ω_k. This application of the PILE to integrating the PIGS Hamiltonian via the DCT we refer to as LE-PIGS.

3.3 Implementation

By beginning with the existing implementation of the PILE in MMTK, converting the code to perform LE-PIGS itself is rather straightforward. A `setTrialWave()` method was added to MMTK's `Atom` class in order to set a trial wavefunction. This method adds a forcefield to the system that is intended to only act on the first and last bead. Due to restrictions in the underlying code, the trial wavefunction is assumed to be of the form

$$\psi_T(x) = e^{-\tau\left[f(x)+\frac{V(x)}{2}\right]} \ , \tag{3.39}$$

and only evaluation of $f(x)$ needs to be programmed. The reason that this form is adopted is that in PIMD code the potential energy function is scaled evenly for all beads, where it should be scaled by 1/2 for the end beads. The form follows directly from this restriction. In order to compare to literature, it is of interest to examine the $\psi_T(x) = 1$ case. To implement this, it is required that $f(x) = -V(x)/2$ so that

$$\psi_T(x) = e^{-\tau\left[f(x)+\frac{V(x)}{2}\right]} = e^{-\tau\left[-\frac{V(x)}{2}+\frac{V(x)}{2}\right]} = e^{-\tau[0]} = 1 \ . \tag{3.40}$$

In this simple case

$$\langle E_0 \rangle_{\text{mixed}} = \langle V(x_P) \rangle \ , \tag{3.41}$$

since the wavefunction is constant and therefore has a derivative of zero. Regrettably, the MMTK code does not allow for access to V at a specific bead, so these energy estimators must be coded by hand.

The PILE code from MMTK was converted to use the `FFTW_REDFT10` method of FFTW [52] for the forward DCT and `FFTW_REDFT01` as the reverse DCT. The FFTW package provides $O(n \log n)$ time algorithms for both the forward and backward DCT. The modes of the DCT are weighted unevenly (i.e. C_k varies for $k = 0$), so the unitary transform was applied. This means that $\{\tilde{x}_i\}$ are normalized both on the forward and reverse transforms. The normalization therefore formally takes twice as long as PIMD but this time is quite small, and the normal mode space code is slightly simplified in this basis. The only other changes required to implement LE-PIGS are changing

$$\omega_k = 2\omega_P \sin\left(\frac{k\pi}{P}\right) \tag{3.42}$$

to

$$\omega_k = 2\omega_{P-1} \sin\left(\frac{k\pi}{2P}\right) \tag{3.43}$$

3.4 Results and Examples

Consider the simple example of a free particle. In this case $V(x) = 0$, and so the only potential term derives from the inter-bead spring term:

$$\tilde{H}_{\text{free}}(\bar{p}, q) = \sum_{i=1}^{P} \frac{\bar{p}^2}{2\bar{m}} + \sum_{k=0}^{P-1} \frac{1}{2} m\omega_k^2 \tilde{q}_k^2 \ . \tag{3.44}$$

Assuming γ_c is set to zero, the results of this path integral are known analytically. The $k = 0$ mode will experience a free ballistic trajectory, while all higher modes will be distributed as a gaussian with $\sigma = (\tau m \omega_k^2)^{-1}$. This result is what is obtained from the MMTK LE-PIGS code, as shown in Figure 3.1. This result is reassuring that the LE-PIGS integrator functions as expected.

For a slightly more complicated example, consider the harmonic oscillator. The potential for this system is

$$V_{\text{HO}}(x) = \frac{1}{2} k_{\text{HO}} x^2 \ , \tag{3.45}$$

where $k_{\text{HO}} = 1$ in atomic units. The ground state density, $\langle x \rangle_0$, is known to form a gaussian distribution. When expressed in atomic units, the ground state energy is known to be $1E_h$, with kinetic and potential energies each contributing $0.5E_h$.

The issue of limits must be discussed in order to examine the energies of this system.

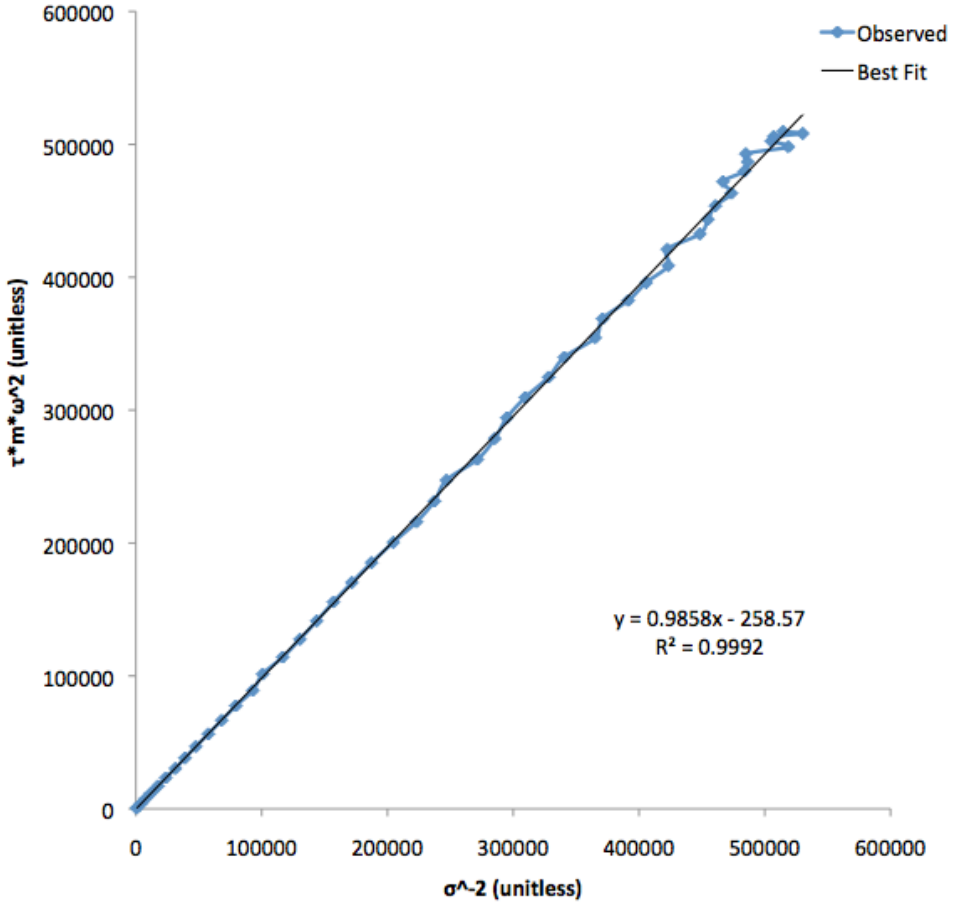

Figure 3.1: The variance of the position distribution of the various modes of a free particle. The distributions reproduce the expected variance from Equation (3.44).

As β tends towards ∞, the energy reported by the LE-PIGS energy estimator will tend towards the true E_0 as $e^{-\beta}$. However, as P tends towards ∞, the energy will tend towards E_0 as $-\tau^{-2}$. This effect is due to the nature of the Trotter factorization employed. In essence,

$$e^{A+B} \approx \left(e^{\frac{B}{2P}} e^{\frac{A}{P}} e^{\frac{B}{2P}}\right)^P + O(P^2) \ , \tag{3.46}$$

with the exact result being obtained in the limit of $P \to \infty$. A plot of reported E_0 vs τ should yield a parabolic convergence to the exact value of E_0 [53]. In this manner, the limit of $P \to \infty$ may be extrapolated from the graph, as seen in Figure 3.2.

Convergence to the correct value (within standard error) is achieved. The optimal choice of γ_0 for the harmonic oscillator is an interesting problem. Formally, the relationship

$$\gamma_0 = \frac{1}{2\tau_0} \tag{3.47}$$

should apply, with τ_0 equal to the correlation time of the centroid mode. In the harmonic oscillator, the centroid motion does not decorrelate, so the correlation time is infinite, leading to a γ_0 value of 0. For this model, Müser's rule of thumb [54] is employed and γ_0 is set to $0.01/\Delta t$.

