Quantum Monte Carlo Methods for Studying Quantum Dots

Alexander Fleischer

Thesis submitted for the degree of
Master in Computational Physics

60 credits

Department of Physics

Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2018

Quantum Monte Carlo Methods for Studying Quantum Dots

Alexander Fleischer

Abstract

This thesis aims to give an understanding of quantum dots trapped in different potentials by computationally finding the ground state energies of the system. The potentials studied are the single and double harmonic oscillator wells and the finite square well.

To achieve this, we have implemented a robust Variational Monte Carlo framework that allows for different single-particle wavefunctions. This allowed us to study the use of a linear expansion of the single harmonic oscillator basisfunctions as an approximation to the single-particle wavefunctions of the other potential wells. We investigated how many of these basisfunctions needed to achieve sufficiently good single-particle wavefunctions.

The framework has several optimizations that drastically reduce the computational time of the Monte Carlo method. The framework has a rigorous testing regime that ensures the authenticity of the results.

To verify the results, we have compared them to the results of other Variatonal Monte Carlo computations, as well as Diffusion Monte Carlo, the Coupled Cluster method, the Similarity Renormalization Group method and the Full Configuration Interaction method. In addition, we have verified the results of the finite square well by comparing with a general series solution.

Contents

List of Figures

List of Tables

Acknowledgement

First off I want to thank Christian for being a great brother and super physicist, and for our discussions.

I wish to thank my thesis supervisor Morten Hjorth-Jensen for being tremendously good person, educator and scientist.

Lastly I want to thank Anette, Robert, Roar, Eigil and Khoa for being the best people,

1 Introduction

In this thesis we are looking at fermionic quantum dots confined to some potential. Experimentally, there are ways to create and tune quantum dot systems. We however, simulate these systems on a computer. Though physicists are able to create the systems, it can be challenging to find the right one to study. By simulating an array of different systems computationally, we can rule out those that doesn't make the cut and experimentally create the ones that seem promising.

In modern physics there is no way around computations. The better our computers get, the larger systems we are able to simulate. But when the computing power increases, so does the need for good programming. In this thesis we have created a Variational Monte Carlo framework[9], called a VMC solver, to find the ground state energies of electrons[19][8][1][6]. We have used both knowledge of computational science and physics to do this. The solver is written in C++ and we have taken advantage of the object-oriented nature of the programming language to create describe the physical objects needed. This ranges from single particles to wavefunctions, from the Hamiltonian to the variational method *steepest/gradient descent*.

We study the single harmonic oscillator, the double harmonic oscillator potential, as well as the finite square well. Before finding the ground state energies of particles in these potentials using the Variational Monte Carlo method, we solve the Schrödinger equation to find the separable single-particle wavefunctions of the systems. These wavefunctions describe a single quantum energy level. It turns out we can expand the single-particle wavefunctions with the single harmonic oscillator basisfunctions. This is one of the main subjects we investigate in this thesis; how we can use this expansion in the VMC calculations. We use both the analytical single-particle wavefunctions and the basisfunction expansions to find the ground state energies with VMC and benchmark the results against the results of other methods. To get an intuition for the physical system, we draw the one-body densities of the systems. This lets us see how the particles place themselves in the potential wells. Finally we look at some specific optimization schemes and how we implement them in the solver.

Structure

- The first chapter introduces all the theory needed to understand the systems we are studying and how we can build the computational framework of this thesis. We start by describing the quantum systems, Hamiltonians, potentials, wavefunctions and the Schrödinger equation. We continue by outlining the Variational Monte Carlo (VMC) method, derived from a diffusion approach. To improve the method, we introduce importance sampling. Trial wavefunctions are an important part of VMC and we explain how they are formed using a correlations, or Jastrow, part and a Slater determinant. Furthermore we show how to effieciently compute the Slater determinant and related matrices. To efficiently calculate the variational parameters, the steepest descent method is. Finally we show we can do statistical analysis with the blocking technique.
- In the next chapter we discuss some computational science and testing of the framework. We then look at the structure of the programs, mainly the VMC solver and the program that finds the single-particle wavefunction expansions.
- Chapter four contains the optimization techniques we use to reduce the computational time of the solver.
- In chapter five we show how we can verify the results of our computations and the implementation.
- The results chapter and the conclusion concludes the thesis. We show our results and benchmark them against the result of both VMC and other methods of finding the ground state energies.

Part I

Theory

2

Quantum Monte-Carlo Methods

2.1 Quantum Mechanical Background

Potentials obviously play an important role in physics, and most potentials behave harmonically around an equilibrium. This makes the *harmonic oscillator potential* a particullary useful model in quantum mechanics. It is also one of the few problems which can be solved exactly analytically. The form of the potential is[12]

$$V_c(\mathbf{r}) = \frac{1}{2} m\omega^2 r^2 \tag{2.1}$$

where ω is the *oscillator frequency*, which governs the strength of the potential. m is the mass of the particle (electron in this thesis). This is similar to the potential of a classical spring. By venturing away from the center, we see that the potential goes towards infinity.

The *Schrödinger equation (SE)*[12] for a particle trapped in a potential $V(\mathbf{r})$ with three-dimensional spatial coordinates $\mathbf{r} = (x, y, z)$ is described by

$$\left[-\frac{1}{2}\hat{\nabla}^2 + V(\mathbf{r})\right]\psi(\mathbf{r}) = E\psi(\mathbf{r}) \tag{2.2}$$

and gives us the total energy E as an eigenvalue with corresponding eigenstates ψ.

The expression inside the parenthesis of equation 2.2 is the *Hamiltonian* $\hat{\mathbf{H}}_0$ of the system. It corresponds to the total energy of the system and thus contains all the kinetic and potential terms needed to describe the energy. When we include more particles, we can include another potential term that models the interaction between the particles. The *relative distance* between two particles $\mathbf{r}_i = (x_i, y_i, z_i)$ and $\mathbf{r}_j = (x_j, y_j, z_j)$ is

$$r_{ij} = |\mathbf{r}_j - \mathbf{r}_i| \tag{2.3}$$

In this thesis we are interested in studying confined electrons, and will therefore use the *Coulomb electron-electron interaction*. The Hamiltonian for this interaction is given by

$$\hat{\mathbf{H}}_1 = \sum_{j<i} \frac{1}{r_{ij}} \tag{2.4}$$

and the total Hamiltonian is then $\hat{\mathbf{H}} = \hat{\mathbf{H}}_0 + \hat{\mathbf{H}}_1$. There are cases where we will not include the Coulomb interaction. This can be a good test for the accuracy of our implementations, as the solution will be easier to verify against the analytical solution.

