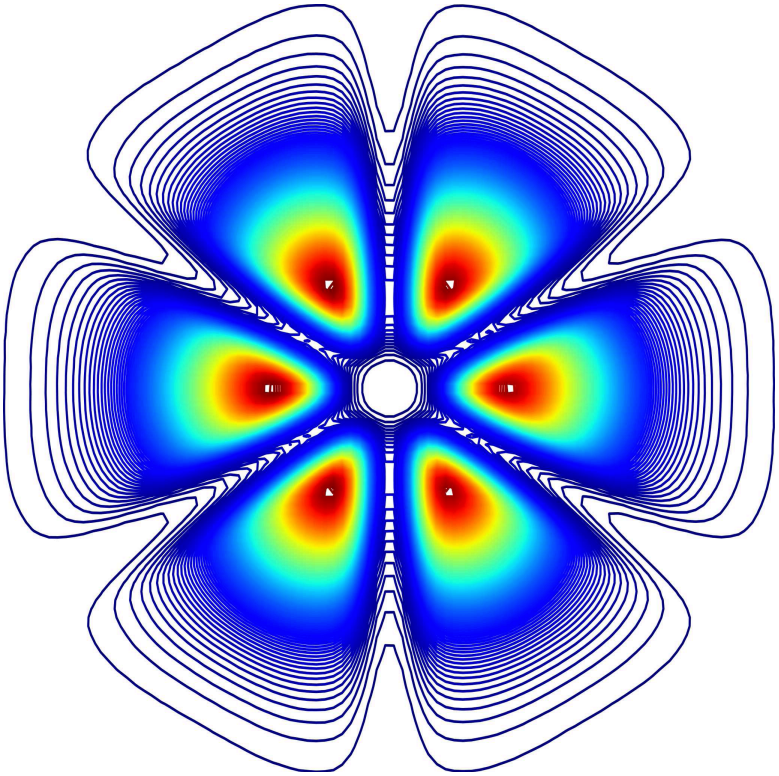
CHALMERS



GPU Implementation of the Feynman Path-Integral Method in Quantum Mechanics

Bachelor of Science Thesis for the Engineering Physics Programme

OLOF AHLÉN, GUSTAV BOHLIN, KRISTOFFER CARLSSON,
MARTIN GREN, PATRIC HOLMVALL, PETTER SÄTERSKOG

Department of Fundamental Physics
Division of Subatomic Physics
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden, 2011

Bachelor Thesis No. FUFX02-11-01

GPU Implementation of the Feynman Path-Integral Method in Quantum Mechanics.

Olof Ahlén[a], Gustav Bohlin[b], Kristoffer Carlsson[c], Martin Gren[d], Patric Holmvall[e], Petter Säterskog[f]

Email:
[a]aolof@student.chalmers.se
[b]bohling@student.chalmers.se
[c]kricarl@student.chalmers.se
[d]gmartin@student.chalmers.se
[e]holmvall@student.chalmers.se
[f]spetter@student.chalmers.se

FUFX02 - Bachelor thesis at Fundamental Physics
Bachelor Thesis No. FUFX02-11-01

Supervisor: Christian Forssén, Division of Subatomic Physics
Examiner: Christian Forssén

Department of Fundamental Physics
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
+46 (31) 772 1000

Printed by Chalmers reproservice
Göteborg, Sweden 2011

Cover: Probability distribution of six bosons calculated with the Monte Carlo Path Integral method. The interaction potential consists of a short-range repulsion and a mid-range attraction modelled by Gaussian functions. See Chapter 8 for details.

Abstract

The Path-Integral Formulation of Quantum Mechanics is introduced along with a detailed mathematical description of how it is used in quantum computations. The important concept of the kernel is explained, along with the free particle and harmonic oscillator as examples. Furthermore, the method for calculating expectation values of quantum operators is explained.

The expectation values are naturally calculated by importance sampled Monte Carlo integration and by use of the Metropolis algorithm. This is due to the discretization of the path integral results in an integral with a high number of integration variables. The mathematical concepts of this calculation are explained. Also, a method for obtaining the probability density of the treated system is presented.

The calculations are performed by a GPU, due to its high capabilities for numerical operations. This requires the mathematical computations to be parallelized and is done by use of the free software PyOpenCL. A thorough introduction to these concepts are given.

The resulting ground state energies and probability densities for many particle systems interacting with harmonic as well as attractive and repulsive gaussian potentials are presented. The calculations worked exceedingly well for many particle systems.

Source code is available at `https://sourceforge.net/projects/feynmangpu/files/`

Contents

1 Introduction

In 1948 R. P. Feynman presented a new way to describe quantum mechanics mathematically in his famous paper *The Space-Time Formulation of Non-relativistic Quantum Mechanics* [1]. This is called the Path-Integral Formalism. Prior to this achievement two different approaches to quantum mechanics existed: These where the Schrödinger equation and Heisenberg's matrix algebra. Feynman's aim with his new approach was to make the connection between quantum mechanics and classical mechanics more clear than the other two approaches did. The general idea of Feynman's approach is that all possible paths in space and time, which starts and ends at fixed points, for a system adds up with specific probability amplitudes. The sum over the probability amplitudes for all paths gives the total probability amplitude for the system. The absolute square of the probability amplitude gives the probability distribution.

The path-integral formalism describes how to perform the sum over all paths. The method to find a probability amplitude results in an integral over an infinite number of variables. In numerical approximations, this results in a high dimensional integral. Therefore, even simple problems like a harmonic oscillator in one dimension, becomes a rather long computation. As we see in this report the benefit of path-integral formalism appears when the number of degrees of freedom is high. For example, path integrals have been used to investigate non-relativistic many-body physics such as helium superfluidity and Bose condensation [3]. Even more important is the use of path integrals for solving quantum field equations. In particular, K. G. Wilson formulated a discretized version of Quantum Chromodynamics [4] that he used to simulate quark confinement. Today, this area of research is very active [5].

1.1 Purpose

Our main purpose is to use the path-integral formalism to find ground-state energies of different quantum-mechanical systems composed of one or many particles. We will restrict ourselves to the study of systems with distinguishable particles and avoid the additional difficulties of treating boson and (in particular) fermion statistics, see [6]. We will also find probability distributions using path integrals.

A specific aim has been that the numerical calculations presented in this work should be tractable on a desktop computer. The design of computational devices in the more recent years has shifted from increasing clock speeds to increasing the number of cores performing calculations. The computational unit on a graphics card, the GPU, is designed with a very large amount of computational cores and is thus capable of performing many more calculations per second than a modern CPU. It is therefore very interesting to explore the possibility of performing the calculations on a GPU. Utilizing a GPU as the computational device requires one to parallelize the computations that are to be performed, which in essence means

that many path integrals will be computed simultaneously, and will prove to be a challenge. We will verify our computational algorithm by solving various examples analytically and comparing the analytical and numerical results.

1.2 Method

First we introduce Feynman's path-integral formalism of quantum mechanics. Our main source of theoretical background on path integrals is Feynman's lecture notes written by A. R. Hibbs [2]. We then theoretically derive expressions involving path integrals to calculate the expectation value of operators (in particular the Hamiltonian) and the probability density for the ground state. These expressions are high-dimensional integrals of a form that can be evaluated efficiently using Monte-Carlo methods.

1.3 Reading Guide

The first section, *Theory*, focuses on the theory of the path-integral formalism. The path-integral formalism is motivated by the double-slit thought experiment and the method to calculate probability amplitudes for each path is introduced. We define the path integrals and introduce important concepts such as the least action principle and Euclidean time. We also theoretically derive expressions involving path integrals to calculate the expectation value of the energy and the probability density for the ground state. The Quantum Virial Theorem is proved. The reader is assumed to have basic knowledge of quantum mechanics and to understand the bra-ket notation.

In the next section, *Analytical Examples*, we present some analytical solutions for the energy and probability density for some simple quantum-mechanical systems. In particular we study a free particle, a particle in a harmonic potential and several particles with harmonic interaction. The Jacobi coordinates, that are useful to describe few body systems, are also introduced.

Next, in *Numerical Methods*, we introduce numerical Monte Carlo methods and sketch the outline of an algorithm to evaluate the ground-state energies and probability densities.

The Monte Carlo methods introduced in the above section will be implemented in OpenCL for performing the calculations on a GPU. This is done in two steps. First, in the section *Implementation in C*, a C-program is presented that performs the calculations outlined in the Numerical Methods-section on a CPU. Since C is such a widely used programming language for calculations, this should be a natural way of introducing how the path integral formalism is used in computer calculations. The transition to OpenCL is then made by first discussing some important concepts of parallel programming and going into detail about the functionality of the Python implementation OpenCL, *PyOpenCL*, in the section *Parallelization and GPU programming*. After that, the source code for the parallelized compu-

tation is explained in section *Implementation in OpenCL*. Here, the C-program presented earlier can be recycled with only minor modifications.

In the *Result* section, we present some different results from our program. Here are also comparisons with the analytical methods presented.

2 Theory

In this section, the theory behind the path-integral formalism will be introduced. There will be a thought experiment that motivates the introduction of the path-integral formalism. Important concepts such as the least-action principle and Euclidean time will be introduced. Expressions involving path integrals to calculate the energy and the probability density for the ground state will be derived.

2.1 Feynman's Approach To Quantum Mechanics

In the standard approach to quantum mechanics the probability amplitude (wave function) for a quantum mechanical system is given by the Schrödinger equation. This Schrödinger viewpoint focuses on the wave properties of quantum mechanical particles. It is not very well connected with classical mechanics since classical particles are not described as waves, but rather as point-like entities. Feynman introduces his path integral formalism in order to use more of the classical point-like entity properties of quantum mechanical particles to calculate the probability amplitude [1]. He also wanted classical mechanics to emerge in the limit $h \to 0$, where h is the Planck constant.

2.2 The Double Slit Experiment

The double-slit experiment is an important experiment that shows the difference between classical mechanics and quantum mechanics [2]. Moreover, it gives a conceptual motivation for introducing the path integral formalism.

Consider a source S of almost monoenergetic particles placed at A, see figure 1. The particles could be protons, electrons etc. The particles come out in all directions. The flux of particles is measured on a screen placed at B. A screen with two slits, that can be opened and closed, is placed in between A and B at position C. When the first slit is opened and the second slit is closed a flux P_1 is measured, when the second slit is opened and the first slit is closed a flux P_2 is measured, and when both are opened a flux P is measured. The flux at a specific position on the detector represents the probability density for the particles to reach the screen at that position.

In classical mechanics each particle is expected to go through either slit 1 or slit 2 since these are excluding events. Therefore, the fluxes at C are expected to satisfy the relation $P_1 + P_2 = P$. Experimental data tells us that this is not the case. The general result is $P_1 + P_2 + P_{\text{interference}} = P$, where $P_{\text{interference}}$ corresponds to the interference between two waves, ϕ_1 and ϕ_2, that goes through slits 1 and 2 respectively according to

$$P = |\phi_1 + \phi_2|^2 = \underbrace{|\phi_1|^2}_{P_1} + \underbrace{|\phi_2|^2}_{P_2} + \underbrace{2\text{Re}(\phi_1^* \phi_2)}_{P_{\text{interference}}} . \tag{2.1}$$

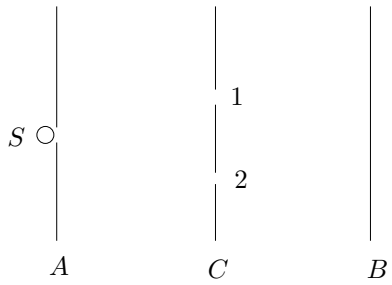


Figure 1: Double-slit experiment set-up

The result is the same as a double slit experiment with light where the "particles" are photons. Moreover, the interference holds even when the intensity of the source is so low that the detector records pulses, at specific positions on the screen, that are separated by gaps in time during which nothing happens. This represents individual particles arriving at the detector and it is the reason why for example electrons are referred to as particles. Still, because of the interference it seems like the particle is a de-localized object that passes through both slits. If an attempt is done to detect through which slit the particle went through the result is that it either goes through slit 1 or 2. However, when performing this intermediate observation at C the interference at B vanishes. This particle/wave duality seems contradictory and it require some explanation.

The flux of particles represents the probability for the particles to reach the detector. The intensity $|\phi|^2$ should therefore be interpreted as the probability density for the position of the particles. It is the probability amplitude ϕ that adds up, not the density. Heisenberg's quantum uncertainty principle says that any determination of the alternative taken by a process capable of following more than one alternative destroys the interference between the alternatives. Therefore, when an attempt to observe which slit the particle went through, the interference between the two options disappear.

The double-slit experiment tells us that the probability amplitude for an event, here particles arriving at the detector, is the sum of the probability amplitudes for which the event can happen. This result can be used to get the probability amplitude for the particles to go from the source to the detector, and this will be explained below.

Imagine that more holes are drilled in the screen at C until there are N slits in the screen at positions $y_C(i)$, where i goes from 1 to N, see figure 2. From the results above one can see that the total probability density is given by

$$\Phi = \sum_{i=1}^{N} \phi(y_C(i)) \ , \tag{2.2}$$

where $\phi(y_C(i))$ is the amplitude for the wave going through the slit at $y_C(i)$. If

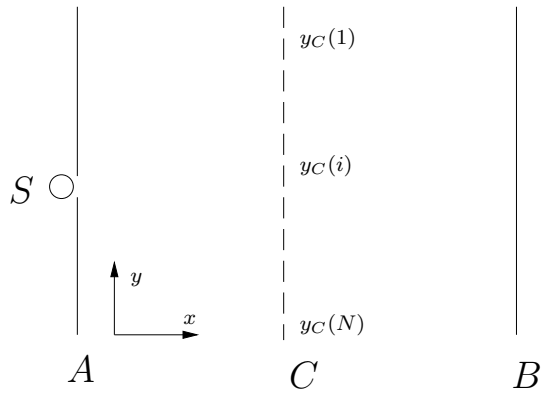


Figure 2: Set-up for thought experiment with N slits

the number of holes goes toward infinity, so that they fill the whole screen, the screen becomes a purely fictive device. The sum then transforms into an integral over all values for y_C

$$\Phi = \int dy_C \phi(y_C) \ . \tag{2.3}$$

Now imagine the case with M number of screens between A and B, placed at x_j and each with an infinite number of slits. The source and the detector are placed at x_A and x_B respectively. The total probability amplitude is then given by

$$\Phi = \int \prod_{j=1}^{M} (dy_j)\, \phi(y_1, x_1; y_2, x_2; \ldots y_M, x_M) \ , \tag{2.4}$$

where $\phi(y_1, x_1; y_2, x_2; \ldots y_M, x_M)$ is the probability amplitude for the particle to go through the screens at specific values of x_j, y_J.

If the number of screens between A and B goes towards infinity so that the distance between two screens are infinitesimally small a specific set of (y_i, x_i) represent one single path the particle can take between A and B. This is only true if the motion along the x-axis is not restricted in the direction from x_A to x_B as before, but allowed to be in both the positive and the negative directions. A path between A and B can be parametrised with respect to time as $\mathbf{x}(t) = (x(t), y(t))$ where each point along the path has a given x and y value for a given time t see figure 3. The time goes from t_A to t_B for all paths.

The total probability amplitude is thus

$$\Phi = \sum_{\substack{\text{all paths} \\ \text{from A to B}}} \phi[\mathbf{x}(t)] \ , \tag{2.5}$$

where $\phi[\mathbf{x}(t)]$ is the probability amplitude for the given path. Since all of these screens are purely fictive, this represent the probability amplitude for a particle to

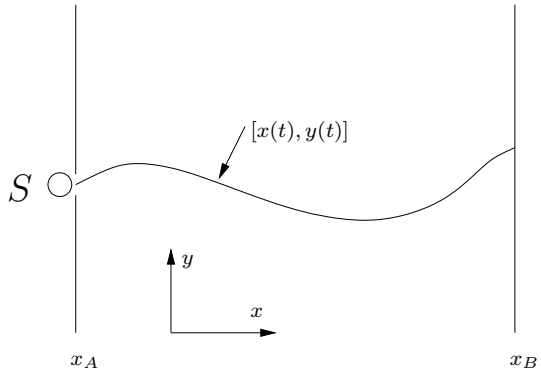


Figure 3: A complete path between A and B

go from A to B in the time interval $t_B - t_A$ when there are no screens in between A and B. This gives an alternative way of calculating the total probability amplitude that emphasizes the particle properties of a particle rather than the wave properties as in the Schrödinger approach.

The path integral formalism gives a method of calculating the probability amplitude for each path and how to make the sum over all paths.

2.3 The Least-action Principle

To make things easier to understand the path integral formalism will be introduced with one particle in one space dimension x. The transition to more particles and more space dimensions is not hard and it will be shown later in section 2.5.1. An event for a particle is its position x at a given time t. The event (x_i, t_i) will be referred to as event i. In classical mechanics there is only one trajectory a particle can take between two events [2]. In this section, an equation to find the classical path will be introduced. In classical mechanics the trajectory is defined by the principle of least action[1]. The action S is given by the expression

$$S[x(t)] = \int_{t_a}^{t_b} L(\dot{x}, x, t)dt \ , \tag{2.6}$$

where L is the Lagrangian for the system. The Lagrangian is given by

$$L = T - V \tag{2.7}$$

where T is the kinetic energy and V is the potential energy. The condition that the action should have an extremum gives the equation

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{x}}\right) - \frac{\partial L}{\partial x} = 0 \ . \tag{2.8}$$

[1]The real condition is actually that the action should have an extremum and not necessarily a minimum.

This is the classical Lagrangian equation of motion. A proof that the least action gives the Lagrangian equation of motion is found in [2]. The solution is named the classical path and is here denoted $\bar{x}(t)$. The quantum of action is the Planck constant h, and this is important when the probability amplitude for each trajectory is introduced.

2.4 The Probability Amplitude for Each Trajectory

In this section, the form of the amplitude for each trajectory shall be given.

As said before, classical mechanics should emerge when the quantum of action $h \to 0$. In classical mechanics only the classical path, with an extremum in action, should contribute to the probability amplitude. Feynman postulated that the magnitude of the probability amplitude for each trajectory is the same, but they have different phases [1]. The phase given in radians is the action in units of the quantum of action times 2π. The probability amplitude from each path is then expressed as

$$\phi[x(t)] = C \cdot e^{i2\pi \frac{S[x(t)]}{h}} = C \cdot e^{i \frac{S[x(t)]}{\hbar}} \ , \tag{2.9}$$

and the total probability amplitude $K(b,a)$ to go from event a to event b can be expressed as

$$K(b,a) = \sum_{\substack{\text{all paths} \\ \text{from a to b}}} \phi[x(t)] = C \cdot \sum_{\substack{\text{all paths} \\ \text{from a to b}}} e^{i \frac{S[x(t)]}{\hbar}} \ . \tag{2.10}$$

The constant C is chosen to normalize K correctly.

2.4.1 The Classical Limit

Now an argument that the classical limit $h \to 0$ works for this choice of probability amplitude will be given. For larger systems a small change in the trajectory corresponds to a relatively large change in the action in comparison to h and thus a big change in phase [2]. Because the phase changes rapidly, the contributions from paths other than the classical path tend to cancel each other. Because of this one can let $h \to 0$ so that $S[x(t)] \gg h$ to get classical mechanics. For small systems the action will take small values. Therefore, small changes in the trajectory give relative small change in the action in comparison to h and thus a small change in phase. Because there is only small changes in phase for trajectories close to the classical path these trajectories give a net contribution to the total probability amplitude.

2.4.2 Probability

The probability P to go from a to b is related to the probability amplitude by

$$P(b,a) = |K(b,a)|^2 \ . \tag{2.11}$$

2.5 Path Integrals

In this section, the path integrals will be defined. The basic idea to sum the contributions from all paths is quite simple, but a more precise mathematical definition of such a sum must be given. Because the number of paths is a high order of infinity it is not evident what measure the space of paths should be given [2]. The path integrals are analogous to the Riemann integral. In Riemann integration the area A under a curve is approximated by using a subset $\{x_i\}$ of all points in the interval over which the area is calculated. The approximation is the sum of the given function's values taken at evenly spaced points x_i times a normalization factor that depends upon the space between the points. The normalization factor ensures that the sum approaches the right value when the spacing between the points becomes smaller. For Riemann integration the normalization factor is just the space between the points, referred to as h, and thus

$$A = \lim_{h\to 0} h \sum_i f(x_i) \ . \tag{2.12}$$

An analogous procedure can be followed when defining the sum over all paths. In order to construct the path integrals a subset of all paths must be found. Divide the independent variable time into N steps of width ϵ which gives a set of values t_i evenly spaced between t_a and t_b. For each time t_i there is a position x_i. For each path there is a specific set of x_i, t_i. (x_0, t_0) is the starting point (x_a, t_a), and (x_N, t_N) is the end point (x_b, t_b). A path is constructed by connecting all the points selected according to this, see figure 4. One can define a sum over all paths constructed in this manner by taking multiple integrals over all values of x_i for i between 1 and $N-1$. The integrals should be multiplied with a normalization factor that ensures that the path integral approaches the right value when the time step ϵ becomes smaller. The normalization factor is a function of ϵ and it is therefore denoted $N(\epsilon)$. The integration does not go over x_0 and x_N because these are fixed end points. If the magnitude C of the probability amplitude is collected in $N(\epsilon)$ the Kernel can be written as

$$\begin{aligned} K(b,a) &= C \cdot \sum_{\substack{\text{all paths} \\ \text{from a to b}}} e^{i\frac{S[x(t)]}{\hbar}} \\ &= \lim_{\epsilon\to 0} N(\epsilon) \cdot \int e^{i\frac{S[x(t)]}{\hbar}} dx_1 \dots dx_{N-1} \ . \end{aligned} \tag{2.13}$$

Unlike the case of Riemann integrals it is hard to find a general expression for the normalization factor, but one can find definitions for many situations which have practical value. However, this leads into rather long calculations. The normalization factors in this paper are therefore taken directly from Feynman [2].

For a particle in a potential the Lagrangian has the form

$$L = \frac{m}{2}\dot{x}^2 - V(x,t) \ . \tag{2.14}$$

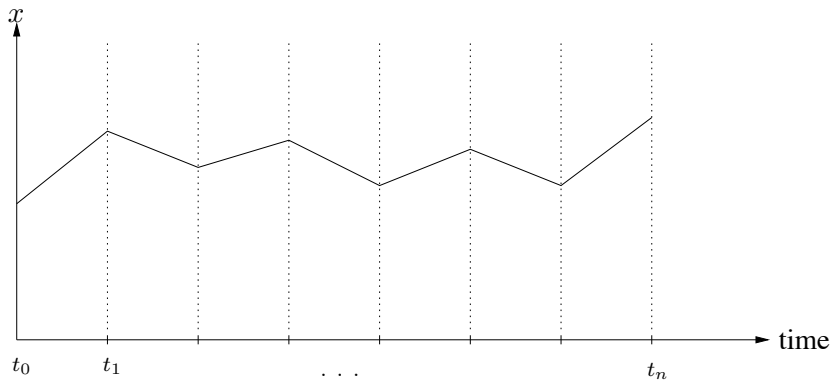


Figure 4: A discrete path

The normalization factor is then A^{-N}, where

$$A = \left(\frac{2\pi i\hbar\epsilon}{m}\right)^{\frac{1}{2}} \text{ [2].} \tag{2.15}$$

When the normalization factor is A^{-N} the probability amplitude can be expressed as

$$K(b,a) = \lim_{\epsilon\to 0}\frac{1}{A}\int \phi[x(t)]\frac{dx_1}{A}\ldots\frac{dx_{N-1}}{A}\ . \tag{2.16}$$

With the definition of $\phi[x(t)]$ this is

$$K(b,a) = \lim_{\epsilon\to 0}\frac{1}{A}\int e^{i\frac{S[x(t)]}{\hbar}}\frac{dx_1}{A}\ldots\frac{dx_{N-1}}{A}\ , \tag{2.17}$$

where $x(t)$ is the path constructed in the manner above. This is defined, in a more compact form, as

$$K(b,a) = \int e^{i\frac{S[x(t)]}{\hbar}}\mathcal{D}x(t)\ . \tag{2.18}$$

This is referred to as a path integral. $K(b,a)$ has many names: kernel, propagator, Greens function, time-evolution operator etc.

Feynman proved that this way of calculating the probability amplitude satisfies the time-dependent Schrödinger equation [1]. This proof will not be shown in this report.

2.5.1 Many Degrees of Freedom

The path integrals can be generalized into more degrees of freedom. The degrees of freedom grows when adding more particles as well as adding more space dimensions. x_i then becomes a vector $\mathbf{x_i}$ of dimension $d = m \cdot n$, for m particles in n space dimensions. The path is thus a vector valued function of time, describing all

the particles coordinates in all dimension. The integrals in the path integral are thus d-dimensional. Using vector notation the general path integral can be written as

$$K(b,a) = \int e^{i\frac{S[\mathbf{x}(t)]}{\hbar}} \mathcal{D}\mathbf{x}(t) \ . \tag{2.19}$$

This only works for many particles if the particles are distinguishable. If they are identical it is necessary to integrate over paths for the system where particles change places because the final states they lead to are the same states as when they don't change places.

2.5.2 The Composition Theorem

In this section, the composition theorem for events occurring in succession for the propagators shall be derived. Suppose there is an event c at point (x_c, t_c) where $t_a < t_c < t_b$ so that it is between the events a and b. The kernel $K(b,a)$ can be related to the kernels $K(b,c)$ and $K(c,a)$.

Since the action is an integral of the Lagrangian over time the action between a and b can be written as

$$S[b,a] = S[b,c] + S[c,a] \ . \tag{2.20}$$

According to equation (2.18) the kernel can be written as

$$K(b,a) = \int e^{i\frac{S[b,a]}{\hbar}} \mathcal{D}x(t) = \int e^{i\frac{S[b,c]+S[c,a]}{\hbar}} \mathcal{D}x(t) \ . \tag{2.21}$$

Now one can use the fact that any path can be split into two paths, and split the path at c. After this one can first integrate over all paths from a to c, then over all paths from c to b and finally integrate the result over all possible values of x_c. Because $S[b,c]$ is constant when integrating from a to c the result after the first integration can be written as

$$K(b,a) = \int_{-\infty}^{\infty} \int_{c}^{b} e^{i\frac{S[b,c]}{\hbar}} K(c,a) \mathcal{D}x(t) dx_c \ . \tag{2.22}$$

Now the integral between c and b is taken for a general x_c, which leaves only an integral over all possible values of x_c so that

$$K(b,a) = \int_{-\infty}^{\infty} K(b,c)K(c,a) dx_c \ . \tag{2.23}$$

This says that the probability amplitude to go from a to c and then to b is the probability amplitude to go from a to c times the probability amplitude to go from c to b. If this is extended by adding more events between a and b it leads to the conclusion that probability amplitudes for events occurring in succession in time multiply. This is called the composition theorem. If the process of adding

more events between a and b is continued so the time scale is divided into N equal intervals, $K(b,a)$ can be expressed as

$$K(b,a) = \int_{x_{N-1}} \dots \int_{x_2} \int_{x_1} K(b,N-1)K(N-1,N-2) \cdots K(j+1,j) \cdots K(1,a) dx_1 dx_2 \cdots dx_{N-1} \ . \tag{2.24}$$

With this result in hand it would be fortunate if one could find an approximation of the propagator for short time evolution and then multiply these and integrate to get an approximation of longer time propagators.

2.6 The Propagator for Short Times

In this section, an approximation for the short time propagator shall be derived. Let us start with equation (2.17). Since the action is an integral one can split it over time. The action between a and b can be expressed as the sum of the actions in each time interval (t_j, t_{j+1})

$$S[b,a] = \sum_{j=0}^{N-1} S[j+1,j] \ . \tag{2.25}$$

If this is put into equation (2.17) one gets

$$K(b,a) = \lim_{\epsilon \to 0} \frac{1}{A} \int e^{i \frac{\sum_{j=0}^{N-1} S[j+1,j]}{\hbar}} \frac{dx_1}{A} \dots \frac{dx_{N-1}}{A} = \tag{2.26}$$
$$= \lim_{\epsilon \to 0} \int \prod_{j=0}^{N-1} \frac{e^{i \frac{S[j+1,j]}{\hbar}}}{A} dx_1 \dots dx_{N-1} \ .$$

where $\epsilon = t_{j+1} - t_j$. If this is compared to equation (2.24) one can identify under the limit $\epsilon \to 0$ that

$$K(j+1,j) = \frac{e^{i \frac{S[j+1,j]}{\hbar}}}{A} \ . \tag{2.27}$$

Since ϵ is small, it is interesting to see how the action grows with ϵ. Therefore, expand the action in a Taylor series

$$S[x(t)] = \int_{t_j}^{t_j+\epsilon} L dt = 0 + L(j)\epsilon + O(\epsilon^2) \ . \tag{2.28}$$

This tells us that to the first order of ϵ the action along any path grows as the Lagrangian at the starting point times ϵ. This means that the action can be replaced by $L(j)\epsilon$ in equation (2.27)

$$\lim_{\epsilon \to 0} K(j+1,j) = \lim_{\epsilon \to 0} \frac{e^{i \frac{L(j)\epsilon}{\hbar}}}{A} \ . \tag{2.29}$$

The short-time evolution propagator is therefore approximated as

$$K(j+1,j) \approx \frac{e^{i\frac{L(j)\epsilon}{\hbar}}}{A} \ . \tag{2.30}$$

This method of approximating the propagator requires that the value of the Lagrangian is known at each point along the path. The method of making the path described in section 2.5 makes the velocity discontinuous at each point along the path. It would therefore be better to take the value of the Lagrangian at the midpoint of each time interval where the velocity is given exactly by $\frac{x_{j+1}-x_j}{\epsilon}$. This procedure gives the same result as above when the limit $\epsilon \to 0$ is taken, see equation (2.29). It can therefore also be used to approximate the short time evolution operator. If the potential energy is only dependent on the position and the time the approximation for the action between j and $j+1$ is given by

$$S[j+1,j] \approx L\left(x = \frac{x_{j+1}+x_j}{2},\ \dot{x} = \frac{x_j - x_{j+1}}{\epsilon},\ t = \left(\frac{t_j + t_{j+1}}{2}\right)\epsilon\right)\epsilon \ . \tag{2.31}$$

The long time propagator is thus

$$\begin{aligned} K(b,a) = \int_{x_{N-1}} \cdots \int_{x_2}\int_{x_1} K(b,N-1)K(N-1,N-2) \\ \cdots K(j+1,j)\cdots K(1,a)dx_1 dx_2 \cdots dx_{N-1} \ , \end{aligned} \tag{2.32}$$

where

$$K(j+1,j) = \frac{e^{\frac{i}{\hbar}L\left(x=\frac{x_{j+1}+x_j}{2},\ \dot{x}=\frac{x_{j+1}-x_j}{\Delta t},\ t=\left(j+\frac{1}{2}\right)\epsilon\right)\epsilon}}{A} \ . \tag{2.33}$$

A more formal proof for the short-time evolution operator is found in *Path Integrals in Quantum Mechanics* by J. Zinn-Justin [6].

2.7 Euclidean Time

The exponents in the expressions for the kernel are imaginary and thus the kernel is non-real. It will make calculation on computer much easier if the kernel was real. As will be shown later it also makes the excited energy states decay and is therefore essential when calculating the energy and probability distribution of the ground state. Here follows a method that makes the kernel real [14]. If the Lagrangian is expressed in Euclidean time $\tau = it$ one gets

$$\begin{aligned} L\left(x,\frac{dx}{dt},t\right) =& L\left(x,-\frac{dx}{id\tau},-i\tau\right) = \frac{m}{2}\left(-\frac{dx}{id\tau}\right)^2 - V(x,-i\tau) \\ =& -\frac{m}{2}\left(\frac{dx}{d\tau}\right)^2 - V(x,-i\tau) = -T - V = -H \ . \end{aligned} \tag{2.34}$$

Therefore, the Lagrangian becomes the Hamiltonian when Euclidean time is introduced. By a simple variable substitution the kernel becomes

$$K(b,a) = \int e^{-\frac{\int_a^b H d\tau}{\hbar}} \mathcal{D}x(t) = \int e^{-\frac{S_E[b,a]}{\hbar}} \mathcal{D}x(t) \ , \tag{2.35}$$

where the $S_E[b,a]$ is the action expressed in Euclidean time. If the Euler approximation of the action is expressed in Euclidean time one gets

$$\begin{aligned} S_E[j+1,j] \approx & \quad H(j+1,j)\epsilon \\ = & \quad H\left(x = \tfrac{x_{j+1}+x_j}{2},\ \dot{x} = \tfrac{x_{j+1}-x_j}{\epsilon},\ t = -\left(j+\tfrac{1}{2}\right) i\epsilon\right)\epsilon \ . \end{aligned} \tag{2.36}$$

$K(j+1,j)$ can then be approximated by

$$K(j+1,j) \approx \frac{1}{A} e^{-\frac{H(j+1,j)\epsilon}{\hbar}} \ . \tag{2.37}$$

An approximation for the long-time propagator in Euclidean time is thus

$$K(b,a) = \int \prod_{j=0}^{N-1} \frac{1}{A} e^{-\frac{H(j+1,j)\epsilon}{\hbar}} dx_1 dx_2 \cdots dx_{N-1} \ . \tag{2.38}$$

These propagators are real and they will be used in calculations.