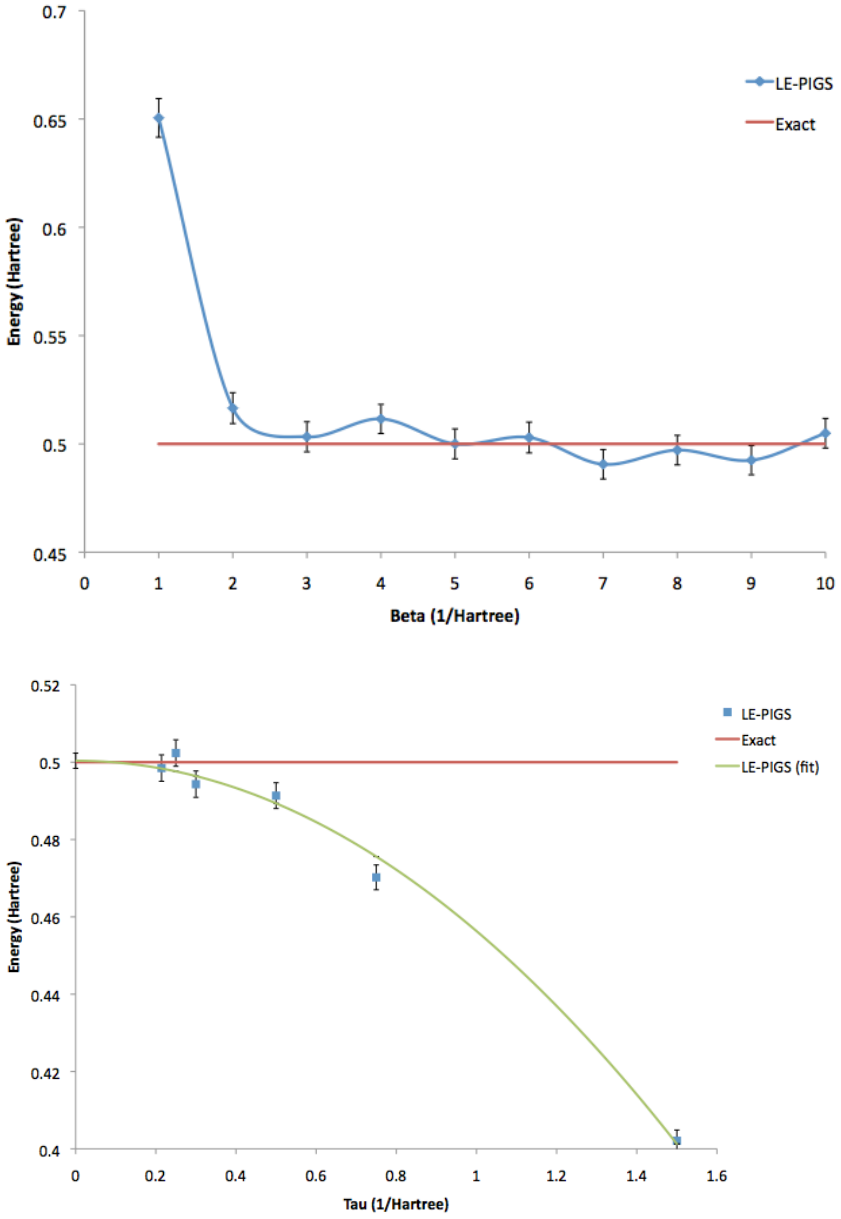

Figure 3.2: Convergence of E_0 with respect to simulation parameters β (top panel) and τ (bottom panel) for the harmonic oscillator.

For the quartic and double well potentials the following potentials are used:

$$V_{\mathrm{Q}}(x) = k_{\mathrm{Q}} x^4 \ , \tag{3.48}$$

$$V_{\mathrm{DW}}(x) = a_{\mathrm{DW}} x^2 + b_{\mathrm{DW}} x^4 \ , \tag{3.49}$$

with $k_{\mathrm{Q}} = 0.25$, $a_{\mathrm{DW}} = -0.5$, and $b_{\mathrm{DW}} = 0.1$ in atomic units. Numerical solutions for the convergence in τ can be evaluated by explicitly constructing the density matrix ρ and performing matrix multiplication. This technique is used to generate the exact values of E_0 for the quartic oscillator and double well potentials, seen in Figures 3.3 and 3.4, respectively.

The convergence in τ approaches the exact value for all three systems, within standard error. This is a strong indicator that the implementation is functioning as intended. The varying magnitude of the error across the models is a reflection of the difficulty of converging that parameter in simulation time. It is concluded that a novel method for ground state property prediction, LE-PIGS, has been created as a synthesis of the PIGS and PILE methods. The full extent of the power of this method has yet to be explored, but by analogy to related PIMD methods, it should provide better results than vanilla PIGS when high-P/low-τ regimes are under consideration because of the known ergodicity issue with ω_P.

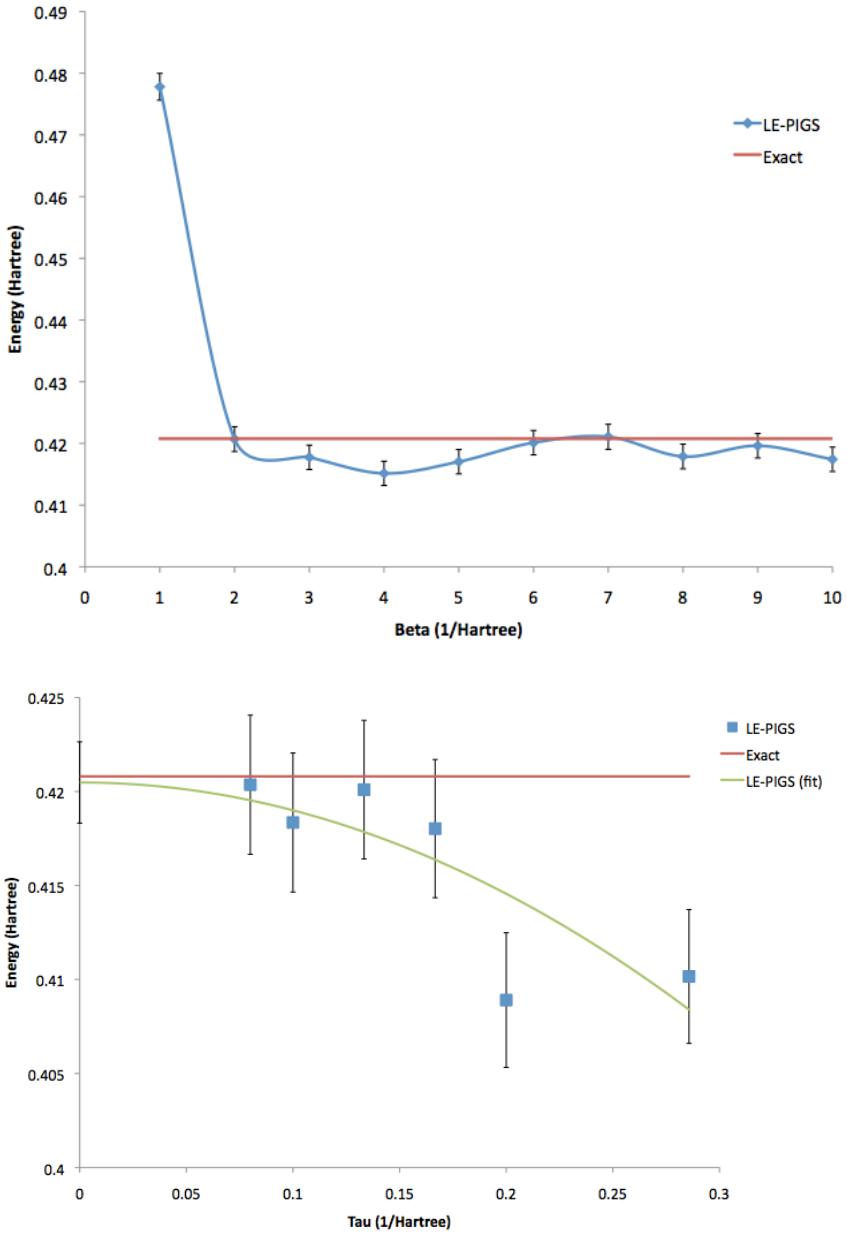

Figure 3.3: Convergence of E_0 with respect to simulation parameters β (top panel) and τ (bottom panel) for the quartic oscillator.