2.1.1 The Double-Well Potential

The particle obviously behaves differently when we vary the potential $V(\mathbf{r})$. A *double-well potential* is similar to the standard harmonic oscillator, but with some kind of barrier or wall somewhere in the potential. A simple form of a double-well potential is described by

$$V_c(\mathbf{r}) = \frac{1}{2} m^* \omega_0^2 \min \left[\sum_{j=1}^{M} (\mathbf{r} - \mathbf{L}_j)^2 \right] \tag{2.5}$$

The difference between this potential and the normal harmonic oscillator well is the $\mathbf{L}_j$ term. It describes an M dimensional barrier, that creates minimas in the potential at $\mathbf{r} = \mathbf{L}_j$. The barriers can be set in one or all spatial directions with different potentials as a result.

An electron in the well can be situated at each side of the barrier. In addition, it has the potential to tunnel through to another side.

2.1.2 The Finite Square Well

A similar potential to the harmonic oscillator well is the *finite square well*. As the name suggests it has the shape of a rectangle. The potential has a constant value on the inside of the well and another constant value on the outside. This potential is useful as it is easy to recreate in a laboratory setting. For simplicity we often set the value inside the well to 0 and the outside to some constant V_0.

A conefinement potential V_c in one dimension with distance L from the center can then be modelled as

$$V_c = \begin{cases} 0, \text{ if } -L < x < L \\ V_0, \text{ otherwise} \end{cases} \tag{2.6}$$

In the case of a large potential outside the well, that is $V_0 \to \infty$, the energy approaches a limit given by

$$\lim_{V_0 \to \infty} E_n = \frac{\pi^2 \hbar^2 n^2}{8mL^2} \tag{2.7}$$

for energy levels $n = 1, 2, 3, \ldots$

The eigenfunctions are given by the following relation[3]

$$\psi_n(x) = \begin{cases} \sqrt{1/L}\cos(k_n x) \text{ when n is odd.} \\ \sqrt{1/L}\sin(k_n x) \text{ when n is even.} \end{cases} \tag{2.8}$$

where $k_n = \sqrt{2mE_n}/\hbar$ is determined by the boundary conditions.

However when the potential V_0 is closer in value to the potential inside the well, we have a much more complex situation, and no analytic solution for the energy states[3].

The eigenfunctions are given by

$$\psi_n(x) = \begin{cases} A_n e^{\kappa_n x} \text{ when } x \leq -L \\ B_n \cos(k_n x) \text{ when } |x \leq L|,\ n \text{ is odd.} \\ A_n e^{-\kappa_n x} \text{ when } x \geq L \end{cases} \tag{2.9}$$

and

$$\psi_n(x) = \begin{cases} -A_n e^{\kappa_n x} \text{ when } x \leq -L \\ B_n \sin(k_n x) \text{ when } |x \leq L|,\ n \text{ is even.} \\ A_n e^{-\kappa_n x} \text{ when } x \geq L \end{cases} \tag{2.10}$$

where B_n are normalization constants and

$$A_n = \begin{cases} e^{\kappa_n L}\cos(k_n L) B_n \text{ when n is odd.} \\ e^{\kappa_n L}\sin(k_n L) B_n \text{ when n is even.} \end{cases} \tag{2.11}$$

Sinxe ψ' should be continous at $x = \pm L$, we have the following constraints

$$\kappa_n = \begin{cases} k_n \tan(k_n L) \text{ when n is odd.} \\ -k_n \cot(k_n L) \text{ when n is even.} \end{cases} \tag{2.12}$$

Note that the eigenfunctions are actually even while n is odd, and odd while n is even.

2.2 Diagonalizing the Schrödinger Equation

A *separable* wavefunction can be factorized as the product of functions that each depend on only *one* spatial coordinate, $\psi(\mathbf{r}) = \psi_x(x)\psi_y(y)\psi_z(z)$. Such wavefunctions are much more favorable to solve, since we can now solve for each dimensions independently. This is particulary useful for the Schrödinger equation. In one dimension it can be formulated as

$$-\frac{1}{2}\frac{\mathrm{d}^2\psi(x)}{\mathrm{d}x^2} + V(x)\psi(x) = E_x\psi(x) \tag{2.13}$$

and by separating the wavefunction, we can solve the two or three dimensional Schrödinger equation as if it were one-dimensional. We will in the following section see how we can use a diagonalization scheme to achieve this.

2.2.1 Discretization

A continous wavefunction $u(x)$ can be discretized on a grid of finite size. That is, for a given dimension we choose a value h that defines the resolution of the grid. The double derivative of the function u, can on discretized form now be written

$$u''(x) = \frac{u(x+h) - 2u(h) + u(x-h)}{h^2} + \mathcal{O}(h^2) \tag{2.14}$$

By giving each point on the grid an index $i = 1, \dots, N$, where N is the number of points on the grid, we can simplify the notation of eq. 2.14. First we must define the step h by the end points of the grid

$$h = \frac{x_{\text{max}} - x_{\text{min}}}{N} \tag{2.15}$$

which yields the value at a point x_i by

$$x_i = x_{\text{min}} + ih \tag{2.16}$$

and $x_{i\pm1} = x_i \pm h$. This means we can rewrite the wavefunction and the double derivative as

$$u(x_i) = u_i \tag{2.17}$$

$$u''(x_i) = u_i'' = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} + \mathcal{O}(h^2) \tag{2.18}$$

Then, when we discard the higher-order terms, the Schrödinger equation eq. 2.13 can be approximated by

$$-\frac{1}{2}\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} + V_i u_i = \lambda u_i \tag{2.19}$$

Or even simpler

$$e_{i-1}u_{i-1} + d_i u_i + e_{i+1}u_{i+1} = \lambda u_i \tag{2.20}$$

where

$$e_i = -\frac{1}{2h^2} \qquad \text{and} \qquad d_i = \frac{1}{h^2} - V_i \tag{2.21}$$

This eigenvalue problem can also be represented on matrix form as

$$\begin{bmatrix} d_1 & e_2 & 0 & \dots & \dots & \dots & 0 \\ e_1 & d_2 & e_3 & \ddots & & & \vdots \\ 0 & e_2 & d_3 & e_4 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & \ddots & e_{N-2} & d_{N-1} & e_N \\ 0 & \dots & \dots & \dots & 0 & e_{N-1} & d_N \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ u_N \end{bmatrix} = \lambda \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ u_N \end{bmatrix}. \tag{2.22}$$

which can be solved easily, since the matrix is tridiagonal.