2.8 The Schrödinger and the Heisenberg Pictures

In the next sections the Dirac bra-ket notation will be used. An introduction to the bra-ket notation can be found in *Modern Quantum Mechanics* by J.J. Sakurai [7]. A time-dependent state $|\alpha\rangle$ at a time t will be denoted as $|\alpha, t\rangle$. The time-evolution operator $\hat{U}(t_b, t_a)$ for a system evolves the states from an initial time t_a to a final time t_b. For a system with a time independent Hamilton operator $\hat{H}$ the time-evolution operator is given by

$$\hat{U}(t_b, t_a) = \exp\left(-i\frac{\hat{H}(t_b - t_a)}{\hbar}\right) \ [7] \ . \tag{2.39}$$

and thus

$$|\alpha, t_b\rangle = \exp\left(-i\frac{\hat{H}(t_b - t_a)}{\hbar}\right) |\alpha, t_a\rangle \ . \tag{2.40}$$

Often the initial time $t_a \equiv 0$ and the final time is just called t. The time evolution operator then becomes

$$\hat{U}(t) = \exp\left(-i\frac{\hat{H}t}{\hbar}\right) \ . \tag{2.41}$$

This way of evolving the states with the time-evolution operator is referred to as the Schrödinger picture. There is another way of describing time evolution of

systems. Instead of evolving the state the operator is evolved in time. This is known as the Heisenberg picture. A time-depending operator $\hat{A}(t)$ can be written as

$$\hat{A}(t) = \hat{U}^{\dagger}(t)\hat{A}(t=0)\hat{U}(t) \text{ [7]} . \tag{2.42}$$

This is equivalent to

$$\hat{A}(t) = \exp\left(i\frac{\hat{H}t}{\hbar}\right)\hat{A}(t=0)\exp\left(-i\frac{\hat{H}t}{\hbar}\right) . \tag{2.43}$$

In the next section the Schrödinger and Heisenberg pictures will be used to derive expressions for expectation values for operators using path integrals.

2.9 Expectation Values of Operators in Path Integrals

Path integrals can be used to calculate expectation values of operators. In this section, the procedure of this will be shown. First consider an operator $\hat{A}$. The operator has eigenvalues $A(x)$ to the position eigenstates[2] according to

$$\hat{A}|x\rangle = A(x)|x\rangle . \tag{2.44}$$

The position eigenstates can have a time dependence, as they do along a path. The operator $\hat{A}$ can therefore be denoted as $\hat{A}(\hat{x}(t))$. Now consider the matrix element A_{ba}

$$A_{ba} = \langle x_b, t_b|\hat{A}(\hat{x}(t_1))|x_a, t_a\rangle . \tag{2.45}$$

With the definition of the time-evolution operator, equation (2.40), this can be written as

$$A_{ba} = \langle x_b|e^{-i\frac{\hat{H}t_b}{\hbar}}\hat{A}(\hat{x}(t_1))e^{i\frac{\hat{H}t_a}{\hbar}}|x_a\rangle . \tag{2.46}$$

The operator can be rewritten using the Heisenberg picture definition, equation (2.43), and the matrix element becomes

$$A_{ba} = \langle x_b|e^{-i\frac{\hat{H}(t_b-t_1)}{\hbar}}\hat{A}e^{-i\frac{\hat{H}(t_1-t_a)}{\hbar}}|x_a\rangle . \tag{2.47}$$

Use the continuous identity operator for the position x_1 at time t_1 and the eigenvalues $A(x_1)$ to get

$$A_{ba} = \int dx_1 \underbrace{\langle x_b|e^{-i\frac{\hat{H}(t_b-t_1)}{\hbar}}|x_1\rangle}_{K(x_b,t_b;x_1,t_1)} A(x_1) \underbrace{\langle x_1|e^{-i\frac{\hat{H}(t_1-t_a)}{\hbar}}|x_a\rangle}_{K(x_1,t_1;x_a,t_a)} . \tag{2.48}$$

Here two propagators can be identified and because of the composition theorem this is equal to

$$A_{ba} = \int \mathcal{D}x(t)e^{i\frac{S[x(t)]}{\hbar}}A(x(t_1)) . \tag{2.49}$$

[2] $\hat{x}|x\rangle = x|x\rangle$

Now express A_{ba} using Euclidean time instead and set $(t_b - t_a)i \equiv \beta_{ba}$ and $t_a \equiv 0$ to get

$$A_{ba}(\beta_{ba}) = \langle x_b, \beta_{ba}|\hat{A}(\hat{x}(\tau_1))|x_a, 0\rangle \ . \tag{2.50}$$

With the same procedure as above one gets

$$A_{ba}(\beta_{ba}) = \int dx_1 \langle x_b|e^{-\frac{\hat{H}\beta_{b1}}{\hbar}}|x_1\rangle A(x_1)\langle x_1|e^{-\frac{\hat{H}\beta_{1a}}{\hbar}}|x_a\rangle \tag{2.51}$$

where the notations $\beta_{b1} \equiv \tau_b - \tau_1$ and $\beta_{1a} \equiv \tau_1 - \tau_a$ have been introduced.

The time-independent Schrödinger equation gives a spectrum of energy eigenstates $|\psi_n\rangle$ with energy eigenvalues E_n according to

$$\hat{H}|\psi_n\rangle = E_n|\psi_n\rangle \ . \tag{2.52}$$

Because the Hamiltonian is hermitian the energy eigenstates $|\psi_n\rangle$ form a complete orthonormal basis. Express $A_{ba}(\beta_{ba})$ in the energy eigenstates using the discrete identity operator from equation (A.1)

$$\begin{aligned} A_{ba}(\beta_{ba}) = \int dx_1 \left(\sum_n \langle x_b|e^{-\frac{\hat{H}\beta_{b1}}{\hbar}}|\psi_n\rangle\langle\psi_n|x_1\rangle\right) A(x_1) \\ \cdot \left(\sum_m \langle x_1|e^{-\frac{\hat{H}\beta_{1a}}{\hbar}}|\psi_m\rangle\langle\psi_m|x_a\rangle\right) \ . \end{aligned} \tag{2.53}$$

Now the exponents can be expanded into a Taylor series

$$e^{-\frac{\hat{H}\beta}{\hbar}} = 1 - \frac{\hat{H}\beta}{\hbar} + \left(\frac{\hat{H}\beta}{\hbar}\right)^2 - \left(\frac{\hat{H}\beta}{\hbar}\right)^3 \ldots \ . \tag{2.54}$$

For each term in the sums above the Hamilton operators can be interchanged with the energy eigenvalues for the given eigenstates in the term. This means that the Taylor expansion can be reversed using the energy eigenvalue instead of the Hamiltonian. The result is

$$\begin{aligned} A_{ba}(\beta_{ba}) = \int dx_1 \left(\sum_n \langle x_b|\psi_n\rangle\langle\psi_n|x_1\rangle e^{-\frac{E_n\beta_{b1}}{\hbar}}\right) A(x_1) \\ \cdot \left(\sum_m \langle x_1|\psi_m\rangle\langle\psi_m|x_a\rangle e^{-\frac{E_m\beta_{1a}}{\hbar}}\right) \ . \end{aligned} \tag{2.55}$$

Now take the trace, i.e. set $x \equiv x_a \equiv x_b$ and integrate over all values for x

$$\begin{aligned} A_{tr}(\beta_{ba}) &= \int dx_b dx_a \delta(x_b - x_a) A_{ba}(\beta) \qquad (2.56) \\ &= \int dx_1 \underbrace{\int dx \sum_{n,m} \langle x|\psi_n\rangle\langle\psi_m|x\rangle}_{\delta_{mn}} \langle\psi_n|x_1\rangle A(x_1)\langle x_1|\psi_m\rangle e^{-\frac{E_n\beta_{b1}}{\hbar}} e^{-\frac{E_m\beta_{1a}}{\hbar}} \ . \end{aligned}$$

The Kronecker δ-function can be identified because of the orthogonality between eigenstates, see equation (A.5). This reduces the sum over both n and m to a single sum over n

$$A_{tr}(\beta_{ba}) = \int dx_1 \sum_n \langle\psi_n|x_n\rangle A(x_1)\langle x_1|\psi_n\rangle \ .e^{-\frac{E_n\beta_{ba}}{\hbar}} \tag{2.57}$$

The integral can be put inside the sum and there one can identify the continuous identity operator, see equation (A.2), for the position x_1 acting on $\hat{A}$ and thus

$$A_{tr}(\beta_{ba}) = \sum_n \langle\psi_n|\hat{A}|\psi_n\rangle e^{-\frac{E_n\beta_{ba}}{\hbar}} \ . \tag{2.58}$$

For the identity operator this is

$$Z_{tr}(\beta_{ba}) = \sum_n e^{-\frac{E_n\beta_{ba}}{\hbar}} \ . \tag{2.59}$$

This is often referred to as the partition function as in statistical physics. In statistical physics energy eigenstates are distributed according to the Boltzmann exponentials if the system is in thermal equilibrium [6]. The probability of finding a system in energy eigenstate $|\psi_n\rangle$ for a given temperature T is given by

$$P_n(T) = \frac{e^{-\frac{E_n}{k_BT}}}{\sum_n e^{-\frac{E_n}{k_BT}}} \ . \tag{2.60}$$

If one defines

$$\frac{\beta_{ba}}{\hbar} \equiv \frac{1}{k_BT} \tag{2.61}$$

then

$$\frac{A_{tr}(\beta_{ba})}{Z_{tr}(\beta_{ba})} = \sum_n \langle\psi_n|\hat{A}|\psi_n\rangle P_n(T) \ . \tag{2.62}$$

The introduction of Euclidean time has apparently lead us to statistical physics where the energy eigenstates are distributed according to the Boltzmann distribution $P_n(T)$. The sum above is therefore the expectation value of the operator $\hat{A}$ at temperature T

$$\frac{A_{tr}(\beta_{ba})}{Z_{tr}(\beta_{ba})} = \langle\psi|\hat{A}|\psi\rangle \ . \tag{2.63}$$

According to equation (2.49), $A(\beta_{ba})$ and $Z(\beta_{ba})$ can be expressed as

$$A_{tr}(\beta_{ba}) = \int_{-\infty}^{\infty} dx' \int_{x'}^{x'} \mathcal{D}x(t) e^{-\frac{S_E[x(t)]}{\hbar}} A(x(t_1)) \tag{2.64}$$

$$Z_{tr}(\beta_{ba}) = \int_{-\infty}^{\infty} dx' \int_{x'}^{x'} \mathcal{D}x(t) e^{-\frac{S_E[x(t)]}{\hbar}} \tag{2.65}$$

and thus

$$\langle\psi|\hat{A}|\psi\rangle = \frac{\int_{-\infty}^{\infty} dx' \int_{x'}^{x'} \mathcal{D}x(t) e^{-\frac{S_E[x]}{\hbar}} A(x(t_1))}{\int_{-\infty}^{\infty} dx' \int_{x'}^{x'} \mathcal{D}x(t) e^{-\frac{S_E[x(t)]}{\hbar}}} \ . \tag{2.66}$$

Here $x(t_1)$ represents a position at an arbitrary time along each path in the path integral.

2.9.1 Ground-state Expectation Values of Operators

This paper focuses on calculating the ground state energy and it is therefore necessary to find an expression for the expectation value on the ground state. In statistical physics only the ground state is populated when the temperature is zero. According to equation (2.61) $\beta_{ba} = \infty$ when $T = 0$. The expectation value of the operator on the ground state can therefore be expressed as

$$\langle\psi_0|\hat{A}|\psi_0\rangle = \lim_{\beta_{ba}\to\infty} \frac{A_{tr}(\beta_{ba})}{Z_{tr}(\beta_{ba})} \ . \tag{2.67}$$

This can easily be verified. From equations (2.58) and (2.59) one can see that

$$\lim_{\beta_{ba}\to\infty} A_{tr}(\beta_{ba}) = \langle\psi_0|\hat{A}|\psi_0\rangle e^{-\frac{E_0\beta_{ba}}{\hbar}} \tag{2.68}$$

and

$$\lim_{\beta_{ba}\to\infty} Z_{tr}(\beta_{ba}) = e^{-\frac{E_0\beta_{ba}}{\hbar}} \ . \tag{2.69}$$

The ratio between these gives the expectation value on the ground-state. The expectation value on the ground state can be expressed using path integrals as

$$\langle\psi_0|\hat{A}|\psi_0\rangle = \lim_{\beta_{ba}\to\infty} \frac{\int_{-\infty}^{\infty} dx' \int_{x'}^{x'} \mathcal{D}x(t) e^{-\frac{S_E[x(t)]}{\hbar}} A(x(t_1))}{\int_{-\infty}^{\infty} dx' \int_{x'}^{x'} \mathcal{D}x(t) e^{-\frac{S_E[x(t)]}{\hbar}}} \ . \tag{2.70}$$

This equation will be used in the computer calculations. This method demands that the ground-state is non-degenerate. If the ground-state is degenerate this method will give the mean of the expectation values for the operator on the different ground-states.

2.10 The Quantum Virial Theorem and Ground-state Energy

Equation (2.70) gives the expectation value of an operator $\hat{A}$ on the ground state of a system. The operator must be of the form

$$\hat{A} = \int |x\rangle A(x) \langle x| \mathrm{d}x \tag{2.71}$$

where $A(x)$ is a complex-valued function of position in space. If the energy of the ground state is sought one would like to have $\hat{A} = \hat{H}$ where $\hat{H}$ is the Hamiltonian of the system. But the Hamiltonian of systems consisting of particles in potentials are of the form

$$\hat{H} = \frac{\hat{p}_1^2}{2m_1} + \ldots + \hat{V}(x_1, \ldots) \tag{2.72}$$

where x_i and p_i are position and momentum of the n different particles in the d different dimensions, $i = 1, 2, \ldots, nd$. This cannot be written as equation (2.71). The expectation value of the energy is the sum of the expectation values of the potential energy and the kinetic energy. The potential energy operator can be written

$$\hat{V} = \int |x\rangle V(x) \langle x| \mathrm{d}x \tag{2.73}$$

and can thus be computed with equation (2.70). The expectation value of the kinetic energy can, as now will be shown[3], be written as the expectation value of another operator which can be written as equation (2.71).
The ground-state of the system is a stationary state and expectation values of all operators on it will be constant in time, thus

$$0 = \frac{\mathrm{d}}{\mathrm{dt}} \langle \hat{x}_i \hat{p}_i \rangle. \tag{2.74}$$

Ehrenfests theorem gives

$$0 = \frac{1}{i\hbar} \langle [\hat{x}_i \hat{p}_i, \hat{H}] \rangle. \tag{2.75}$$

Equation (2.72) gives

$$0 = \frac{1}{i\hbar} \langle [\hat{x}_i \hat{p}_i, \frac{p_1^2}{2m_1} + ... + \hat{V}] \rangle. \tag{2.76}$$

The position and momentum operators of different particles commute, the position and momentum operators of the same particles follow the canonical commutation relation thus

$$[\hat{x}_i, \hat{p}_j] = \delta_{ij} i\hbar. \tag{2.77}$$

Equation (2.76) now becomes

$$\begin{aligned} 0 &= \frac{1}{i\hbar} \langle [\hat{x}_i \hat{p}_i, \frac{p_1^2}{2m_1} + ...] + [\hat{x}_i \hat{p}_i, \hat{V}] \\ &= \frac{1}{i\hbar} \langle [\hat{x}_i \hat{p}_i, \frac{p_i^2}{2m_i}] + [\hat{x}_i \hat{p}_i, \hat{V}] \rangle. \end{aligned} \tag{2.78}$$

The commutation relations in equation (2.77) gives for the first commutator:

$$[\hat{x}_i \hat{p}_i, \frac{p_i^2}{2m_i}] = \frac{1}{2m_i} \left(\hat{x}_i \hat{p}_i \hat{p}_i \hat{p}_i - \hat{p}_i \hat{p}_i \hat{x}_i \hat{p}_i \right) = \frac{i\hbar \hat{p}_i^2}{m_i}. \tag{2.79}$$

[3]Idea for beginning of proof from [16]

The momentum operator $\hat{p}_i$ acts as a derivation operator on the x_i-wavefunction

$$\langle x|\hat{p}_i|\psi\rangle = -i\hbar\frac{\mathrm{d}}{\mathrm{d}x_i}\langle x|\psi\rangle \ , \tag{2.80}$$

and thus

$$\hat{p}_i|\psi\rangle = \int |x\rangle\langle x|\hat{p}|\psi\rangle \mathrm{d}x = \int |x\rangle(-i\hbar)\frac{\mathrm{d}}{\mathrm{d}x_i}\langle x|\psi\rangle \mathrm{d}x. \tag{2.81}$$

The second commutator is by use of equation (2.83)

$$\begin{aligned}
& [\hat{x}_i\hat{p}_i, \hat{V}] = \\
= \ & \hat{x}_i\hat{p}_i \int |x\rangle V(x)\langle x|\mathrm{d}x - \int |x\rangle V(x)\langle x|\hat{x}_i\hat{p}_i \mathrm{d}x \\
= \ & \hat{x}_i \int |x'\rangle(-i\hbar)\frac{\mathrm{d}}{\mathrm{d}x_i'}\langle x'|x\rangle V(x)\langle x|\mathrm{d}x\mathrm{d}x' \\
- \ & \int |x\rangle V(x)\langle x|\hat{x}_i|x'\rangle(-i\hbar)\frac{\mathrm{d}}{\mathrm{d}x_i'}\langle x'|\mathrm{d}x\mathrm{d}x' \\
= \ & \int |x\rangle x_i(-i\hbar)\frac{\mathrm{d}}{\mathrm{d}x_i}V(x)\langle x|\mathrm{d}x \\
- \ & \int |x\rangle V(x)x_i(-i\hbar)\frac{\mathrm{d}}{\mathrm{d}x_i}\langle x|\mathrm{d}x \\
= \ & \int |x\rangle x_i(-i\hbar)\left(\frac{\mathrm{d}V(x)}{\mathrm{d}x_i}\langle x| + V(x)\frac{\mathrm{d}}{\mathrm{d}x_i}\langle x|\right)\mathrm{d}x \\
- \ & \int |x\rangle V(x)x_i(-i\hbar)\frac{\mathrm{d}}{\mathrm{d}x_i}\langle x|\mathrm{d}x \\
= \ & -i\hbar\hat{x}_i\frac{\mathrm{d}\hat{V}}{\mathrm{d}x_i}.
\end{aligned} \tag{2.82}$$

Where $\frac{\mathrm{d}\hat{V}}{\mathrm{d}x_i}$ is defined by

$$\frac{\mathrm{d}\hat{V}}{\mathrm{d}x_i} = \int |x\rangle\frac{\mathrm{d}V(x)}{\mathrm{d}x_i}\langle x|\mathrm{d}x. \tag{2.83}$$

Thus by equation (2.78)

$$\left\langle \frac{\hat{p}_i^2}{m_i} \right\rangle = \left\langle \hat{x}_i\frac{\mathrm{d}\hat{V}}{\mathrm{d}x_i} \right\rangle. \tag{2.84}$$

This is true for all i, which gives

$$2\langle\hat{T}\rangle = \langle\hat{x}\cdot\nabla\hat{V}\rangle. \tag{2.85}$$

Which will be true for all stationary states. The expectation value of the energy of the ground state system can now be found through equation (2.70) by finding the expectation value of the operator $\hat{O}$

$$\hat{O} = \hat{V} + \frac{1}{2}\hat{x}\nabla\hat{V} \tag{2.86}$$

because $\hat{O}$ can be written in the form specified by equation (2.71). The quantum virial theorem only works for stationary states so it cannot be used in (2.66). Taking the expectation value of the operator O for temperatures above zero doesn't work because a linear combination of stationary states is not a stationary state.

2.11 Calculating the Ground-state Probability Density from the Kernel

The propagator gives the probability amplitude to go from event (x_a, t_a) to event (x_b, t_b). The wave function gives the probability density for a given position x_a for a specific state $|\psi\rangle$ at time t_a. Therefore an integral over all values of x_a for the propagator, that evolves from event (x_a, t_a) to (x_b, t_b), times the wave function for events at time t_a for the state, must give the probability density for position x_b of the state at time t_b. The kernel $K(x_b, t_b; x_a, t_a)$ should therefore evolve the wave function $\langle x_a, t_a|\psi\rangle$ to $\langle x_b, t_b|\psi\rangle$ according to

$$\langle x_b, t_b|\psi\rangle = \int K(x_b, t_b; x_a, t_a)\langle x_a, t_a|\psi\rangle dx_a \ . \tag{2.87}$$

One can easily see from the definition of the continuous identity operator that the kernel can be expressed as

$$K(x_b, t_b; x_a, t_a) = \langle x_b, t_b|x_a, t_a\rangle \ . \tag{2.88}$$

This is equal to

$$K(x_b, t_b; x_a, t_a) = \langle x_b, t_b|\hat{I}|x_a, t_a\rangle \ , \tag{2.89}$$

where I is the identity operator. Now introduce Euclidean time and set $\beta_{ba} \equiv (t_b - t_a)i$ and $t_a \equiv 0$. From equation (2.55) one can see that

$$K(x_b, \beta_{ba}; x_a, 0) = \int dx_1 \sum_{n,m} \langle x_b|\psi_n\rangle\langle\psi_n|x_1\rangle e^{-\frac{E_n\beta_{b1}}{\hbar}} \langle x_1|\psi_m\rangle\langle\psi_m|x_a\rangle e^{-\frac{E_m\beta_{1a}}{\hbar}} \ . \tag{2.90}$$

The identity operator, see equation (A.2), for x_1 can be removed and then one can identify $\langle\psi_n|\psi_m\rangle = \delta_{mn}$. The double sum thus becomes a single sum and the energy exponents can be put together to get

$$K(x_b, \beta_{ba}; x_a, 0) = \sum_n \langle x_b|\psi_n\rangle\langle\psi_m|x_a\rangle e^{-\frac{E_n\beta_{ba}}{\hbar}} \ . \tag{2.91}$$

When the start and end points are the same, $x \equiv x_b \equiv x_a$, this is

$$K(x, \beta_{ba}; x, 0) = \sum_n \langle x|\psi_n\rangle\langle\psi_n|x\rangle e^{-\frac{E_n\beta_{ba}}{\hbar}} = \sum_n |\psi_n|^2 e^{-\frac{E_n\beta_{ba}}{\hbar}} \ . \tag{2.92}$$

Multiply both sides by $e^{\frac{E_0\beta_{ba}}{\hbar}}$ and let $\beta_{ba} \to \infty$ to get

$$\lim_{\beta_{ba}\to\infty} K(x,\beta_{ba};x,0)e^{\frac{E_0\beta_{ba}}{\hbar}} = \lim_{\beta_{ba}\to\infty} \left(|\psi_0(x)|^2 + |\psi_1(x)|^2 e^{-\frac{E_1-E_0\beta_{ba}}{\hbar}} + \dots\right) . \tag{2.93}$$

Because E_0 is the smallest energy eigenvalue only the term without an exponential will survive the limit and thus

$$\lim_{\beta_{ba}\to\infty} K(x,\beta_{ba};x,0)e^{\frac{E_0\beta_{ba}}{\hbar}} = |\psi_0(x)|^2 . \tag{2.94}$$

This cannot be used directly since the ground-state energy is not known. Instead use the fact that the ground-state wave function is normalized

$$\int_{-\infty}^{\infty} |\psi_0(x)|^2 dx = 1 . \tag{2.95}$$

Integrate equation (2.94) and use equation (2.95) to get

$$\int_{-\infty}^{\infty} \lim_{\beta_{ba}\to\infty} K(x,\beta_{ba};x,0)e^{\frac{E_0\beta_{ba}}{\hbar}} dx = 1 . \tag{2.96}$$

Now shift the integral and the limit. Because $e^{\frac{E_n\tau}{\hbar}}$ has no x-dependence this can be moved outside the integral so that

$$\lim_{\beta_{ba}\to\infty} e^{\frac{E_0\beta_{ba}}{\hbar}} \int_{-\infty}^{\infty} K(x,\beta_{ba};x,0)dx = 1 . \tag{2.97}$$

The exponential can now be identified as

$$\lim_{\beta_{ba}\to\infty} e^{\frac{E_0\beta_{ba}}{\hbar}} = \frac{1}{\lim_{\beta_{ba}\to\infty}\int_{-\infty}^{\infty} K(x,\beta_{ba};x,0)dx} . \tag{2.98}$$

If this is put into equation (2.94) the result is

$$|\psi_0(x)|^2 = \lim_{\beta_{ba}\to\infty} \frac{K(x,\beta_{ba};x,0)}{\int_{-\infty}^{\infty} K(x,\beta_{ba};x,0)dx} \text{ [14].} \tag{2.99}$$

Using path integrals this can be written as

$$|\psi_0(x)|^2 = \lim_{\beta_{ba}\to\infty} \frac{\int_{x'}^{x'} \mathcal{D}x(t) e^{-\frac{S_E[x(t)]}{\hbar}}}{\int_{-\infty}^{\infty} dx' \int_{x'}^{x'} \mathcal{D}x(t) e^{-\frac{S_E[x(t)]}{\hbar}}} \tag{2.100}$$

With the knowledge of the kernel the probability distribution of the ground state can be calculated.

3 Analytical Examples

Some examples have been solved analytically in order to verify the computer calculations. Detailed derivations of the formulas presented below are found in appendix B.

3.1 The Free Particle Propagator

The free particle propagator is

$$K(b,a) = \sqrt{\frac{m}{2i\pi\hbar(t_b - t_a)}} \exp\left(\frac{im(x_b - x_a)^2}{2\hbar(t_b - t_a)}\right) . \tag{3.1}$$

3.2 The Harmonic Oscillator

The propagator for the one dimensional harmonic oscillator is given by

$$K(b,a) = \sqrt{\frac{m\omega}{2i\pi\hbar\sin(\omega(t_b - t_a))}} \exp\left(\frac{im\omega\left((x_a^2 + x_b^2)\cos\left(\omega(t_b - t_a)\right) - 2x_a x_b\right)}{2\hbar\sin(\omega(t_b - t_a))}\right) . \tag{3.2}$$

This can be expressed in dimensionless form by introducing

- a characteristic length $x_c = \sqrt{\frac{\hbar}{m\omega}}$
- a characteristic time $T_c = \frac{1}{\omega}$
- a characteristic energy $E_c = \hbar\omega$

and expressing physical quantities as multiples of them

$$x = \xi x_c \tag{3.3}$$

$$t_b - t_a = \tau T_c \tag{3.4}$$

$$E = \varepsilon E_c , \tag{3.5}$$

Equation (3.2) becomes

$$K(b,a) = \frac{1}{x_c}\sqrt{\frac{1}{2i\pi\sin(\tau)}} \exp\left(\frac{i\left((\xi_a^2 + \xi_b^2)\cos\left(\tau\right) - 2\xi_a\xi_b\right)}{2\sin(\tau)}\right) . \tag{3.6}$$

The eigenstates for the harmonic oscillator are the Hermite functions. These are given by

$$g_n(\xi) = (-1)^n(2^n n!\sqrt{\pi})^{-1/2} \exp\left(\frac{\xi^2}{2}\right) \frac{\mathrm{d}^n \exp\left(-\xi^2\right)}{\mathrm{d}\xi^n} . \tag{3.7}$$

The energy eigenvalues of the system are $E_n = \hbar\omega\left(n + \frac{1}{2}\right)$.

3.2.1 Verifying the Propagator and Ground State for the Harmonic Oscillator

In this section, the propagator and the ground state for the harmonic oscillator will be verified using equation (2.99). The propagator for the harmonic oscillator expressed in dimensionless coordinates is

$$K(\xi_b, t; \xi_a, 0) = \frac{1}{x_c}\sqrt{\frac{1}{2i\pi\sin(t)}}\exp\left(\frac{i\left((\xi_a^2+\xi_b^2)\cos(t) - 2\xi_a\xi_b\right)}{2\sin(t)}\right) . \tag{3.8}$$

In Euclidean time this is

$$K(\xi_b, \tau; \xi_a, 0) = \frac{1}{x_c}\sqrt{\frac{1}{2i\pi\sin(-i\tau)}}\exp\left(\frac{i\left((\xi_a^2+\xi_b^2)\cos(-i\tau) - 2\xi_a\xi_b\right)}{2\sin(-i\tau)}\right) . \tag{3.9}$$

If $\xi_a = \xi_b = \xi$ this becomes

$$K(\xi_b, \tau; \xi_a, 0) = \frac{1}{x_c}\sqrt{\frac{1}{2i\pi\sin(-i\tau)}}\exp\left(-\xi^2\tanh\left(\frac{\tau}{2}\right)\right) . \tag{3.10}$$

According to equation (2.99) the square of the ground state wave function should be given by

$$\begin{aligned}|\psi_0(\xi)|^2 &= \lim_{\tau\to\infty}\frac{K(\xi,\tau;\xi,0)}{\int_{-\infty}^{\infty}K(\xi,\tau;\xi,0)d\xi} = \lim_{\tau\to\infty}\frac{\exp\left(-\xi^2\tanh\left(\frac{\tau}{2}\right)\right)}{\int_{-\infty}^{\infty}\exp\left(-\xi^2\tanh\left(\frac{\tau}{2}\right)\right)d\xi} \\ &= \lim_{\tau\to\infty}\frac{\exp\left(-\xi^2\tanh\left(\frac{\tau}{2}\right)\right)}{\sqrt{\frac{\pi}{\tanh\left(\frac{\tau}{2}\right)}}} = \frac{\exp\left(-\xi^2\right)}{\sqrt{\pi}} .\end{aligned} \tag{3.11}$$

This is equal to $|g_0(\xi)|^2$, for g_o given in the previous section.

3.3 The Perturbed Harmonic Oscillator

In this section, a perturbed harmonic oscillator will be studied. The perturbing term is proportional to x^4. The Hamiltonian for the harmonic oscillator is given by

$$\hat{H}^{(0)} = \frac{\hat{p}^2}{2m} + \frac{1}{2}m\omega^2\hat{x}^2 . \tag{3.12}$$

With dimensionless parameters the Hamiltonian becomes

$$\hat{H}^{(0)} = \frac{E_c}{2}\left(-\frac{\partial^2}{\partial\xi^2} + \xi^2\right) . \tag{3.13}$$

The eigenstates are the Hermite functions

$$g_n(x) = (-1)^n\sqrt{\frac{1}{2^n n!\sqrt{\pi}}}e^{\frac{\xi^2}{2}}\frac{d^n}{d\xi^n}\left(e^{-\xi^2}\right) . \tag{3.14}$$

Now introduce a perturbation

$$\hat{H}^{(1)} = \lambda E_c \xi^4 \ . \tag{3.15}$$

The new Hamiltonian becomes

$$\hat{H} = \hat{H}^{(0)} + \hat{H}^{(1)} = \frac{E_c}{2}\left(-\frac{\partial^2}{\partial \xi^2} + \xi^2\right) + \lambda E_c \xi^4 \ . \tag{3.16}$$

The new energies can be calculated using perturbation theory. This is described in the appendix C. The ground state energy for the disturbed harmonic oscillator to the second order is found to be

$$E_0 = E_0^{(0)} + E_0^{(1)} + E_0^{(2)} = E_c\left(\frac{1}{2} + \frac{3}{4}\lambda - \frac{21}{8}\lambda^2\right) \ . \tag{3.17}$$

3.4 Few-particle Systems and Jacobi Coordinates

A system of particles with a potential that only depends on the relative positions of the particles can be reduced by one degree of freedom through a change of coordinates. This makes numerical calculations simpler. How to do this will be described below.