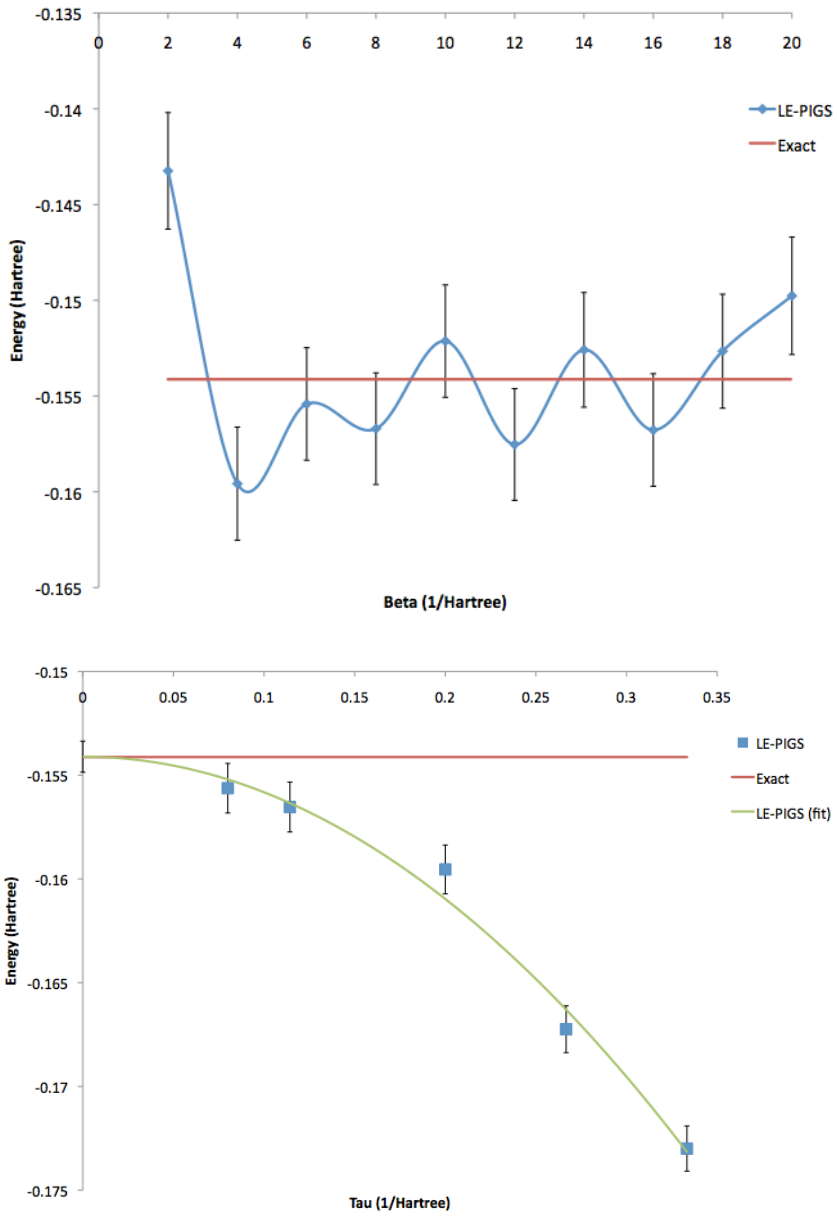

Figure 3.4: Convergence of E_0 with respect to simulation parameters β (top panel) and τ(bottom panel) for the double well potential.

Chapter 4

Conclusions

With the advent of widespread GPU computing and specialized GPGPU computing units, MD simulations of large systems for long timescales have become feasible. By leveraging the work of the OpenMM project, GPU computing support has been incorporated into MMTK and the result is a marked drop in overall runtime for both classical MD and PIMD. While initial forcefield support is limited, arbitrary system configurations are supported. Furthermore, additional forcefields can be supported, but OpenMM implementations must be provided and there is currently no way to load these implementations dynamically. The O-O, OH, and H-H radial distribution functions of water were reproduced with moderate accuracy by the new MMTK-OpenMM code, implying that it there may be some underlying bugs. The computational overhead associated with the addition of many path integral

beads was shown to be ameliorated when running simulations on the GPU.

A new tool for studying the ground state properties of arbitrary molecular systems has been realized in the LE-PIGS method. While the method is in its infancy and hasn't been comprehensively benchmarked against popular methods such as DMC, it has given good results for a variety of model systems. The main advantages over currently available codes are the ability to handle general systems without designing moves (a limitation of Monte Carlo approaches) and an efficient normal mode representation currently unavailable in existing PIGSMD codes. It is known that analogous PIMD codes suffer from ergodicity problems when integrated in cartesian space, so this representation should provide enhanced sampling of the canonical ensemble.

4.1 Future Work

At the time when this project was undertaken, OpenMM existed solely as a computational backend meant to be integrated into existed MD codes. However, under pressure from the scientific community, a Python-based frontend has been introduced to the OpenMM project, not unlike the MMTK-OpenMM code described here. However, compared to MMTK, this OpenMM Python environment has only the basic functionality required to run a simulation. Since both codes are open source and written in Python, there is potential

for cross-pollination of features between the two codebases. In particular, it may be possible to rewrite the MMTK-OpenMM bridge such that an intermediate C layer is not required, depending on the approach taken by the OpenMM Python code.

While MMTK is rapidly scriptable and it is easy for anyone to code their own custom potential, this flexibility does not necessarily extend to OpenMM. OpenMM provides a fixed number of forcefield primitives that have been integrated into MMTK so that AMBER-type molecular mechanics force fields are supported, with arbitrary parameter files. However, for many interesting systems, a custom potential function is desired. In MMTK this can be programmed with Python and/or C, but there is currently no way to inform the OpenMM backend of any nonstandard potentials. It is therefore desirable that the MMTK `ForceField` class be extended to provide an option of embedding OpenMM code, for example in a `OpenMMImplementation()` function, that will be passed into the underlying C code and executed. This is feasible since OpenMM embeds a small language that can be used to provide the functional form of a potential in terms of basic mathematical functions. Another option would be to provide a precomputed array of values describing the potential on a grid, for which OpenMM has some limited support.

The benchmarking of various codes have confirmed the notion that for Nvidia GPUs, the fastest execution results from programs written in CUDA C code. It is therefore desirable to extend the current OpenMM PIMD code to the CUDA `Platform` so that

available hardware may be used to its fullest extent. Due to the open source nature of the OpenMM libraries, it should be feasible to go through the current OpenCL implementation of PIMD and replace OpenCL-specific syntax with the equivalent CUDA code, especially since both languages are based on C. A CUDA-enabled PIMD integrator may achieve even greater speedups than those already highlighted in this report.

The integrator used for PIMD in OpenMM is the PILE. Due to the similarities between the LE-PIGS method and the PILE method, this OpenMM code can fairly easily be rewritten to provide GPU accelerated ground state code. To my knowledge, this would be the first such code in existence. The advantages of such a code are twofold: Larger systems up to the size of proteins and polymers could have their ground state energies deduced, and the increased number of beads and length of time afforded by the newfound efficiency would allow for vibrational spectroscopy of macromolecules and clusters. The ground state of a protein could be of interest, especially at the active site of an enzyme, where complicated reaction kinetics take place which may be affected by the gap between the bottom of the potential and the zero-point energy. Vibrational spectra can be extracted from PIGS trajectories by computing a correlation function in τ [55]. By utilizing OpenMM to perform the integration of PIGS equations it will be possible to achieve these long simulations in shorter wall clock time, as demonstrated for PIMD, especially where large numbers of beads are required.