2.2.2 Linear Combination of Harmonic Oscillator Eigenstates

When we solve the Schrödinger equation for some potential, we obtain a set of eigenstates $\psi_{n'}$ and eigenvalues $E_{n'}$, each corresponding to the wavefunction and the total energy of the particle for given energy levels n'.

Each of these eigenstates can be approximated with a linear combination of some other basis. For example, we may solve the Schrödinger equation for the harmonic oscillator potential, and obtain eigenstates φ_n^{ho}. The approximated eigenstate (in one dimension) for the desired well is then given by

$$\psi_{n'}(x) \simeq \sum_{n=0}^{\Lambda-1} c_{n'n} \varphi_n^{\text{ho}}(x) \tag{2.23}$$

where Λ is the number of harmonic oscillator basis functions we want to use for the approximation and n represents the *quantum number*.

Since we know both $\psi_{n'}$ and φ_n^{ho}, we can find the overlap coefficients[31] as

$$c_{n'n} = \langle \psi_{n'} | \varphi_n^{\text{ho}} \rangle = \sum_{i=0}^{N-1} \psi_{n'}(x_i) \varphi_n^{\text{ho}}(x_i) \tag{2.24}$$

that is, the innerproduct of the eigenstates with respect to the position. N is the resolution of our discretization. A higher number will generally yield a better result.

The harmonic oscillator functions are on the form

$$\varphi_n^{\text{ho}} = \frac{1}{\sqrt{2^n n!}} \left(\frac{m\omega}{\pi\hbar}\right)^{1/4} \exp\left(-\frac{m\omega}{2\hbar}x^2\right) H_n\left(\sqrt{\frac{m\omega}{\hbar}}x\right) \tag{2.25}$$

$$= \frac{1}{\sqrt{2^n n!}} \left(\frac{\omega}{\pi}\right)^{1/4} \exp\left(-\frac{\omega}{2}x^2\right) H_n\left(\sqrt{\omega}x\right) \tag{2.26}$$

On the second line we have introduced natural units. The *harmonic oscillator frequency* is denoted by ω.

To solve this equation, we need the solution of the *Hermite polynomials*, $H_n(y)$. The set of solutions are obtained from the second-order differential equation

$$\frac{\mathrm{d}^2}{\mathrm{d}y^{\,2}}H(y) - 2y\frac{\mathrm{d}}{\mathrm{d}y}H(y) + (a-1)H(y) = 0 \tag{2.27}$$

with a as a constant. The equation has a recursion relation given by

$$H_{n+1}(y) = 2yH_n(y) - 2nH_{n-1}(y) \tag{2.28}$$

The initial values are

$$H_0(y) = 1 \tag{2.29}$$
$$H_1(y) = 2y \tag{2.30}$$

2.2.3 The Single-Particle Wavefunction

The wavefunction in equation 2.23 is one dimensional. For two and more dimensions, the harmonic oscillator basis function is separable. In three dimensions this equates to $\varphi_n^{\text{ho}}(\mathbf{r}) = \varphi_{n_x}^{\text{ho}}(x)\varphi_{n_y}^{\text{ho}}(y)\varphi_{n_z}^{\text{ho}}(z)$, where each spatial dimension has its own quantum number. The energy level of the eigenstate is given by the sum of the quantum numbers, $n_x + n_y + n_z$. In our notation n denotes which number the eigenstate has (up to N). In the one-dimensional case, the eigenstate number n equals the energy level. For higher dimensions, each energy level can be populated by several states (so-called *degeneracy of eigenstates*).

In three dimensions we have the following first two energy levels

$$\varphi_0^{\text{ho}}(\mathbf{r}) = \varphi_0^{\text{ho}}(x)\varphi_0^{\text{ho}}(y)\varphi_0^{\text{ho}}(z)$$

$$\varphi_1^{\text{ho}}(\mathbf{r}) = \varphi_1^{\text{ho}}(x)\varphi_0^{\text{ho}}(y)\varphi_0^{\text{ho}}(z), \quad \varphi_2^{\text{ho}}(\mathbf{r}) = \varphi_0^{\text{ho}}(x)\varphi_1^{\text{ho}}(y)\varphi_0^{\text{ho}}(z), \quad \varphi_3^{\text{ho}}(\mathbf{r}) = \varphi_0^{\text{ho}}(x)\varphi_0^{\text{ho}}(y)\varphi_1^{\text{ho}}(z)$$

The second energy level thus has a three-fold degeneracy and we have made a choice to make the x component be the first that increase with increasing energy levels. It should now be clear that the way we order these eigenstates are not unique, and care must be taken when choosing the order.

Since we have separated the harmonic oscillator basisfunctions, the constant $c_{n'n}$ in equation 2.23 is also separated into three. The approximated wavefunction will therefore have the form of

$$\psi_{n'}(\mathbf{r}) \simeq \sum_{n=0}^{\Lambda-1} c_{n'n_x}\varphi_{n_x}^{\text{ho}}(x)c_{n'n_y}\varphi_{n_y}^{\text{ho}}(y)c_{n'n_z}\varphi_{n_z}^{\text{ho}}(z) \tag{2.31}$$

in three dimensions. The spatial quantum numbers n_x, n_y, n_z are given by the eigenstate number n.