A system of n particles with positions $\mathbf{x}_i$ (in any number of dimensions) and masses m_i can be described with so called Jacobian coordinates, [12] page 102. The new coordinates are defined by a linear transformation

$$\begin{aligned}
\mathbf{y}_1 &= \mathbf{x}_1 - \mathbf{x}_2 \\
&\vdots \\
\mathbf{y}_i &= \left(\frac{1}{m_{0i}}\sum_{j=1}^{i} m_j \mathbf{x}_j\right) - \mathbf{x}_{i+1} \\
&\vdots \\
\mathbf{y}_n &= \frac{1}{m_{0n}}\sum_{j=1}^{i} m_j \mathbf{x}_j \ ,
\end{aligned} \tag{3.18}$$

where m_{0i} is defined as

$$m_{0i} = \sum_{j=1}^{i} m_j . \tag{3.19}$$

The kinetic energy expressed in these new coordinates is, [12] page 103

$$T = \sum_{i=1}^{n} \frac{m_i}{2}\dot{\mathbf{x}}_i^2 = \sum_{i=1}^{n} \frac{M_i}{2}\dot{\mathbf{y}}_i^2 \ , \tag{3.20}$$

where M_i is defined as

$$\begin{aligned} M_1 &= \frac{m_1 m_2}{m_1 + m_2} \\ &\vdots \\ M_i &= \frac{m_{0i}}{m_{0i+1}} m_{i+1} \\ &\vdots \\ M_n &= m_{0n}. \end{aligned} \tag{3.21}$$

The coordinate $\mathbf{y}_i$ gives for $i < n$ the relative position of particle i from the mass center of all particles with index less than i. So given the position $\mathbf{x}_1$ the position of all other particles are given if $\mathbf{y}_i$ is given for $i < n$. Thus if the potential energy V of the system only depends on the relative positions of the particles it can always be expressed by just the first $n-1$ new coordinates because they can always give all relative positions of the particles. This together with (3.20) gives that if the particles are free of any external forces the system can be considered as two separate systems, one (1) with kinetic energy $T_1 = \frac{M_n}{2}\dot{\mathbf{y}}_n^2$ and no potential energy, $V_1 = 0$. The other system (2) having the potential of the original system, $V_2(\mathbf{y}_1, ..., \mathbf{y}_{n-1}) = V$ and kinetic energy

$$T_2 = \sum_{i=1}^{n-1} \frac{M_i}{2} \dot{\mathbf{y}}_i^2. \tag{3.22}$$

The original system thus has kinetic energy $T = T_1 + T_2$ and potential energy $V = V_1 + V_2$. System 1 only depends on coordinate $\mathbf{y}_n$ while system 2 only depends on coordinates $\mathbf{y}_i$ where $i < n$, the systems are thus independent and can be separated and viewed individually.
A normlization of variables is useful,

$$\tilde{y}_i = y_i \sqrt{M_i}. \tag{3.23}$$

This makes the calculation of kinetic energy treat all coordinates the same weight

$$T = \sum_{i=1}^{n} \frac{1}{2} \dot{\tilde{\mathbf{y}}}_i^2. \tag{3.24}$$

3.4.1 Two Particles with a Harmonic Potential

Now study the case of two particles, both with mass m, in one dimension having an harmonic potential V between them:

$$V(x_1, x_2) = \frac{m\omega^2}{2} (x_2 - x_1)^2 \ . \tag{3.25}$$

The kinetic energy T of the system is

$$T(x_1, x_2) = \frac{m}{2}\left(\dot{x}_1^2 + \dot{x}_2^2\right) \ . \tag{3.26}$$

Now change the coordinates to Jacobi coordinates:

$$\begin{aligned} \tilde{y}_1 =& \frac{x_2 - x_1}{\sqrt{2}} \\ \tilde{y}_2 =& \frac{x_1 + x_2}{\sqrt{2}} \ . \end{aligned} \tag{3.27}$$

The inverse transformation is

$$\begin{aligned} x_1 =& \frac{\tilde{y}_2 - \tilde{y}_1}{\sqrt{2}} \\ x_2 =& \frac{\tilde{y}_1 - \tilde{y}_2}{\sqrt{2}} \ . \end{aligned} \tag{3.28}$$

The kinetic energy of the system in Jacobi coordinates is

$$T(\tilde{y}_1, \tilde{y}_2) = \frac{m}{2}\left(\dot{\tilde{y}}_1^2 + \dot{\tilde{y}}_2^2\right) \ . \tag{3.29}$$

The potential in the new coordinates is

$$V(\tilde{y}_1, \tilde{y}_2) = m\omega^2 \tilde{y}_1^2 \ . \tag{3.30}$$

The Hamiltonian is thus

$$H(\eta_1, \eta_2) = \frac{m}{2}\dot{\tilde{y}}_1^2 + m\omega^2 \tilde{y}_1^2 + \frac{m}{2}\dot{\tilde{y}}_2^2 \ . \tag{3.31}$$

One can see that this is the same as a free particle $\tilde{y}_2$ plus a particle $\tilde{y}_1$ in an external harmonic potential $m\omega^2\tilde{y}_1^2$. The free particle has a continuous energy spectrum and the particle in the potential has the energy eigenvalues

$$E_n = \left(n + \frac{1}{2}\right)\hbar\sqrt{2}\omega \ . \tag{3.32}$$

The factor $\sqrt{2}$ comes from the harmonic potential being $m\omega^2\tilde{y}_1^2$, rather than $\frac{m\omega^2}{2}\tilde{y}_1^2$ as before. The ground-state energy is $\frac{\hbar\omega}{\sqrt{2}}$.

3.4.2 Three Particles with Harmonic Potentials

Consider a system consisting of three particles in one dimension with positions $\mathbf{x}_i$ and equal masses m. The potential energy V of the system is:

$$V = \frac{m\omega^2}{2}\left((\mathbf{x}_1 - \mathbf{x}_2)^2 + (\mathbf{x}_2 - \mathbf{x}_3)^2 + (\mathbf{x}_3 - \mathbf{x}_1)^2\right) \tag{3.33}$$

Performing the changes of coordinates described in section 3.4 gives new coordinates

$$\tilde{y}_1 = \frac{x_1 - x_2}{\sqrt{2m}} \tag{3.34}$$

$$\tilde{y}_2 = \left(\frac{1}{2}x_1 + \frac{1}{2}x_2 - x_3\right)\sqrt{\frac{2}{3m}} \tag{3.35}$$

$$\tilde{y}_3 = \left(\frac{1}{3}x_1 + \frac{1}{3}x_2 + \frac{1}{3}x_3\right)\sqrt{\frac{3}{m}}. \tag{3.36}$$

The potential is only dependent on the relative positions of the particles and the system can thus be separated into a free particle part and a system of two particles, see section 3.4. The system of two particles have kinetic energy, (3.20)

$$T_2 = \frac{1}{2}\dot{\tilde{y}}_1^2 + \frac{1}{2}\dot{\tilde{y}}_2^2. \tag{3.37}$$

where the derivative is taken in dimensionless time $\tilde{t}$. The relative positions of all particles can be expressed with $\tilde{y}_1$ and $\tilde{y}_2$

$$x_1 - x_2 = \sqrt{2}\tilde{y}_1 \tag{3.38}$$

$$x_2 - x_3 = -\frac{1}{\sqrt{2}}\tilde{y}_1 + \sqrt{\frac{3}{2}}\tilde{y}_2 \tag{3.39}$$

$$x_3 - x_1 = -\frac{1}{\sqrt{2}}\tilde{y}_1 - \sqrt{\frac{3}{2}}\tilde{y}_2. \tag{3.40}$$

The potential energy thus becomes

$$V = \frac{m\omega^2}{2}\left(2\tilde{y}_1^2 + \left(-\frac{1}{\sqrt{2}}\tilde{y}_1 + \sqrt{\frac{3}{2}}\tilde{y}_2\right)^2 + \left(-\frac{1}{\sqrt{2}}\tilde{y}_1 - \sqrt{\frac{3}{2}}\tilde{y}_2\right)^2\right) \tag{3.41}$$

$$= \frac{m3\omega^2}{2}\tilde{y}_1^2 + \frac{m3\omega^2}{2}\tilde{y}_2^2. \tag{3.42}$$

This means that the system can be regarded as composed of a independent free particle and two independent harmonic oscillators, the harmonic oscillators having frequencies $\omega_1 = \sqrt{3}\omega$ and $\omega_2 = \sqrt{3}\omega$ and thus the total energy of the ground state of the system is $\sqrt{3}\hbar\omega$.

3.4.3 Many Particles with Harmonic Potentials

We have in the same way as above shown that the kinetic energy for 4-8 particles of mass m in 1 dimension with potential defined as

$$V = \frac{m\omega^2}{2}\sum_{i<j}\left(\mathbf{x}_i - \mathbf{x}_j\right)^2 \tag{3.43}$$

have ground-state energies as shown in table 1.

Table 1: The ground-state energies for different number of particles coupled with harmonic oscillator potentials.

Number of particles	Ground-state energy
2	$1/\sqrt{2}\hbar\omega$
3	$\sqrt{3}\hbar\omega$
4	$3\hbar\omega$
5	$2\sqrt{5}\hbar\omega$
6	$5\sqrt{3/2}\hbar\omega$
7	$3\sqrt{7}\hbar\omega$
8	$7\sqrt{2}\hbar\omega$

4 Numerical methods

First, some theory about Monte Carlo integration and generation of generating samples from a probability distribution using the Metropolis algorithm will be presented. An algorithm will then be presented that gives the operator mean and the square of the wavefunction for a system that can contain many interacting distinguishable particles.

4.1 Monte Carlo Integration

Monte Carlo methods are a general class of algorithms using random numbers. We can use Monte Carlo methods to evaluate integrals as expected values of functions of continuous random variables. The point of Monte Carlo integration is faster convergence for multidimensional integrals compared to conventional methods.

4.1.1 Brute Force Approach to Monte Carlo Integration

For a continuous random variable X with a probability density function (PDF) given by $p(x)$ the expected value of $f(X)$ is given by

$$_{p}[(f(X)] = \int_{-\infty}^{\infty} f(x)p(x)\,\mathrm{d}x. \tag{4.1}$$

The PDF of a uniform distribution is given by

$$p(x) = \begin{cases} \frac{1}{b-a} & a < x < b \\ 0 & \text{otherwise,} \end{cases} \tag{4.2}$$

and in the case of X being uniformly distributed on the interval $[a, b]$ the expectation value is given by

$$\mathbb{E}_{U(a,b)}[(f(X)] = \int_a^b \frac{f(x)}{b-a}\,\mathrm{d}x. \tag{4.3}$$

In this way we can express an integral I of one variable as an expectation value.

$$I = \int_a^b f(x)\,\mathrm{d}x = (b-a)\,\mathbb{E}_{U(a,b)}[f(X)] \tag{4.4}$$

As an estimator of the expectation value[9] you can generate an ensemble of samples $\{X_i\} \sim U(a, b)$ and calculate the sample mean of the function as

$$\langle f \rangle = \frac{1}{n}\sum_{i=1}^{n} f(X_i). \tag{4.5}$$

In this way you can approximate an integral as

$$I = \int_a^b f(x)\,\mathrm{d}x \approx \frac{b-a}{n}\sum_{i=1}^{n} f(X_i), \tag{4.6}$$

where the X_i are uniformly randomly distributed. Note the similarity to a rectangular integration given by

$$I = \int_a^b f(x)\,\mathrm{d}x \approx \frac{b-a}{n}\sum_{i=1}^{n} f(x_i), \tag{4.7}$$

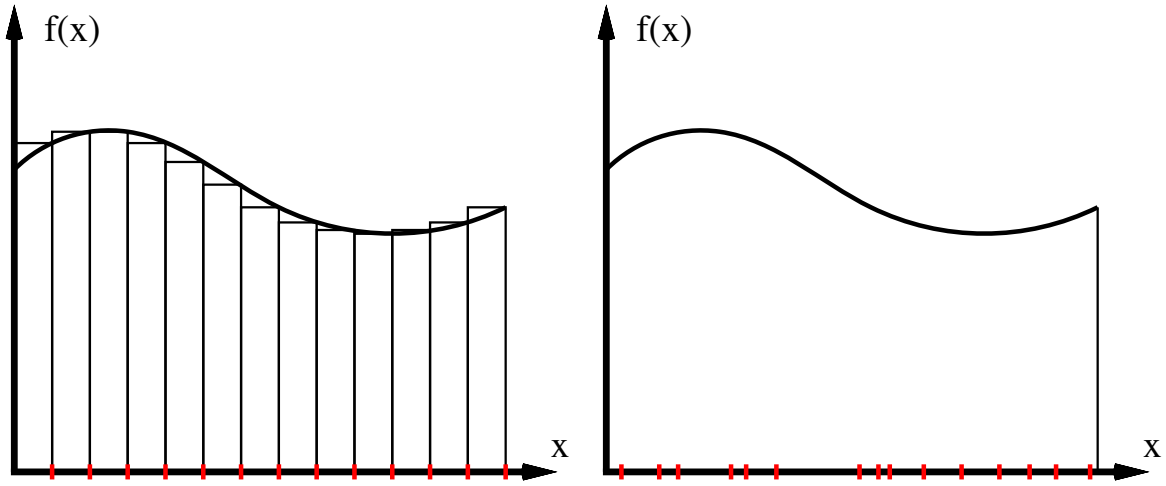
where the x_i are evenly spaced lattice points on the integration interval.

We can see that for large n the two methods should yield similar results (since the X_i are uniformly randomly distributed they should cover the integration inerval quite evenly for large n, see figure 5). In fact, the law of large numbers [8][9] states that

$$\lim_{n\to\infty} \langle f \rangle = \mathbb{E}_{U(a,b)}[f(X)], \tag{4.8}$$

while the Riemann sum given by equation (4.7) converges to the integral in the limit $n \to \infty$.

The approximation in equation (4.6) can easily be generalised to multidimensional integrals as



(a) Rectangular integration. (b) Monte Carlo integration.

Figure 5: Comparing rectangular integration with simple Monte Carlo integration.

$$\begin{aligned} I &= \int_{a_1}^{b_1} \mathrm{d}x_1 \int_{a_2}^{b_2} \mathrm{d}x_1 ... \int_{a_D}^{b_D} \mathrm{d}x_D f(x_1, x_2, ..., x_D) \\ &= \int_V f(\mathbf{x})\, \mathrm{d}\mathbf{x} \\ &\approx \frac{V}{n} \sum_{i=1}^{n} f(\mathbf{X}_i), \end{aligned} \tag{4.9}$$

where V is the volume of the integration region.

Next, we will see why we bother with this seemingly complicated way of evaluating integrals. The variance of $f(X)$ is given by

$$\sigma_f^2 = \_\_{}_p \left[\left(f(X) - \_\_{}_p[f(X)] \right)^2 \right] = \_\_{}_p[f(X)^2] - \left(\_\_{}_p[f(X)] \right)^2 . \tag{4.10}$$

The central limit theorem [8] states that the mean of a large number of independent samples is approximatly normally distributed with the variance

$$\sigma_n^2 = \frac{\sigma_f^2}{n}, \tag{4.11}$$

which gives the standard deviaton

$$\sigma_n = \frac{\sigma_f}{\sqrt{n}}. \tag{4.12}$$

The standard deviation is effectivly a measure of the error in the sample mean of the function. If one ensemble $\{X_i\}$ of samples is called one measurement, then a series of measurements will be approximatly normally distributed. About 68.3% of the measurements will generate a sample mean of the function within one standard deviation $\pm\sigma_n$ of the true mean (expectation value), and 95.4% will be within $\pm 2\sigma_n$.

The error σ_n is independet of the dimension [8] of the random variable (or the dimension of the integral) and goes like

$$\sigma_n \propto \frac{1}{\sqrt{n}}. \tag{4.13}$$

This can be compared with for example a simple trapezoidal rule whose error goes like $n^{-2/d}$ where d is the dimension [8].

To estimate the variance of $f(X)$ you can use the biased estimator

$$\sigma_f^2 \approx \left\langle (f - \langle f \rangle)^2 \right\rangle = \langle f^2 \rangle - \langle f \rangle^2 \tag{4.14}$$

(since our n are always large the difference from an unbiased estimator is negligible).

4.1.2 Importance Sampled Monte Carlo Integration

To get a good estimation of the integral with a Monte Carlo evaluation you want to make the error σ_n as small as possible. As seen in equation (4.12), one way of doing this is of course to increase the number of samples n. Another way of achieving this is decreasing the variance σ_f.

The expectation value of $f(X)$ can be rewritten as

$$\begin{aligned} \mathrm{E}_p[f(X)] &= \int_{-\infty}^{\infty} f(x)p(x)\,\mathrm{d}x \\ &= \int_{-\infty}^{\infty} g(x)\,\underbrace{\frac{p(x)}{g(x)}}_{w(x)}\,f(x)\,\mathrm{d}x \\ &= \mathrm{E}_g[w(\tilde{X})f(\tilde{X})], \end{aligned} \tag{4.15}$$

where $\tilde{X} \sim g(x)$. The function $g(x)$ must of course be a proper PDF fulfilling the normalization condition

$$\int_{-\infty}^{\infty} g(x)\,\mathrm{d}x = 1. \tag{4.16}$$

In this way we can instead approximate the integral as

$$I = \int_a^b f(x)\,\mathrm{d}x = \int_a^b g(x)\,\underbrace{\frac{1}{g(x)}}_{w(x)}\,f(x)\,\mathrm{d}x \tag{4.17}$$
$$= \mathbb{E}_g[w(\tilde{X})f(\tilde{X})] \approx \frac{1}{n}\sum_{i=1}^{n} w(\tilde{X}_i)f(\tilde{X}_i),$$

where $\tilde{X}_i$ are chosen at random with probability $g(x)$ within the integration interval.

By choosing a $g(x)$ that resembles $f(x)$ you can drastically decrease the new variance

$$\sigma_g^2 = \mathbb{E}_g[w(\tilde{X})^2 f(\tilde{X})^2] - \Big(\mathbb{E}_g[w(\tilde{X})f(\tilde{X})]\Big)^2, \tag{4.18}$$

making the error

$$\sigma_n = \frac{\sigma_g}{\sqrt{n}} \tag{4.19}$$

smaller (thus decreasing the number n of samples needed).

In a sense you can say that we, when using the brute-force approach, might be wasting time generating samples in areas where the function $f(x)$ gives very small contributions to the integral. By generating more samples in regions where $f(x)$ gives large contributions, we can dramatically decrease the number of needed samples n. We must of course weight (with $w(\tilde{X}_i)$) the values of $f(\tilde{X}_i)$ to get the right answer.

The method of importance sampling will later be used when evaluating the ground-state expectation value of quantum mechanical operators by the path-integral expression, derived in Section 2.9, given by

$$\langle\psi_0|\hat{A}|\psi_0\rangle \approx \frac{\int_{-\infty}^{\infty} dx' \int_{x'}^{x'} \mathcal{D}x(t) e^{-\frac{S_E[x(t)]}{\hbar}} A[x]}{\int_{-\infty}^{\infty} dx' \int_{x'}^{x'} \mathcal{D}x(t) e^{-\frac{S_E[x(t)]}{\hbar}}}, \tag{4.20}$$

where x now denotes a path starting and ending in x'. We will identify

$$p[x] = \frac{e^{-\frac{S_E[x(t)]}{\hbar}}}{\int_{-\infty}^{\infty} dx' \int_{x'}^{x'} \mathcal{D}x(t) e^{-\frac{S_E[x(t)]}{\hbar}}} \tag{4.21}$$

as our (properly normalized) PDF of paths and in this way approximate the path integral as

$$\langle\psi_0|\hat{A}|\psi_0\rangle \approx \int_{-\infty}^{\infty} dx' \int_{x'}^{x'} \mathcal{D}x(t)p[x]A[x]$$
$$\approx \frac{1}{n}\sum_{\{x_i\}} A[x]. \tag{4.22}$$

Note that we have discretisized the paths in the approximation, and that $\{x_i\}$ is an ensemble of paths where each path is represented by a set of coordinates.

4.1.3 Markov Chains and the Metropolis-Hastings Algorithm

A stochastic process is a process that transitions from one state to another in a random and chainlike manner, and thus generates a sequence of states (or samples) $\{..., X_i, X_{i+1}, ...\}$. The process is called stationary if the probabilistic laws remain unchanged in time. In other words, if the joint probability distribution of a series of n samples $\{X_i, X_{i+1}, ..., X_{i+n}\}$ is equal to the joint PDF of a series of n later samples $\{X_{i+h}, X_{i+h+1}, ..., X_{i+h+n}\}$.

A Markov chain is a simple stochastic process that depends only on the current state, and it is called ergodic if it always converges to a unique stationary distribution. Thus far we have assumed the samples to be statistically independent of each other but the samples generated from a Markov chain are generally correlated. It can fortunatly be shown that the law of large numbers, used in equation (4.8), hold for a stationary stochastic process [10].

The Metropolis-Hastings algorithm is a way of generating samples from a PDF using an ergodic Markov chain [10][11]. The algorithm uses relative probabilities, so we only need to be able to calculate a function $f(x)$ proportional to the wanted PDF $p(x)$ according to

$$\frac{1}{C}f(x) = p(x). \tag{4.23}$$

Suppose the last sampled value is x_i, and we want to generate a new sample x_{i+1}. The process can be summarized as follows:

1. Generate a proposal sample x' by displacing x_i with a number taken from a symmetric probability distribution.
2. Calculate the relative probability ratio $\alpha = \frac{p(x')}{p(x_i)} = \frac{f(x')}{f(x_i)}$.
3. Accept or reject the proposal sample:

$$\text{if } \alpha \geq 1 : x_{i+1} = x'$$
$$\text{else } x_{i+1} = \begin{cases} x' \text{ with probability } \alpha; \\ x_i \text{ with probability } 1-\alpha. \end{cases}$$

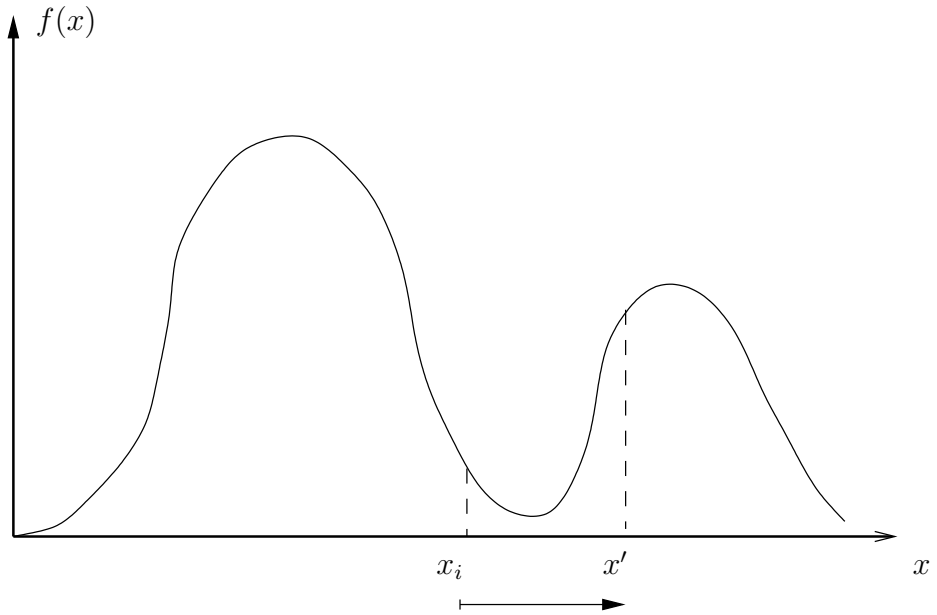


Figure 6: An illustration of one random step in the Metropolis-Hastings algorithm.

Note that the normalization constants cancel out, so we can use $f(x)$ directly for sampling.

In our application we will be using a Random Walk Metropolis-Hastings algorithm, which takes a uniformly randomly distributed step from the initial state x_i. We will use an initial state x_0 and then let the algorithm run to stabilize (sometimes called thermalizing), discarding the "burn-in" samples.

One important parameter to the algorithm is the acceptance rate of new samples. A too high or too low acceptance rate will make the chain converge slowly. You can understand the effect of the acceptance rate by looking at Figure 6. By taking too large steps, you might probe outside the likely areas, get a low acceptance rate, and thus move around the PDF slowly. If you have a too small stepsize, and thus a high acceptance rate, you might get stuck probing only one area of the PDF for a very long time.

4.2 Discretization of the Path Integral

We will now connect all the theoretical derivations and results to see how we can discretisize a path integral in a way suitable for computer calculations.

As we have previously seen, in Section 2.5, the kernel to go from x_a at time t_a to x_b at time t_b can be approximated as

$$K(x_b, t_b; x_a, t_a) \approx \frac{1}{A^N} \int e^{i\frac{S[x_b,x_a]}{\hbar}} \, \mathrm{d}x_1 \, \mathrm{d}x_2 ... \, \mathrm{d}x_{N-1}. \tag{4.24}$$

We have divided the time interval $t_b - t_a$ in N short intervals with length $\epsilon = (t_b - t_a)/N$, making x_i the coordinate for time t_i. When working in Euclidean time, setting $t_a = 0$ and defining $\beta = i(t_b - t_a)$, this becomes

$$K(x_b, \beta; x_a, 0) \approx \frac{1}{A^N} \int e^{-S_E[x_b, x_a]} \, \mathrm{d}x_1 \, \mathrm{d}x_2 ... \, \mathrm{d}x_{N-1}, \tag{4.25}$$

as shown in Section 2.7, where S_E is the Euclidean action. This allows us to discretisize a path as an array of coordinates $\{x_0, x_1, ..., x_N\}$, where $x_0 = x_a$ and $x_N = x_b$.

We have also seen, in Section 2.6, that the action (or the Euclidean action) can be written as

$$S_E[x_b, x_a] = \sum_{i=0}^{N-1} S_E[x_{i+1}, x_i]. \tag{4.26}$$

For short times it has been shown that

$$S_E[x_{i+1}, x_i] \approx \epsilon H(x_i, \dot{x}_i) \tag{4.27}$$

where the Hamiltonians we work with can be written as

$$H(x_i, \dot{x}_i) = T(\dot{x}_i) + V(x_i). \tag{4.28}$$

To discretisize the velocity at time t_i we use a simple Eulers rule

$$\dot{x}_j \approx \frac{x_{i+1} - x_i}{\epsilon}. \tag{4.29}$$

This then gives

$$S_E[x_b, x_a] \approx \epsilon \sum_{i=0}^{N-1} \left[T\left(\frac{x_{i+1} - x_i}{\epsilon} \right) + V(x_i) \right]. \tag{4.30}$$

As a final note, the expressions for the expectation value of a quantum mechanical operator, derived in Section 2.9, and expression for the squared ground-state wave function, derived in Section 2.11, utilize periodic boundary conditions $x_a = x_b$. This allows us to store only $\{x_0, x_1, ..., x_{N-1}\}$ and use x_0 whenever x_N is needed.

4.3 Algorithm for Operator Mean and Probability Density

In this section will start with describing the algorithm used to find the quantum mechanical expectation value of an operator and the squared wave function (probability density) without too much consideration why it works. The actual implementation of this algorithm is discussed later in the report.

1. Generate an initial path. This can be done by choosing N numbers at random between two integers or it can simply be N zeros.

2. Make a suggested modification to the path. This is done by displacing the i-th node in the path where i is a random integer between 1 and N. The displacement take the form

$$x_i' = x_i + \delta(2\sigma_1 - 1) \tag{4.31}$$

where σ_1 is a random number between 0 and 1 and δ is a parameter that determines the greatest distance we can move a node in a path in one displacement.

3. Calculate the differences in energy $\Delta E[x]$ between the modified and unmodified paths. If we define the energy for one path link, using equation (4.30), as

$$E(x_i, x_{i+1}) = \left(\frac{1}{2} m \left(\frac{x_{i+1} - x_i}{\epsilon} \right)^2 + V(x_i) \right)_{x_N = x_0} \tag{4.32}$$

then the full energy of a path is given by

$$E_{\text{path}} = \sum_{i=1}^{N} E(x_i, x_{i+1}). \tag{4.33}$$

Again, notice that $x_N = x_0$ which means that the start and end point is the same in the path. We want the paths to be distributed as $\exp(-\epsilon E[x]/\hbar)$ which can be done by using the Metropolis algorithm. We first calculate

$$\frac{\exp(-\epsilon E_{\text{mod}}/\hbar)}{\exp(-\epsilon E_{\text{unmod}}/\hbar)} = \exp(-\epsilon \Delta E[x]/\hbar). \tag{4.34}$$

The modification to the path is then accepted if $\Delta E < 0$. Otherwise we accept it with probability $\exp(-\epsilon \Delta E[x]/\hbar)$. This can be done by generating a random number, σ_2, between 0 and 1 and accept the modified path if $\exp(-\epsilon \Delta E[x]/\hbar) > \sigma_2$. Accepting the modified path means that we change the x_i in the path to x_i'.

Since we are only changing one node of the path it is not necessary to do the whole sum in (4.33). Instead, the differences in energies can be calculated as

$$\Delta E[x] = E(x_{i-1}, x_i') + E(x_i', x_{i+1}) - E(x_{i-1}, x_i) - E(x_i, x_{i+1}). \tag{4.35}$$

This is useful for saving computation time since we are now calculating $E(x_i, x_{i+1})$ four times for a new path instead of $2N$ times.

4. Depending on if we want the squared wave function or the operator mean two different things is done in this step.

   (a) If we want to calculate the squared wave function we store the displaced x_i' if we accepted the suggested modification else x_i is stored. These can for example be saved in an array and for further references to this array of saved values it will be denoted X where X_i is the i'th element in X.

   (b) If the desired quantity is the mean of an operator $O(x)$ we save the value $\sum_{i=1}^{N} O(x_i)$ in an array O. This can then be used to calculate the average value of the operator for the path.

5. Repeat steps 2 to 4 until convergence.

6. Remove some of the values saved from the first iterations since the Metropolis need some iterations to thermalize. The number of iterations for thermalization can not be given generally and depends on the system and the parameters of the path.

7. (a) The wave function squared is calculated by first doing a histogram of the saved x_i-values. This can be done by dividing an interval into subintervals. We call these subintervals bins. Next, a count is added to the bin for every x_i whose value is inside the interval of the bin. If a value is outside all bins the total interval should be increased, or if one just want to see a specific interval of the
   ction the data point should be disregarded. This can be regarded as a function that assigns the midpoint of a bin to the number of counts in that bin. We then divide with the normalizing factor that is the number of bins multiplied with the total amount of counts in the bins. The now normalized function will be the probability density.

   (b) The mean of the operator is the mean of the saved values in step 5.

8. The whole process can then be repeated multiple times to get the mean and standard error of the results.

4.4 Explanation of algorithm

The explanation that the normalized histogram created in step 7a will give the probability density is as follows. From the Metropolis algorithm we know that the x_i values will be distributed as the probability density function that is proportional to $\exp(-\epsilon E[x]/\hbar)$. We can thus write

$$x_i \sim \frac{e^{-\frac{S_E[x(t)]}{\hbar}}}{\int_{-\infty}^{\infty} dx' \int_{x'}^{x'} \mathcal{D}x(t) e^{-\frac{S_E[x(t)]}{\hbar}}}. \tag{4.36}$$

From equation (2.99) we can see that for large times this is equal to the probability density of the system. A normalized histogram of the saved x_i-values will thus give probability distribution of the ground state.

The operator mean is calculated using importance sampling. For the derivation of how it works we refer to the latter parts of section 4.1.2 in particular equations eqn:OperatorMean to (4.22).

4.5 Generalization to More Particles

The generalization of the algorithm to more than one particle is not very difficult. If we for n particles in the system define the path of the system as $\{\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{N-1}\}$ where $\mathbf{x}_i = (x_{1i}, x_{2i}, x_{2n})$ the algorithm is basically the same. We chose a random node and a random particle and do the displacement like we did in step . To get the probability density one do a n-dimensional normalized histogram of saved $\mathbf{x}_i$. The mean of the operator is computed the same way as before.

4.6 Choice of parameters

Since we are interested in the ground state probability density and operator mean the time duration of a path needs to be set large enough so that excited states have decayed. We found that the number of nodes in a path, N, should be quite large (in the order of 100-1000). Too large N causes the time step to be very small, making the kinetic energy term in the energy estimator (4.32) to diverge. Better energy estimators would be needed to solve this problem. For discussion about better energy estimators in the path integral formalism see [32]. With increased N one also needs to increase the number of iterations for thermalization. The choice of good parameters have from our experience not been the same between when finding the probability density and the operator mean. In general a greater node count, N, of a path and higher end time ,t_b, is needed to get a good value for the probability density than the operator mean. For some results about different choice of parameters see section 8.6.

5 Implementation in C

This section will present a full C-program that implements that Feynman path integral formalism with use of the Metropolis algorithm as discussed earlier in this report. Since random numbers play a big role in Monte Carlo calculations, this section will begin with a discussion about how random numbers are generated in computer calculations, and what random number generator is used for this work. To help understand the program, the source code is explained on a line by line basis. Understanding the C-program presented here will help a great deal when making the transition to the parallelized computations in OpenCL.

5.1 Random Number Generators

Since the Monte Carlo method for integration utilizes the theory of mathematical statistics, a source for random numbers is needed. Having access to random numbers uniformly distributed in the interval $[0, 1]$ suffices to produce samples from any other distribution. Computers are deterministic machines and have no possibility whatsoever to produce truly random numbers in the sense that tossing a dice would. What can be done however is producing a sequence of numbers that to an outside observer appear random. If there is no way for that observer to determine that the set of numbers are calculated deterministically instead of generated by tossing a dice, then that set of numbers will work as as a substitute to true random numbers. These kind of numbers are called *Pseudo Random Numbers* and are generated by a Pseudo Random Number Generator, which in essence is a sequence of mathematical operations with numbers that result in a single pseudo random number. In order for a PRNG to produce a sequence of pseudo random numbers, an initial number (or set of numbers) is required, on which to build the subsequent calculation of pseudo random numbers. This initial number is called the seed for the PRNG.

There are many PRNG's available for use, one of the most common types is the *linear congruential generator*, where given a seed X_0, pseudo random numbers are generated with the calculation

$$X_{n+1} = (aX_n + b) \mod m$$

where the paramteters a, b and m can be tuned to obtain a sequence of random numbers with good quality.