The trial wavefunction, $\psi_T(x)$, is of central importance in PIGS. The trial function should try to maximize the overlap with the true ground state to accelerate convergence of the simulation. For simple model systems it is possible to provide a trial function that is exactly the form of the true ground state, but for complex molecular systems this is not generally true. One possible idea for a trial function of a molecular system is a sum of gaussian functions of the distance between atoms. This description arises from fact that the exact quantum mechanical solution of the harmonic oscillator is a gaussian, and bonds are often treated as harmonic oscillators for large systems. Such a trial function would be of the form:

$$\psi_T(x) = \sum_{i=1}^{N} e^{-\left(\frac{r_i-\mu_i}{\sigma_i}\right)^2} \tag{4.1}$$

for some total number N of gaussians, each having mean μ_i and deviation σ_i, and r_i being the distance between two atoms of interest. Clearly μ_i is related to the equilibrium length of the bond and σ_i to its strength, but how exactly to construct these functions is an area yet to be explored. The energy of such a system will be easy to compute, since

$$\frac{\partial^2}{\partial x^2} e^{-x^2} = (x^2 - 1)e^{-x^2} \tag{4.2}$$

and the laplacian distributes linearly across the sum. With proper tuning, a trial wavefunction could potentially be constructed automatically from the parameters of an AMBER-

style potential. This approach could be especially useful in dealing with organic molecules, which are well described by these kinds of potentials.

One open question in the LE-PIGS theory is that of centroid dynamics. In PIMD, the centroid (or $k = 0$ Fourier mode) is identified as the "most classical" degree of freedom of the quantum system. The centroid is implicated in various semi-classical theories such a Quantum Transition State Theory (QTST) [56] and Centroid Molecular Dynamics (CMD) [57]. In LE-PIGS we have identified that the DCT diagonalizes the Hessian of the inter-bead spring potential, and thus the $k = 0$ mode may also play a special role in the dynamics of the ground state. One notable difference is that in PIMD the centroid consists of contributions from all beads, each of which may be considered as contributing equally of the average of A since the path integral for PIMD is invariant under cyclic relabelling of the bead indices, whereas in PIGS the only bead that contributes to the average is the middle bead. However, for properties that commute with the density matrix ρ such as the Hamiltonian $\hat{H}$, it is possible to move the operator to an arbitrary bead and act with it there, and in such a manner contributions from all beads could be considered. It is currently unknown whether or not such an approach is theoretically justified, but it may be worthy of further investigation as it would yield correlation functions in time for ground state systems.

Of course, one obvious area of study is the application of LE-PIGS beyond model sys-

tems. Investigations in this area are currently underway, in particular with applications to the ground state energies of small metal clusters. Preliminary results show good agreement with results from PIMC simulations. In the future, it is expected that GPU accelerated LE-PIGS will be able to handle a wide variety of systems that were previously unreachable due to computational constraints.

References

[1] H. Goldstein, *Classical mechanics*, Addison-Wesley series in physics (Addison-Wesley Pub. Co., 1980).

[2] L. Verlet, Physical Review **159**, 98 (1967).

[3] J. Kestemontand, Journal of Computational Physics **458**, 451 (1976).

[4] M. E. Tuckerman and G. J. Martyna, The Journal of Physical Chemistry B **104**, 159 (2000).

[5] D. McQuarrie, *Statistical Mechanics* (University Science Books, 2000).

[6] D. Chandler, *Introduction to Modern Statistical Mechanics* (Oxford University Press, USA, 1987).

[7] R. Wilde and S. Singh, *Statistical mechanics: fundamentals and modern applications*, A Wiley-Interscience publication (Wiley, 1998).

[8] H. C. Andersen, The Journal of Chemical Physics **72**, 2384 (1980).

[9] D. J. Evans and B. L. Holian, The Journal of Chemical Physics **83**, 4069 (1985).

[10] G. J. Martyna, M. L. Klein, and M. Tuckerman, The Journal of Chemical Physics **97**, 2635 (1992).

[11] M. Ceriotti, M. Parrinello, T. E. Markland, and D. E. Manolopoulos, The Journal of Chemical Physics **133**, 124104 (2010).

[12] D. S. Lemons, A. Gythiel, and P. Langevin, American Journal of Physics **65**, 1079 (1997).

[13] G. Bussi and M. Parrinello, Physical Review E **75** (2007).

[14] Mizumoto, Chemical Physics Letters **501**, 304 (2010).

[15] K. F. Wong, J. L. Sonnenberg, F. Paesani, T. Yamamoto, J. Vanicek, W. Zhang, H. B. Schlegel, D. A. Case, T. E. Cheatham III, and W. H. Miller, Journal of Chemical Theory and Computation **6**, 2566 (2010).

[16] R. Feynman and A. Hibbs, *Quantum mechanics and path integrals*, International series in pure and applied physics (McGraw-Hill, 1965).

[17] M. F. Herman, E. J. Bruskin, and B. J. Berne, The Journal of Chemical Physics **76**, 5150 (1982).

[18] H. Kono, A. Takasaka, and S. H. Lin, The Journal of Chemical Physics **88**, 6390 (1988).

[19] D. Chandler and P. G. Wolynes, The Journal of Chemical Physics **74**, 4078 (1981).

[20] K. S. Schweizer, The Journal of Chemical Physics **75**, 1347 (1981).

[21] R. G. Woolley and B. T. Sutcliffe, Chemical Physics Letters **45**, 393 (1977).

[22] J. Anderson, International Reviews in Physical Chemistry **14**, 85 (1995).

[23] M. E. Tuckerman, Path Integration via Molecular Dynamics, in *Quantum Simulations of Complex Many-Body Systems*, volume 10, pages 269–298, John von Neumann Institute for Computing, 2002.

[24] H. F. Trotter, Proceedings of the American Mathematical Society **10**, 545 (1959).

[25] M. Müser, Computer physics communications **147**, 83 (2002).

[26] M. F. Herman, The Journal of Chemical Physics **76**, 5150 (1982).

[27] R. W. Hall and B. J. Berne, The Journal of Chemical Physics **81**, 3641 (1984).

[28] M. Ceriotti, G. Bussi, and M. Parrinello, Journal of Chemical Theory **6**, 1170 (2010).

[29] C. Ing, K. Hinsen, J. Yang, T. Zeng, H. Li, and P.-N. Roy, The Journal of Chemical Physics **136**, 224309 (2012).

[30] K. Hinsen, Journal of Computational Chemistry **21**, 79 (2000).

[31] U. Borštnik, B. T. Miller, B. R. Brooks, and D. Janežič, Journal of Computational Chemistry **32**, 3005 (2011).

[32] M. S. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houston, S. Legrand, A. L. Beberg, D. L. Ensign, C. M. Bruns, and V. S. Pande, Journal of Computational Chemistry **30**, 864 (2009).

[33] J. E. Stone, D. Gohara, and G. Shi, Computing in Science & Engineering **12**, 66 (2010).

[34] V. Pande, Computing in Science & Engineering **12**, 34 (2010).

[35] F. Paesani, W. Zhang, D. A. Case, T. E. Cheatham, and G. A. Voth, The Journal of chemical physics **125**, 184507 (2006).

[36] A. Rahman, Physical Review **136**, A405 (1964).

[37] A. Sarsa, K. E. Schmidt, and W. R. Magro, The Journal of Chemical Physics **113**, 1366 (2000).

[38] P. Whitlock, D. Ceperley, and G. Chester, Physical Review B **19**, 5598 (1979).

[39] P. Sindzingre and M. Klein, Physical Review Letters **63**, 1601 (1989).

[40] D. Ceperley, Reviews of Modern Physics **67** (1995).

[41] J. E. Cuervo, P.-N. Roy, and M. Boninsegni, The Journal of Chemical Physics **122**, 114504 (2004).

[42] S. Miura, Chemical Physics Letters **482**, 165 (2009).

[43] R. Rota, J. Casulleras, F. Mazzanti, and J. Boronat, Physical Review E **81**, 1 (2010).