In this thesis we are interested in how many harmonic oscillator basisfunctions we need in order to get a good representation of the single-particle wavefunction for the system we are looking at. Therefore we need to know the total number of basisfunctions M for a given energy level m. The relations for two and three dimensions respectively are

$$M_{2D} = \frac{m(m+1)}{2} \tag{2.32}$$

and

$$M_{3D} = \frac{m(m+1)(m+2)}{6} \tag{2.33}$$

2.3 Quantum Monte Carlo Methods

In this section we will give an overview of how we can derive *Quantum Monte Carlo methods (QMC)*. They are methods used to solve the Schrödinger equation statistically, implementing *Markov Chains* in the form of *random walks*. We have used the [15], [13], [11], [4] and [16] as sources for this section.

We will use an approach where we model the Schrödinger equation as a diffusion equation[20]. The result is the *Diffusion Monte Carlo (DMC)* method. Variational Monte Carlo (VMC) is a less complex part of the DMC method and is the main focus of this thesis.

Throughout this chapter, we will use the atomic units, meaning $\hbar = m_e = e = 1$.

2.3.1 The Variational Principle

A quantum mechanical system given by the Hamiltionian $\hat{\mathbf{H}}$ has a set of eigenvectors ψ. The Hamiltonian will yield the eigenvalues of the system when it acts on the eigenvectors. If we do not have the eigenstates of the Hamiltonian, we can guess a set of states that we believe will produce a result close to the exact eigenvalues. We call these states *trial wavefunctions* ψ_T. Usually these functions have some parameters α that we can tweak to improve the results of the initial guess.

The variational principle is a method to estimate the ground state energy E_0 of the Hamiltonian $\hat{\mathbf{H}}$. It states that the expectation value of the Hamiltonian,

$$\langle \hat{\mathbf{H}} \rangle = \frac{\int d\mathbf{R}\, \psi_T^*(\mathbf{R};\boldsymbol{\alpha})\hat{\mathbf{H}}\psi_T(\mathbf{R};\boldsymbol{\alpha})}{\int d\mathbf{R}\, \psi_T^*(\mathbf{R};\boldsymbol{\alpha})\psi_T(\mathbf{R};\boldsymbol{\alpha})} \tag{2.34}$$

is an upper bound for the ground state energy, such that

$$E_0 \leq \langle \hat{\mathbf{H}} \rangle \tag{2.35}$$

This is the basis for the Variational Monte Carlo method.

2.3.2 The Diffusion Problem

In quantum mechanics, the wavefunction squared, $|\Psi(\mathbf{r}, t)|^2$, is the probability function of the system. Since we are operating with probability functions, we are able to look at the Schrödinger equation as a diffusion problem, where the electrons of the system are the diffusing particles.

The key concept of the Monte Carlo methods is to generate an ensemble of so-called random walkers. These walkers will traverse space randomly, giving many different position values. Then the average of the paths of the walkers in an infinitesimal volume element $d\mathbf{r}$ corresponds to the probability distribution $|\Psi(\mathbf{r}, t)|^2 d\mathbf{r}$.

In this chapter we will connect the use of Markov chains and the diffusion of the random walkers to the quantum mechanical system.

Let us first look at the eigenequation representation of the Schrödinger equation[29],

$$\hat{\mathbf{H}}\varphi_i = \epsilon_i \varphi_i \tag{2.36}$$

for stationary states φ_i.

The superposition of the stationary states[15][17],

$$\Psi(\mathbf{r}) = \sum_i c_i \varphi_i(\mathbf{r}) \tag{2.37}$$

is then also a solution of the eigenequation. Furthermore, the time-dependent Schrödinger equation[12] is given as

$$i\frac{\partial}{\partial t}\Psi(\mathbf{r}, t) = \hat{\mathbf{H}}\Psi(\mathbf{r}, t) \tag{2.38}$$

The time variable t in the time-dependent Schrödinger equation can be transformed to imaginary time as $t \to it = \tau$, giving

$$-\frac{\partial}{\partial t}\Psi(\mathbf{r}, \tau) = \hat{\mathbf{H}}\Psi(\mathbf{r}, \tau) \tag{2.39}$$

A solution of this equation, given a time-independent Hamiltonian, is simply

$$\Psi(\mathbf{r}, \tau) = e^{-\hat{\mathbf{H}}\tau/\hbar}\Psi(\mathbf{r}) = \sum_i c_i e^{-\epsilon_i \tau/\hbar}\varphi_i(\mathbf{r}) \tag{2.40}$$

The eigenvalues in this equation will decide the faith of the exponential as $\tau \rightarrow \infty$,

$$e^{-\epsilon_i \tau} \rightarrow \begin{cases} 0 \text{ for positive } \epsilon_i \\ \infty \text{ for negative } \epsilon_i \end{cases} \tag{2.41}$$

The goal of the variatonal Monte Carlo method was to end up with an approximation of the ground state energy, ϵ_0. Assume now that we could have an ideal situation where $\Psi(\mathbf{r}, \tau) \rightarrow c_0 \varphi_0$ as $\tau \rightarrow \infty$. This motivates the introduction of a *trial energy* term E_T. By adding this constant to the Hamiltonian's potential term, we get

$$\Psi(\mathbf{r}, \tau) = e^{-(\hat{\mathbf{H}} - E_T)\tau/\hbar} \Psi(\mathbf{r}) = \sum_i c_i e^{-(\epsilon_i - E_T)\tau/\hbar} \varphi_i(\mathbf{r}) \tag{2.42}$$

An ideal situation here would be that the trial energy was equal to the ground state energy, yielding

$$\Psi(\mathbf{r}, \tau) = e^{-(\hat{\mathbf{H}} - \epsilon_0)\tau/\hbar} \Psi(\mathbf{r}) = c_0 \varphi_0 + \sum_{i>0} c_i e^{-(\epsilon_i - \epsilon_0)\tau/\hbar} \varphi_i(\mathbf{r}) \tag{2.43}$$

and

$$\lim_{\tau \rightarrow \infty} \Psi(\mathbf{r}, \tau) = c_0 \varphi_0 \tag{2.44}$$

The result is that any wavefunction will be *projected* onto the ground state, as long as the overlap coefficient $c_0 \neq 0$. We will call these wavefunctions the trial wavefunctions, ψ_T, in the following. The choice of the trial wavefunction will determine how close we will get to the true ground-state energy.