5.1.1 Quality of Pseudo Random Number Generators

What characterizes a good PRNG is a long period (the number of pseudo random numbers that can be generated before the sequence is repeated), low correlation (the degree to which new random numbers depend on previously generated random numbers) and speed (the amount of computation necessary to generate a new

sample). To test for randomness (low correlation), a number of statistical tests have been developed, among them are the Diehard Battery of Tests [17]. One example of these tests is the Gorilla test, considered one of the harder tests. This is a strong version of a monkey test with the analogy that the PRNG is a "monkey" sitting at a typewriter and typing "letters of an alphabet" at random (the alphabet in this case consists of a range of numbers) to produce a long string. The number of "k-letter words" that don't appear in this string should be approximately normally distributed. Poor PRNG seem to fail when the string of "letters" gets very long [18].

When using the Monte Carlo method, it is very important that the PRNG is of good quality, otherwise the results will not be very credible. With demands on quality, certain PRNGs have been developed and others have been discarded. The linear congruential generator does not do very well in statistical tests [4]. Instead, a popular PRNG is the Mersenne Twister, developed in 1997 by Makoto Matsumoto with a periods up to a whooping $2^{19937} - 1$. Implementing the Mersenne Twister as the PRNG used in this work would bring about an unnecessary large amount of extra code. Instead, the simpler Xorshift PRNG is used.

5.1.2 Xorshift

The Xorshift PRNG was developed in 2002 by George Marsaglia at Florida State University [19]. It works by applying the bitwise XOR-operation on a number with a bit shifted version of itself.

The logical operator XOR (exclusive OR) is defined by it's truth table

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

The operator that performs XOR bit by bit on two numbers in C is represented by the `^`-operator. For example, two 8-bit unsigned integers `a` and `b` with binary representations `a` = 00110011_2 = `51` and `b` = 01010101_2 = `85` would give the new number `a ^ b` = 01100110_2 = `102`.

Bit shifting a number involves shifting all the bits in the binary representation of that number a certain number of steps either to the left or the right. Bits that are shifted out of place are discarded and new bits come in as zeros. The left- and right-shift operators in C are represented by the `<<` and `>>`-operators. For

[4] A classic example is the infamous RANDU PRNG that fails tests of randomness dramatically. As a result, scientific publications that makes use of RANDU does not carry a large amount of credibility.

example shifting the 8-bit unsigned integer `a = 00110011`$_2$ `= 51` 3 steps to the left is written as `a << 3 = 10011000`$_2$ `= 152`.

An example of an Xorshift PRNG is

Listing 1: Xorshift PRNG with period $2^{128} - 1$.

```
1  unsigned int xor128(void) {
2      static unsigned int x = 123456789;
3      static unsigned int y = 362436069;
4      static unsigned int z = 521288629;
5      static unsigned int w = 88675123;
6      unsigned int t;
7
8      t = x ^ (x << 11);
9      x = y; y = z; z = w;
10     return w = w ^ (w >> 19) ^ (t ^ (t >> 8));
11 }
```

Calling `xor128()` will return a random unsigned 32-bit integer, meaning an integer in the interval $[2^0 - 1, 2^{32} - 1] = [0, 294967295]$. A uniformly distributed random floating point number in the interval $[0, 1]$ can be obtained by dividing this unsigned integer with $2^{32}-1$, or multiplying with it's inverse $(2^{32}-1)^{-1} \approx 2.32830644 \cdot 10^{-10}$.

5.2 Example: C-program to Evaluate Mean of an Operator

In this example, the Metropolis algorithm is used to calculate the ground state energy of a system of three particles interacting with a harmonic potential.

5.2.1 Theory for the System

Consider the system of three particles in one dimension with coordinates x_1, x_2 and x_3 interacting with a harmonic potential as described in section 3.4.2, each particle with mass $m = 1$ and frequency $\omega = 1$. Performing the change of variables

$$\tilde{y}_1 = \frac{x_1 - x_2}{\sqrt{2}} \tag{5.1}$$

$$\tilde{y}_2 = \left(\frac{1}{2}x_1 + \frac{1}{2}x_2 - x_3\right)\sqrt{\frac{2}{3}} \tag{5.2}$$

$$\tilde{y}_3 = \left(\frac{1}{3}x_1 + \frac{1}{3}x_2 + \frac{1}{3}x_3\right)\sqrt{3} \tag{5.3}$$

$$\tag{5.4}$$

implies that Hamiltonian of the system can be separated into one free particle part and a system of two particles. Considering that the ground state energy of a free particle is 0, one may disregard this and focus only on the system of two particles[5].

[5]Actually, referring to this as a system of two particles is not entirely accurate. It is better to refer to it as a quantum mechanical system with two degrees of freedom.

The kinetic energy for this system is given by

$$T = \frac{1}{2}\dot{\tilde{y}}_1^2 + \frac{1}{2}\dot{\tilde{y}}_2^2 \tag{5.5}$$

and the potential is given by

$$V = \frac{3}{2}\tilde{y}_1^2 + \frac{3}{2}\tilde{y}_2^2. \tag{5.6}$$

Employing the virial theorem yields

$$\langle \hat{H} \rangle = \langle T \rangle + \langle V \rangle = \frac{1}{2}\langle (\tilde{y}_1, \tilde{y}_2) \cdot \nabla V \rangle + \langle V \rangle = \tag{5.7}$$

$$= \frac{1}{2}\langle (\tilde{y}_1, \tilde{y}_2) \cdot (3\tilde{y}_1, 3\tilde{y}_2) \rangle + \frac{3}{2}\langle \tilde{y}_1^2 + \tilde{y}_2^2 \rangle = 3\langle \tilde{y}_1^2 + \tilde{y}_2^2 \rangle \tag{5.8}$$

$$\tag{5.9}$$

Thus, in order to find the ground state energy of the system, it is enough to find the mean of the operator $A = 3(\tilde{y}_1^2 + \tilde{y}_2^2)$, in accordance with (2.86). The use of the virial theorem is motivated by the desire to find an operator that is local in time and therefore can be written in the form (2.71) and whose mean gives the energy of the system.

The mean of an operator in terms of Feynman Path Integrals is given by equation (4.20). Calculating this by use of the Metropolis algorithm and importance sampling is discussed in section 4.1.2. One arrives at the expression (4.22). In order to make computations on a computer, one always needs to discretize a continuous expression. Here the question rises, what is the functional $A[x]$ used in expression (4.22) for representing the operator $A = 3(\tilde{y}_1^2 - \tilde{y}_2^2)$?

Assume that the paths $\tilde{y}_i(t)$ where $t \in [0, T]$ and $i = 1, 2$ for particle 1 or 2 have been discretized into a lattice with $N + 1$ points $\tilde{y}_i^j, j = 0, \ldots, N$ for each particle with j corresponding to times $t_j = j\varepsilon$ where $\varepsilon = \frac{T}{N}$, so that $\tilde{y}_i^0 = \tilde{y}_i(0)$ and $\tilde{y}_i^N = \tilde{y}_i(T)$, and that the particle moves in straight lines between the lattice points. It is then sufficient to take for example

$$A[\tilde{y}] = 3((\tilde{y}_1^j)^2 + (\tilde{y}_2^j)^2) \tag{5.10}$$

for any j. This is because the operator in question is local in time and therefore does not depend on what point in time it is evaluated. It is however better for convergence to use

$$A[\tilde{y}] = \frac{1}{N}\sum_{j=1}^{j=N} 3((\tilde{y}_1^j)^2 + (\tilde{y}_2^j)^2). \tag{5.11}$$

Notice here that the lattice points $\tilde{y}_i^0$ are omitted. This is because when applying the periodic boundary conditions $\tilde{y}_i(0) = \tilde{y}_i(T)$ required by the $\int_{x'}^{x'} \mathcal{D}(x)$-part of equation (4.22), the points $\tilde{y}_i^0$ and $\tilde{y}_i^N$ are in some sense the same and counting them twice would bias the value for A (think about it in the sense that there are $N+1$ lattice points but only N links between the lattice points, where the particle spends ε of it's total time T).

5.2.2 Construction of C-program

In appendix E.1, a complete C-program is presented that calculates the mean of the operator A for the two particles in the harmonic potential, i.e. the ground state energy. This program will be explained here.

First, some general concepts need to be clear. `path` is a field with `pathLength=nbrOfParticles*N` floats. This corresponds to the spacial coordinates of the N first lattice points of the 2 particles.. Since periodic boundary conditions are employed, the coordinate for lattice point number $N+1$ will be the same as the coordinate for lattice point number 0, therefore this lattice point is not stored explicitly in `path`. Also, whenever the text below mentions iterations, they refer to the iterations of the loop at line 83.

Lines 1—8: Relevant C-libraries are included and problem specific constants are chosen. In particular, the number of particles is set to 2 and the time interval for which the particles travel in one path is set to 10. The choice of time affects how much higher energy states have decayed. The choice of 10 can be motivated by solving the problem analytically and examining how T affects the mean of the operator A. If that is not possible, one will need to proceed empirically.

Lines 10—24: The Xorshift PRNG works through these two methods. Calling `xorshift()` performs the bit shifts and XOR operations discussed in section 5.1.2 with the result that a fresh random unsigned 32-bit integer is stored in `w` for use in `randFloat()`.

Lines 26—36: These two funtions return the value of the operator A and potential V respectively for one lattice point (one discrete point in time, as for example in equation (5.10) for a specified j). To understand how this works, let's say that the lattice point being modified in the Metropolis algorithm has index j where $j = 0 \dots N-1$ (corresponding to time $t_j = j\varepsilon$). The spacial coordinate for the first particle for this point in time is stored in `path[`j`]` and the corresponding spacial coordinate for the second particle is stored in `path[`$j+N$`]`. Examining the functions `operator()` and `potential()`, it's apparent that they take a *poiner* `x` to a `float`. If this pointer is the pointer to `path[j]` then `x[0]` will be the value stored in in `path[j]` and `x[N]` will be the value stored in `path[j + N]` that is, the spacial coordinate corresponding to time t_j for the second particle.

Lines 38—46: Changing a spacial coordinate in a lattice point from an old value to a new changes two links. The particle moves in straight lines between the lattice points and hence there is a slight change in the kinetic part of the action between the old path and the new. This change is calculated here, by specifying the old and new spacial coordinates as well as the coordinates to the left and right of the modified points.

Line 49: This is the header for the `metropolis()`-function which performs roughly `nbrOfLoopings` metropolis steps. Evidently, the arguments for `metropolis()` are five pointers. The reason for using pointers will be evident when going into the parallelization part to perform the calculations on a GPU.

Lines 51—59: The variables that are utilized throughout the calculation are

initialized here. `modPathPoint` will keep track of the index of the element in `path` that is being modified in each of the iterations starting at line 79. `modPoint` hold the index corresponding to the point in time that is being modified in each of the iterations, hence `modPoint = modPathPoint % N` (an integer in the interval $[0, N-1]$). `leftIndex` and `rightIndex` will hold the index for the two adjacent lattice points, that is, the index for spacial coordinates for the two points occurring ε before and after the point in time t_j for the point being modified. `modx` hold the new proposed spacial coordinate for the point being modified while `oldX` stores the spacial coordinate before modifying it. `loopLimit` stores the number of iterations should be performed. This value is obtained from the argument `nbrOfLoopings` and is thus modifiable (to handle burn in as well as regular runs).

Lines 61—62: The idea with `partialOp` is that for each iteration in the loops a new sample (path) is generated with the Metropolis algorithm. The operator is then evaluated for this path according to equation (5.11). This calculated value is then added to `partialOp`. To prevent risk of cancellation when adding a small floating point number to a large floating point number — something that could be the case after say 400000 iterations (`partialOp` would then be rather large) — a buffer `opMeanBuffer` is created, to where every once in a while (specifically every `threeLoopRuns`) the value in `partialOp` is transfered.

Lines 65—68: The Xorshift PRNG needs four seeds to function, these are set here. Note also that `x`, `y`, `z` and `w` are changed whenever `xorshift()` gets called. In order to be able to run `metropolos()` several times in sequence and have each run pick up where the last run finished, these seeds are stored outside of `metropolis()` and are exported at the end of `metropolis()` in lines 127—130.

Lines 70—71: Since only one spacial coordinate is changed in each iteration, it is unnecessary to calculate the value of A for the whole path in each iteration. Instead, the value of A for the initial path (before the iterations begin) is calculated and then updated when a spacial coordinate is changed. These two lines calculate A for the initial path.

Lines 73—74: `opMean[0]` will eventually store the calculated mean of the entered operator after `nbrOfLoopings` Metropolis steps have been performed. `accepts[0]` will hold the number of accepted changes to calculate the acceptance rate of the Metropolis algorithm.

Lines 79—83: One iteration of the loop at line 83 performs a Metropolis step and hence produces a sample from the desired PDF (a path distributed as (4.21)). New paths are generated by proposing a change of one spacial coordinate in a lattice point in the current path and then accept the change with a certain probability. The lattice points are looped through sequentially from with increasing j, particle by particle. One full loop for all lattice points and all particles correspond through one full loop through the `pathSize` elements `paths` field. This is done in the two loops at lines 80—84. The purpose of the loop at line 79 is to repeat this calculation. The outer loop runs `threeLoopRuns` times (`threeLoopRuns` = `nbrOfLoopings/pathSize` (integer division)) and thus in reality the total number

of Metropolis steps will not equal `nbrOfLoopings`, but rather the closest lower multiple of `pathSize`. This ensures that all lattice points undergo the same number of proposed modifications.

Lines 88—96: These important lines of code are responsible for setting $\tilde{y}_i^0 = \tilde{y}_i^N$ (remember that these are the boundary conditions for the Feynman Path Integral $\int \mathcal{D}[\tilde{y}] \dots$). Since the value for $\tilde{y}_i^N$ is not stored (only spacial coordinates with index j between 0 and $N-1$ are stored), the variable `rightIndex` is set to 0 if `modPoint` happens to equal $N-1$. Likewise, if `modPoint` equals 0, `leftIndex` is set to $N-1$. In other cases, `leftIndex` and `rightIndex` are just the integers adjacent to `modPoint`.

Lines 98—99: The spacial coordinate that is proposed to change is stored in `oldX`. The new proposed value is stored in `modx` and is simply a step of length α in either direction (positive or negative) from the current value.

Lines 102—106: These lines calculate the difference in energy (from which later the difference in action is obtained by multiplication with the time step ε). The difference in energy is stored in `diffE` and contains one term for kinetic energy and one for potential energy. Again, the change in kinetic energy depends on the change of the *two* links that are changed when modifying the spacial coordinate for one lattice point, while the change in potential energy is dependent solely on the change of the single spacial coordinate (the virial theorem cannot be employed here!).

Lines 109—114: This is the condition for accepting the change in the path. If the new path is accepted, the value for `partialOp` needs to be updated (remember that `partialOp` contains the value for the given operator evaluated for the *whole* current path). `partialOp` is updated by subtracting the value of A for the now old spacial coordinate and adding the value of A for the new spacial coordinate (hence the two calls to `operator()`). Also, `accepts[0]` is incremented.

Lines 116—124: At the end of each iteration, the calculated value of A for the sample path (stored in `partialOp`) is added to a pool `opMeanBuffer` of completed functional values. Once every `pathSize` iteration, this pool is emptied into `opMean` to avoid numerical cancellation (lines 119—120). Lastly, on line 124 the value in `opMeanBuffer` is normalized, in the sense that the values already stored in `opMeanBuffer` are functional values A for different paths x distributed as (4.21) with the exception that the division with N in equation (5.11) is not performed when calculating `partialOp` (this saves roughly `nbrOfLoopins` divisions by N.), instead this is done once and for all on this line. This corresponds to one of the factors of `N` in the denominator, the other factor `threeLoopRuns*nbrOfParticles*N` is there to return the mean of A instead of returning the sum of `threeLoopRuns*nbrOfParticles*N` values of A for different paths.

The `metropolis()`-function is now complete and the calculated mean of the opefootnotesrator is stored in `opMean`. The lines 133—169 constitute a simple C-program that calls `metropolis() runs` times and use the results to compute 65%- and 95% confidence intervals. This assumes that the calculated means are

independent and identically distributed random variables (otherwise a t-test would be needed). This is a good assumption however, since the resulting means are sums of a very large amount of random variables that follow some (unknown) distribution. The Central Limit Theorem then yield that `opMean` are normally distributed random variables. Some sample output from the program follows here

Listing 2: Sample output from C-program in appendix E.1.

```
Doing burn in...
Run  1:
Mean of operator: 1.741713
Acceptance rate: 35.09%

Run  2:
Mean of operator: 1.723673
Acceptance rate: 35.02%

Run  3:
Mean of operator: 1.719937
Acceptance rate: 35.06%

Run  4:
Mean of operator: 1.719481
Acceptance rate: 35.06%

Run  5:
Mean of operator: 1.708518
Acceptance rate: 35.03%

Run  6:
Mean of operator: 1.718654
Acceptance rate: 35.07%

Run  7:
Mean of operator: 1.716800
Acceptance rate: 35.05%

Run  8:
Mean of operator: 1.729634
Acceptance rate: 35.02%

Run  9:
Mean of operator: 1.722076
Acceptance rate: 35.06%

Run 10:
Mean of operator: 1.762898
Acceptance rate: 35.01%

Mean: 1.726338
Standard error: 0.004660
65% CI: [1.721678, 1.730999]
95% CI: [1.717017, 1.735659]
```

The analytical solution for the ground state energy of a system of three particles interacting with harmonic potentials is given at the end of section 3.4.2, and with $\hbar = \omega = 1$ it is $E_0 = \sqrt{3} \approx 1.732$ which is in good agreement with the calculated value.

It should be noted that examining the expression for V and T above show that the system can be treated as two independent and identical harmonic oscillators, and thus it would be sufficient to calculate the ground state energy for only one of them and then multiply this value by 2. This was not done however, since the purpose of this section was to demonstrate how the path integral formalism is used in computer code.

6 Parallelization and GPU Programming

In section 5.2, a C-program was presented that calculates the mean of an operator according to the outlines of the algorithm in section 4.3. This section will cover the background for parallelizing the code and implementing it in OpenCL. The final OpenCL code will be presented in section 7.

In **section 6.1**, GPU programming concepts will be introduced and compared with CPU programming. This will be related to the GPU architecture. This is a good starting point if not familiar with GPU terminology and concepts.

Section 6.2 covers OpenCL background, how OpenCL programs are designed and how they are implemented in Python through PyOpenCL. A short comparison will be carried out between OpenCL and CUDA. This section offers some useful background, but is not imperative for further reading.

Section 6.3 is a brief guide to OpenCL programming in Python. It will cover the essentials in order to understand the code later on in the report. It also includes a very condensed list of best practices when writing OpenCL code that will hugely improve performance.

Section 6.4 covers two of the biggest challenges that emerged when implementing the code on a GPU with OpenCL, and how they were dealt with.

6.1 GPU versus CPU Programming Concepts

The purpose of this section is to introduce some of the concepts that make GPU programming different from CPU programming. The differences will be related to the architecture of the GPU. Furthermore, some basic terminology will be defined, like work group, work unit, thread, global size and folding. For the sake of simplicity, the scope of this section will only cover current series of ATI and nVidia Graphics cards. A brief comparison will be made between the two brands.

6.1.1 Background

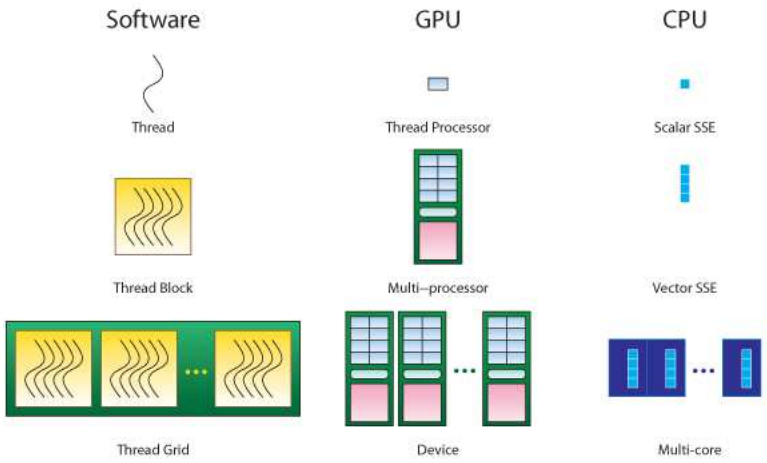
What mainly sets Graphical Processing Units apart from Central Processing Units is the amount of *cores*. This amount represent how many calculations that can be carried out simultaneously. A GPU contain several *Multi Processors* (MP), each having an array of cores. Until quite recently, it was very unusual for CPUs to even have more than one core. GPUs have meanwhile had up to several hundred physical cores for a long time. The reason as to why the CPU is the main computational device is that the clock frequency is so much higher than on the GPU. Since the CPU has few cores and a very high clock frequency, it performs serial computations efficiently. This is even more true when the calculations are based on previous results. The GPU on the other hand is excellent at handlig many small routine tasks with little to no correlation to eachother. A typical example is the matrix and vector operations behind what colors each pixel on a screen shall use when representing three-dimensional objects.

Altough the CPU have seen an insane trend in increased clock frequency the last decade, more focus has recently been diverted toward implementing more cores. The clock frequency has pretty much stagnated around 3500 MHz per core. This will probably lead to an increased interest in parallelization in the nearest future.

6.1.2 Multiprocessors, Work groups, Threads and Global Sizes

As previously stated, each GPU consist of a number of multiprocessors. The amount varies heavily between different companies and architectures, but what they all have in common is that they consist of arrays of smaller single processors. These are also known as cores, *work units* and *threads*. Together they form a *work group*, or *thread grid*, on each multiprocessor. All the multiprocessors will together compromise a *thread grid*.

Figure 7: Representation of GPU and CPU compute units, as well as desired thread computations in software. Figure taken from[31].



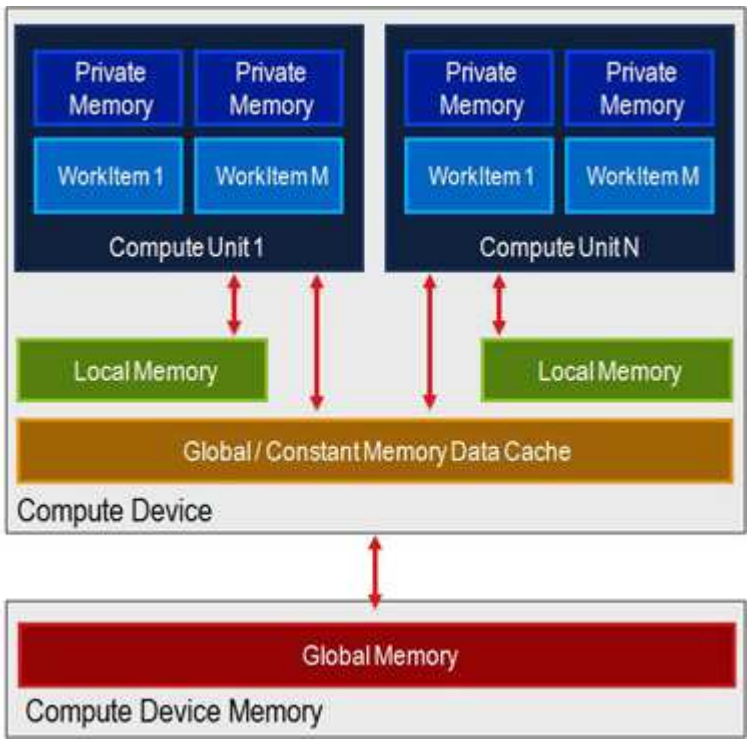
Different work groups will generally not be able to use the same memory or communicate with eachother. This is due to the fact that the GPU is meant to execute similair instructions in parallel over the different work groups. When each instruction is finished, a new instruction will be handed to the work group. Since there is no way of knowing which work group finishes first, writing to the same memory is often out of the picture. This is either resolved by using *Memory Barriers*, but that goes beyond the scope of this project. The GPU is a nifty little device, because the work units can continue to perform their next set of instructions while waiting for memory. Thus, it is very important to make sure that the total amount of instructions is a factor of the total amount of available cores on the graphics card. Another advantage of this is that some cores will finish slower than others, and the probability of having the whole array of cores waiting for a slow

core decreases with a larger set of instructions. This total amount of instructions is often called *global size* or *thread strides*. In some cases, it is important to instruct the device how to split the global size in order to assign the multiprocessors with the correct amount of instructions. These divisions will be called *block sizes* or *local sizes*, each being a 2-dimensional list. OpenCL often does this automatically, as opposed to CUDA where local size and block size needs to be defined explicitly. Local size is always set according to hardware, and global size is set as a factor of the local size.

6.1.3 Memory

There are basically four memory types available on the device. These are `__global`, `__constant`, `__local` and `__private`.

Figure 8: OpenCL Memory Hierarchy. Figure taken from[30].



__global is the memory of the entire GPU, the slowest to work toward. This is usually somewhere above 1 GB for newer graphics cards. This entire memory will not be available for allocation if not using Image2D[40] variables. For regular variables, roughly a third of this will be available for allocation. The global memory is very useful if the kernel code is designed to be re-run, saving the user the effort of having to transfer the data back and forth between the device and host for each run.

__constant is read only memory, optimized for being widely accessed. Constant should always be used with variables that are read only. Constant memory can be combined with the other memory types.

__local memory is basically restricted to each work group, or more exactly each Streaming Multiprocessor (SM). The local memory is much faster to access than the global memory, and thus a better option for frequently used data. The draw back is the size, being typically 16 kB for new graphics cards.

__private is a register-like memory, tailored for individual work items/threads. This is by far the fastest memory, but also the smallest. It should be used for frequently used control data.

6.1.4 ATI versus nVidia

It is very hard to compare performance of AMD/ATI[6] and nVidia graphics cards on paper. The big difference is mainly due to the different architectures they are basing their hardware on. Take two of their cards as an example, *ATI Radeon HD 5850*[25] and *nVidia GeForce GTX 470*[26] (these happen to be two of the cards used during the project). GTX 470 has 14 *Streaming Multiprocessors* (SM), each consisting of 32 *Shader Processors* (SP). That is a total of 14·32 = 448 so called *CUDA Cores*. HD 5850 on the other hand has 288 multiprocessors, each having 5 *Arithmetic Logical Units* (ALU). This leads to a total of 5·288 = 1440 *Stream Processing Units* (SPU). Some would argue that only 4 ALUs should be counted as the 5th ALU is transcendental, lacking some functionalities. 4 or 5 ALUs aside, HD 5850 SPUs are a lot more than GTX 470s CUDA cores. These numbers basically represent the amount of threads/work units, thus determining how many computations the graphics card can perform in parallel. Bear in mind though that these numbers alone says nothing about performance. GTX 470 has a higher clock frequency and the CUDA cores are generally a bit better optimized than the SPUs. Add to this a whole bunch of other parameters and you end up in a very confusing situation, as is customary with GPU architectures. How then can a global size be determined that works well with both ATI and nVidia? There is no rule that fit all sizes. Generally though, a global size/total thread count that is a factor of 32 is a good choice since it is a very reoccuring number in different dimensions (32 work units, 32·45 = 1440 stream processors etc). If unsure though, refer to AMD[24] and nVidias[23] recommendations respectively. With GTX 470 as an example, the local size is beneficially set to (14,32), relating to (SM,SP). The global size could then be set to `(4*448,1)` or something similair, depending on the application. Note: OpenCL has another way of counting ATI multiprocessors, called *compute units*. The HD 5850 has 18 compute units with 80 *OpenCL processing elements* each. Notice that 18·80 = 1440, the same amount as SPUs.

A side note for someone interested in determining performance between different

[6] AMD bought ATI in 2006, but still label their graphic cards with ATI Technologies.

kinds of graphics cards. Benchmarking with some kind of software or calculation-heavy program is generally the best approach. Bear in mind that different benchmarks give varying results, as the graphics cards and multiprocessors are optimized for different kinds of calculations and instructions. Some of the most common benchmarks are performed with DirectX, OpenCL, 3D Mark or FPS measurements in games.

6.1.5 CPU versus GPU Calculation: A Brief Example

Up to this point, some concepts have been introduced that makes GPU calculations different from CPU calculations. A practical example might be in place. Take for example the evaulation of equation (6.1). It represents the function f of two variables x and y, summed over i values of x and y. For complete source code of a GPU implementation of this problem, please refer to section 6.3.2, page 57.

$$\sum_{i=0}^{n} f(x_i, y_i) = e^{-\frac{x_i^2+y_i^2}{2}} \tag{6.1}$$

A typical CPU program would simply loop over i, calculate the exponent and sum the results to a variable. A GPU on the other hand could instruct each of its multiprocessors G to calculate as many values of f simultaneously as there are work units U. In effect, that would be $G{\cdot}U$ evaluations simultaneously. If i is greater than $G{\cdot}U$, work units will simply be assigned new calculations of x and y once they are done with their current ones. The summation of the result would have to be carried out a bit differently though, as the work groups generally do not have access to the same memory. There are several different approaches to this issue. If a pure GPU implementation is required, there exist a method called *reduction*[37][38] or *folding*[39]. This method is not used in the project, but instead the result of the GPU is typically a vector or matrix. The summation is then carried out on the CPU.

6.1.6 Speed-up Factor of Parallelization

Previously, it was claimed that the amount of cores is not enough to determine the performance. The amount of cores generally needs to be combined with other factors in order to be useful. Take for example Amdahl's law(6.2), which is an estimation of the maximum speed-up factor S for parallelization on N cores. It also uses a factor P, which is *the fraction of the total serial execution time taken by the portion of code that can be parallelized*[23].

$$S = \frac{1}{(1-P) + \frac{P}{N}} \tag{6.2}$$

If N happens to be a very large number, the speed-up factor will act like equation (6.3). That means with $P = 75\%$ parallelization, the maximum speed-up

factor will be $S = 4$. Thus, the greater P is, the greater the speed-up.

$$S = \frac{1}{1 - P} \tag{6.3}$$

In the guide, nVidia argues that if P is a small number, increasing the amount of available cores does little to improve the overall performance. That is, an expensive GPU will not out-perform a single core CPU if the code is poorly parallelized. nVidia thus argues that the most bang for the buck lies in increasing P before spending too much resources on increasing N.

6.2 OpenCL Description

The previous section covered GPU concepts, architecture and terminology. These concepts are actualized in this project through OpenCL. This section contains a short outline of how OpenCL programs work, a comparison of OpenCL versus CUDA, and how OpenCL is implemented in Python through PyOpenCL. For more info on Python please refer to section D on page 96. For installation please refer to the official PyOpenCL installation guide[43].

6.2.1 Background

OpenCL stands for Open Computing Language and is a framework for parallel programming on heterogeneous systems, like CPUs and GPUs. More exactly, it is a specification managed by the Khronos Group[44], determining how hardware manefacturers should design the interfacing with their hardware. OpenCL code is basically a programming language based on the modern C dialect *C99*[48], with some restrictions and additions like special keywords and data types. OpenCL provides the ability to write *Application Programming Interfaces* (API) which handle the devices (GPU or CPU) and execute kernels (functions) on them. OpenCL is the first cross-platform specification to be open and royalty free, unlike its competitors *CUDA*[45] and *DirectCompute*[46], which are restricted to nVidia and Windows platforms respectively. These are all examples of GPGPU languages, short for *General-Purpose computing on Graphics Processing Units*[47]. A big advantage of GPGPU is that it unlocks the GPU and its parallel computing capabilities to non-graphical applications.

6.2.2 OpenCL versus CUDA

There are mainly two reasons as to why OpenCL was chosen instead of CUDA. First of all, OpenCL is compatible with many different platforms, which makes this bachelor project more available to others. One of the computers used in this project was in fact an AMD/ATI platform, which is obviously not compatible with CUDA. Second of all, OpenCL is newer and not as established as CUDA, making it a bit more interesting and fresh. This is also a downside since this might lead

to lack of some functionalities, not as good support or patching, as well as worse run-time and efficiency optimization. OpenCL will probably never be able to compete with CUDAs performance. This is due to the fact that CUDA is tailored for one specific environment, while OpenCL has to sacrifice a lot of optimization when adapting to many different platforms. nVidia has a lot of interest in CUDA, investing a lot of resources in developing and optimizing it. Should there exist an interest in changing from OpenCL to CUDA, it is supposedly not that hard. CUDA and OpenCL works pretty much the same way, and have similair syntaxes. For people more familiar to CUDA, please refer to AMDs guide[49] for basic terminology comparison. One important note for CUDA programmers is that OpenCL usually is very good at assigning what block sizes to use. This means that the user often only has to specify global size, then leave local size and block size to OpenCL to figure out. As far as integration with Python goes, the difference is trivial. CUDA and OpenCL are integrated with modules called PyCuda and PyOpenCL respectively. Both are conveniently managed by the same person, Anders Klöckner[51]. Please refer to his guide *CudaVsOpenCL*[50] for a more in-depth discussion about the difference in their respective implementation with Python.