[44] B. Hetenyi, E. Rabani, and B. J. Berne, The Journal of Chemical Physics **110**, 6143 (1999).

[45] D. Ceperley, Journal of Statistical Physics **43** (1986).

[46] J. Carlson, Physical Review C **36**, 2026 (1987).

[47] D. E. Galli and L. Reatto, Molecular Physics **101**, 1697 (2003).

[48] R. J. Hinde, Chemical Physics Letters **418**, 481 (2006).

[49] J. E. Cuervo and P.-N. Roy, The Journal of Chemical Physics **125**, 124314 (2006).

[50] P. Garcia and A. Peinado, Signal Processing, **43**, 2631 (1995).

[51] C. J. Burnham, G. F. Reiter, J. Mayers, T. Abdul-Redah, H. Reichert, and H. Dosch, Physical Chemistry Chemical Physics **8**, 3966 (2006).

[52] M. Frigo and S. G. Johnson, Proceedings of the IEEE **93**, 216 (2005), Special issue on "Program Generation, Optimization, and Platform Adaptation".

[53] N. Blinov, X. Song, and P.-N. Roy, The Journal of Chemical Physics **120**, 5916 (2004).

[54] M. H. Müser, The Journal of Chemical Physics **114**, 6364 (2001).

[55] S. Moroni, M. Botti, S. De Palo, and A. R. W. McKellar, The Journal of Chemical Physics **122**, 094314 (2005).

[56] F. McLafferty and P. Pechukas, Chemical Physics Letters **27**, 511 (1974).

[57] J. Cao and G. A. Voth, The Journal of Chemical Physics **101**, 6168 (1994).

Appendix A

List of Code

A.1 Python Layer: LangevinDynamics.py

```
from MMTK import Dynamics, Environment, Features, Trajectory,

Units, ParticleProperties

import numpy as N

from MMTK.Features import PathIntegralsFeature

import MMTK_langevin

import MMTK_forcefield

import operator, copy


#
```

```
# Langevin integrator
#
class LangevinIntegrator(Dynamics.Integrator):


def __init__(self, universe, **options):
Dynamics.Integrator.__init__(self, universe, options)
# Supported features:  path integrals, to do PIMD
self.features = [PathIntegralsFeature]
assert PathIntegralsFeature.isInUniverse(universe)
self.nbeads = universe.maxNumberOfBeads()


#def initOmmSystem(self):
# atoms = self.universe.atomList()
# masses = [atom.mass() for atom in atoms]
# MMTK_langevin.initOpenMM(N.array(masses, dtype=N.float64), len(atoms))


def initOmmSystem(self):
atoms = copy.copy(self.universe.atomList())
atoms.sort(key=operator.attrgetter('index'))
masses = [atom.mass() for atom in atoms]
MMTK_langevin.initOpenMM(N.array(masses, dtype=N.float64), len(atoms))
```

```
def destroyOmmSystem(self):
MMTK_langevin.destroyOpenMM()


def addOmmCMRemover(self, skip):
MMTK_langevin.addCMRemover(skip)


def createOmmForces(self):
params = self.universe.energyEvaluatorParameters()
natoms = len(self.universe.atomList())
nbeads = self.nbeads


#here we convert MMTK's internal representation of the forcefield parameters
#to OpenMM. For a new MMTK forcefield to work, it must implement the
#energyEvaluatorParameters method, and this code must be modified to make use
#of those parameters


#first, lets start with the bonded forcefield
bonds = params['harmonic_distance_term']


atom_index_1 = []
atom_index_2 = []
eq_distance = []
```

```
spring_k = []


for bond in bonds:

if ((bond[0] % nbeads == 0) and (bond[1] % nbeads == 0)):

atom_index_1.append(bond[0]/nbeads)

atom_index_2.append(bond[1]/nbeads)

eq_distance.append(bond[2])

spring_k.append(bond[3])


MMTK_langevin.makeOmmBondedForce(N.array(atom_index_1, dtype=N.int32),

N.array(atom_index_2, dtype=N.int32),

N.array(eq_distance, dtype=N.float64),

N.array(spring_k, dtype=N.float64),

len(atom_index_1))


#angles

angles = params['harmonic_angle_term']


atom_index_1 = []

atom_index_2 = []

atom_index_3 = []

eq_angle = []
```

```
spring_k = []


for angle in angles:
if ((angle[0] % nbeads == 0) and (angle[1] % nbeads == 0) and (angle[2] % nbeads == 0)):
atom_index_1.append(angle[0]/nbeads)
atom_index_2.append(angle[1]/nbeads)
atom_index_3.append(angle[2]/nbeads)
eq_angle.append(angle[3])
spring_k.append(angle[4])


MMTK_langevin.makeOmmAngleForce(N.array(atom_index_1, dtype=N.int32),
N.array(atom_index_2, dtype=N.int32),
N.array(atom_index_3, dtype=N.int32),
N.array(eq_angle, dtype=N.float64),
N.array(spring_k, dtype=N.float64),
len(atom_index_1))


#dihedrals
dihedrals = params['cosine_dihedral_term']


atom_index_1 = []
atom_index_2 = []
```

```
atom_index_3 = []

atom_index_4 = []

periodicity = []

eq_dihedral = []

spring_k = []


for dihedral in dihedrals:

if ((dihedral[0] % nbeads == 0) and (dihedral[1] % nbeads == 0) and (dihedral[2] % nbeads

== 0) and (dihedral[3] % nbeads == 0)):

atom_index_1.append(dihedral[0]/nbeads)

atom_index_2.append(dihedral[1]/nbeads)

atom_index_3.append(dihedral[2]/nbeads)

atom_index_4.append(dihedral[3]/nbeads)

periodicity.append(dihedral[4])

eq_dihedral.append(dihedral[5])

spring_k.append(dihedral[6])


MMTK_langevin.makeOmmDihedralForce(N.array(atom_index_1, dtype=N.int32),

N.array(atom_index_2, dtype=N.int32),

N.array(atom_index_3, dtype=N.int32),

N.array(atom_index_4, dtype=N.int32),

N.array(periodicity, dtype=N.int32),

N.array(eq_dihedral, dtype=N.float64),
```

```
N.array(spring_k, dtype=N.float64),

len(atom_index_1))


#now, nonbonded forces

lj = params['lennard_jones']

lj_14_factor = lj['one_four_factor']

lj_cutoff = lj['cutoff']


e_s_matrix = lj['epsilon_sigma']

epsilon_list = []

sigma_list = []


for i in range(0, len(e_s_matrix)):

epsilon, sigma = e_s_matrix[i][i]

epsilon_list.append(epsilon)

sigma_list.append(sigma)


#contains per-particle information, but we want per-atom

e_s_types = lj['type']

epsilon_p = [] # per-particle epsilons

sigma_p = [] # per-particle sigmas
```

```
for type in e_s_types:

epsilon_p.append(epsilon_list[type])

sigma_p.append(sigma_list[type])


epsilon = [] # per-atom epsilons

sigma = [] # per-atom sigmas


for i in range(0,natoms*self.nbeads,self.nbeads):

epsilon.append(epsilon_p[i])

sigma.append(sigma_p[i])


#now epsilon and sigma contain the correct values in the correct order


elec = params['electrostatic']

elec_14_factor = elec['one_four_factor']

elec_cutoff = elec['real_cutoff'] #for step_0_

#elec_cutoff = elec['cutoff'] #for example


charge_p = elec['charge'] # per-particle charges

elec_method = elec['algorithm']


charge = [] # per-atom charges
```

```
for i in range(0,natoms*self.nbeads,self.nbeads):

charge.append(charge_p[i])


MMTK_langevin.makeOmmEsAndLjForce(N.array(sigma, dtype=N.float64),

N.array(epsilon, dtype=N.float64),

N.array(charge, dtype=N.float64),

elec_14_factor,

lj_14_factor,

elec_cutoff,

len(sigma))


def __call__(self, **options):

# Process the keyword arguments

self.setCallOptions(options)

# Check if the universe has features not supported by the integrator

Features.checkFeatures(self, self.universe)


#the following two lines are required due to MMTK mixing up atom indices.  This sorts them.

#atoms = self.universe.atomList()

#atoms.sort(key=operator.attrgetter('index'))


# Get the universe variables needed by the integrator
```

```
configuration = self.universe.configuration()

velocities = self.universe.velocities()

if velocities is None:

raise ValueError("no velocities")


# Get the friction coefficients.  First check if a keyword argument

# 'friction' was given to the integrator.  Its value can be a

# ParticleScalar or a plain number (used for all atoms).  If no

# such argument is given, collect the values of the attribute

# 'friction' from all atoms (default is zero).

friction = self.getOption('friction')


#call this method to create the OpenMM System.  Without this, creating

#the forces will fail!!

self.initOmmSystem()


#call this method to ensure that the forcefield parameters get

#passed to OpenMM. Without this, there will be no forces to integrate!

self.createOmmForces()


#ugly hack to get timestep skip information

skip = -1
```

```
if ('actions' in self.call_options):

actions = self.call_options['actions']

for action in actions:

if isinstance(action, Dynamics.TranslationRemover):

self.addOmmCMRemover(action.skip)

if isinstance(action, Trajectory.TrajectoryOutput):

skip = action.skip

action.skip = 1



if skip < 0 :

for action in actions:

if issubclass(action.__class__, Trajectory.TrajectoryOutput):

skip = action.skip

action.skip = 1

break



#we are not logging anything, so we don't have to report any intermediate values

if skip < 0:

skip = 100

steps = 1



# Run the C integrator
```

```
atoms = self.universe.atomList()

masses = [atom.mass() for atom in atoms]

natoms = len(atoms)

MMTK_langevin.integrateLD(self.universe, configuration.array,

velocities.array, N.array(masses, dtype=N.float64),

friction,

self.getOption('temperature'),

self.getOption('delta_t'),

self.getOption('steps'), skip, natoms, self.nbeads,

self.getActions())


#call this to clean up after ourselves

self.destroyOmmSystem()
```