It can be useful to mention that the physical properties are kept, as a constant shift in the point which the energy is zero will not affect the system. Thus, by introducing a constant term to the Hamiltonian's potential term, we are able to control the behavior of the exponential.

2.3.3 The Projection Operator

The time evolution operator is an operator that evolves a state in time from some starting point τ_0, by giving the state a time dependence. From equation (2.42), the time-evolution operator is just

$$\hat{\mathbf{U}}(\tau, \tau_0) = e^{-(\hat{\mathbf{H}} - E_T)(\tau - \tau_0)/\hbar} \tag{2.45}$$

with $\tau_0 = 0$. As the time-evolution operator in this case *projects* out the ground state from $\Psi(\mathbf{r})$, it can be referred to as a *projection operator*. Monte Carlo methods that makes projections in Hilbert space are often called *projector Monte Carlo methods*.

2.3.4 The Green's Function

In Dirac notation, one would write the time-dependent wavefunction in momentum space in the following way

$$\Psi(\mathbf{r},\tau) = \langle \mathbf{r}|\hat{\mathbf{U}}(\tau,0)|\Psi\rangle \tag{2.46}$$

The completeness relation

$$\int \mathrm{d}x' \; |x'\rangle\langle x'| = \mathbb{1} \tag{2.47}$$

let's us rewrite this in a more convenient way

$$\boxed{\Psi(\mathbf{r},\tau) = \int \mathrm{d}\mathbf{r}' \; \langle \mathbf{r}|e^{-(\hat{\mathbf{H}}-E_T)\tau}|\mathbf{r}'\rangle\langle\mathbf{r}'|\Psi\rangle = \int \mathrm{d}\mathbf{r}' \; G(\mathbf{r},\mathbf{r}';\tau)\Psi(\mathbf{r}') \qquad (2.48)}$$ where we have defined the *Green's function* as

$$G(\mathbf{r},\mathbf{r}';\tau) = \langle \mathbf{r}|e^{-(\hat{\mathbf{H}}-E_T)\tau}|\mathbf{r}'\rangle \tag{2.49}$$

For a Hamiltonian on the form $\hat{\mathbf{H}} = \hat{\mathbf{T}} + \hat{\mathbf{V}} = -\frac{\hbar^2\hat{\nabla}^2}{2m} + \hat{\mathbf{V}}(\mathbf{r})$, it would be beneficial if the Hamiltonian exponential could be split into separate parts for the kinetic and potential operator. Since $\hat{\mathbf{T}}$ and $\hat{\mathbf{V}}$ are operators, we must apply the *Baker-Campbell-Haussdorf formula*[1]

$$e^{\hat{\mathbf{T}}}e^{\hat{\mathbf{V}}} = e^{\hat{\mathbf{T}}+\hat{\mathbf{V}}+\frac{1}{2}[\hat{\mathbf{T}},\hat{\mathbf{V}}]+\dots} \tag{2.50}$$

The Green's function can then be written

$$\begin{aligned}
G(\mathbf{r},\mathbf{r}';\tau) &= \langle \mathbf{r}|e^{-\hat{\mathbf{T}}\tau}e^{-(\hat{\mathbf{V}}-E_T)\tau}|\mathbf{r}'\rangle + \mathcal{O}(\tau^2) && (2.51)\\
&= \int \mathrm{d}\mathbf{r}'' \; \langle \mathbf{r}|e^{-\hat{\mathbf{T}}\tau}|\mathbf{r}''\rangle\langle\mathbf{r}''|e^{-(\hat{\mathbf{V}}-E_T)\tau}|\mathbf{r}'\rangle + \mathcal{O}(\tau^2) && (2.52)\\
G(\mathbf{r},\mathbf{r}';t) &= \langle \mathbf{r}|e^{-\hat{\mathbf{T}}\tau}e^{-(\hat{\mathbf{V}}-E_T)\tau}|\mathbf{r}'\rangle + \frac{1}{2}[\hat{\mathbf{T}},\hat{\mathbf{V}}] + \mathcal{O}(\tau^3) && (2.53)\\
&= \int \mathrm{d}\mathbf{r}'' \; \langle \mathbf{r}|e^{-\hat{\mathbf{T}}\tau}|\mathbf{r}''\rangle\langle\mathbf{r}''|e^{-(\hat{\mathbf{V}}-E_T)\tau}|\mathbf{r}'\rangle + \mathcal{O}(\tau^2) && (2.54)\\
& && (2.55)
\end{aligned}$$

We have again used the completeness relation. The $\mathcal{O}(\tau^2)$ term will fade away when $\tau \to 0$. Therefore we call this the *short time approximation*[13]. The remaining expression is now made up of a separate kinetic and potential part, which we will name the *diffusion* and *branching* term, given respectively by

$$G_d = e^{\frac{1}{2}\hat{\nabla}^2\tau} \tag{2.56}$$

$$G_b = e^{-(\hat{\mathbf{V}}-E_T)\tau} \tag{2.57}$$

[1]Reference to formula. Appendix or source.

The kinetic (or diffusing part) may be calculated analytically. It has the following proportionality relationship

$$G_d \propto e^{-\frac{(\mathbf{r}''-\mathbf{r})^2}{4D\tau}} \tag{2.58}$$

This is called *isotropic diffusion*. It is the simplest type of diffusion where the particles has equal probability of diffusing in any direction. The general form of isotropic diffusion is

$$\frac{\partial P(\mathbf{r},t)}{\partial t} = D\nabla^2 P(\mathbf{r},t) \tag{2.59}$$

where D is the diffusion constant. In our equation $D = 1/2$.

In the next section we will see how we can use the *Langevin*[7] and *Fokker-Planck*[26] equations to get a suggestion for the next move of the walker. The Fokker-Planck equation will also let us introduce a drift term in the diffusion equation. Thus the diffusion is not equally probable in every direction, it will rather follow a gradient.