6.2.3 OpenCL Programs

OpenCL programs are divided into two parts: one that is executed on the host and the other on the device. The host application is in this case written in Python and run on the CPU. The host application provides the necessary API to handle the device execution and control the OpenCL environment through *Contexts* and *Command Queues* etc. The device application is the main focus of this project. It is where OpenCL strengths come into play, it is where the data and algorithm parallelism occurs. Code is executed on the device through kernels (functions) written in OpenCL. The device is primarily the GPU in this project. However, through ATI Stream SDK[41], an AMD multi-core processor will be used as device in OpenCL for reference and benchmarking purposes.

6.2.4 PyOpenCL

OpenCL is compatible with a variety of host programming languages. In Python, the integration is done through a module, called PyOpenCL[34][36][35][22]. PyOpenCL basically wraps every OpenCL API construct into a Python class, including garbage collection and conversion of errors into Python exceptions. What makes PyOpenCL so good and sets it appart from other similair implementations is[35]:

- **Object cleanup tied to lifetime of objects**. This idiom, often called RAII[52] in C++, makes it much easier to write correct, leak- and crash-free code.
- **Completeness**. PyOpenCL puts the full power of OpenCLs API at your disposal, if you wish.

- **Convenience**. While PyOpenCLs primary focus is to make all of OpenCL accessible, it tries hard to make your life less complicated as it does so–without taking any shortcuts.
- **Automatic Error Checking**. All OpenCL errors are automatically translated into Python exceptions.
- **Speed**. PyOpenCLs base layer is written in C++, so all the niceties above are virtually free.
- **Helpful**, complete documentation[22] and a wiki[36].
- **Liberal licensing** (MIT).

6.3 OpenCL Programming in Python: A Brief Guide

This will be a short but to-the-point introduction to OpenCL implementation in Python through PyOpenCL. Please note that PyOpenCL slightly differs from pure OpenCL when it comes to syntax, but basically works the same way. This guide will contain enough information for people to be able to start experimenting and continue developing on their own. For more in-depth guides, please refer to one of the many OpenCL tutorials available online. For complete source codes, please refer to section E.2.3 on page 112.

6.3.1 Best Practices: Writing Optimized OpenCL Code

This section will list some key points for writing optimized OpenCL applications that leads to increased performance. For in-depth analasys, please refer to nVidias best pratices[23] and AMDs OpenCL Programming guide[24].

Maximize Parallelization: This is probably the single most important optimization. Find ways to parallelize algorithms and instructions as much as possible.

Use fast memory: Avoid using global memory. Instead use local or even private memory if possible. Remember to define memory only used for reading as constant.

Reduce instructions: Avoid calculating the same set of instructions several times, by calculating and storing the result for that instruction *once*, then pointing to that adress space.

Minimize data transfer: Stick to few but large transfers between host and device, rather than having many transfers.

Avoid division: Use multiplication of inversed quantities as this is much faster for the compiler.

Use native functions: Native functions are special sets of functions, that are greatly optimized for the device. Take for example `cos(`ω`)` versus `native_cos(`ω`)`, representing 200 versus 1 hardware instruction for small ω.

For larger ω, the amount of hardware instructions for `cos` will increase even further.

Avoid using pow: Try to write out powers explicitly, rather than using the function `pow`. This has been one of the biggest speed-up factors in programs during this project.

Avoid `double` type: OpenCL currently has performance issues with double emulation. Always make sure to manually type cast to float instead of double, as this will save the compiler several execution cycles. Example: Use `0.5f` instead of `0.5`

Division and Modulo: If possible, avoid integer division and modulo operators as these are very costly. Instead, use bitwise operation. Example: If n is a power of 2, $\frac{i}{n}$ is equivalent to $(i >> \log_2 n)$, and $(i\%n)$ is equivalent to $(i\&(n-1))$. $>>$ and & are bitwise operators, corresponding to bitshift right and logical AND respectively.

Time measurement: Use profiling for time measurements instead of measuring with Python. This is shown later in this section.

The following recommendations apply for nVidia only[23]:

MAD: Enabling the device option `-cl-mad-enabled` lets the compiler group add and multiply instructions into a single *FMAD* instruction whenever possible. This is recommended as it can lead to large performance gains. Note however that the FMAD truncates the intermediate result of the multiplication.

Compiler Math Relaxation: Enabling the device option `-cl-fast-relaxed-math` activates many *aggressive* compiler optimizations. That is, very fast run-time with some sacrifice of precision.

6.3.2 A Simple OpenCL Application

Lets consider a simple GPU program that evaluates equation(6.4).

$$\sum_{i=0}^{nbrOfPoints} f(x_i, y_i) = e^{-\frac{1.5 \cdot x_i^2 + 2.8 \cdot y_i^2}{2}} \tag{6.4}$$

Please refer to the official documentation[22] for a complete list of available PyOpenCL commands and their specifications.

Listing 3: OpenCL Tutorial Application

```
1 # Filename: demo_gaussian.py
2 # Info: Simple PyOpenCL demo to evaluate a gaussian function.
3 # Author: Patric Holmvall, 2011-05-14, Chalmers, Gothenburg
4
5 import pyopencl as cl
```

```
6  import pyopencl.array
7  import numpy as np
8
9  nbrOfThreads = 4*448
10 nbrOfPoints = 100*nbrOfThreads
11 globalSizeA = nbrOfPoints
12 globalSizeB = 1
13 localSizeA = 1
14 localSizeB = 1
15
16 ctx = cl.create_some_context()
17 queue_properties=cl.command_queue_properties.PROFILING_ENABLE
18 queue = cl.CommandQueue(ctx, properties=queue_properties)
19
20 x_rand = np.random.rand(nbrOfPoints).astype(np.float32)
21 y_rand = np.random.rand(nbrOfPoints).astype(np.float32)
22
23 x = cl.array.to_device(ctx, queue, x_rand).astype(np.uint32)
24 y = cl.array.to_device(ctx, queue, y_rand).astype(np.float32)
25 gaussian = cl.array.zeros(ctx, queue, (nbrOfPoints,), np.float32)
26
27 KERNEL_CODE = """
28 __kernel void sum(__global const float *x, __global const float *y,
29                   __global float *gaussian)
30 {
31     int gid = get_global_id(0);
32     gaussian[gid] = native_exp(-(0.5f)*(%(A)s*x[gid]*x[gid] +
33                                         %(B)s*y[gid]*y[gid]));
34 }
35 """
36 KERNEL_PARAMS = {'A':'1.5f','B':'2.8f'}
37
38 prg = cl.Program(ctx, KERNEL_CODE % KERNEL_PARAMS).build()
39
40 kernel = prg.sum(queue, (globalSizeA,globalSizeB), (localSizeA,localSizeB),
41                  x.data, y.data, gaussian.data)
42 kernel.wait()
43 runTime = 1e-9*(kernel.profile.end - kernel.profile.start)
44
45 gaussianResult = np.sum(gaussian.get())
46
47 print("Result: "+str(gaussianResult))
48 print("Kernel run time: %0.5f s" % runTime)
```

In this program, three libraries are used. The first is `PyOpenCL`, imported with the alias cl. This makes it possible to write `cl.someCommand` instead of `pyopencl.someCommand`. The second library is a class of PyOpenCL called `pyopencl.array`, used for passing pyopencl arrays between the host and the device. The last library is `numpy`[55], a scientific computational library for Python.

Lines 16—18 in the code creates the *context* and *command queue*, and is a fundamental step in making an OpenCL application. Using the definitions from Khronos OpenCL specification[20][21]: *An OpenCL context is created with one or more devices. Contexts are used by the OpenCL runtime for managing objects such as command-queues, memory, program and kernel objects and for executing kernels on one or more devices specified in the*

context. If no argument is specified to `cl.create_some_context`, the first available device will be used. This is usually every available GPU in some arbitrary order, then the CPU. The command queue is an object that is used to enqueue OpenCL commands to be executed by the device. The `queue_properties` variable makes sure that command queues have profiling enabled, making it possible to measure time on the device instead of in the Python code. This is always recommended since Python will not be able present a correct timing. For more information on this topic, please refer to nVidia best practices[23].

Lines 23—25 deal with memory allocation. There are several ways of allocating a memory buffer for data transferred to the device. Without too much concern for details, it becomes a choice of preference. In this example, a very condensed method is used. The command `cl.array.to_device` tells the device to allocate memory equivalent to the size of the variable in its argument. In this case, it is the variables x and y both being `float32` vectors the size of `numberOfPoints`. `cl.array.to_device` returns a python pointer object that is used as a kernel argument, linking to the vectors x and y in the kernel code. The command `cl.array.zeros` works pretty much the same way. It allocates the same amount of memory since the gaussian has the same dimenson as x and y, but will initially be an array filled of zeros. The pointer ojbect will later be used to fetch the result from the device.

Lines 27—35 contain the kernel code. It is the C99-based code executed on the device. Note that it is wrapped in a python string. The variable `int gid = get_global_id(0)` is a global unique identifier for threads. In this example, it is used to make sure that threads access unique parts of the arrays x and y before making operations on them. `get_global_id(0)` will thus have the same dimension as x and y, in this case having an index from `0` to (`nbrOfPoints`-1). There are two other things worthy of noting, the first is writing powers explicitly as `x[gid]*x[gid]` instead of using the `pow` function. This has been one of the biggest performance gains in various programs during this project. The second performance gain is the usage of `native_exp` instead of `exp`. The text `__global` in front of the kernel arguments indicate that the variables will be stored in the global memory of the graphics card. If an array is going to be used frequently, there will be performance gain in using local memory instead with `__local`. Remember though that the local memory is severely limited to only around 16 kB per Streaming Multiprocessor on a nVidia GeForce 470 GTX. That means that 32 threads will have to share the 16 kB besides the small registers available to each thread.

`KERNEL_PARAMS` on line 36 is a Python tuple used to paste constants into the kernel code, saving the effort of having to transfer them to the device and

allocate memory. Note that A and B are strings, making it possible to make a float type cast in OpenCL by writing `1.5f` instead of `1.5`. This might seem like a small difference, but such type casts could potentially increase performance and avoid unwanted problems. This is especially true for code that will be run a lot. Later in the project, code on the device will typically be re-run somewhere between 10^5 and 10^{10} times. Thus, small changes will have huge impact on overall performance.

Lines 38—43 contain the build and execution of the kernel. Notice how the device is instructed to use $globalSizeA \times globalSizeB$ as global block size and $localSizeA \times localSizeB$ as local block size. Since everything but `globalSizeA` is 1, it is the same as leaving everything but `globalSizeA` undefined. The purpose of this is however to illustrate how such arguments would be passed. Notice the `kernel.wait()` command, instructing Python to wait for the kernel to be finished executed before continuing. In this portion of the code, it is used together with the kernel profiling to make sure that the kernel execution time is properly measured.

Line 45 fetches the resulting vector of the gaussian, and sums every element in the vector with `np.sum`. The reason as to why the device is returning a vector is that it has no way of storing all the values of the gaussian into the same memory space. That is without using memory barriers[27], reduction[37][38] or folding[39].

6.4 Challenges with GPU Implementation

During the GPU implementation of the algorithms and theoretical models, numerous challenges have arisen and blocked the path to success. Most noteworthy are the harsch run-time and memory limitations. This section cover how these challenges were overcome.

6.4.1 5 second Kernel Run-Time Limitation

Graphics Processing Units are raw computational devices. They are designed to break down and compute vast amounts of data, often much larger than those computed on the Central Processing Units. There is however one important fact that sets the GPU apart from the CPU in this aspect. The data computed on the GPU is mainly intented for visualization of graphical objects on the screen. As humans seem to be a very impatient species, there must excist some kind of run-time limitations, preventing the graphics card from calculating too long before outputting results. Run-time limitations are also necessary to prevent damage on the hardware through overheating, which could occur if the graphics card works at maximum performance for

a long time. A common limitation on nVidia GPUs is 5 second run time per kernel, which basically means that our kernel code needs to be executed within 5 seconds. A similar limit exists on ATI based platforms. This poses a significant problem for someone dealing with scientific computations, since he or she might be used to do calculations for several hours or even days. Luckily, there are several work-arounds that are more or less optimal:

- Split your kernel up into several smaller kernels. Program them to be re-runnable. This is generally the most recommended solution.
- Install an additional GPU solely for non-graphical computations. Note: do not connect to a screen!
- Buy an extremely expensive graphics card like the nvidia Tesla[54]
- Disable the X server in Linux or do a registry hack in Windows. Try to avoid this as it disables an important safety mechanism for your GPU.

The first suggestion is a good practice that keeps your code tidy and user-friendly. It is probably the most employed method on desktop computers. If the kernel functions are used to compute serial data, as is the case of this project, there need to be some way for the GPU to pick up where it left off between different re-runs. This can be achieved by transfering the data back and forth between host and device between runs, protecting it from being dumped as unused data by the GPU. This is probably not very likely if the computer only has one active user at a time, and can be very costly performance-wise. A better option in that case is to store the data in the global memory of the device, and pass a token between runs if necessary. Working directly towards the global memory on the device might have its performance disadvantages. Therefore, it might be a good idea to copy the data to shared/local memory when a kernel is using it.

6.4.2 Memory Limitation and Random Numbers

A big issue with the 0.92 and 2011.1 version of PyOpenCL is the random number generator, `pyopencl.random`. It is based on *MD5 hashing*[53] and is pretty flawed when it comes to stability. On some platforms, there are huge performance leaks causing extremely long execution times, up to tens of seconds for single random numbers. As a side note, these are well known issues and there are currently improvements being worked out. As this project aims to examine path integrals leading to many variables and iterations, `pyopencl.random` is simply not satisfactory. Being unable to create the random

numbers with the built-in functions basically leads to two options. That is, either creating the random numbers on the host and transferring them to the device, or implementing a Pseudo Random Number Generator by hand in the kernel code. The former method quickly provides high quality random numbers since it typically utilizes Mersenne Twister[63] in numpys built in random function. The downside is slow-downs during host to device transfer and extremely high memory occupation. A way of generating them on the device without using `pyopencl.random` is to implement some minimal PRNG by hand in the kernel code. In this project, the PRNG *xorshift* was used, more thoroughly described in section 5.1.2 on page 41. xorshift still requires data in form of seeds to be transferred from host to device. The 32 bit int seeds for xorshift typically needs to allocate $4 \cdot NT \cdot 32$ bit where the 4 comes from the 4 variables xorshift uses in the algorithm and NT is the number of threads. With 512 threads, the seeds allocate 4 kB and can be used to generate any desired amount of variables on the device. Take for example $2 \cdot 10^5$ random numbers on each thread. That would be 1562 MB allocated memory if generated on the host and 4 kB if generated on the device with xorshift. The quality of the xorshift random numbers may be worse, and the xorshift algorithm would be required to run $2 \cdot 10^5$ times. The performance and quality loss is however neglible. The amount of memory saved is a huge improvement. The nVidia GTX 470 used in this project is a relatively new graphics card. It has 1280 MB global memory, but only 320 MB maximum allowed memory allocated. GTX 470 will not able to handle the storage of the $2 \cdot 10^5$ random numbers in other words, if not using *Image2D Variables*[40] that is. Image2D variables were not used in this project.

7 Implementation in OpenCL

Now it is time to present the complete source code for calculating the Feynman path integrals on a GPU. The program will consist of a kernel[7] code (written in C) that contains the actual calculations that are to be performed and runs on the GPU, and a host code (written in Python) that runs on the CPU. The host code communicates with the GPU, allocates memory and transfers data to and fro the graphics card. Additionally, another piece of code is required to use the features provided by the host code. This code will be referred to as the application code, and is customized to each quantum mechanical system of particles and whatever calculations that are to be performed. The results that are presented in this report are obtained by writing an application program that make use of the kernel and host code presented below. The kernel-, host- and example application code is discussed in detail below. Corresponding source code is put in the appendix.

The idea of presenting a C-program in section 5.2 is that the kernel code presented below is very similar and should thus be easy to follow, if one has taken time to understand the C-program. Only small changes are needed to convert the C-program to a functioning OpenCL kernel.

7.1 OpenCL Kernel Code

The OpenCL kernel code is presented in appendix E.2.1 and will be explained here.

First, some general concepts will be explained. The host code utilize placeholders as a means for getting user input to the kernel. For example, when running the host code, all instances like `%(variablename)d` will be replaced with relevant information. The trailing `d` indicates what data type `variablename` should be converted to. The different datatypes are `d` for integer, `f` for float and `s` for string. This concept is very similar to `#define` used in the C-program, where the preprocessor manges the replacements.

Lines 1—47: The reader should be familiar with the functions defined in this section, as they are explained in detail in section 5.2, when discussing the C-program for calculating Feynman path integrals. One difference however is the `inline` prefix, which is a hint to the compiler that copying and pasting the code to wherever the functions are called could bring about a speedup. At line 25, a placeholder for a string is inserted as a way for the user to formulate his or her own C-functions in the application code to help write

[7]Not to be confused with the mathematical concept of the quantum mechanical propagator/kernel $K(b, a)$.

out the potential and operator (see the comments in the example application code).

Line 50: The function header differs in the OpenCL kernel from the C-program. The `metropolis()` function now has the `__kernel` prefix, indicating that is is an OpenCL kernel and the arguments `__global` keyword to specify where in the memory the input is stored. The arguments are created as `pyopencl.array` objects in the host code (see below).

Lines 52—55: These lines are important and necessary changes from the C-program. The command `get_global_id()` returns the id for the running thread (thread ids range from 0 to `nbrOfThreads`-1). This is how different threads are distinguished from one another. One example how this is used is when passing the seeds for the Xorshift PRNG. The seeds are generated by Numpy in the host code as a `nbrOfThreads`$\times 4$ array of random unsigned integers. When this array is passed to the kernel, it is no longer indexed by row and column, but rather by a single index that goes sequentially from left to right and top to bottom in the array (like when reading a book). One row in this array contains the four seeds for the thread with thread id equal to the row number (given that first row has row number 0). Therefore, `seeds[4*gid]` is the first seed for thread with id `gid`. The special number `4*gid` is saved under the name `seedsVectNbr`. In a similar way, `paths` is a long vector containing the spacial coordinates for all lattice points and particles for all the threads. It has a total of `N` $\times$ `nbrOfParticles` $\times$ `nbrOfThreads` elements, where each chunk of `N` $\times$ `nbrOfParticles` $=$ `pathSize` elements belong to one individual thread. This can be viewed as the many `path`-fields of the C-program stored one after another when this is run in many independent instances (which is what happens in the OpenCL implementation). The element `paths[gid*pathSize]` is the first spacial coordinate for the first particle for thread with id `gid`. The special number `pathSize*gid` is saved under the name `pathsVectNbr`.

Lines 147—148: Here, the `pathSize` spacial coordinates for the running thread are copied from `paths` to a local field `path` for easier reference.

Lines 206—224: These lines are new and did not appear in the C-program. They perform the time-saving trick to obtain a numerical representation of the probability density as discussed in section 4.4. The array `binCounts` stores the number of hits in each bin. The array is in global memory and the part this thread uses thus starts at `gid*nbrOfBins`. The bins are arranged symmetrically in a hypercube spanning from `xmin` to `xmax`. The hypercube contains `binsPerPart` bins in each dimension and thus $\texttt{binsPerPart}^{\texttt{nbrOfParticles}}$ bins in total. The array is split into `binsPerPart` parts where each part corresponds to different positions for particle number `nbrOfParticles`. The parts are then further split for different positions for particle number `nbrOfParticles-`

1 and so on until all particles positions have been considered and the size of the parts are just one bin. The variable ok tells whether the current sample will be in the hypercube and thus should be registered. The value of ok is found by looping through all particles and checking whether they are inside the bounds. ok is set to 0 if any particle is outside the bounds. If the value of ok is 1 a hit should be registered. The index in binCounts to be incremented is called binIndex. binIndex is calculated while looping to find ok by incrementing it inc $\times$ bin position for current particle for each particle. inc is the distance in binCounts between different bin positions for the current particle and it thus starts at 1 and is multiplied by binsPerPart in every loop iteration.

Lines 236—237: Here, the path as it is after all the Metropolis steps have been taken is saved back to paths to make the next kernel run able to pick up where this kernel run ended.

7.2 Python Host Code

The Python host code is presented in appendix E.2.2 and will be explained here.

First, some general concepts will be explained. The code contains three methods: getOperatorMean(), _loadKernel() and _runKernel(). The two methods with names starting with an underscore are not meant to be called from an outside user, but are rather called by the getOperatorMean()-method.

Line 55: getOperatorMean() is the one method that is provided to the user and supervises the whole operation of building the program, running the kernel and returning the results of the run. A description of the arguments that need to be provided are given in lines 13—29. First, the kernel and important OpenCL objects are created by calling the _loadKernel()-function at line 60. After that, the kernel can be run and therefore perform burn in. This is done by running the kernel burnInRuns number of times without taking care of the output at lines 65—67. Remember that each new kernel run picks up where the last kernel left off, meaning that a long burn in can be accomplished by running the kernel many times in sequence. Each execution of nbrOfThreads kernels will be called a run. Each run produces a calculated operator mean and standard deviation, as well as a numerical representation of the probability density from the time saving trick. Lines 73—83 allocate memory for storing these results after runs runs. On lines 86—90 the runs are executed and the results are stored. On lines 105—106, the results from the runs sets of nbrOfThreads kernel executions are merged into one resulting value for the operator mean, it's standard deviation and numerical

representation of probability density by calculating the appropriate means. This is what `getOperatorMean()` returns.

Line 115: `_loadKernel` is a neat little function for creating the necessary OpenCL objects such as command queue and context as well as filling the place holders with data and building the program. Especially interesting are the lines 118—120, that converts the user inputed potential and operator to valid C-code. For example, the potential for two particles interacting with a harmonic potential should be written by the user as `'0.5f*(x1-x2)*(x1-x2)'` (see the example application code). This string should be replaced with `'0.5f*(x[0]-x[N])*(x[0]-x[N])'`. Such replacements are performed at lines 118—120. Next, in lines 123—143, the placeholders in the kernel are filled and the program is built. The kernel is identified as the `metropolis()`-function. Lastly, at lines 146—161 memory for the kernel arguments is allocated. On lines 164—165 the context, command queue, kernel object and `pyopenck.array`-objects for the kernel arguments are returned.

Line 174: `_runKernel` is the method that actually runs the kernel and returns the results from each kernel run. The kernel is executed at lines 188—190 by calling `kernel()` with appropriate arguments. The `queue`-argument is the command queue. `(nbrOfThreads,)` specifies that the global work size should be one dimensional and of length `nbrOfThreads` (this specifies that `nbrOfThreads` threads should be executed). The `None` lets the compiler group the threads into local groups instead of the user. The remaining arguments containing `.data` pass the arguments to the `metropolis()`-kernel. Line 192 prevent halts the execution of the host code until the GPU has finished it's instructions. In lines 197-201 the `pyopencl.array`-objects that were passed to the kernel as arguments are now fetched from the graphics card to the RAM with the `.get()`-method, sinc they are now filled with calculated operator means and more. The mean acceptance rate for all the `nbrOfThreads` threads is calculated at line and the bins counts from all individual threads are summed into one resulting array of bin counts. Lines 203—207 return the results from this kernel run of `nbrOfThreads` kernel executions.

7.3 Example of Python Application Code

As stated earlier, this part of the program will need to be customized for different applications of the Feynman path integral. Some examples of what this code can accomplish include examining how a perturbation parameter affect the ground state energy of a quantum mechanical system, plotting the probability density of a system or examining how different choices of parameters such as N and $t_B - t_A$ affect the calculated operator means. An example of application code is shown in E.2.3 and will be explained here.

Lines 45—65: Here, some physical parameters are set, as well as the desired number of threads and kernel runs. The variables are explained by their comments on their right

Lines 117—135: Here, some example potentials and operators are formulated. The syntax for using several particles should be clear. To recap: The potential for the quantum mechanical system needs to be specified in order to use path integral formalism. This is done by first entering the number of particles in the system and then writing a valid C-expression where `x1`, `x2`, ... and so on are used as particle positions. After that, an operator that depends on the position of the particles can be entered and the mean of this operator will then be calculated. This operator can be chosen cleverly by use of the virial theorem as an operator whose mean is the energy of the system. In addition to this, a numerical representation of the probability density of the system will be returned when calling `getOperatorMean()`. This can be used for plotting or doing further calculations, if desired. If only the probability density is of interest, then one may set `operator = '0.0f'` for faster execution. Lines 117—125 are good examples of how the `userCode` variable is utilized in lines 78—108. The `userCode` variable is simply a string that will be pasted into the OpenCL kernel and can thus contain additional C-functions that can be used when specifying `potential` and `operator`.

Lines 140—143: Here, the call to `getOperatorMean()` is made and thus the path integral calculations are started. The results are saved to the variables on the left hand side in lines 140—141. All the following lines analyze the data in different ways.

Lines 146—223: First, the calculated operator mean and standard deviation is printed, as well as the mean acceptance rate for the Metropolis algorithm and the time taken for the whole calculation. If the calculation was done for one or two particles, then the probability density can be easily represented visually, either by a one dimensional plot, or by isocurves in two dimensions. In lines 161—234, these plots are created and saved to a file.

8 Results

8.1 Theoretical Results

A system of particles in a potential can through a change of coordinates be transformed so that the kinetic energy of the system can be expressed in an expression where all coordinates are treated the same way. This simplifies future calculations. The system can be reduced by one degree of freedom if the potential only depends on the relative locations of the particles, using the so called Jacobi coordinates.
The propagator of a system can be evaluated using:

$$K(b,a) = \int e^{i\frac{S[x(t)]}{\hbar}} \mathcal{D}x(t) \tag{8.1}$$

This can be evaluated numerically by discretising the integral.
The probability density function for finding the ground state system with coordinates x can be found by evaluating

$$|\psi_0(x)|^2 = \lim_{\beta_{ba}\to\infty} \frac{\int_{x'}^{x'} \mathcal{D}x(t) e^{-\frac{S_E[x(t)]}{\hbar}}}{\int_{-\infty}^{\infty} dx' \int_{x'}^{x'} \mathcal{D}x(t) e^{-\frac{S_E[x(t)]}{\hbar}}} \tag{8.2}$$

The expectation value of an operator A of the form

$$\hat{A} = \int |x\rangle A(x) \langle x| \mathrm{d}x \tag{8.3}$$

on a system at temperature $T = \frac{\hbar}{K_B \beta_{ba}}$ can be found by solving

$$\langle \hat{A} \rangle = \frac{\int_{-\infty}^{\infty} dx' \int_{x'}^{x'} \mathcal{D}x(t) e^{-\frac{S_E[x(t)]}{\hbar}} A(x(t_1))}{\int_{-\infty}^{\infty} dx' \int_{x'}^{x'} \mathcal{D}x(t) e^{-\frac{S_E[x(t)]}{\hbar}}} \tag{8.4}$$

where the time evolution in euclidean time is β_{ba}. The ground state energy can't be directly found this way because the Hamilton operator can not be written in said form. The expectation value of the Hamilton operator on the ground state can be shown by the virial theorem to be the expectation value of another operator which can be written in said form. This operator is given by

$$\hat{O} = \hat{V} + \frac{1}{2} \hat{x} \nabla \hat{V} \ . \tag{8.5}$$

The ground state energy is given when the temperature approaches 0, which is the same as when β_{ba} approaches infinity.

8.2 Numerical Algorithms

8.3 Implementation Calculations

Here we will present a series of tests done with the implemented algorithm given in section . The purpose of these test is to see how well our algorithm performs for different systems and what parameters that give good results. As as an operator we have in this paper only used the Hamiltonian with the kinetic energy term rewritten in terms of the potential energy using the virial theorem, see 2.10. This operator is written as

$$\hat{H} = \hat{V} + \frac{1}{2}\hat{x} \cdot \nabla \hat{V} \tag{8.6}$$

We want to once again emphasize that the following results are valid only for distinguishable particles.

8.4 Harmonic oscillator

The quantum harmonic oscillator, e.g $V(x) = \frac{m\omega^2}{2}x^2$, is a good potential to start doing simulations of. Analytical solutions to energies and probability densities exists which can be used to check how well our implementation can reproduce already known results.

The ground state energy for a harmonic oscillator is $\frac{1}{2}\hbar\omega$. We will work in units such that $m = 1, \hbar = 1$ and $\omega = 1$. This gives ground-state energy $\frac{1}{2}$ for the harmonic oscillator.

8.4.1 Uncoupled Harmonic Oscillators

The ground state energies for uncoupled harmonic oscillator were calculated. These can be seen in table 2. In the case with one particle the probability distribution was also calculated with 200 bins, 9. Since the particles do not interact with each other the ground state energy of the system is simply the number of particles multiplied with the ground state energy for one harmonic oscillator which in our units is $\frac{1}{2}$. For this calculation we used 128 nodes in a path for every particle. $4 \cdot 448$ threads did $8 \cdot 10^5$ iterations of the algorithm. For desired thermalization the algorithm ran $1.5 \cdot 10^3$ iterations. The results are given in table 2.

Table 2: Numerical results for the energy in a system of non interacting particles in a harmonic potential

Number of particles	Mean	Standard error	Analytical
1	0.4990	0.00017	0.5
2	0.9987	0.00034	1.0
3	1.4972	0.00050	1.5
4	1.9959	0.00067	2.0
5	2.4946	0.00083	2.5
6	2.9922	0.00098	3.0
7	3.4918	0.00113	3.5
8	3.9933	0.00128	4.0

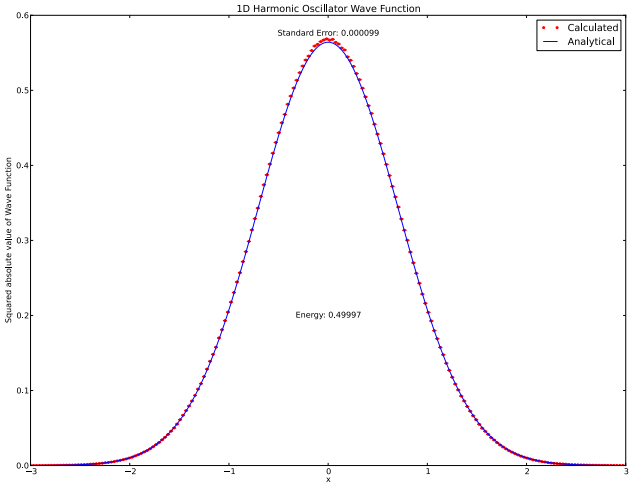


Figure 9: $|\psi|^2$ for particle in a harmonic oscillator potential.

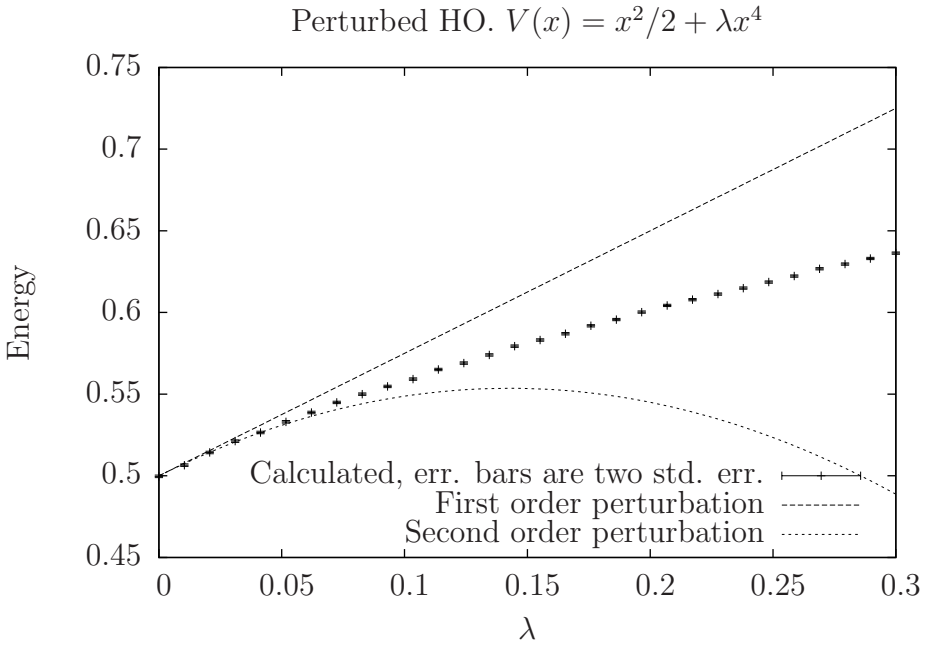
8.4.2 Perturbed single harmonic oscillator

To further analyse the capabilities of our program we gave it a perturbed harmonic oscillator. The potential was of the form

$$V(x) = \frac{x^2}{2} + \lambda x^4 \tag{8.7}$$

with λ determining the magnitude of the perturbation. The program looped over lambda and calculated the energy for the system for every loop. The results are given in figure 10. An analytical solution is derived in 3.17 to second order in λ, this is used as reference. As can be seen for very small values of λ the first order works well. The second order is accurate a while longer but quite quickly both the first and second order approximation accuracies become bad.