A.2 C Layer: MMTK_langevin.c

```
#include "MMTK/universe.h"

#include "MMTK/forcefield.h"

#include "MMTK/trajectory.h"

#include "OpenMMCWrapper.h"

#include <stdio.h>
```

```
/* Global variables */

OpenMM_System* omm_system;

OpenMM_BondArray* bonded_list;

double kB; /* Boltzman constant */


/* Allocate and initialize Output variable descriptors */

static PyTrajectoryVariable *

get_data_descriptors(int n, PyUniverseSpecObject *universe_spec,

PyArrayObject *configuration, PyArrayObject *velocities,

PyArrayObject *gradients, PyArrayObject *masses,

int *ndf, double *time,

double *p_energy, double *k_energy,

double *temperature, double *pressure,

double *box_size) {

PyTrajectoryVariable *vars = (PyTrajectoryVariable *)

malloc((n+1)*sizeof(PyTrajectoryVariable));

int i = 0;

if (vars != NULL) {

if (time != NULL && i < n) {

vars[i].name = "time";

vars[i].text = "Time:  %lf";

vars[i].unit = time_unit_name;

vars[i].type = PyTrajectory_Scalar;
```

```
vars[i].class = PyTrajectory_Time;

vars[i].value.dp = time;

i++;

}

if (p_energy != NULL && i < n) {

vars[i].name = "potential_energy";

vars[i].text = "Potential energy:  %lf, ";

vars[i].unit = energy_unit_name;

vars[i].type = PyTrajectory_Scalar;

vars[i].class = PyTrajectory_Energy;

vars[i].value.dp = p_energy;

i++;

}

if (k_energy != NULL && i < n) {

vars[i].name = "kinetic_energy";

vars[i].text = "Kinetic energy:  %lf";

vars[i].unit = energy_unit_name;

vars[i].type = PyTrajectory_Scalar;

vars[i].class = PyTrajectory_Energy;

vars[i].value.dp = k_energy;

i++;

}

if (temperature != NULL && i < n) {
```

```
vars[i].name = "temperature";

vars[i].text = "Temperature:  %lf";

vars[i].unit = temperature_unit_name;

vars[i].type = PyTrajectory_Scalar;

vars[i].class = PyTrajectory_Thermodynamic;

vars[i].value.dp = temperature;

i++;

}

if (pressure != NULL && i < n) {

vars[i].name = "pressure";

vars[i].text = "Pressure:  %lf";

vars[i].unit = pressure_unit_name;

vars[i].type = PyTrajectory_Scalar;

vars[i].class = PyTrajectory_Thermodynamic;

vars[i].value.dp = pressure;

i++;

}

if (configuration != NULL && i < n) {

vars[i].name = "configuration";

vars[i].text = "Configuration:

n";

vars[i].unit = length_unit_name;

vars[i].type = PyTrajectory_ParticleVector;
```

```
vars[i].class = PyTrajectory_Configuration;

vars[i].value.array = configuration;

i++;

}

if (box_size != NULL && i < n) {

vars[i].name = "box_size";

vars[i].text = "Box size:";

vars[i].unit = length_unit_name;

vars[i].type = PyTrajectory_BoxSize;

vars[i].class = PyTrajectory_Configuration;

vars[i].value.dp = box_size;

vars[i].length = universe_spec->geometry_data_length;

i++;

}

if (velocities != NULL && i < n) {

vars[i].name = "velocities";

vars[i].text = "Velocities:

n";

vars[i].unit = velocity_unit_name;

vars[i].type = PyTrajectory_ParticleVector;

vars[i].class = PyTrajectory_Velocities;

vars[i].value.array = velocities;

i++;
```

```
}
if (gradients != NULL && i < n) {
vars[i].name = "gradients";
vars[i].text = "Energy gradients:
n";
vars[i].unit = energy_gradient_unit_name;
vars[i].type = PyTrajectory_ParticleVector;
vars[i].class = PyTrajectory_Gradients;
vars[i].value.array = gradients;
i++;
}
if (masses != NULL && i < n) {
vars[i].name = "masses";
vars[i].text = "Masses:";
vars[i].unit = mass_unit_name;
vars[i].type = PyTrajectory_ParticleScalar;
vars[i].class = PyTrajectory_Internal;
vars[i].value.array = masses;
i++;
}
if (ndf != NULL && i < n) {
vars[i].name = "degrees_of_freedom";
vars[i].text = "Degrees of freedom:  %d";
```

```
vars[i].unit = "";
vars[i].type = PyTrajectory_IntScalar;
vars[i].class = PyTrajectory_Internal;
vars[i].value.ip = ndf;
i++;
}
vars[i].name = NULL;
}
return vars;
}


/* Langevin dynamics integrator */
static PyObject *
integrateLD(PyObject *dummy, PyObject *args)
{
/* The parameters passed from the Python code */
PyObject *universe; /* universe */
PyArrayObject *configuration; /* array of positions */
PyArrayObject *velocities; /* array of velocities */
PyArrayObject *masses; /* array of masses */
PyArrayObject *friction; /* array of friction coefficients */
PyListObject *spec_list; /* list of periodic actions */
double ext_temp; /* temperature of the heat bath */
```

```
double delta_t; /* time step */

int steps; /* number of steps */

int skip; /* number of steps to skip */

int natoms; /* number of atoms */


/* Other variables, see below for explanations */

PyThreadState *this_thread;

PyUniverseSpecObject *universe_spec;

PyArrayObject *gradients, *random1, *random2;

PyTrajectoryOutputSpec *output;

PyTrajectoryVariable *data_descriptors = NULL;

vector3 *x, *v, *g;

double *f;

energy_data p_energy;

double time, k_energy, temperature;

int atoms, df;

int pressure_available;

int i, j, k;


/* OpenMM required variables */

OpenMM_Context* context;

OpenMM_Platform* platform;

OpenMM_LangevinIntegrator* integrator;
```

```
OpenMM_State* state;
OpenMM_Vec3Array* omm_pos;
OpenMM_Vec3Array* omm_vel;
OpenMM_Vec3* pos_i;
OpenMM_Vec3* vel_i;


/* Get arguments passed from Python code */
if (!PyArg_ParseTuple(args, "OO!O!O!O!ddiiiO!", &universe,
&PyArray_Type, &configuration,
&PyArray_Type, &velocities,
&PyArray_Type, &masses,
&PyArray_Type, &friction,
&ext_temp, &delta_t, &steps,
&skip, &natoms, &PyList_Type, &spec_list))
return NULL;


/* Obtain the universe specification */
universe_spec = (PyUniverseSpecObject *)
PyObject_GetAttrString(universe, "_spec");
if (universe_spec == NULL)
return NULL;
```

```
/* Create the array for energy gradients */
#if defined(NUMPY)
gradients = (PyArrayObject *)PyArray_Copy(configuration);
#else
gradients = (PyArrayObject *)PyArray_FromDims(configuration->nd,
configuration->dimensions,
PyArray_DOUBLE);
#endif
if (gradients == NULL)
return NULL;

/* Set some convenient variables */
atoms = natoms; /* number of atoms */
x = (vector3 *)configuration->data; /* pointer to positions */