Langevin and Fokker-Planck Equations

The Langevin equation is a stochastic differential equation that originally described Brownian motion. It relates the time evolution of variables, or degrees of freedom, to the time evolution of the distribution. With one degree of freedom, it can be written as an equation of motion on the following general form[26]

$$\frac{\mathrm{d}y}{\mathrm{d}t} = A(y,t) + B(y,t)\xi(t) \tag{2.60}$$

where $\xi(t)$ is a given stochastic process. It has the following properties for a Markov process

$$\langle \xi(t) \rangle = 0 \tag{2.61}$$

$$\mathrm{var}(\xi(t)) = 2D\delta t \tag{2.62}$$

$A(y,t)$ and $B(y,t)\xi(t)$ are called the *drift* and *diffusion* terms respectively. The probability distribution obeys a Fokker-Planck equation given by[26]

$$\frac{\partial P}{\partial t} = -\frac{\partial}{\partial y}\left(\left[A(y,t) + DB(y,t)\frac{\partial B(y,t)}{\partial y}\right] P\right) + D\frac{\partial^2}{\partial y^2}\left[B^2(y,t)P\right] \tag{2.63}$$

Relating these two equations for the simple isotropic diffusion equation, we get the differential equation

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \xi(t) \tag{2.64}$$

Solving this "numerically" with the Euler method, we get an expression for the new position of the walker,

$$x_{i+1} = x_i + \xi \tag{2.65}$$

Let's set the diffusion part $B(y,t) = 1$ and the drift part $A(y,t) = DF(y,t)$ in the Fokker-Planck equation. Now we can rewrite equation 2.63 in three spatial dimensions as

$$\frac{\partial P(\mathbf{r},t)}{\partial t} = D\hat{\nabla} \cdot \left[\left(\hat{\nabla} - \mathbf{F}(\mathbf{r},t) \right) P(\mathbf{r},t) \right] \tag{2.66}$$

We call this drift term $\mathbf{F}(\mathbf{r},t)$ the *quantum force*.

The goal of a Markov process is to converge to a stationary state. Such a state is obtained when the left side of equation 2.63 is zero. When this is the case, we have that

$$\hat{\nabla}^2 P(\mathbf{r},t) = P(\mathbf{r},t)\hat{\nabla} \cdot \mathbf{F}(\mathbf{r},t) + \mathbf{F}(\mathbf{r},t) \cdot \hat{\nabla} P(\mathbf{r},t) \tag{2.67}$$

For the left side to cancel the right side, the quantum force must be on the form $\mathbf{F}(\mathbf{r},t) = g(\mathbf{r},t)\hat{\nabla}P(\mathbf{r},t)$, giving

$$\nabla^2 P = P\frac{\partial g}{\partial P}\left|\hat{\nabla}P\right|^2 + Pg\hat{\nabla}^2 P + g\left|\hat{\nabla}P\right|^2 \tag{2.68}$$

Thus the function $g(\mathbf{r},t) = {}^1/_{P(\mathbf{r},t)}$. The quantum force is then

$$\begin{aligned} \mathbf{F}(\mathbf{r},t) &= \frac{1}{P(\mathbf{r},t)}\hat{\nabla}P(\mathbf{r},t) & (2.69) \\ &= \frac{2}{\Psi(\mathbf{r},t)}\hat{\nabla}\Psi(\mathbf{r},t) & (2.70) \\ & & (2.71) \end{aligned}$$

It is responsible for *pushing* the walkers into regions where the trial wavefunction takes large values. This type of diffusion is called *anisotropic diffusion*.

The Langevin equation of the anisotropic Fokker-Planck equation can now be written on discretized form as

$$\frac{\mathrm{d}x_i}{\mathrm{d}t} = DF_i(\mathbf{r}) + \xi(t) \tag{2.72}$$

and we can solve it as we did in the isotropic case. This yields

$$x_{i+1} = x_i + \xi + DF_i(\mathbf{r})\delta t \tag{2.73}$$

The closed-form solution of the Green's function is now

$$G_d \propto e^{-(x-x''-D\delta t F(x_i))^2/(4D\delta t)} \tag{2.74}$$

2.3.5 Importance Sampling

It is now time to introduce the time-independent *trial wavefunction* $\Psi_T(\mathbf{r})$ into our calculations. The related *local energy* is given by

$$E_L \equiv \frac{1}{\Psi_T}\hat{\mathbf{H}}\Psi_T \tag{2.75}$$

When we include the trial-energy offset to the time-dependent Schrödinger equation, we have that

$$\frac{\partial \Psi(\mathbf{r},\tau)}{\partial \tau} = D\hat{\nabla}^2\Psi(\mathbf{r},\tau) - (\hat{\mathbf{V}} - E_T)\Psi(\mathbf{r},\tau) \tag{2.76}$$

where the diffusion constant is $D \equiv \frac{\hbar^2}{2m}$.

To relate the recently introduced trial wavefunction to the analytical wavefunction Ψ, we define

$$f(\mathbf{r},\tau) \equiv \Psi(\mathbf{r},\tau)\Psi_T(\mathbf{r}) \tag{2.77}$$

Substituting $\Psi = f/\Psi_T$ into equation 2.76 gives us

$$\frac{1}{\Psi_T}\frac{\partial f(\mathbf{r},\tau)}{\partial \tau} = D\hat{\nabla}^2\left(\frac{f(\mathbf{r},\tau)}{\Psi_T(\mathbf{r})}\right) - (\hat{\mathbf{V}} - E_T)\frac{f(\mathbf{r},\tau)}{\Psi_T(\mathbf{r})} \tag{2.78}$$

The first term on the right-hand side is the kinetic part of the Hamiltonian. It will be our first focus. We double differentiate f/Ψ_T with respect to the position using the product rule repeatedly. To simplify calculations further, we will also insert the quantum force $\mathbf{F}(\mathbf{r},t) = \frac{2}{\Psi_T}\hat{\nabla}\Psi_T$.