Figure 10: Plot of energy vs lambda for a perturbed harmonic oscillator



8.4.3 Coupled Harmonic Oscillators

Simulations of coupled harmonic oscillators were also done. This is a system where every particle exerts a force on the other particles proportional to

Table 3: Numerical results for coupled harmonic oscillators.

Number of particles	Mean	Standard error	Analytical
2	0.7053	0.00026	$1/\sqrt{2} \approx 0.7071$
3	1.7248	0.00041	$\sqrt{3} \approx 1.7321$
4	2.9854	0.00055	3
5	4.4445	0.00067	$2\sqrt{5} \approx 4.4721$
6	6.0762	0.00079	$5\sqrt{3/2} \approx 6.1237$
7	7.8615	0.00089	$3\sqrt{7} \approx 7.9372$
8	9.8016	0.00100	$7\sqrt{2} \approx 9.8995$

the distance between them. Such a system would for three particles have a potential energy

$$V = \left((x_1 - x_2)^2 + (x_1 - x_3)^2 + (x_2 - x_3)^2\right)/2 \tag{8.8}$$

The analytical energies were derived by the method of Jacobi coordinates, see section 3.4. The calculations were done with $4 \cdot 448$ threads, 20 runs with $8 \cdot 10^5$ loopings and $1.5 \cdot 10^6$ burnin runs. The system was transformed to Jacobi coordinates to remove the free-particle part. The results are given in table 3.

8.5 Gaussian potential

Some measurements on a potential that used two gaussian terms to simulate repulsive short-range and attractive mid-range nucleon-nucleon two-body potentials were done. The model is taken from [33]. It have the form

$$V(x) = \frac{V_1}{\sigma_1\sqrt{\pi}} e^{-x^2/\sigma_1^2} + \frac{V_2}{\sigma_2\sqrt{\pi}} e^{-x^2/\sigma_2^2}$$

We have as in [33] done calculations on two potentials, V_α with $V_1 = 12$, $V_2 = -12$, $\sigma_1 = 0.2$ and $\sigma_2 = 0.8$. V_β with $V_1 = 0$, $V_2 = -2$, $\sigma_2 = 0.8$. The first potential has thus both an attractive and a repulsive part while the second potential is attractive. The calculated energies where for -10.2 V_α and -1.78 for V_β. Note that these results are not comparable to the results in [33] because we consider distinguishable particles. The probability density functions are shown in figures 11 and 12.

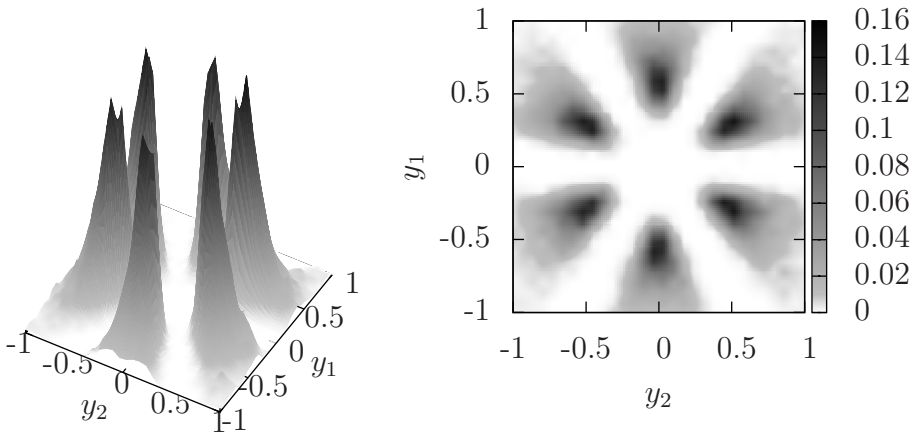


Figure 11: $|\psi_0|^2$ for V_α in Jacobi coordinates

We can see six peaks in the probability density. This is likely related to the number of ways the order of three objects can be permuted. The particles have lowest energy at specific distances from each other and they thus place themselves at a line with equal distances between them.

8.6 Choice of parameters

To see how our parameters affected the computed result of the energy operator we did measurements when where we varied the nodes in a path, N, and the time interval, t_b, for a path. The results are shown in figure 13. We can see that for times less than 3 the resulting energies are quite erratic. It is likely that the particle is in a mix of the ground state together with excited states. This causes many of our assumptions made when creating the algorithm to fail and it can no longer be used. For time equal to 4 the energy starts to go down at around $N = 200$. It is possible that the kinetic energy term in the action is diverging and causing instabilities. For times

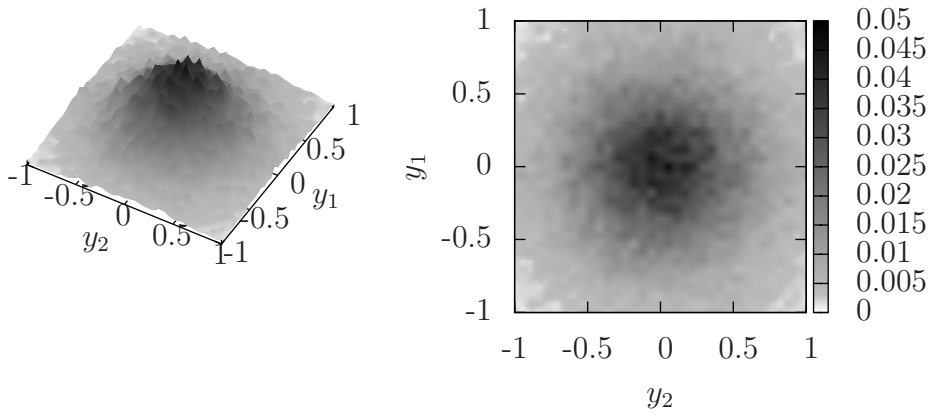


Figure 12: $|\psi_{=}|^2$ for V_β in Jacobi coordinates

larger than 4 the energies seem to converge quite nicely. However, it is likely that the energies for these times will do the same decay as for the

$$t_b = 4$$

energies when N become large enough. Good parameters is therefore a compromise between fast convergence and stability.

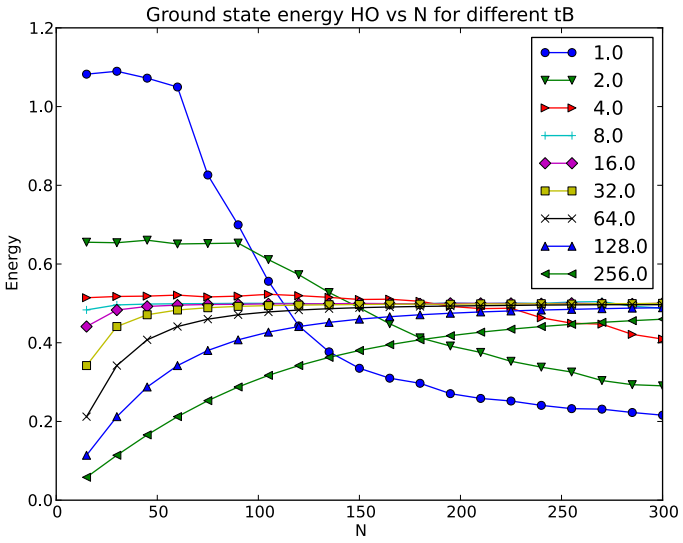


Figure 13: Energies for the one particle harmonic oscillator was computed for different time intervals ,t_b, and different number of nodes in a path, N.

9 Discussion

9.1 Theory

In our introductory courses in quantum mechanics it has been very hard to relate to quantum mechanics. These courses only use the Schrödinger approach which does not really have any obvious connections to classical mechanics. We feel that it is easier to relate to the path-integral formalism of quantum mechanics than the Schrödinger approach because it uses the action from classical mechanics to describe quantum-mechanical systems. However, in the mathematical aspect it is far more complicated than the Schrödinger approach. Maybe it is because of this that most introductory quantum mechanical courses use the simpler Schrödinger approach.

It was interesting to see how statistical physics emerged from our quantum mechanical equations. This gave us information about observables at temperatures above 0 K even though that was not what we set of to find in the first place.

During this project we have understood that the path-integral formalism demands a lot of calculations even for simple problems such as the one-dimensional harmonic oscillator. However, we have seen that this method is easily adapted to various problems and it does not demand a lot more

calculations for systems with more degrees of freedom.

Possible theoretical extension could be to consider identical particles (bosons and fermions) and derive expression to calculate the energy and probability distribution for the ground state. This is described in for example [3]. It would also be interesting to calculate expectation values of operators on excited states. As said in the introduction an interesting application of our program would be lattice QCD calculations, [4].

9.2 Numerical methods

In our results for the energy of the system we are usually a bit off from analytically calculated values even by allowing room for quite many standard errors. It is likely that our algorithm does not converge to the correct value due to how we discretized the action. Improvements on this are needed for better results.

One problem that frequently came up in the algorithm was the divergence of the kinetic energy term in the energy estimator. The estimation for the velocity in a path was a naive first order approximation and it is not surprising this gave us inaccuracies. It would be interesting to study better approximations of the energy. Work on this has already been done[32] and implementing their result is likely to give better results. Another improvement if you use their energy estimators is that convergence is faster and you can reduce the number of nodes in a path in the calculation.

9.3 Implementation

None of the group members have had previous experiences with GPGPU programming. As of such, the implementation with OpenCL has been a new and exotic experience for everyone. Working with the GPU is not as simple as learning a new programming language. It is rather a matter of learning a new programming paradigm, in this case parallelism. This challenges the programmer to break free from much of the philosophy and previous experience with CPU programming. This has imposed several restrictions for the implementation, as the time span has been relatively short. These restrictions are both result and method based.

First off all, the optimization in the implementation is less than optimal. GPGPU programming is pretty advanced and requires a lot of programming experience to master. More time spent on learning OpenCL programming would lead to faster and more powerful applications, but probably not much improvement in extent of the problems solved. This is rather a matter of

creating new numerical models. The result code is quite versatile in what kind of models it can handle.

The main performance boost of GPU and OpenCL calculations comes from parallelizing as big portion of the code and algorithm as possible. Parallelizing Markov Chain Monte Carlo calculations however is a tough challenge, since each path is a serial computation. Every change to the path is based on all previous changes. There exist no efficient way to split a path into different parts for several threads to handle. Instead, the parallelization has to occur over other dimensions. In this case, the advantage of many threads resulted in many ensembles of calculated paths, as each thread managed an individual path. Since all the particles interact with each other, it is not possible to parallelize over them.

One major drawback of parallelizing the algorithm by running many independent instances is the fact that burn-in has to be performed on each individual instance. This means that the more threads that are used, the greater amount of ”useless” computation has to be performed. Considering the fact that a graphics card with 14 multiprocessors, each consisting of 32 Shader Processors for a total of 448 computing CUDA cores need much more than 448 threads to perform optimally, this becomes an even greater setback.

An important aspect is to determine whether parallelizing Markov Chain Monte Carlo problems with GPU is worth it or not. An estimation of this was done by running the OpenCL code and comparing with the equivalent C-code looped as many times as there were threads. Quick tests showed that the OpenCL code out-performed the C-code by a factor of 10. These tests were not rigorous enough though. Improvements might be a subject for an extension. On the AMD platforms used, ATI Stream SDK was used to enable the CPU as a device in OpenCL. This turned up with different results on different platforms, one being roughly 5% faster on the CPU and the other 10 times as slow on the CPU. Both CPUs had 4 cores and were as new as the GPUs, so the difference probably lies in clock speed and hyper-threading capabilites. With hyper-threading enabled, the CPU will have boosted parallelization performance. A good benchmark would include an estimation of floating point operations per second and memory bandwidth. The subject of this benchmarks would preferably encompass a bunch of different ATI and nVidia graphics cards, Windows and Linux operative systems running in 32-bit and 64-bit mode. This in order to take into account effects of different platforms.

10 Acknowledgements

Christian Forssén - For his excellent wiki that helped us get a good start and his (almost) endless patience answering our questions.
Håkan Johansson - For a lot of help with general computer stuff.
Ulf Assarsson and Markus Billeter - For helping us get a better understanding of GPU architecture.
The Fundamental Physics instiutions - For letting us use one of their room, two computers with good graphics cards.
Andreas Klöckner and the PyOpenCL mailing list[64] - For a lot of in-depth help with the GPU implementation.
And last but not least a big thanks to the elevator repair man who rescued half of the authors from a stuck elevator at the evening of the deadline.

References

[1] Feynman, R. P. (1948) The Space-Time Approach to Non-Relativistic Quantum Mechanics.*Rev. Mod. Phys.* vol. 20, pp. 367-387

[2] Feynman, R. P., Hibbs, A.R., Styer, D.F. (2005) *Quantum Mechanics and Path Integrals.* Emended Edition. New York: Dover.

[3] Ceperley, D. M. (1995) Path integrals in the theory of condensed helium.*Rev. Mod. Phys.* vol. 67, pp. 279-355

[4] Wilson, K. G. (1974) Confinement of quarks.*Phys. Rev. D* vol. 10, pp. 2445-2459

[5] Gattringer, C., Lang, C. B. (2009) *Quantum Chromodynamics on the Lattice* (Lecture Notes in Physics, 788) :Springer

[6] Zinn-Justin, J. (2005) *Path Integrals in Quantum Mechanics.* New York: Oxford University Press Inc.

[7] Sakurai, J.J. (1993) *Modern Quantum Mechanics.* Revised Edition. Boston: Addison Wesley.

[8] Hjort-Jensen, M. (2010) *Computational Physics.* Oslo: University of Oslo.

[9] Rice, J. A. (2007) *Mathematical Statistics and Data Analysis.* Third edition. Pacific Grove: Duxbury Press.

[10] Morningstar, C. (2007) The Monte Carlo method in quantum field theory. *arXiv.* `http://arxiv.org/abs/hep-lat/0702020v1` (2011/04/14)

[11] Berg, B.A. (2004) *Markov Chain Monte Carlo Simulation and their Statistical Analysis.* Singapore: World Scientific Publishing.

[12] Cornille, P. (2003) *Advanced Electromagnetism and Vacuum Physics.* Vol. 21, Singapore: World Scientific Publishing.

[13] Rattazzi, R. (2009) *The Path Integral approach to Quantum Mechanics - Lecture Notes for Quantum Mechanics IV* New Jersey: Rutgers University.

[14] Landau, R.H., Páez, M.J., Bordeianu, C.C. (2007) *Computational Physics: Problem Solving with Computers.* 2nd rev. Weinheim: Wiley-VCH.

[15] Institutionen fö r Teknisk Fysik och Fundamental Fysik (2010) *Kvantfysik del 2* Göteborg: Chalmers Tekniska Högskola

[16] Bobrowski, M. *Ehrenfest's Theorem and Quantum Virial Theorem.* `http://www.caelestis.de/dateien/UEA05_2.pdf` (2011/05/11).

[17] Marsaglia, George, 1995, The Marsaglia Random Number CDROM, with The Diehard Battery of Tests of Randomness, produced at Florida State University. Access available at `www.stat.fsu.edu/pub/diehard`.

[18] Marsaglia, George and Tsang, Wai Wan, 2002, Some difficult-to-pass tests of randomness, *Journal Statistical Software*, **Vol. 7**, Issue 3.

[19] Marsaglia, July 2002, Xorshift RNGs, *Journal Statistical Software*, **Vol. 7**, Issue 3.

[20] Khronos Group. *clCreateContext specification.* `http://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/clCreateContext.html` (2011/05/14).

[21] Khronos Group. *OpenCL Specification 1.x.* `http://www.khronos.org/registry/cl/specs/opencl-1.x-latest.pdf` (2011/05/14).

[22] Andreas Klöckner. *PyOpenCL Official Documentation.* `http://documen.tician.de/pyopencl/` (2011/05/14).

[23] nVidia. *nVidia OpenCL Best Practices Guide.* `http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf` (2011/05/14).

[24] AMD/ATI. *ATI Stream SDK OpenCL Programming Guide.* `http://developer.amd.com/gpu_assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdf` (2011/05/14).

[25] AMD/ATI. *ATI Radeon HD 5850 Specification.* `http://www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/hd-5850/Pages/ati-radeon-hd-5850-overview.aspx#2` (2011/05/14).

[26] nVidia. *nVidia GeForce GTX 470 Specification.* `http://www.nvidia.com/object/product_geforce_gtx_470_us.html` (2011/05/14).

[27] CMSoft. *Advanced Aspects of the OpenCL C99 Language.* `http://www.cmsoft.com.br/index.php?option=com_content&view=category&layout=blog&id=92&Itemid=144` (2011/05/17).

[28] Python.org. *Python 3.0 Official Release Info.* http://www.python.org/download/releases/3.0/ (2011/05/14).

[29] Lutz, Mark (2009) *Learning Python.* Fourth Edition. Sebastopol: O'Reilly Media.

[30] CodeProject.com. *image001.jpg.* Part 2: OpenCL - Memory Spaces. http://www.codeproject.com/KB/showcase/Memory-Spaces.aspx (2011/05/14).

[31] The Portland Group. *v2n4a1i2.png.* PGI CUDA-x86: CUDA Programming for Multi-core CPUs . http://www.pgroup.com/lit/articles/insider/v2n4a1.htm (2011/05/14).

[32] J. Grujic, A. Bogojevic, A. Balaz. (2006) *Energy estimators and calculation of energy expectation values in the path integral formalism.* 360, 2, 217-223

[33] Jurgenson, E.D., Furnstahl, R.J. (2009) Similarity renormalization group evolution of many-body forces in a one-dimensional model. *Nuclear Physics A*, 818, 152-173.

Further Reading

[34] Python.org, PyPI. *PyOpenCL Package Index and Download.* http://pypi.python.org/pypi/pyopencl (2011/05/14).

[35] Andreas Klöckner. *PyOpenCL Description.* http://mathema.tician.de/software/pyopencl (2011/05/14).

[36] Andreas Klöckner. *PyOpenCL Official Wiki.* http://wiki.tiker.net/PyOpenCL (2011/05/14).

[37] Andreas Klöckner. *Custom Reduction.* http://documen.tician.de/pyopencl/array.html#module-pyopencl.reduction (2011/05/14).

[38] http://git.tiker.net/pyopencl.git/blob/HEAD:/pyopencl/reduction.py (2011/05/14).

[39] Wikipedia. *Folding (higher-order function).* https://secure.wikimedia.org/wikipedia/en/wiki/Fold_%28higher-order_function%29 (2011/05/14).

[40] CMSoft. *Using OpenCL Image2D Variables (Guide).* http://www.khronos.org/news/permalink/cmsoft-image2d-tutorial-utilizing-opencl (2011/05/18)

[41] AMD/ATI. *ATI Stream Technology.* http://www.amd.com/US/PRODUCTS/TECHNOLOGIES/STREAM-TECHNOLOGY/Pages/stream-technology.aspx (2011/05/14).

[42] Christoph Gohlke. *Unofficial Windows Binaries for Python Extension Packages.* http://www.lfd.uci.edu/~gohlke/pythonlibs/ (2011/05/14).

[43] Andreas Klöckner. *PyOpenCL Installation Guide (Official).* http://wiki.tiker.net/PyOpenCL (2011/05/14).

[44] Khronos Group. *OpenCL Specification Website.* http://www.khronos.org/opencl/ (2011/05/14).

[45] nVidia. *CUDA Zone.* http://www.nvidia.com/object/cuda_home.html (2011/05/15).

[46] Wikipedia. *Microsoft DirectCompute.* http://en.wikipedia.org/wiki/DirectCompute (2011/05/15).

[47] Wikipedia. *GPGPU.* http://en.wikipedia.org/wiki/GPGPU (2011/05/15).

[48] Wikipedia. *C99.* http://en.wikipedia.org/wiki/C99 (2011/05/14).

[49] AMD/ATI. *Porting CUDA Applications to OpenCL (Guide).* http://developer.amd.com/zones/OpenCLZone/programming/pages/portingcudatoopencl.aspx (2011/05/14).

[50] Andreas Klöckner. *CUDA vs OpenCL: Which should I use (Guide).* http://wiki.tiker.net/CudaVsOpenCL (2011/05/14).

[51] Andreas Klöckner. *Personal Website.* http://mathema.tician.de/ (2011/05/14).

[52] Wikipedia. *Resource Acquisition Is Initialization.* http://en.wikipedia.org/wiki/RAII (2011/05/14).

[53] Wikipedia. *MD5 Hashing.* http://en.wikipedia.org/wiki/MD5 (2011/05/14).

[54] nVidia. *High Performance Computing - Supercomputing with Tesla GPUs.* http://www.nvidia.com/object/tesla_computing_solutions.html (2011/05/14).

[55] SciPy.org. *NumPy Package Index.* http://numpy.scipy.org/ (2011/05/14).

[56] Sourceforge. *MatPlotLib Package Index.* http://matplotlib.sourceforge.net/ (2011/05/14).

[57] SciPy.org. *PyLab Package Index.* http://www.scipy.org/PyLab (2011/05/14).

[58] SciPy.org. *SciPy Package Index.* http://www.scipy.org/ (2011/05/14).

[59] Boost.org. *Boost C++ Libraries for Python.* http://www.boost.org/doc/libs/1_46_1/libs/python/doc/ (2011/05/14).

[60] Code.Google. *PyMC - Markov Chain Monte Carlo for Python.* http://code.google.com/p/pymc/ (2011/05/14).

[61] PyGTK. *PyGTK: GTK+ for Python.* http://www.pygtk.org/ (2011/05/14).

[62] Python.org . *SetupTools Package Index.* http://pypi.python.org/pypi/setuptools (2011/05/14).

[63] Wikipedia. *Mersenne Twister.* http://en.wikipedia.org/wiki/Mersenne_twister (2011/05/18).

[64] PyOpenCL Mailing List. *PyOpenCL - Development and Support List for PyOpenCL.* http://lists.tiker.net/listinfo/pyopencl (2011/05/18).

A Formula

This section contains basic quantum mechanics formula that are used in the paper.

- The identity operator for a space spanned by the discrete orthonormal basis $\{|i\rangle\}$ can be expressed as

$$\hat{\mathbf{I}} = \sum_i |i\rangle\langle i| \ . \tag{A.1}$$

  Generalized to a space spanned by the continuous orthonormal basis $\{|\xi\rangle\}$ the identity operator is given by

$$\hat{\mathbf{I}} = \int d\xi |\xi\rangle\langle\xi| \ . \tag{A.2}$$

- An arbitrary state can be expressed using the identity operator. Using the discrete identity operator a state can be expressed as

$$|\psi\rangle = \sum_i |i\rangle\langle i|\psi\rangle = \sum_i \psi_i |i\rangle \ . \tag{A.3}$$

  With the continuous identity operator we get

$$|\psi\rangle = \int d\xi |\xi\rangle\langle\xi|\psi\rangle = \int d\xi \psi(\xi)|\xi\rangle \ . \tag{A.4}$$

  $\psi_i = \langle i|\psi\rangle$ and $\psi(\xi) = \langle\xi|\psi\rangle$ are the discrete and continuous wave functions for a system.

- The inner product between orthonormal basis states can be written as

$$\langle i|j\rangle = \delta_{ij} \tag{A.5}$$

  for a discrete system, and written as

$$\langle\xi'|\xi\rangle = \delta(\xi' - \xi) \tag{A.6}$$

  for a continuous system.

B Derivation of examples

B.1 Free-particle propagator

The Lagrangian for a free particle in one dimension is

$$L = \frac{m\dot{x}^2}{2} \tag{B.1}$$

where x is the position of the particle. The action for a path $x(t)$ from time t_a to time t_b is thus

$$S[x(t)] = \int_{t_a}^{t_b} \frac{m\dot{x}^2}{2} \mathrm{d}t. \tag{B.2}$$

The propagator from x_a to x_b for the particle is the path-integral of $\mathrm{e}^{\frac{i}{\hbar}S[x(t)]}$ over all paths connecting x_a to x_b at the given times:

$$K(x_b, t_b, x_a, t_a) = \int_{x_a}^{x_b} \mathrm{e}^{S[x(t)]} \mathcal{D}x(t) \tag{B.3}$$

The path integral is carried out as a limit. The time is discretized in $t_0, ..., t_N$ where $t_{i+1} = t_i + \epsilon$. The particles position at time t_i is x_i. The paths are by definition of path integrals linear functions of time between the times t_i. This gives the speed between time t_i and t_{i+1}:

$$\dot{x} = \frac{x_{i+1} - x_i}{\epsilon}. \tag{B.4}$$

The action for these discretized paths is then

$$S[x(t)] = \sum_{i=0}^{N-1} \frac{(x_{i+1} - x_i)^2}{2m\epsilon}. \tag{B.5}$$

The integral over all paths is carried out by integrating over all x_i, except x_0 and x_N which are given by x_a and x_b. The limit is taken as ϵ goes to 0, and thus the number of integrals goes to infinity. This is the definition of path integrals:

$$K = \lim_{\epsilon \to 0} \frac{1}{A} \int ... \int \exp\left(\frac{im}{2\hbar\epsilon} \sum_{i=0}^{N-1} (x_{i+1} - x_i)^2 \right) \frac{\mathrm{d}x_1}{A} ... \frac{\mathrm{d}x_{N-1}}{A} \tag{B.6}$$

A is an unknown normalization factor needed for the limit to exist. A depends on N or alternatively on ϵ. The limit is obtained by finding an expression

$I(N)$ for N time slices and taking the limit as N approaches ∞. The expression for N time slices is found by a iterative process. The first integration gives

$$I(N) = \frac{1}{A}\int ... \int \exp\left(\frac{im}{2\hbar\epsilon}\sum_{i=2}^{N-1}(x_{i+1}-x_i)^2\right)$$
$$\int \exp\left(\frac{im}{2\hbar\epsilon}((x_2-x_1)^2+(x_1-x_0)^2)\right)\frac{\mathrm{d}x_1}{A}\frac{\mathrm{d}x_2}{A}...\frac{\mathrm{d}x_{N-1}}{A} \tag{B.7}$$
$$= \frac{1}{A}\int ... \int \exp\left(\frac{im}{2\hbar\epsilon}\sum_{i=2}^{N-1}(x_{i+1}-x_i)^2\right)$$
$$\frac{1+i}{A}\sqrt{\frac{\pi\hbar\epsilon}{2m}}\exp\left(\frac{im}{2\cdot 2\hbar\epsilon}(x_2-x_0)^2\right)\frac{\mathrm{d}x_2}{A}...\frac{\mathrm{d}x_{N-1}}{A} \tag{B.8}$$
$$= \frac{1}{A}\frac{1+i}{A}\sqrt{\frac{\pi\hbar\epsilon}{2m}}\int ... \int \exp\left(\frac{im}{2\hbar\epsilon}\sum_{i=3}^{N-1}(x_{i+1}-x_i)^2\right)$$
$$\int \exp\left(\frac{im}{2\hbar\epsilon}\left(\frac{(x_2-x_0)^2}{2}+(x_3-x_2)^2\right)\right)\frac{\mathrm{d}x_2}{A}\frac{\mathrm{d}x_3}{A}...\frac{\mathrm{d}x_{N-1}}{A} \tag{B.9}$$

The integration of one integral gave a similar expression as the one we started with, equation (B.7). It gave a factor

$$\frac{1+i}{A}\sqrt{\frac{\pi\hbar\epsilon}{2m}}, \tag{B.10}$$

removed one term in the sum and changed a constant factor on one of the terms in the exponent to be integrated. Integration of an integral like this, but with an arbitrary factor k_i in the exponent, gives:

$$\int \exp\left(\frac{im}{2\hbar\epsilon}\left(k_i(x_i-x_{i-1})^2+(x_{i+1}-x_i)^2\right)\right)\frac{\mathrm{d}x_i}{A}$$
$$= \frac{1+i}{A}\sqrt{\frac{2\pi\hbar\epsilon}{2(k_i+1)m}}\exp\left(\frac{k_i}{k_i+1}\frac{im(x_{i+1}-x_{i-1})^2}{2\hbar\epsilon}\right) \tag{B.11}$$

Integrating over first x_1 and then $x_2...x_{N-1}$ using the result above is thus posible because the integrand is always of the form above. The constants k_i are different for each integral. The first integral gives $k_1 = 1$ and the result above gives the next value after each integral $k_{i+1} = k_i/(k_i+1)$. This can be written as:

$$k_{i+1}^{-1} = \frac{k_i+1}{k_i} = k_i^{-1}+1 \tag{B.12}$$

k_i^{-1} will then go from 1 to $N-1$. The expression after all integrals have been performed will then become:

$$I(N) = \left(\prod_{k^{-1}=1}^{N-1} \frac{1+i}{A} \sqrt{\frac{\pi\hbar\epsilon}{(k+1)m}} \right) \exp\left(\frac{(N-1)^{-1}}{(N-1)^{-1}+1} \frac{im(x_N - x_0)^2}{2\hbar\epsilon} \right) \tag{B.13}$$

Recognizing that $\epsilon = \frac{t_b - t_a}{N}$ gives:

$$I(N) = A^{-N} \left(\sqrt{\frac{2i\pi\hbar\epsilon}{m}} \right)^{N-1} \left(\prod_{k^{-1}=1}^{N-1} \sqrt{\frac{k^{-1}}{(1+k^{-1})}} \right) \exp\left(\frac{im(x_N - x_0)^2}{2\hbar(t_b - t_a)} \right) \tag{B.14}$$

The denominator in the product cancels with the numerator of the next factor leaving:

$$I(N) = A^{-N} \left(\sqrt{\frac{2i\pi\hbar\epsilon}{m}} \right)^{N-1} \sqrt{\frac{1}{1+(N-1)}} \exp\left(\frac{im(x_N - x_0)^2}{2\hbar(t_b - t_a)} \right) \tag{B.15}$$

The normalization factor A is for a particle in a potential [2]:

$$A = \sqrt{\frac{2i\pi\hbar\epsilon}{m}} \tag{B.16}$$

Our particle is in the zero potential so equation (B.16) applies. This gives:

$$I(N) = \sqrt{\frac{m}{2i\pi\hbar(t_b - t_a)}} \exp\left(\frac{im(x_b - x_a)^2}{2\hbar(t_b - t_a)} \right) \tag{B.17}$$

This is independent of N so the limit is unessesary and the propagator for a free particle is thus:

$$K(x_b, t_b, x_a, t_a) = \sqrt{\frac{m}{2i\pi\hbar(t_b - t_a)}} \exp\left(\frac{im(x_b - x_a)^2}{2\hbar(t_b - t_a)} \right) \tag{B.18}$$

B.2 The harmonic oscillator

The harmonic oscillator is often encountered in the description of natural phenomena. Many quantum mechanical systems can be approximated as harmonic oscillators, an example is a vibrating diatomic molecule, another example is phonons in a crystal. We will here derive the propagator for the system usig path integrals. We will also find the eigenstates and eigenvalues of the system. A perturbated oscillator will also be studied. The reason for these studies is to be able to verify our numerical results.