v = (vector3 *)velocities->data; /* pointer to velocities */
g = (vector3 *)gradients->data; /* pointer to energy gradients */
f = (double *)friction->data; /* pointer to friction constant */
df = 3*atoms; /* number of degrees of freedom */

/* Initialize trajectory output and periodic actions */
data_descriptors =
get_data_descriptors(10 + pressure_available,
universe_spec,
```

```
configuration, velocities,
gradients, masses, &df,
&time, &p_energy.energy, &k_energy,
&temperature, NULL,
(universe_spec->geometry_data_length > 0) ?
universe_spec->geometry_data :  NULL);
if (data_descriptors == NULL){
goto error2;}
output = PyTrajectory_OutputSpecification(universe, spec_list,
"Langevin Dynamics",
data_descriptors);
if (output == NULL){
goto error2;}


/* Initial coordinate correction (for periodic universes etc.)  */
universe_spec->correction_function(x, atoms, universe_spec->geometry_data);


if (universe_spec->is_periodic == 1) {
if (universe_spec->is_orthogonal == 1) {
/*double min = universe_spec->geometry_data[0];
int geometry_counter = 0;
for (geometry_counter = 0; geometry_counter < universe_spec->geometry_data_length;
geometry_counter++) {
```

```
if (universe_spec->geometry_data[geometry_counter] < min) {
min = universe_spec->geometry_data[geometry_counter];
}
}*/
OpenMM_Vec3 a = {universe_spec->geometry_data[0],0.0,0.0};
OpenMM_Vec3 b = {0.0,universe_spec->geometry_data[1],0.0};
OpenMM_Vec3 c = {0.0,0.0,universe_spec->geometry_data[2]};

OpenMM_System_setDefaultPeriodicBoxVectors(omm_system, &a, &b, &c);

printf("REMARK Using OpenMM periodic box with dimensions (%f, %f, %f)
n", universe_spec->geometry_data[0],
universe_spec->geometry_data[1], universe_spec->geometry_data[2]);

} else {
/* abort!  OpenMM does not support non-orthorhombic periodic universes */
printf("ERROR OpenMM does not support non-orthorhombic universes
n");
goto error2;
}
}
```

```
integrator = (OpenMM_Integrator*) OpenMM_LangevinIntegrator_create(ext_temp, *f, delta_t);

/* Let OpenMM Context choose best platform.  */

/*context = OpenMM_Context_create(omm_system, integrator);

platform = OpenMM_Context_getPlatform(context);*/

printf( "REMARK Using OpenMM platform %s

n",

OpenMM_Platform_getName(platform));


/* Set starting positions and velocities of the atoms.  Leave time zero.  */

omm_pos = OpenMM_Vec3Array_create(atoms);

omm_vel = OpenMM_Vec3Array_create(atoms);


for (i = 0; i < atoms; i++) {

pos_i = x[i];

vel_i = v[i];

OpenMM_Vec3Array_set(omm_pos, i, *pos_i);

OpenMM_Vec3Array_set(omm_vel, i, *vel_i);

}

OpenMM_Context_setPositions(context, omm_pos);

OpenMM_Context_setVelocities(context, omm_vel);


/* Collect properties of interest for which to query OpenMM */
```

```
int infoMask = OpenMM_State_Positions + OpenMM_State_Energy;

/*

* Main integration loop

*/

time = 0.;


printf("completing %d steps while skipping %d per log output

n", steps, skip);

for (k = 0; k < steps; k+= skip) {

/* Calculation of thermodynamic properties */

state = OpenMM_Context_getState(context, infoMask, 1);


omm_pos = OpenMM_State_getPositions(state);

for (i = 0; i < atoms; i++) {

pos_i = OpenMM_Vec3Array_get(omm_pos, i);

x[i][0] = pos_i->x;

x[i][1] = pos_i->y;

x[i][2] = pos_i->z;

}

k_energy = OpenMM_State_getKineticEnergy(state);

p_energy.energy = OpenMM_State_getPotentialEnergy(state);

time = OpenMM_State_getTime(state);
```

```
temperature = 2.*k_energy/(df*kB);


OpenMM_State_destroy(state);

/* Trajectory and log output */

PyTrajectory_Output(output, k, data_descriptors, NULL);


OpenMM_LangevinIntegrator_step(integrator, skip);

/* Coordinate correction (for periodic universes etc.)  */

/* universe_spec->correction_function(x, atoms, universe_spec->geometry_data); */

}

/** End of main integration loop **/


/* Final thermodynamic property evaluation */

state = OpenMM_Context_getState(context, infoMask, 1);


omm_pos = OpenMM_State_getPositions(state);

for (i = 0; i < atoms; i++) {

pos_i = OpenMM_Vec3Array_get(omm_pos, i);

x[i][0] = pos_i->x;

x[i][1] = pos_i->y;

x[i][2] = pos_i->z;

}
```

```
k_energy = OpenMM_State_getKineticEnergy(state);

p_energy.energy = OpenMM_State_getPotentialEnergy(state);

time = OpenMM_State_getTime(state);

temperature = 2.*k_energy/(df*kB);


OpenMM_State_destroy(state);

/* Final trajectory and log output */

PyTrajectory_Output(output, k, data_descriptors, NULL);


/* Cleanup */

PyUniverseSpec_StateLock(universe_spec, -2);

/*PyEval_RestoreThread(this_thread);*/

PyTrajectory_OutputFinish(output, k, 0, 1, data_descriptors);

free(data_descriptors);

Py_DECREF(gradients);

Py_INCREF(Py_None);

return Py_None;


/* Error return */

error:

PyTrajectory_OutputFinish(output, k, 1, 1, data_descriptors);

error2:
```

```
free(data_descriptors);

Py_DECREF(gradients);

return NULL;

}


static PyObject* initOpenMM(PyObject* dummy, PyObject* args) {

PyArrayObject* masses;

int natoms;


if (!PyArg_ParseTuple(args, "O!i",

&PyArray_Type, &masses,

&natoms))

return NULL;


double* m = (double*)masses->data;

omm_system = OpenMM_System_create();


int iter;

for (iter = 0; iter < natoms; iter++) {

OpenMM_System_addParticle(omm_system, m[iter]);

/* printf("Added particle %d with mass %f

n", iter, m[iter]); */
```

```
}
return Py_None;
}

static PyObject* destroyOpenMM(PyObject* dummy, PyObject* args) {
OpenMM_System_destroy(omm_system);
OpenMM_BondArray_destroy(bonded_list);
return Py_None;
}

static PyObject* addCMRemover(PyObject *dummy, PyObject *args) {
int skip;

if (!PyArg_ParseTuple(args, "i",
&skip))
return NULL;

OpenMM_CMMotionRemover* remover = OpenMM_CMMotionRemover_create(skip);
OpenMM_System_addForce(omm_system, (OpenMM_Force*)remover);
return Py_None;
}
```

```
static PyObject* makeOmmBondedForce(PyObject *dummy, PyObject *args) {
PyArrayObject *atom_index_1;
PyArrayObject *atom_index_2;
PyArrayObject *eq_distance;
PyArrayObject *spring_k;
int nbonds;

if (!PyArg_ParseTuple(args, "O!O!O!O!i",