$$-\hat{\mathbf{K}} = D\hat{\nabla}^2\left(\frac{f}{\Psi_T}\right) \tag{2.79}$$

$$= D\hat{\nabla}\cdot\left(\frac{\hat{\nabla}f}{\Psi_T} - \frac{f\hat{\nabla}\Psi_T}{\Psi_T^2}\right) \tag{2.80}$$

$$= D\left[\frac{\Psi_T\hat{\nabla}f - \hat{\nabla}f\cdot\hat{\nabla}\Psi_T}{\Psi_T^2} - \frac{1}{2}\hat{\nabla}\left(\frac{f\mathbf{F}}{\Psi_T}\right)\right] \tag{2.81}$$

$$= \frac{D}{\Psi_T}\left[\hat{\nabla}^2 f - \frac{\hat{\nabla}f\cdot\hat{\nabla}\Psi_T}{\Psi_T} - \frac{1}{2}\hat{\nabla}(f\mathbf{F}) + \frac{1}{2}\frac{f\mathbf{F}\hat{\nabla}\Psi_T}{\Psi_T}\right] \tag{2.82}$$

We recognize that we can split $\frac{1}{2}\hat{\nabla}(f\mathbf{F})$ into a negative and positive part, so that

$$-\hat{\mathbf{K}} = \frac{D}{\Psi_T}\left[\hat{\nabla}^2 f - \hat{\nabla}(f\mathbf{F}) + \frac{1}{2}\hat{\nabla}(f\mathbf{F}) + \frac{1}{2}\frac{f\mathbf{F}\hat{\nabla}\Psi_T}{\Psi_T} - \frac{\hat{\nabla}f\cdot\hat{\nabla}\Psi_T}{\Psi_T}\right] \tag{2.83}$$

$$= \frac{D}{\Psi_T}\left[\hat{\nabla}^2 f - \hat{\nabla}(f\mathbf{F}) + \frac{1}{2\Psi_T}\left(\Psi_T\hat{\nabla}(f\mathbf{F}) + f\mathbf{F}\hat{\nabla}\Psi_T\right) - \frac{\hat{\nabla}f\cdot\hat{\nabla}\Psi_T}{\Psi_T}\right] \tag{2.84}$$

Now we can combine the parts in the parenthesis to a common derivative and substitute away the **F** term again

$$-\hat{\mathbf{K}} = \frac{D}{\Psi_T}\left[\hat{\nabla}^2 f - \hat{\nabla}(f\mathbf{F}) + \frac{1}{2\Psi_T}\hat{\nabla}(\Psi_T f\mathbf{F}) - \frac{\hat{\nabla} f \cdot \hat{\nabla}\Psi_T}{\Psi_T}\right] \tag{2.85}$$

$$= \frac{D}{\Psi_T}\left[\hat{\nabla}^2 f - \hat{\nabla}(f\mathbf{F}) + \frac{1}{\Psi_T}\hat{\nabla}(f\hat{\nabla}\Psi_T) - \frac{\hat{\nabla} f \cdot \hat{\nabla}\Psi_T}{\Psi_T}\right] \tag{2.86}$$

With the quantum force gone, we differentiate the remaining part. We see that two of the terms cancel, leving only three terms behind.

$$-\hat{\mathbf{K}} = \frac{D}{\Psi_T}\left[\hat{\nabla}^2 f - \hat{\nabla}(f\mathbf{F}) + \frac{1}{\Psi_T}\hat{\nabla}^2\Psi_T f + \frac{1}{\Psi_T}\hat{\nabla} f \cdot \hat{\nabla}\Psi_T - \frac{\hat{\nabla} f \cdot \hat{\nabla}\Psi_T}{\Psi_T}\right] \tag{2.87}$$

$$= \frac{D}{\Psi_T}\left[\hat{\nabla}^2 f - \hat{\nabla}(f\mathbf{F}) + \frac{\hat{\nabla}^2\Psi_T}{\Psi_T} f\right] \tag{2.88}$$

Remembering the definition of the local energy in equation 2.75, the third term above can be referred to as the *kinetic local energy*,

$$\hat{\mathbf{K}}_L = -D\frac{\hat{\nabla}^2\Psi_T}{\Psi_T} \tag{2.89}$$

Inserting the expression for the kinetic energy back into equation 2.78, we see that we get the following expression for the Schrödinger equation

$$\frac{1}{\Psi_T(\mathbf{r})}\frac{\partial f(\mathbf{r},\tau)}{\partial \tau} = \frac{D}{\Psi_T(\mathbf{r})}\left[\hat{\nabla}^2 f(\mathbf{r},\tau) - \hat{\nabla}(f(\mathbf{r},\tau)\mathbf{F}(\mathbf{r}))\right] - (\hat{\mathbf{K}}_L + \hat{\mathbf{V}} - E_T)\frac{f(\mathbf{r},\tau)}{\Psi_T(\mathbf{r})}$$

or

$$\boxed{\frac{\partial f(\mathbf{r},\tau)}{\partial \tau} = D\hat{\nabla}^2 f(\mathbf{r},\tau) - D\hat{\nabla}(f(\mathbf{r},\tau)\mathbf{F}(\mathbf{r})) - (E_L - E_T)f(\mathbf{r},\tau) \qquad (2.90)}$$

$$E_{\text{QMC}} = \frac{1}{N}\int \mathrm{d}\mathbf{R}\, f(\mathbf{R},\tau)\frac{1}{\Psi_T(\mathbf{R})}\hat{\mathbf{H}}\Psi_T(\mathbf{R}) \tag{2.91}$$

$$= \frac{1}{N}\int \mathrm{d}\mathbf{R}\, \Phi(\mathbf{R},\tau)\hat{\mathbf{H}}\Psi_T(\mathbf{R}) \tag{2.92}$$

$$= \frac{1}{N}\langle\Phi(\mathbf{R},\tau)|\hat{\mathbf{H}}|\Psi_T(\mathbf{R})\rangle \tag{2.93}$$

and the normalization is given by

$$N = \int \mathrm{d}\mathbf{R}\, f(\mathbf{r},\tau) \tag{2.94}$$

$$= \int \mathrm{d}\mathbf{R}\, \Phi(\mathbf{r},\tau)\Psi_T(\mathbf{R}) \tag{2.95}$$