B.2.1 Finding the propagator

The propagator for a harmonic oscillator from a to b is to be found. A harmonic oscillator in one dimension have one degree of freedom, x. The Lagrangian is

$$L(\dot{x}, x) = \frac{m}{2}\dot{x}^2 - \frac{m\omega^2}{2}x^2. \tag{B.19}$$

The Lagrange equation of motion is

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial L}{\partial \dot{x}}\right) - \frac{\partial L}{\partial x} = 0. \tag{B.20}$$

The classical path of the system is $\bar{x}(t)$ and it satisfies equation (B.20) and thus

$$m\ddot{\bar{x}} + m\omega^2\bar{x} = 0. \tag{B.21}$$

Solving this ordinary differential equation gives (the subtraction of t_a is just to simplify calculations later on)

$$\bar{x}(t) = A\cos\left(\omega(t - t_a)\right) + B\sin\left(\omega(t - t_a)\right). \tag{B.22}$$

Initial and final time, t_a, t_b, and position, x_a, x_b, for the system, yields

$$A = x_a \tag{B.23}$$

$$B = \frac{x_b - x_a\cos(\omega(t_b - t_a))}{\sin(\omega(t_b - t_a))}. \tag{B.24}$$

Any path, $x(t)$, for the system from a to b can be written as

$$x(t) = \bar{x}(t) + y(t) \tag{B.25}$$

where $y(t)$ is a path from 0 to 0 with the same time interval as $x(t)$. Thus the following is true

$$\begin{aligned} y(t_a) &= 0 \\ y(t_b) &= 0. \end{aligned} \tag{B.26}$$

The action of the path $x(t)$, $S[x(t)]$ is defined by

$$S[x(t)] = \int_{t_a}^{t_b} L(\dot{x}(t), x(t))\mathrm{d}t. \tag{B.27}$$

The substitution of equation (B.25) gives

$$
\begin{aligned}
S[x(t)] &= \int_{t_a}^{t_b} L(\dot{\bar{x}} + \dot{y}, \bar{x} + y)\mathrm{d}t && \text{(B.28)}\\
&= \frac{m}{2}\int_{t_a}^{t_b} \left(\dot{\bar{x}}^2 + \dot{y}^2 + 2\dot{\bar{x}}\dot{y} - \omega^2 x^2 - \omega^2 y^2 - 2\omega^2 \bar{x} y\right)\mathrm{d}t \\
&= \int_{t_a}^{t_b} \left(L(\dot{\bar{x}}, \bar{x}) + L(\dot{y}, y) + m\dot{\bar{x}}\dot{y} - m\omega^2 \bar{x} y\right)\mathrm{d}t \\
&= S[\bar{x}(t)] + S[y(t)] + \int_{t_a}^{t_b} \left(m\dot{\bar{x}}\dot{y} - m\omega^2 \bar{x} y\right)\mathrm{d}t \\
&= \left[\text{partial integration}\right] \\
&= S[\bar{x}(t)] + S[y(t)] + \left[m\dot{\bar{x}} y\right]_{t_a}^{t_b} + \int_{t_a}^{t_b} my\left(-\ddot{\bar{x}} - \omega^2 \bar{x}\right)\mathrm{d}t && \text{(B.29)}
\end{aligned}
$$

The last two terms are each zero by equations (B.26) and (B.21) respectively. This gives

$$S[x(t)] = S[\bar{x}(t)] + S[y(t)]. \tag{B.30}$$

A path integral of the action from a to b gives the kernel $K(b, a)$

$$K(b, a) = \int_a^b \mathrm{e}^{\frac{i}{\hbar}S[x(t)]}\mathcal{D}x(t) \tag{B.31}$$

This is, according to equation (B.30),

$$K(b, a) = \mathrm{e}^{\frac{i}{\hbar}S[\bar{x}(t)]}\int_0^0 \mathrm{e}^{\frac{i}{\hbar}S[y(t)]}\mathcal{D}y(t). \tag{B.32}$$

The action of the classical path $S[\bar{x}(t)]$ is given by

$$
\begin{aligned}
S[\bar{x}(t)] &= \int_{t_a}^{t_b} L(\dot{\bar{x}}(t), \bar{x}(t))\mathrm{d}t \\
&= \frac{m}{2}\int_{t_a}^{t_b} (\dot{\bar{x}}^2 - \omega^2 \bar{x}^2)\mathrm{d}t \\
&= \left[\text{partial integration}\right] \\
&= \frac{m}{2}\left[\dot{\bar{x}}\bar{x}\right]_{t_a}^{t_b} - \frac{m}{2}\int_{t_a}^{t_b} (\ddot{\bar{x}} + \omega^2 \bar{x})\bar{x}\mathrm{d}t. && \text{(B.33)}
\end{aligned}
$$

The integral is zero by equation (B.21). Insertion of equations (B.22) and (B.24) and defining $T \equiv t_b - t_a$ gives

$$\begin{aligned}
S[\bar{x}(t)] =& \frac{m\omega}{2}\left((-A\sin(\omega T) + B\cos(\omega T))x_b - Bx_a\right) \\
=& \frac{m\omega}{2}\left((-x_a\sin(\omega T) + \frac{x_b - x_a\cos(\omega T)}{\sin(\omega T)}\cos(\omega T))x_b\right. \\
& \left. - \frac{x_b - x_a\cos(\omega T)}{\sin(\omega T)}x_a\right) \\
=& \frac{m\omega}{2\sin(\omega T)}(-x_a x_b\sin^2(\omega T) + x_b^2\cos(\omega T) \\
& - x_a x_b\cos^2(\omega T) - x_a x_b + x_a^2\cos(\omega T)) \\
=& \frac{m\omega}{2\sin(\omega T)}((x_a^2 + x_b^2)\cos(\omega T) - 2x_a x_b). \qquad \text{(B.34)}
\end{aligned}$$

The path-integral I in equation (B.32) only depends on the time difference $T \equiv t_b - t_a$ between a and b.

$$I(T) = \int_0^0 \mathrm{e}^{\frac{i}{\hbar}S[y(t)]}\mathcal{D}y(t) \qquad \text{(B.35)}$$

Since the path starts and stops at 0 we can consider the paths as $2T$-periodic functions which are even about 0. They can thus be expressed as sine-series

$$y(t) = \sum_{n=1}^{\infty} a_n \sin\frac{n2\pi t}{2T} = \sum_{n=1}^{\infty} a_n \sin\frac{n\pi t}{T}. \qquad \text{(B.36)}$$

The integral over all paths is defined as the limit of integrals over positions at timeslices of the path. The integrals can instead be taken over the fourier coefficients a_n. This is just a linear variable substituition in the integral, requiring multiplication by the determinant J of the jacobian which by the linearity is constant

$$J = \left|\frac{\partial(x_1 ... x_N)}{\partial(a_1 ... a_N)}\right| \qquad \text{(B.37)}$$

The path-integral becomes

$$\begin{aligned}
I(T) &= \lim_{N\to\infty}\frac{1}{A}\int ... \int \exp\left(\frac{i}{\hbar}S[y(t)]\right)\frac{\mathrm{d}x_1}{A}...\frac{\mathrm{d}x_N}{A} \\
&= \lim_{N\to\infty} J\frac{1}{A}\int ... \int \exp\left(\frac{i}{\hbar}S[y(t)]\right)\frac{\mathrm{d}a_1}{A}...\frac{\mathrm{d}a_N}{A}. \qquad \text{(B.38)}
\end{aligned}$$

The action for a path in the new variables is

$$\begin{aligned} S[y(t)] &= \int_0^T \frac{m}{2}\left(\dot{y}^2 - \omega^2 y^2\right) \mathrm{d}t \\ &= \frac{m}{2}\int_0^T \left(\left(\sum_{n=1}^{N} a_n \frac{n\pi}{T} \cos\frac{n\pi t}{T} \right)^2 - \omega^2 \left(\sum_{n=1}^{N} a_n \sin\frac{n\pi t}{T} \right)^2 \right) \mathrm{d}t \\ &= \frac{m}{2}\int_0^T \left(\sum_{n=1,m=1}^{N,N} a_n a_m \frac{n\pi}{T}\frac{m\pi}{T} \cos\frac{n\pi t}{T}\cos\frac{m\pi t}{T} - \right. \\ &\quad \left. \omega^2 \sum_{n=1,m=1}^{N,N} a_n a_m \sin\frac{n\pi t}{T}\sin\frac{m\pi t}{T} \right) \mathrm{d}t \\ &= \frac{m}{2}\left(\sum_{n=1}^{N} a_n^2 \frac{n^2\pi^2}{T^2}\frac{T}{2} - \omega^2 \sum_{n=1}^{\infty} a_n^2 \frac{T}{2} \right) \\ &= \frac{mT}{4}\sum_{n=1}^{N} a_n^2 \left(\frac{n^2\pi^2}{T^2} - \omega^2 \right). \end{aligned} \tag{B.39}$$

The integrals are put inside the sums and evaluated in the third step, all integrals with $m = n$ evaluates to 0. This gives

$$I(T) = \lim_{N\to\infty} J\frac{1}{A}\int \ldots \int \prod_{n=1}^{N} \exp\left(\frac{i}{\hbar} a_n^2 \left(\frac{n^2\pi^2}{T^2} - \omega^2 \right) \right) \frac{\mathrm{d}a_1}{A} \ldots \frac{\mathrm{d}a_N}{A} \tag{B.40}$$

Each factor in the product just depends on one variable so the integrals can be done individually

$$\begin{aligned} I(T) &= \lim_{N\to\infty} J\frac{1}{A}\prod_{n=1}^{N}\int \exp\left(\frac{i}{\hbar} a_n^2 \left(\frac{n^2\pi^2}{T^2} - \omega^2 \right) \right) \frac{\mathrm{d}a_n}{A} \\ &= \lim_{N\to\infty} J\frac{1}{A}\prod_{n=1}^{N} \frac{1}{A}\sqrt{\frac{\pi i\hbar}{\frac{n^2\pi^2}{T^2} - \omega^2}}. \end{aligned} \tag{B.41}$$

Using the value of A for a particle in a potential, equation (2.15), gives

$$I(T) = \lim_{N\to\infty} J\sqrt{\frac{Nm}{2\pi i\hbar T}}\prod_{n=1}^{N}\sqrt{\frac{Nm}{2\pi i\hbar T}}\sqrt{\frac{\pi i\hbar}{\frac{n^2\pi^2}{T^2} - \omega^2}}. \tag{B.42}$$

If all factors independent of ω are collected in C_N the expression becomes

$$I(T) = \lim_{N\to\infty} C_N \left(\prod_{n=1}^{N} \left(1 - \frac{\omega^2 T^2}{n^2\pi^2} \right) \right)^{-\frac{1}{2}}. \tag{B.43}$$

The infinite product converges to $\sin(\omega T)/\omega T$ giving

$$I(T) \;=\; C\left(\frac{\sin(\omega T)}{\omega T}\right)^{-\frac{1}{2}} \tag{B.44}$$

where $C = \lim_{N\to\infty} C_N$. This, equations (B.32) and (B.34) gives the propagator with the unknown (but independent of ω) constant C:

$$K(b,a) = \exp\left(\frac{i}{\hbar}\frac{m\omega}{2\sin(\omega T)}((x_a^2+x_b^2)\cos(\omega T) - 2x_a x_b)\right) C\left(\frac{\sin(\omega T)}{\omega T}\right)^{-\frac{1}{2}} \tag{B.45}$$

We should get the free particle propagator when $\omega \to 0$ so it must be true that

$$\exp\left(\frac{im(x_b-x_a)^2}{2\hbar T}\right) C = \sqrt{\frac{m}{2i\pi\hbar T}}\exp\left(\frac{im(x_b-x_a)^2}{2\hbar T}\right). \tag{B.46}$$

This gives C

$$C = \sqrt{\frac{m}{2i\pi\hbar T}} \tag{B.47}$$

and thus the final expression for the propagator is

$$K(b,a) = \sqrt{\frac{m\omega}{2i\pi\hbar\sin(\omega T)}}\exp\left(\frac{im\omega\left((x_a^2+x_b^2)\cos(\omega T) - 2x_a x_b\right)}{2\hbar\sin(\omega T)}\right). \tag{B.48}$$

This can be expressed in dimensionless as done in 3.2. Equation (B.48) becomes

$$K(b,a) = \frac{1}{x_c}\sqrt{\frac{1}{2i\pi\sin(\tau)}}\exp\left(\frac{i\left((\xi_a^2+\xi_b^2)\cos(\tau) - 2\xi_a\xi_b\right)}{2\sin(\tau)}\right). \tag{B.49}$$

B.2.2 Finding the eigenstates

The propagator has eigenfunctions such that

$$\int K(b,a)\psi_i(\xi_a)\mathrm{d}(\xi_a x_c) = \exp\left(-i\varepsilon_i\tau\right)\psi_i(\xi_b) \tag{B.50}$$

where $E_i = \varepsilon_i E_c$ is the corresponding eigenenergy. These eigenfunctions are the Hermite functions defined in equation (3.7). That this is true will now

be shown. The first Hermite function $g_0(\xi)$ is an eigenfunction with energy $\varepsilon_0 = \frac{1}{2}$

$$\begin{aligned}
& \int K(b,a) g_0(\xi_a) \mathrm{d}(\xi_a x_c) \\
= & \int \sqrt{\frac{1}{2i\pi \sin(\tau)}} \exp\left(\frac{i\left(\left(\xi_a^2+\xi_b^2\right)\cos(\tau) - 2\xi_a\xi_b\right)}{2\sin(\tau)}\right) \\
& \pi^{-1/4} \exp\left(-\frac{\xi_a^2}{2}\right) \mathrm{d}\xi_a \\
= & \pi^{-3/4} \sqrt{\frac{1}{2i\sin(\tau)}} \exp\left(\frac{i\xi_b^2\cos(\tau)}{2\sin(\tau)}\right) \int \exp\left(\frac{-2i\xi_a\xi_b + \xi_a^2\left(i\cos(\tau) - \sin(\tau)\right)}{2\sin(\tau)}\right) \mathrm{d}\xi_a \\
= & \exp\left(\frac{i\xi_b^2\cos(\tau)}{2\sin(\tau)}\right) C_1 \int \exp\left(-C_2\xi_a - \xi_a^2 C_3\right) \mathrm{d}\xi_a \\
= & \exp\left(\frac{i\xi_b^2\cos(\tau)}{2\sin(\tau)}\right) C_1 \exp\left(\frac{C_2^2}{4C_3}\right) \int \exp\left(-C_3\left(\xi_a + \frac{C_2}{2C_3}\right)^2\right) \mathrm{d}\xi_a \\
= & \exp\left(\frac{i\xi_b^2\cos(\tau)}{2\sin(\tau)}\right) C_1 \exp\left(\frac{C_2^2}{4C_3}\right) \frac{1}{\sqrt{C_3}} \int \exp\left(-\eta_a^2\right) \mathrm{d}\eta_a \\
= & \pi^{-1/4} \exp\left(-\frac{1}{2}i\tau\right) \exp\left(\frac{-\xi_b^2}{2}\right). \qquad \text{(B.51)}
\end{aligned}$$

The Hermite functions obey the following recurrence formula:

$$g_n(\xi) = -(2n)^{-1/2} \exp\left(\frac{\xi^2}{2}\right) \frac{\mathrm{d}}{\mathrm{d}\xi}\left(\exp\left(\frac{-\xi^2}{2}\right) g_{n-1}(\xi)\right) \qquad \text{(B.52)}$$

The proof is made by induction, assuming that Hermite function g_{n-1} is an eigenfunction with energy $\varepsilon_{n-1} = n - \frac{1}{2}$ one can show that g_n is an eigenfunction with energy $\varepsilon_n = n + \frac{1}{2}$:

$$\begin{aligned}
& \int K(b,a) g_n(\xi_a) \mathrm{d}(\xi_a x_c) \\
= & \int \sqrt{\frac{1}{2i\pi \sin(\tau)}} \exp\left(\frac{i\left(\left(\xi_a^2+\xi_b^2\right)\cos(\tau) - 2\xi_a\xi_b\right)}{2\sin(\tau)}\right) \\
& (-1)(2n)^{-1/2} \exp\left(\frac{\xi_a^2}{2}\right) \frac{\mathrm{d}}{\mathrm{d}\xi_a}\left(\exp\left(\frac{-\xi_a^2}{2}\right) g_{n-1}(\xi_a)\right) \mathrm{d}\xi_a \qquad \text{(B.53)}
\end{aligned}$$

by using the recurrence formula of equation (B.52). Integration by parts gives:

$$\begin{aligned}
&\left[\sqrt{\frac{1}{2i\pi\sin(\tau)}}\exp\left(\frac{i\left((\xi_a^2+\xi_b^2)\cos(\tau)-2\xi_a\xi_b\right)}{2\sin(\tau)}\right)\right.\\
&\left.(-1)(2n)^{-1/2}\exp\left(\frac{\xi_a^2}{2}\right)\exp\left(\frac{-\xi_a^2}{2}\right)g_{n-1}(\xi_a)\right]_{-\infty}^{\infty}\\
&-\int\sqrt{\frac{1}{2i\pi\sin(\tau)}}\frac{\mathrm{d}}{\mathrm{d}\xi_a}\left(\exp\left(\frac{i\left((\xi_a^2+\xi_b^2)\cos(\tau)-2\xi_a\xi_b\right)+\xi_a^2\sin(\tau)}{2\sin(\tau)}\right)\right.\\
&\left.(-1)(2n)^{-1/2}\right)\exp\left(\frac{-\xi_a^2}{2}\right)g_{n-1}(\xi_a)\mathrm{d}\xi_a\\
=\;&\int\sqrt{\frac{1}{2i\pi\sin(\tau)}}\frac{i\exp(-i\tau)\xi_a-i\xi_b}{\sin(\tau)}\\
&\exp\left(\frac{i\left((\xi_a^2+\xi_b^2)\cos(\tau)-2\xi_a\xi_b\right)}{2\sin(\tau)}\right)\\
&(2n)^{-1/2}g_{n-1}(\xi_a)\mathrm{d}\xi_a
\end{aligned}\tag{B.54}$$

Multiplying and dividing by $\exp\left(\frac{\xi_b^2}{2}\right)$ gives:

$$\begin{aligned}
=\;&\exp\left(\frac{\xi_b^2}{2}\right)\exp\left(\frac{-{\xi_b}^2}{2}\right)\int\sqrt{\frac{1}{2i\pi\sin(\tau)}}\frac{i\exp(-i\tau)\xi_a-i\xi_b}{\sin(\tau)}\\
&\exp\left(\frac{i\left((\xi_a^2+{\xi_b}^2)\cos(\tau)-2\xi_a\xi_b\right)}{2\sin(\tau)}\right)\\
&(2n)^{-1/2}g_{n-1}(\xi_a)\mathrm{d}\xi_a
\end{aligned}\tag{B.55}$$

Integrating and differentiating gives:

$$\begin{aligned}
=\;&\exp\left(\frac{\xi_b^2}{2}\right)\frac{\mathrm{d}}{\mathrm{d}\xi_b}\int_{-\infty}^{\xi_b}\exp\left(\frac{-{\xi_b'}^2}{2}\right)\int\sqrt{\frac{1}{2i\pi\sin(\tau)}}\frac{i\exp(-i\tau)\xi_a-i\xi_b'}{\sin(\tau)}\\
&\exp\left(\frac{i\left((\xi_a^2+{\xi_b'}^2)\cos(\tau)-2\xi_a\xi_b'\right)}{2\sin(\tau)}\right)\\
&(2n)^{-1/2}g_{n-1}(\xi_a)\mathrm{d}\xi_a\mathrm{d}\xi_b'
\end{aligned}\tag{B.56}$$

Evaluating the integral over ξ_b' gives:

$$\begin{aligned}
&= -(2n)^{-1/2}\exp\left(\frac{\xi_b^2}{2}\right)\frac{\mathrm{d}}{\mathrm{d}\xi_b}\Bigg[\exp\left(-\frac{\xi_b^2}{2}\right)\int\sqrt{\frac{1}{2i\pi\sin(\tau)}}\frac{i}{\sin(\tau)} \\
&\quad \frac{1}{2}\exp\left(\frac{1}{2}i(\xi_a^2+\xi_b^2)\cot(\tau)-i\xi_a\xi_b\csc(\tau)\right)\cdot(\exp(-2i\tau)-1) \\
&\quad \cdot g_{n-1}(\xi_a)\mathrm{d}\xi_a\Bigg] \\
&= -(2n)^{-1/2}\exp(-i\tau)\exp\left(\frac{\xi_b^2}{2}\right)\frac{\mathrm{d}}{\mathrm{d}\xi_b}\Bigg[\exp\left(-\frac{\xi_b^2}{2}\right)\int\sqrt{\frac{1}{2i\pi\sin(\tau)}} \\
&\quad \exp\left(\frac{i((\xi_a^2+\xi_b^2)\cos(\tau)-2\xi_a\xi_b)}{2\sin(\tau)}\right) \\
&\quad g_{n-1}(\xi_a)\mathrm{d}\xi_a\Bigg] \qquad \text{(B.57)}
\end{aligned}$$

Using the kernel of equation (B.49) gives:

$$\begin{aligned}
&= -(2n)^{-1/2}\exp(-i\tau)\exp\left(\frac{\xi_b^2}{2}\right)\frac{\mathrm{d}}{\mathrm{d}\xi_b}\Bigg[\exp\left(-\frac{\xi_b^2}{2}\right) \\
&\quad \int K(b,a)g_{n-1}(\xi_a)\mathrm{d}(\xi_a x_c)\Bigg] \qquad \text{(B.58)}
\end{aligned}$$

Using the induction assumption gives:

$$\begin{aligned}
&= -(2n)^{-1/2}\exp(-i\tau)\exp\left(\frac{\xi_b^2}{2}\right)\cdot \\
&\quad \cdot\frac{\mathrm{d}}{\mathrm{d}\xi_b}\left[\exp\left(-\frac{\xi_b^2}{2}\right)\exp\left(-i\left(n-\frac{1}{2}\right)\tau\right)g_{n-1}(\xi_b)\right] \qquad \text{(B.59)}
\end{aligned}$$

And finally, the recurrence formula in equation (B.52) for the hermite functions gives:

$$= \exp\left(-i\left(n+\frac{1}{2}\right)\tau\right)g_n(\xi_b) \qquad \text{(B.60)}$$

By induction are thus all g_n eigenfunctions with energies $\varepsilon_n = n+\frac{1}{2}$. The Hermite functions form a complete basis for L^2 and they are thus the only eigenfunctions of the type in equation (B.50). The energies of the system are thus $E_n = \hbar\omega\left(n+\frac{1}{2}\right)$

C Perturbation theory

The perturbation theory is the study of how small disturbances effect a system [15]. The eigenstates ψ_n^0 and energy eigenvalues E_n^0 of a Hamiltonian $\hat{H}_0$ are known. Now a small disturbance $\hat{H}_1$ is added to the Hamiltonian and a new set of energy eigenvalues and eigenstates arises

$$(\hat{H}_0 + \hat{H}_1)\psi_n = E_n\psi_n. \tag{C.1}$$

These can approximated by successive series of corrections

$$\psi_n = \psi_n^{(0)} + \psi_n^{(1)} + \psi_n^{(2)} + \dots \tag{C.2}$$

$$E_n = E_n^{(0)} + E_n^{(1)} + E_n^{(2)} + \dots \tag{C.3}$$

The zero-order energy term is the energy for the undisturbed system

$$E_n^{(0)} = E_n^0 \tag{C.4}$$

Introduce the notation $|\psi_n\rangle = |n\rangle$. The first-order energy term for a harmonic oscillator is

$$E_n^{(1)} = \langle n^{(0)}|\hat{H}^{(1)}|n^{(0)}\rangle. \tag{C.5}$$

The second-order energy term is

$$E_n^{(2)} = \sum_{m\neq n} \frac{|\langle m^{(0)}|\hat{H}^{(1)}|n^{(0)}\rangle|^2}{E_n^{(0)} - E_m^{(0)}}. \tag{C.6}$$

The last only holds if the energy eigenvalues are non-degenerated which they are in the harmonic oscillator.

For the harmonic oscillator the first-order change in energy is given by

$$E_n^{(1)} = \langle n^{(0)}|\lambda E_c\xi^4|n^{(0)}\rangle = \int_{-\infty}^{\infty} g_n(\xi)^2\lambda E_c\xi^4 d\xi. \tag{C.7}$$

For the ground state ($n = 0$) this is

$$E_0^{(1)} = \lambda E_c \int_{-\infty}^{\infty} \frac{1}{\sqrt{\pi}} e^{-\xi^2}\xi^4 d\xi = \frac{3\lambda E_c}{4}. \tag{C.8}$$

The second order change in energy for the harmonic oscillator is given by

$$\begin{aligned} E_n^{(2)} &= \sum_{m\neq n} \frac{|\langle m^{(0)}|\lambda E_c\xi^4|n^{(0)}\rangle|^2}{E_n^{(0)} - E_m^{(0)}} \\ &= \sum_{m\neq n} \frac{|\int_{-\infty}^{\infty} g_n(\xi)g_m(\xi)\lambda E_c\xi^4 d\xi|^2}{E_n^{(0)} - E_m^{(0)}} \end{aligned} \tag{C.9}$$

An infinite set of integrals is to be solved:

$$I_m = \int_{-\infty}^{\infty} g_0(\xi) g_m(\xi) \xi^4 d\xi \tag{C.10}$$

The L^2 function $g_0(\xi)\xi^4$ can be expressed as a linear combination of the Hermite functions because they form a complete set:

$$\begin{aligned} g_0(\xi)\xi^4 &= \pi^{-1/4} \exp\left(-\frac{\xi^2}{2}\right) \xi^4 \\ &= \pi^{-1/4} \exp\left(-\frac{\xi^2}{2}\right) \\ &\quad \cdot \left(16^{-1}(16\xi^4 - 48\xi^2 + 12) + \frac{3}{4}(4\xi^2 - 2) + \frac{3}{4}\right) \\ &= \frac{\sqrt{6}}{2} g_4(\xi) + \frac{3}{\sqrt{2}} g_2(\xi) + \frac{3}{4} g_0(\xi) \end{aligned} \tag{C.11}$$

The Hermite functions are orthonormal so the integral I_m only give contributions when $m = 2, 4$ and the sum (C.9) thus becomes:

$$E_n^{(2)} = \left(\frac{9}{2(1/2 - 5/2)} + \frac{6}{4(9/2 - 1/2)}\right) \lambda^2 E_c = -\frac{21}{8} \lambda^2 E_c \tag{C.12}$$

The ground state energy for the undisturbed harmonic oscillator is

$$E_0^{(0)} = \langle \psi_0 | H^{(0)} | \psi_0 \rangle = \frac{E_c}{2\sqrt{\pi}} \int_{-\infty}^{\infty} e^{-\frac{\xi^2}{2}} \left(-\frac{\partial^2 e^{-\frac{\xi^2}{2}}}{\partial \xi^2}\right) d\xi = \frac{E_c}{2} \tag{C.13}$$

The ground state energy for the disturbed harmonic oscillator to the second order is

$$E_0 = E_0^{(0)} + E_0^{(1)} + E_0^{(2)} = E_c \left(\frac{1}{2} + \frac{3}{4}\lambda - \frac{21}{8}\lambda^2\right) \tag{C.14}$$

D Python

During this project, Python 2.6 was used as host API language. Python is a programming language that is very easy to learn if you are accustomed to C-like languages. Python provides both power and versatility, and can easily simulate many other environments and languages, thanks to its wide variety of modules.

D.1 Background

Python was developed in the late 1980s by Guido van Rossum. It is an interpreted, general-purpose, high-level programming language[28]. That said, it still runs like a compiled intermediate level language like C++ in many aspects. This is especially true in Python 3.0 and later. Python supports many programming paradigms like object-orientation, declarative functional programming, imperative procedural programming, and with certain modules parallel programming. The main idea behind Python is that it uses a clear and readable syntax, as well as being compatible and versatile. For example, Python can be implemented within API's like a scripting interface for short ease-of-use executions, or used as a foundation and framework for larger applications. Python is often compared with Perl, TCL, Ruby, Java and Scheme in how it runs. Like Java, it uses a strong exception-based error handling.

D.2 Modules

Another great aspect of Python is the modularity. First of all, it provides extensive standard libraries. Add to this all the third-party modules, and it will handle a very wide variety of tasks. It can easily execute extensions and modules written in:

- **C and C++** using Cython
- **Java** using Jython
- **.NET languages** using IronPython

Python can also simulate the behaviour of many popular languages, like for example MATLAB. MATLAB is probably most famous for its fast calculations of vectors/martrices and the user-friendly plotting tools. The modules Numpy[55], Matplotlib[56] and pylab[57] implement these functionalities in Python in a fashion very similair to MATLAB. Another handy module is SciPy[58], which is a toolbox for mathematics, science and engineering. In order to implement OpenCL in Python, a module called PyOpenCL is used. The modules mainly used in this project were *NumPy*, *PyOpenCL*, *PyLab* and

MatPlotLib. Other modules that might come in handy depending on the environment are Base[42] (Windows, unofficial), Boost.Python[59], PyMC[60], PyGTK[61] and SetupTools[62].

D.3 Why Python 2.6

As of this date, Python 3.2 was released quite recently. Python 3.x[28] is a new version of Python, incompatible with the 2.x releases in many aspects. The fundamental data types and their handling have dramatically changed. Most notably, Python 3.x supposedly[29] brings the run-time performance closer to C in some relevant aspects. This was overlooked when choosing which version to use, since versions newer than 2.6 currently lack much of the 3d-party support required in this project. This is especially true for modules running in 64-bit mode, which is prioritized. Although native support is missing, it is possible with some effort to implement the third party support by hand. This was regarded as way to cumbersome, so eventually 2.6 was chosen.

E Source code

E.1 C-code

Listing 4: C-code for calculating the ground state energy of a system of three particles with harmonic potentials.

```
#include <stdio.h>
#include <math.h>
#define N 400
#define nbrOfParticles 2
#define pathSize 200 //N*nbrOfParticles
#define epsilon 0.1f
#define epsilon_inv 10.0f
#define alpha 1.0f

void xorshift(unsigned int *x, unsigned int *y, unsigned int *z, unsigned int *w)
{
    //This performs the necessary XORs to obtain a new random unsigned integer w.
    unsigned int t;
    t = *x ^ (*x << 11);
    *x = *y; *y = *z; *z = *w;
    *w = (*w ^ (*w >> 19) ^ (t ^ (t >> 8)));
}

float randFloat(unsigned int *x, unsigned int *y, unsigned int *z, unsigned int *
    w)
{
    xorshift(x, y, z, w);
    //This returns a random floating point number by dividing w with UINT32_MAX (
        hardcoded).
    return (*w)*2.328306437080797e-10;
}

float operator(float *x)
{
    //This returns the value of the operator for a specified point in time (index
         in [0, N-1]).
    return 3.0f*(x[0]*x[0] + x[N]*x[N]);
}

float potential(float *x)
{
    //This returns the value of the potential for a specified point in time (
        index in [0, N-1]).
    return 1.5f*(x[0]*x[0] + x[N]*x[N]);
}

float kineticDiff(float leftX, float midX, float rightX, float newMidX)
{
    //This returns the change in kinetic energy when changing a lattice point
        from midX to newMidX.
    float leftChangeOld = midX - leftX;
    float rightChangeOld = midX - rightX;
    float leftChangeNew = newMidX - leftX;
    float rightChangeNew = newMidX - rightX;
    return 0.5f*(leftChangeNew*leftChangeNew + rightChangeNew*rightChangeNew -
        leftChangeOld*leftChangeOld - rightChangeOld*rightChangeOld)*epsilon_inv*
        epsilon_inv;
}
```

```
47
48 //This is the code that performs nbrOfLoopins Metropolis steps and is called from
       main().
49 void metropolis(float *path, float *opMean, int *accepts, unsigned int *seeds,
       unsigned int *nbrOfLoopings)
50 {
51     int modPathPoint = 0;
52     int modPoint;
53     int rightIndex;
54     int leftIndex;
55     float diffE;
56     float modx;
57     float oldX;
58     unsigned int loopLimit = *nbrOfLoopings;
59     unsigned int threeLoopRuns = loopLimit / pathSize;
60
61     float partialOp = 0;
62     float opMeanBuffer = 0;
63
64     //This sets the seeds for the Xorshift PRNG.
65     unsigned int x = seeds[0];
66     unsigned int y = seeds[1];
67     unsigned int z = seeds[2];
68     unsigned int w = seeds[3];
69
70     for (int i = 0; i < N; i++)
71         partialOp += operator(path + i);
72
73     opMean[0] = 0;
74     accepts[0] = 0;
75     /*The following three loops are responsible for creating Metropolis samples.
           This is done by looping trough all the lattice points for all the
           particles.
76     The third loop loops through the lattice points for a given particle.
77     The second loop loops through the different particles.
78     The first loop runs the two inner loops many times to create more samples.*/
79     for (int i = 0; i < threeLoopRuns; i++)
80     {
81         for (int particle = 0; particle < nbrOfParticles; particle++)
82         {
83             for (modPoint = 0; modPoint < N; modPoint++)
84             {
85                 modPathPoint = modPoint + N*particle;
86
87                 //Arrange with the periodic boundary conditions.
88                 rightIndex = modPathPoint + 1;
89                 leftIndex = modPathPoint - 1;
90                 if (modPoint == 0)
91                 {
92                     leftIndex = particle*N + N - 1;
93                 } else if (modPoint == N - 1)
94                 {
95                     rightIndex = particle*N;
96                 }
97
98                 oldX = path[modPathPoint];
99                 modx = oldX + alpha*(2.0f*randFloat(&x, &y, &z, &w) - 1.0f);
100
101                //Calculate the difference in energy (action) for the new path
                       compared to the old, stored in diffE.
102                diffE = kineticDiff(path[leftIndex], path[modPathPoint], path[
                       rightIndex], modx);
```

```
103                 diffE -= potential(path + modPoint);
104                 path[modPathPoint] = modx;
105                 diffE += potential(path + modPoint);
106                 path[modPathPoint] = oldX;
107 
108                 //Determine whether or not to accept the change in the path.
109                 if (exp(-epsilon*diffE) > randFloat(&x, &y, &z, &w)) {
110                     partialOp -=  operator(path+modPoint);
111                     path[modPathPoint] = modx;
112                     partialOp +=  operator(path+modPoint);
113                     accepts[0]++;
114                 }
115 
116                 opMeanBuffer += partialOp;
117             }
118         }
119         opMean[0] += opMeanBuffer;
120         opMeanBuffer = 0;
121     }
122 
123     opMean[0] += opMeanBuffer;
124     opMean[0] /= (float) (threeLoopRuns*nbrOfParticles*N*N);
125 
126     //Store the current state of the Xorshift PRNG for use in the next kernel run
        .
127     seeds[0] = x;
128     seeds[1] = y;
129     seeds[2] = z;
130     seeds[3] = w;
131 }
132 
133 int main()
134 {
135     //Initialize variables
136     float path[pathSize];
137     float opMean;
138     int accepts;
139     unsigned int seeds[4] = {123456789, 362436069, 521288629, 88675123};
140     unsigned int nbrOfLoopings;
141     float sum = 0;
142     float squareSum = 0;
143     int runs = 10;
144     for (int i = 0; i < pathSize; i++)
145         path[i] = 0;
146     nbrOfLoopings = 10000000;
147     //Run kernel once for burn in
148     printf("Doing burn in...\n");
149     metropolis(path, &opMean, &accepts, seeds, &nbrOfLoopings);
150     //Run metropolis() runs times to produce runs values for the energy, each
          call for metropolis() starts where the last one ended (path and seeds are
           saved).
151     for (int i = 1; i <= runs; i++)
152     {
153         printf("Run %2d:\n", i);
154         metropolis(path, &opMean, &accepts, seeds, &nbrOfLoopings);
155         printf("Mean of operator: %f\n", opMean);
156         printf("Acceptance rate: %.2f%%\n\n", 100.0f*((float) accepts)/((float)
              ((nbrOfLoopings/pathSize) * pathSize)));
157         sum += opMean;
158         squareSum += opMean*opMean;
159     }
160     //Print results from runs calls to metropolis().
```

```
161     float mean = sum/((float) runs);
162     float stdDev = sqrt(squareSum/((float) runs) - mean*mean);
163     float stdErr = stdDev/sqrt(runs);
164     printf("Mean: %f\n", mean);
165     printf("Standard error: %f\n", stdErr);
166     printf("65%% CI: [%f, %f]\n", mean - stdErr, mean + stdErr);
167     printf("95%% CI: [%f, %f]\n", mean - 2*stdErr, mean + 2*stdErr);
168     return 0;
169 }
```