&PyArray_Type, &atom_index_1,
&PyArray_Type, &atom_index_2,
&PyArray_Type, &eq_distance,
&PyArray_Type, &spring_k,
&nbonds))
return NULL;

int* i = (int*)atom_index_1->data;
int* j = (int*)atom_index_2->data;
double* r = (double*)eq_distance->data;
double* k_eq = (double*)spring_k->data;

OpenMM_HarmonicBondForce* bonded = OpenMM_HarmonicBondForce_create();
bonded_list = OpenMM_BondArray_create(nbonds);
```

```
printf("Adding bonds

n");


int iter;

for (iter = 0; iter< nbonds; iter++) {

OpenMM_HarmonicBondForce_addBond(bonded, i[iter], j[iter], r[iter], k_eq[iter]);

/*printf("Added bond %d between atoms %d and %d with length %f and strength %f

n", iter, i[iter], j[iter], r[iter], k_eq[iter]);*/

OpenMM_BondArray_append(bonded_list, i[iter], j[iter]);

}


OpenMM_System_addForce(omm_system, (OpenMM_Force*)bonded);

return Py_None;

}


static PyObject* makeOmmAngleForce(PyObject *dummy, PyObject *args) {

PyArrayObject *atom_index_1;

PyArrayObject *atom_index_2;

PyArrayObject *atom_index_3;

PyArrayObject *eq_angle;

PyArrayObject *spring_k;
```

```
int nangles;


if (!PyArg_ParseTuple(args, "O!O!O!O!O!i",

&PyArray_Type, &atom_index_1,

&PyArray_Type, &atom_index_2,

&PyArray_Type, &atom_index_3,

&PyArray_Type, &eq_angle,

&PyArray_Type, &spring_k,

&nangles))

return NULL;


int* i = (int*)atom_index_1->data;

int* j = (int*)atom_index_2->data;

int* k = (int*)atom_index_3->data;

double* theta = (double*)eq_angle->data;

double* k_eq = (double*)spring_k->data;


OpenMM_HarmonicAngleForce* angles = OpenMM_HarmonicAngleForce_create();


printf("Adding angles

n");
```

```
int iter;
for (iter = 0; iter < nangles; iter++) {
OpenMM_HarmonicAngleForce_addAngle(angles, i[iter], j[iter], k[iter], theta[iter], k_eq[iter]);
}

OpenMM_System_addForce(omm_system, (OpenMM_Force*)angles);
return Py_None;
}

static PyObject* makeOmmDihedralForce(PyObject* dummy, PyObject* args) {
PyArrayObject* atom_index_1;
PyArrayObject* atom_index_2;
PyArrayObject* atom_index_3;
PyArrayObject* atom_index_4;
PyArrayObject* periodicity;
PyArrayObject* eq_dihedral;
PyArrayObject* spring_k;
int ndihedrals;

if (!PyArg_ParseTuple(args, "O!O!O!O!O!O!O!i",
&PyArray_Type, &atom_index_1,
&PyArray_Type, &atom_index_2,
```

```
&PyArray_Type, &atom_index_3,

&PyArray_Type, &atom_index_4,

&PyArray_Type, &periodicity,

&PyArray_Type, &eq_dihedral,

&PyArray_Type, &spring_k,

&ndihedrals))

return NULL;


int* i = (int*)atom_index_1->data;

int* j = (int*)atom_index_2->data;

int* k = (int*)atom_index_3->data;

int* l = (int*)atom_index_4->data;

int* n = (int*)periodicity->data;

double* phi = (double*)eq_dihedral->data;

double* k_eq = (double*)spring_k->data;


OpenMM_PeriodicTorsionForce* dihedrals = OpenMM_PeriodicTorsionForce_create();

printf("Adding dihedrals

n");

int iter;

for (iter = 0; iter < ndihedrals; iter++) {

OpenMM_PeriodicTorsionForce_addTorsion(dihedrals, i[iter], j[iter], k[iter], l[iter], n[iter],
```

```
phi[iter], k_eq[iter]);
}

OpenMM_System_addForce(omm_system, (OpenMM_Force*)dihedrals);
return Py_None;
}

static PyObject* makeOmmEsAndLjForce(PyObject* dummy, PyObject* args) {
PyArrayObject* sigma;
PyArrayObject* epsilon;
PyArrayObject* charge;
int natoms;
double es_14_factor, lj_14_factor, cutoff;

if (!PyArg_ParseTuple(args, "O!O!O!dddi",
&PyArray_Type, &sigma,
&PyArray_Type, &epsilon,
&PyArray_Type, &charge,
&es_14_factor, &lj_14_factor,
&cutoff, &natoms))
return NULL;
```

```
double* s = (double*)sigma->data;

double* e = (double*)epsilon->data;

double* z = (double*)charge->data;


OpenMM_NonbondedForce* nonbond = OpenMM_NonbondedForce_create();

printf("Adding nonbonded

n");

int iter;

for (iter = 0; iter < natoms; iter++) {

OpenMM_NonbondedForce_addParticle(nonbond, z[iter], s[iter], e[iter]);

/*printf("help

n");

printf("atom %d:  charge %f, sigma %f, epsilon %f

n", iter, z[iter], s[iter], e[iter]);*/

}


OpenMM_NonbondedForce_createExceptionsFromBonds(nonbond, bonded_list, es_14_factor, lj_14_factor);


OpenMM_NonbondedForce_setNonbondedMethod(nonbond, OpenMM_NonbondedForce_Ewald); /*for step_0*/

/*OpenMM_NonbondedForce_setNonbondedMethod(nonbond, OpenMM_NonbondedForce_NoCutoff);*/ /*for

example*/
```

```
OpenMM_NonbondedForce_setCutoffDistance(nonbond, cutoff);

OpenMM_System_addForce(omm_system, (OpenMM_Force*)nonbond);

return Py_None;

}


static PyMethodDef langevin_methods[] = {

{"integrateLD", integrateLD, 1},

{"initOpenMM", initOpenMM, 1},

{"destroyOpenMM", destroyOpenMM, 1},

{"addCMRemover", addCMRemover, 1},

{"makeOmmBondedForce", makeOmmBondedForce, 1},

{"makeOmmAngleForce", makeOmmAngleForce, 1},

{"makeOmmDihedralForce", makeOmmDihedralForce, 1},

{"makeOmmEsAndLjForce", makeOmmEsAndLjForce, 1},

{NULL, NULL} /* sentinel */

};


/* Initialization function for the module */

void

initMMTK_langevin()

{

PyObject *m, *dict;
```

```
PyObject *universe, *trajectory, *forcefield, *units;


/* Create the module and add the functions */

m = Py_InitModule("MMTK_langevin", langevin_methods);

dict = PyModule_GetDict(m);


/* Import the array module */

import_array();


/* Import MMTK modules */

import_MMTK_universe();

import_MMTK_forcefield();

import_MMTK_trajectory();


/* OpenMM library initialization */

OpenMM_StringArray* pluginList = OpenMM_Platform_loadPluginsFromDirectory(

OpenMM_Platform_getDefaultPluginsDirectory());

int num_plugins = OpenMM_StringArray_getSize(pluginList);

int i;

for (i = 0; i < num_plugins; i++) {

printf("loaded plugin %s

n", OpenMM_StringArray_get(pluginList, i));
```

```
}
OpenMM_StringArray_destroy(pluginList);


/* Get the Boltzman constant factor from MMTK.Units */
units = PyImport_ImportModule("MMTK.Units");
if (units != NULL) {
PyObject *module_dict = PyModule_GetDict(units);
PyObject *factor = PyDict_GetItemString(module_dict, "k_B");
kB = PyFloat_AsDouble(factor);
}


/* Check for errors */
if (PyErr_Occurred())
Py_FatalError("can't initialize module MMTK_langevin");
}
```