E.2 PyOpenCL code

E.2.1 OpenCL Kernel Code

Listing 5: The PyOpenCL kernel code.

```
1  //#############################################################################
2  //kernel.c
3  //-----------------------------------------------------------------------------
4  //Imported by: ../host.py
5  //-----------------------------------------------------------------------------
6  //Description: This is the OpenCL kernel and is the code that runs on the GPU.
7  //             This contains the Monte Carlo calculations that are to be per-
8  //             performed and some helper methods for generating random numbers
9  //             numbers with the Xorshift PRNG.
10 //-----------------------------------------------------------------------------
11 //Authors:     Feynman Bachelors Group, Chalmers University of Technology
12 //Date:        2011-05-17
13 //Licensing:   GPL
14 //#############################################################################
15
16
17 //#############################################################################
18 //#                                  XORSHIFT                                 #
19 //#############################################################################
20 //Description: This performs the necessary XORs to obtain a new random
21 //             unsigned integer w.
22 inline void xorshift(unsigned int *x, unsigned int *y, unsigned int *z,
23                      unsigned int *w)
24 {
25
26     unsigned int t;
27     t = *x ^ (*x << 11);
28     *x = *y; *y = *z; *z = *w;
29     *w = (*w ^ (*w >> 19) ^ (t ^ (t >> 8)));
30 }
31
32
33 //#############################################################################
34 //#                                 randFloat                                 #
35 //#############################################################################
36 //Description: This returns a random floating point number by dividing w with
37 //             UINT32_MAX (hardcoded).
38 inline float randFloat(unsigned int *x, unsigned int *y, unsigned int *z,
39                        unsigned int *w)
40 {
41     xorshift(x, y, z, w);
42
```

```
43      return (*w)*2.328306437080797e-10;
44  }
45
46
47  //##############################################################################
48  //#                                randPathPoint                               #
49  //##############################################################################
50  //Description: This returns a uniformly distributed random unsigned integer
51  //             in the interval [0, pathSize - 1].
52  inline unsigned int randPathPoint(unsigned int *x, unsigned int *y,
53                                    unsigned int *z, unsigned int *w)
54  {
55      xorshift(x, y, z, w);
56      //
57      return (*w)*2.328306437080797e-10*%(pathSize)d;
58  }
59
60
61  //##############################################################################
62  //#                                  userCode                                  #
63  //##############################################################################
64  //Description: This is a placeholder for code that the user can write to help
65  //             specify operator and potential.
66  %(userCode)s;
67
68
69  //##############################################################################
70  //#                                 OPERATOR                                   #
71  //##############################################################################
72  //Description: This returns the value of the operator for a specified point
73  //             in time (index in [0, N-1]).
74  inline float operator(float *x)
75  {
76      return %(operator)s;
77  }
78
79
80  //##############################################################################
81  //#                                 POTENTIAL                                  #
82  //##############################################################################
83  //Description: This returns the value of the potential for a specified
84  //             point in time (index in [0, N-1]).
85  inline float potential(float *x)
86  {
87      return %(potential)s;
88  }
89
90
91  //##############################################################################
92  //#                                kineticDiff                                 #
93  //##############################################################################
94  //Description: This returns the change in kinetic energy when changing a
95  //             lattice point from midX to newMidX.
96  inline float kineticDiff(float leftX, float midX, float rightX, float newMidX)
97  {
98      //
99      float leftChangeOld = midX - leftX;
100     float rightChangeOld = midX - rightX;
101     float leftChangeNew = newMidX - leftX;
102     float rightChangeNew = newMidX - rightX;
103     return 0.5f*(leftChangeNew*leftChangeNew + rightChangeNew*rightChangeNew -
104                  leftChangeOld*leftChangeOld - rightChangeOld*rightChangeOld)*
```

```
105             %(epsilon_inv)f*%(epsilon_inv)f;
106 }
107
108
109 //##############################################################################
110 //#                                  KERNEL CODE                               #
111 //##############################################################################
112 //Description: The following code is the OpenCL kernel. This is the code that is
113 //             called from Python and takes all the input and delivers output.
114 __kernel void metropolis(__global float *paths,
115                          __global float *opMeans,
116                          __global int *accepts,
117                          __global unsigned int *seeds,
118                          __global unsigned int *nbrOfLoopings,
119                          __global unsigned int *binCounts)
120 {
121     int gid = get_global_id(0);
122
123     int pathsVectNbr = gid*%(pathSize)d;
124     int seedsVectNbr = gid*4;
125     int modPathPoint = 0;
126     int modPoint;
127     int rightIndex;
128     int leftIndex;
129     int binIndex;
130     float diffE;
131     float modx;
132     float oldX;
133     unsigned int loopLimit = *nbrOfLoopings;
134     unsigned int threeLoopRuns = loopLimit / %(pathSize)d;
135
136     float path[%(pathSize)d];
137     float partialOp = 0;
138     float opMeanBuffer = 0;
139
140     //This sets the seeds for the Xorshift PRNG.
141     unsigned int x = seeds[seedsVectNbr + 0];
142     unsigned int y = seeds[seedsVectNbr + 1];
143     unsigned int z = seeds[seedsVectNbr + 2];
144     unsigned int w = seeds[seedsVectNbr + 3];
145
146     //This imports the path corresponding to this thread from the collection
147     //of all paths stored in the field paths.
148     for (int i = 0; i < %(pathSize)d; i++)
149          path[i] = paths[i + pathsVectNbr];
150
151     for (int i = 0; i < %(N)d; i++)
152         partialOp += operator(path + i);
153
154     opMeans[gid] = 0;
155     accepts[gid] = 0;
156
157
158     /*
159     The following three loops are responsible for creating Metropolis samples.
160     This is done by looping trough all the lattice points for all the particles.
161
162     The third loop loops through the lattice points for a given particle.
163     The second loop loops through the different particles.
164     The first loop runs the two inner loops many times to create more samples.
165     */
166     for (int i = 0; i < threeLoopRuns; i++)
```

```
167     {
168         for (int particle = 0; particle < %(nbrOfParticles)d; particle++)
169         {
170             for (modPoint = 0; modPoint < %(N)d; modPoint++)
171             {
172                 modPathPoint = modPoint + %(N)d*particle;
173 
174                 //Arrange with the periodic boundary conditions.
175                 rightIndex = modPathPoint + 1;
176                 leftIndex = modPathPoint - 1;
177                 if (modPoint == 0)
178                 {
179                     leftIndex = particle*%(N)d + %(N)d - 1;
180                 } else if (modPoint == %(N)d - 1)
181                 {
182                     rightIndex = particle*%(N)d;
183                 }
184 
185                 oldX = path[modPathPoint];
186                 modx = oldX + %(alpha)f*(2.0f*randFloat(&x, &y, &z, &w) - 1.0f);
187 
188                 //Calculate the difference in energy (action) for the new path
189                 //compared to the old, stored in diffE.
190                 diffE = kineticDiff(path[leftIndex], path[modPathPoint],
191                                     path[rightIndex], modx);
192                 diffE -= potential(path + modPoint);
193                 path[modPathPoint] = modx;
194                 diffE += potential(path + modPoint);
195                 path[modPathPoint] = oldX;
196 
197                 //Determine whether or not to accept the change in the path.
198                 if (native_exp(-%(epsilon)f*diffE) > randFloat(&x,&y,&z,&w)) {
199                     partialOp -=  operator(path+modPoint);
200                     path[modPathPoint] = modx;
201                     partialOp +=  operator(path+modPoint);
202                     accepts[gid]++;
203                 }
204 
205                 opMeanBuffer += partialOp;
206 
207                 int ok = 1;
208                 int inc = 1;
209                 binIndex = 0;
210                 for(int p = 0; p < %(nbrOfParticles)d; p++)
211                 {
212                     if(path[modPoint + %(N)d*p] < %(xmin)f ||
213                        path[modPoint + %(N)d*p] >= %(xmax)f)
214                     {
215                         ok=0;
216                         break;
217                     }
218                     binIndex += (int) ((path[modPoint + %(N)d*p] -
219                                         (%(xmin)f))*%(invBinSize)f)*inc;
220                     inc *= %(binsPerPart)d;
221                 }
222                 if(ok)
223                 {
224                     binCounts[gid*%(nbrOfBins)d+binIndex]++;
225                 }
226             }
227         }
228         opMeans[gid] += opMeanBuffer;
```

```
229             opMeanBuffer = 0;
230         }
231
232         opMeans[gid] += opMeanBuffer;
233         opMeans[gid] /= (float) (threeLoopRuns*%(nbrOfParticles)d*%(N)d*%(N)d);
234
235         //Store the last path back in the paths variable for use in the
236         //next kernel run.
237         for (int i = 0; i < %(pathSize)d; i++)
238             paths[i + pathsVectNbr] = path[i];
239
240         //Store the current state of the Xorshift PRNG for use in the next
241         //kernel run.
242         seeds[seedsVectNbr + 0] = x;
243         seeds[seedsVectNbr + 1] = y;
244         seeds[seedsVectNbr + 2] = z;
245         seeds[seedsVectNbr + 3] = w;
246     }
```

E.2.2 PyOpenCL Host Code

Listing 6: The PyOpenCL host code.

```
1  ##############################################################################
2  # host.py
3  # ----------------------------------------------------------------------------
4  # Imported by: ../application.py
5  #
6  # Imports:     GPUsrc/kernel.c
7  # ----------------------------------------------------------------------------
8  # Description: Calling this method will cause the GPU to calculate the mean of
9  #              a given operator on a quantum mechanical system. Also, a
10 #              numerical representation of the probability density is calculated
11 #              and returned.
12 # ----------------------------------------------------------------------------
13 # Arguments:   userCode: C-code to help the user formulate the operator (string)
14 #              potential: Potential (string)
15 #              operator: Operator (string)
16 #              N: Number of lattice points (int)
17 #              beta: Time interval (float)
18 #              alpha: Metropolis step length (float)
19 #              nbrOfThreads: Number of threads that run the kernel (int)
20 #              nbrOfLoopings: Number of metropolis steps in each thread (int)
21 #              runs: Number of times to run the kernel on each thread (int)
22 #              burnIn: Number of Metropolis steps that are taken without saving
23 #                      results (int)
24 #              burnInRuns: Number of times to run the burn in process (int)
25 #              xmin: Lower boundary for each degree of freedom when calculating
26 #                    the probability density with the time saving trick (float)
27 #              xmax: Like xmin but an upper boundary (float)
28 #              binsPerPart: Number of bins into which the interval
29 #                           [xmin, xmax] is divided (int)
30 # ----------------------------------------------------------------------------
31 # Authors:     Feynman Bachelors Group, Chalmers University of Technology
32 # Date:        2011-05-17
33 # Licensing:   GPL
34 ##############################################################################
35
36
```

```
37 ##############################################################################
38 #                                  LIBRARIES                                 #
39 ##############################################################################
40 import numpy as np
41 import pyopencl as cl
42 import pyopencl.array
43 import pyopencl.clrandom
44 import pyopencl.clmath
45 from time import time
46 import sys
47
48
49 ##############################################################################
50 #                           getOperatorMean Function                         #
51 ##############################################################################
52 # Description: This method takes physically relevant parameters to calculate
53 #              the mean of a specified quantum mechanical operator, as well as
54 #              a numerical representation of the probability density.
55 #
56 def getOperatorMean(userCode, potential, N, beta, alpha, operator, nbrOfThreads,
57                     nbrOfLoopings, runs, burnIn, burnInRuns, nbrOfParticles,
58                     xmin, xmax, binsPerPart):
59
60     #Fetch results from load kernel into a list
61     kernelEnvironment = _loadKernel(userCode, potential, operator, N, beta,
62                                     alpha, nbrOfThreads, nbrOfParticles,
63                                     xmin, xmax, binsPerPart)
64
65     #Perform burn in ('burnIn' Metropolis steps, 'burnInRuns' times)
66     for i in range(0, burnInRuns):
67         [burnEnergy, burnSTD, burnAcceptance, burnTime,
68          dummy] =_runKernel(kernelEnvironment, burnIn, xmax, xmin, binsPerPart)
69
70     #Resets the bin counts from the burn in executions
71     kernelEnvironment[8].fill(0)
72
73     #Allocate space for the results from each set of 'nbrOfThreads' kernel runs.
74     energy = np.zeros(runs)
75     energySTD = np.zeros(runs)
76     acceptanceRate = np.zeros(runs)
77     runTime = np.zeros(runs)
78
79     #Create the multidimensional array for storing the resulting number of bin
80     #counts (dimension of the array is nbrOfParticles) after 'runs' executions.
81     binTouple = (binsPerPart,)
82     for i in range(1, nbrOfParticles):
83        binTouple += (binsPerPart,)
84     binCounts = np.zeros(binTouple).astype(np.float32)
85
86     #Execute 'nbrOfThreads' kernels, 'runs' times and store the results.
87     for i in range(0, runs):
88         print "Run " + str((i+1)) + " of " + str(runs)
89         [energy[i], energySTD[i], acceptanceRate[i], runTime[i],
90          binCounts] = _runKernel(kernelEnvironment, nbrOfLoopings, xmin,
91                                  xmax, binsPerPart)
92
93     pathSize = N*nbrOfParticles
94
95     #'nbrOfLoopings' is truncated if not a factor of 'pathSize'
96     effNbrOfLoopings = float((nbrOfLoopings//pathSize)*pathSize)
97
98     #Create a normalized histogram from the bin counts.
```

```
99      totBinCount = nbrOfThreads*nbrOfLoopings*runs
100     width = float(xmax-xmin)/float(binsPerPart)
101     binCountsNormed = binCounts.astype(np.float32)/(width*effNbrOfLoopings)
102
103     #The calculated operator mean, standard error, mean acceptance rate, total
104     #run time and numerical representation of probability density is returned.
105     return [energy.mean(), energySTD.mean()/np.sqrt(runs),
106             acceptanceRate.mean(), runTime.sum(), binCountsNormed]
107
108
109 ###############################################################################
110 #                              _loadKernel Function                           #
111 ###############################################################################
112 # Description: Set up the kernel and prepare the GPU for the execution of the
113 #              desired number of threads.
114 #
115 def _loadKernel(userCode, potential, operator, N, beta, alpha, nbrOfThreads,
116                 nbrOfParticles, xmin, xmax, binsPerPart):
117
118     for i in range(1, nbrOfParticles + 1):
119         potential = potential.replace("x"+str(i), "x["+str(N*(i-1))+"]")
120         operator = operator.replace("x"+str(i), "x["+str(N*(i-1))+"]")
121
122     #Import the kernel code and fill the placeholders with relevant data.
123     replacements = {'N': N, 'epsilon': beta/float(N),
124                     'epsilon_inv': float(N)/beta,
125                     'pathSize': (nbrOfParticles*N), 'alpha': alpha,
126                     'userCode':userCode, 'potential':potential,
127                     'operator':operator, 'nbrOfParticles': nbrOfParticles,
128                     'xmin':xmin, 'xmax':xmax, 'binsPerPart':binsPerPart,
129                     'nbrOfBins':binsPerPart**nbrOfParticles,
130                     'invBinSize':float(binsPerPart)/float(xmax-xmin)}
131
132     #Import kernel code and paste 'replacements' into it
133     kernelCode_r  = open(sys.path[0]+'/src/GPUsrc/kernel.c', 'r').read()
134     kernelCode  = kernelCode_r%replacements
135
136     #Create the important OpenCL context and command queue
137     ctx = cl.create_some_context()
138     queue_properties = cl.command_queue_properties.PROFILING_ENABLE
139     queue = cl.CommandQueue(ctx, properties=queue_properties)
140
141     #Build the program and identify metropolis as the kernel
142     prg = cl.Program(ctx, kernelCode).build()
143     kernel = prg.metropolis
144     #pyopencl.array objects are created for storing the calculated operator
145     #means from each thread
146     operatorValues = cl.array.zeros(ctx, queue, (nbrOfThreads, ), np.float32)
147     #Initial paths are created (the initial path vector is filled with zeros,
148     #meaning no movement of the particles)
149     paths = cl.array.zeros(ctx, queue, (nbrOfThreads, N*nbrOfParticles),
150                            np.float32)
151     #Buffer for storing number of accepted values and seeds for the xorshfitPRNG
152     accepts = cl.array.zeros(ctx, queue, (nbrOfThreads, ), np.int32)
153     seeds = cl.array.to_device(ctx, queue, np.random.randint(0, high=10**9,
154                                size=(nbrOfThreads, 4))).astype(np.uint32)
155
156     #A buffer for the multidimensional array for storing the resulting number
157     #of bin counts
158     binTouple = (binsPerPart,)
159     for i in range(1, nbrOfParticles):
160         binTouple += (binsPerPart,)
```

```
161     binCounts = cl.array.zeros(ctx, queue, (nbrOfThreads,)+binTouple, np.uint32)
162
163     #Return the kernel object, context, command queue and pyopencl.array objects
164     return [kernel, ctx, queue, paths, seeds, operatorValues, accepts,
165             nbrOfThreads, binCounts]
166
167
168 ##############################################################################
169 #                              _runKernel Function                           #
170 ##############################################################################
171 # Description: This method takes care of running the kernel with given input
172 #              and return the results from this kernel run.
173 #
174 def _runKernel(kernelEnvironment, nbrOfLoopings, xmin, xmax, binsPerPart):
175     #Pick out kernel, context, command queue and pyopencl.array objects from
176     #passed list kernelEnvironment
177     [kernel, ctx, queue, paths, seeds, operatorValues, accepts,
178      nbrOfThreads, binCounts] = kernelEnvironment
179     #Create pyopencl.array object with one element to pass the nbrOfLoopings
180     #argument to the kernel
181     runDim = np.uint([nbrOfLoopings])
182     nbrOfLoopingsOCL=cl.array.to_device(kernelEnvironment[1],
183                                         kernelEnvironment[2],
184                                         runDim.astype(np.uint32))
185
186     #Run kernel. The buffers for all pyopencl.array objects are passed
187     #by the .data parameter.
188     kernel_obj = kernel(queue, (nbrOfThreads, ), None, paths.data,
189                         operatorValues.data, accepts.data, seeds.data,
190                         nbrOfLoopingsOCL.data, binCounts.data)
191     #Wait until the threads have finished and then calculate total run time
192     kernel_obj.wait()
193     runTime = 1e-9*(kernel_obj.profile.end - kernel_obj.profile.start)
194
195     #Fetch the operatorValues and acceptanceRate pyopencl.array objects from
196     #the graphics card to the RAM.
197     operatorValuesPython = operatorValues.get()
198     acceptanceRate = accepts.get()/float(nbrOfLoopings)
199
200     #Sum the bin counts from all individual threads.
201     binCountsResult = np.sum(binCounts.get(),axis=0).astype(np.float32)
202
203     return [operatorValuesPython.mean(),
204             operatorValuesPython.std()/np.sqrt(len(operatorValuesPython)),
205             acceptanceRate.mean(),
206             runTime,
207             binCountsResult]
```

E.2.3 Example of Application Code

Listing 7: An example of application code.

```
1  ##############################################################################
2  # application.py
3  # ----------------------------------------------------------------------------
4  # Imports:      src/host.py
5  #               src/GPUsrc/kernel.c   (indirectly through above file)
6  # ----------------------------------------------------------------------------
7  # Description: Calculates the ground state mean value of a user-defined operator
```

```
8  #               and plots if there are 1 or 2 degrees of freedom. This particular
9  #               application contains coupled Harmonic Oscillators as well as
10 #               Jacobi-transformed systems for up to three particles.
11 # ----------------------------------------------------------------------------
12 # Usage:         Uncomment or add the potential/operator of interest to variables
13 #                'potential' and 'operator' respectively. Modify 'userCode' if
14 #                necessary to contain C-functions to calculate these.
15 # ----------------------------------------------------------------------------
16 # Results:       * Calculates mean of a user-defined operator
17 #                * Plots probability density (1 or 2 dimensions) with pylab
18 #                * Saves figure to .png-file
19 #                * Saves plot data into .dat-file
20 # ----------------------------------------------------------------------------
21 # Note:          Plancks constant and k are conveniently set to 1.
22 # ----------------------------------------------------------------------------
23 # Authors:       Feynman Bachelors Group, Chalmers University of Technology
24 # Date:          2011-05-17
25 # Licensing:     GPL
26 ##############################################################################
27
28
29
30 ##############################################################################
31 #                                   LIBRARIES                                #
32 ##############################################################################
33 import numpy as np
34 import pylab as pl
35 from datetime import datetime
36 import sys
37
38 # Set system path for importation of host.py and kernel.c
39 sys.path.append(sys.path[0]+'/src/')
40 import host
41
42 ##############################################################################
43 #                          VARIABLES AND KERNEL ARGUMENTS                    #
44 ##############################################################################
45 nbrOfThreads = 4*448              # Amount of threads to use on the GPU
46
47 N = 1024                          # Number of lattice points per particle
48 nbrOfParticles = 2                # Amount of particles/degrees of freedom
49 pathSize = nbrOfParticles*N       # Total amount of time steps for all particles
50
51 burnIn = 1e5                      # Number of path changes before saving results
52 burnInRuns = 30                   # Amount of re-runs of burnin
53 nbrOfLoopings = 1e5               # Number of path changes, will be truncated if
54                                   #    not a factor of pathSize
55 runs = 60                         # Amount of re-runs for each kernel
56
57 # Variables affecting acceptance rate
58 beta = 300                        # Size of time interval
59 alpha = 1.0                       # Metropolis step length
60
61 # Plot variables
62 xmin = -1.5                       # Plot min of interval for each particle
63 xmax = 1.5                        # Plot max of interval for each particle
64 binsPerPart = 30                  # Number of bins/points used for plotting
65 binSize = (xmax-xmin)/binsPerPart # Size of each bin
66
67 ##############################################################################
68 #                             OPERATOR and POTENTIAL                         #
69 ##############################################################################
```

```
# userCode: Through this variable, it will be possible to define C-functions
#           used for calculating the operator and potential. The variables
#           'potential' and 'operator' might be used to define how these
#           functions are going to be called in order to get the full
#           operator/potential.
#
# Note: For plotting only, set operator to 0.0f for faster run-time
#
userCode = """
float operatorFunction(float V, float sigma, float x, float y)
{
    const float pi = 3.1415926535f;
    const float sqrt3 = 1.73205081f;
    float A = 4.0f*x*x;

    float B = (x-sqrt3*y)*(x-sqrt3*y);
    float C = (x+sqrt3*y)*(x+sqrt3*y);
    float D = 2.0f*sigma*sigma;
    float D_inv = 1.0f/D;

    return -V*D_inv/(native_sqrt(pi)*sigma) * ((A-D)*native_exp(-A*D_inv) +
                                               (B-D)*native_exp(-B*D_inv) +
                                               (C-D)*native_exp(-C*D_inv));
}

float potentialFunction(float V, float sigma, float x, float y)
{
    const float pi = 3.1415926535f;
    const float sqrt3 = 1.73205081f;
    float A = 4.0f*x*x;

    float B = (x-sqrt3*y)*(x-sqrt3*y);
    float C = (x+sqrt3*y)*(x+sqrt3*y);
    float D_inv = 0.5f/(sigma*sigma);

    return V/(native_sqrt(pi)*sigma) * (native_exp(-A*D_inv) +
                                        native_exp(-B*D_inv) +
                                        native_exp(-C*D_inv));
}
"""

#
# Examples of different potentials
#
##################################
#           V_ALPHA              #
##################################
potential = 'potentialFunction(12.0f, 0.2f, x1, x2) + potentialFunction(-12.0f,
    0.8f, x1, x2)'
operator = 'operatorFunction(12.0f, 0.2f, x1, x2) + operatorFunction(-12.0f, 0.8f
    , x1, x2)'
##################################
#            V_BETA              #
##################################
#potential = 'potentialFunction(-2.0f, 0.8f, x1, x2)'
#operator = 'operatorFunction(-2.0f, 0.8f, x1, x2)'
##################################
#              1D HO             #
##################################
#potential = '0.5f*x1*x1'
#operator = 'x1*x1'
##################################
```

```
130 # Coupled Harmonic Oscillators #
131 ###################################
132 #potential = '0.5f*((x1-x2)*(x1-x2)+(x2-x3)*(x2-x3)+(x3-x1)*(x3-x1))'
133 #operator = '((x1-x2)*(x1-x2)+(x2-x3)*(x2-x3)+(x3-x1)*(x3-x1))'
134
135 ##############################################################################
136 #                                HOST CODE EXECUTION                         #
137 ##############################################################################
138 [operator, standardError, acceptanceRate,runTime,
139  binCount] = host.getOperatorMean(userCode,potential,N,beta,alpha,operator,
140                                   nbrOfThreads,nbrOfLoopings,runs,burnIn,
141                                   burnInRuns,nbrOfParticles,
142                                   xmin,xmax,binsPerPart)
143
144 # Print results
145 print("Kernel Run Time: %0.4f s" % runTime)
146 print("Mean Operator: "+str(operator))
147 print("Operator Standard Error: "+str(standardError))
148 print("Acceptance Rate: %0.1f%%" % (acceptanceRate*100.0))
149
150
151 # Output-file parameters
152 t = datetime.now()
153 timestamp = t.strftime("%Y-%m-%d_%H-%M-%S")
154 filename = "Wavefunction" + "-" + timestamp
155 f_data = open(filename+".dat",'w')
156
157 ##############################################################################
158 #                        PLOT for 1 particle/degree of freedom               #
159 ##############################################################################
160 if nbrOfParticles == 1:
161     # x interval for plot, +0.5*binSize to have each value in the middle of bins
162     x_ival = np.linspace(xmin,xmax,binsPerPart)+0.5*binSize
163
164     # Calculate and plot analytical probability density
165     psi_a = np.exp(-(x_ival**2))/(np.sqrt(np.pi))
166     pl.plot(x_ival,psi_a,'g-',label='analytical')
167
168     # Plot calculated probability density with errorbars (standard error)
169     pl.errorbar(x_ival,binCount,ecolor='r',fmt='r.-',yerr=binSTD.mean(axis=0),
170                 label='Calculated')
171
172     # Plot information
173     pl.title('Probability density plot')
174     pl.xlabel('x')
175     pl.ylabel('|psi_0|^2')
176     pl.legend()
177
178     #Save plot data to file
179     f_data.write("#x" + "\t" + "|psi_0|^2" + "\n")
180     for i in range(0,binsPerPart):
181         f_data.write( str(x_ival[i])  + "\t" + str(binCount[i]) + "\n")
182     f_data.close()
183
184     # Save figure
185     pl.savefig(filename+".png")
186
187     # Show plot
188     pl.show()
189
190 ##############################################################################
191 #                       PLOT for 2 particles/degrees of freedom              #
```

```
192 ###############################################################################
193 elif nbrOfParticles == 2:
194     # x interval for plot, +0.5*binSize to have each value in the middle of bins
195     x_ival = np.linspace(xmin,xmax,binsPerPart)+0.5*binSize
196 
197     # Filled Contour Plot
198     pl.contourf(x_ival, x_ival, binCount,60, cmap = None)
199 
200     # Plot information
201     pl.xlabel('x1')
202     pl.ylabel('x2')
203     pl.title('Contour plot for 2 dimensions/degrees of freedom')
204     pl.figure()
205 
206     # Contour Plot
207     pl.contour(x_ival, x_ival, binCount,60, cmap = pl.cm.jet)
208 
209     #Save plot data to file
210     f_data.write("#x" + "\t" + "y" + "\t" + "|psi|^2" + "\n")
211     for x in range(0,binsPerPart):
212           f_data.write( "\n" )
213           for y in range(0,binsPerPart):
214                f_data.write( str(x_ival[x])  + "\t" + str(x_ival[y]) + "\t"
215                                              + str(binCount[x,y])    + "\n")
216     f_data.close()
217 
218     # Save figure
219     pl.savefig(filename+".png")
220 
221     # Show plot
222     pl.show()
```

E.3 Device Dump Properties

This script outputs some properties for all detectable devices.

Listing 8: Device Dump Property Application

```
1  # Filename: dump_properties.py
2  # Info: Used to fetch device info for all available devices
3  # Author: Example in PyOpenCL module
4  
5  import pyopencl as cl
6  
7  def print_info(obj, info_cls):
8      for info_name in sorted(dir(info_cls)):
9          if not info_name.startswith("_") and info_name != "to_string":
10             info = getattr(info_cls, info_name)
11             try:
12                 info_value = obj.get_info(info)
13             except:
14                 info_value = "<error>"
15 
16             print "%s: %s" % (info_name, info_value)
17 
18 for platform in cl.get_platforms():
19     print 75*"="
20     print platform
21     print 75*"="
```

```
22      print_info(platform, cl.platform_info)
23
24      for device in platform.get_devices():
25          print 75*"-"
26          print device
27          print 75*"-"
28          print_info(device, cl.device_info)
```

E.4 Device Benchmark Properties

This is a script that outputs device info and performs a simple benchmark between Python code run on the CPU, and an equivalent code run in OpenCL on all OpenCL-usable devices. The benchmark is a simple vector operation, written by Roger Pau Monn'e and included with PyOpenCL as an example.

Listing 9: Benchmarking Application

```
# Filename: benchmark-all.py
# Info: Perform a simple benchmark on all available devices
# Author: Roger Pau Monn'e, available in PyOpenCL module

import pyopencl as cl
import numpy
import numpy.linalg as la
import datetime
from time import time

a = numpy.random.rand(1000).astype(numpy.float32)
b = numpy.random.rand(1000).astype(numpy.float32)
c_result = numpy.empty_like(a)

# Speed in normal CPU usage
time1 = time()
for i in range(1000):
        for j in range(1000):
                c_result[i] = a[i] + b[i]
                c_result[i] = c_result[i] * (a[i] + b[i])
                c_result[i] = c_result[i] * (a[i] / 2.0)
time2 = time()
print "Execution time of test without OpenCL: ", time2 - time1, "s"


for platform in cl.get_platforms():
    for device in platform.get_devices():
        print "==========="
        print "Platform name:", platform.name
        print "Platform profile:", platform.profile
        print "Platform vendor:", platform.vendor
        print "Platform version:", platform.version
        print "---------------"
        print "Device name:", device.name
        print "Device type:", cl.device_type.to_string(device.type)
        print "Device memory: ", device.global_mem_size//1024//1024, 'MB'
        print "Device max clock speed:", device.max_clock_frequency, 'MHz'
        print "Device compute units:", device.max_compute_units

        # Simnple speed test
        ctx = cl.Context([device])
        queue = cl.CommandQueue(ctx,
                properties=cl.command_queue_properties.PROFILING_ENABLE)

        mf = cl.mem_flags
        a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
        b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
        dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes)

        prg = cl.Program(ctx, """
            __kernel void sum(__global const float *a,
```

```
52              __global const float *b, __global float *c)
53              {
54                      int loop;
55                      int gid = get_global_id(0);
56                      for(loop=0; loop<1000;loop++)
57                      {
58                              c[gid] = a[gid] + b[gid];
59                              c[gid] = c[gid] * (a[gid] + b[gid]);
60                              c[gid] = c[gid] * (a[gid] / 2.0);
61                      }
62              }
63              """).build()
64
65      exec_evt = prg.sum(queue, a.shape, None, a_buf, b_buf, dest_buf)
66      exec_evt.wait()
67      elapsed = 1e-9*(exec_evt.profile.end - exec_evt.profile.start)
68
69      print "Execution time of test: %g s" % elapsed
70
71      c = numpy.empty_like(a)
72      cl.enqueue_read_buffer(queue, dest_buf, c).wait()
73      error = 0
74      for i in range(1000):
75              if c[i] != c_result[i]:
76                      error = 1
77      if error:
78              print "Results doesn't match!!"
79      else:
80              print "Results OK"
```