$$= \langle\Phi(\mathbf{R},\tau)|\Psi_T(\mathbf{R})\rangle \tag{2.96}$$

If the walkers converge to the ground state, $|\Phi(\mathbf{R},\tau)\rangle \rightarrow |\Phi_0(\mathbf{R})\rangle$, as they should, the QMC energy is

$$E_{\text{QMC}} = \frac{\langle \Phi(\mathbf{R},\tau)|\hat{\mathbf{H}}|\Psi_T(\mathbf{R})\rangle}{\langle \Phi(\mathbf{R},\tau)|\Psi_T(\mathbf{R})\rangle} \tag{2.97}$$

$$= E_0 \frac{\langle \Phi_0(\mathbf{R})|\Psi_T(\mathbf{R})\rangle}{\langle \Phi_0(\mathbf{R})|\Psi_T(\mathbf{R})\rangle} \tag{2.98}$$

$$= E_0 \tag{2.99}$$

2.3.6 Metropolis Sampling

It is intuitive to think of diffusion as a random walk. The particles spread in a seemingly random manner. This picture of the diffusion process allows us to connect the problem at hand to *Markov chains*.

Given two states i and j, we can define a probability that we go from state i to j, $W(i \rightarrow j)$. A reversible Markov process can then be written as

$$P_i W(i \rightarrow j) = P_j W(j \rightarrow i) \tag{2.100}$$

where the probability density of the i^{th} configuration is given by P_i.

$$P_i g(i \rightarrow j) A(i \rightarrow j) = P_j g(j \rightarrow i) A(j \rightarrow i) \tag{2.101}$$

In our system, the wavefunction determines the probability function and we can think of the Green's function as the selection probaility. When we insert this in the equation above, we get

$$|\psi_i|^2 G(i \rightarrow j) A(i \rightarrow j) = |\psi_j|^2 G(j \rightarrow i) A(j \rightarrow i) \tag{2.102}$$

$$\frac{A(j \rightarrow i)}{A(i \rightarrow j)} = R_G(j \rightarrow i) R_\psi(j \rightarrow i)^2 \tag{2.103}$$

where we have the ratios $R_G = {}^{G(i \rightarrow j)}/{}_{G(j \rightarrow i)}$ and $R_\psi^2 = {}^{|\psi_i|^2}/{}_{|\psi_j|^2}$.

To satisfy the conditions above[11], we let the move be accepted if the acceptance $A(i \rightarrow j) > 1$ and rejected otherwise. This can be formulated mathematically as

$$A(i \rightarrow j) = \min(1,\ R_G(i \rightarrow j) R_\psi(i \rightarrow j)^2) \tag{2.104}$$

and is called the *Metropolis choice*. When the proposal distribution is symmetric, we will have a cancellation of the Green's functions such that the acceptance rule simplifies to

$$A(i \rightarrow j) = \min(1,\ R_\psi(i \rightarrow j)^2) \tag{2.105}$$

which is the case for isometric diffusion and is called the *Metropolis algorithm*. In the anistropic case, there won't be any cancellation and is called the *Metropolis-Hastings algorithm*[27].

This means that we have to calculate the ratio between two Green's functions on the form of equation 2.74. By taking the logarithm of the Green's functions ratio, we see that the calculation can be done more efficiently by just subtracting the exponentials,

$$\log R_G(i \to j) = \log\left(G_d(j \to i)/G_d(i \to j)\right) \tag{2.106}$$

$$= \frac{1}{2}\left(F(x_j) - F(x_i)\right) \cdot \left(D\delta t(F(x_j) - F(x_i))/2 + x_j - x_i\right) \tag{2.107}$$

Thus we can write the acceptance-rejectance rules in the following manner

$$A(i \to j) = \min(1,\ \exp[\log R_G(i \to j)]R_\psi(i \to j)^2) \tag{2.108}$$

2.3.7 Branching of the Walkers

With help from the Metropolis algorithm, the job of the ensemble of walkers is to span a distribution that in a good way represents the trial wavefunction. The key element of diffusion Monte Carlo is that we make changes to the distribution by branching. What this means is that we create new or destroy the current walker based on the branching part of the Green's function. We set the following rules for determining wether we branch or not

- $G_B = 1$: No branching.
- $G_B = 0$: The current walker is destroyed.
- $G_B > 1$: We copy the current walker. On average $G_B - 1$ times.

To compute the G_B quantity efficiently we use the following rule. Given a random number drawn from a uniform distribution of numbers, $a \in [0, 1)$, we calculate

$$\bar{G}_B = \text{floor}(G_B + a) \tag{2.109}$$

The branching term for the isotropic and anisotropic diffusion are

$$G_B^{\text{iso}}(i \to j) = \exp(-((V(x_i) + V(x_j))/2 - E_T)\delta t) \tag{2.110}$$

$$G_B(i \to j) = \exp(-((E_L(x_i) + E_L(x_j))/2 - E_T)\delta t) \tag{2.111}$$

respectively.

In this thesis we have implemented the Variational Monte Carlo method, and thus we are looking at the case where $G_B = 1$. An outline of the VMC algorithm can be seen in algorithm 1.

Algorithm 1 VMC Algorithm

```
1: procedure INITIALIZATION
2:     Set a fixed number of MC steps.
3:     Choose initial position $\mathbf{R}$ and variational
         parameters $\boldsymbol{\alpha}$.
4:     Calculate $|\psi_T(\mathbf{R})|^2$.
5: procedure INITIALIZE ENERGY AND VARIANCE, START THE MC CALCULATION
6:     Find the trial position $\mathbf{R}' = \mathbf{R} + \delta \times r$,
         where $r \in [0, 1]$ is randomly selected.
7:     Use the Metropolis algorithm to determine if the
         move $w = \frac{P(\mathbf{R}')}{P(\mathbf{R})}$ is accepted or rejected.
8:     Given that the move is accepted, set $\mathbf{R} = \mathbf{R}'$.
9:     Update averages.
10: procedure COMPUTE FINAL AVERAGES
```

2.4 The Trial Wavefunction and Slater Determinants

The arguably most important part of a VMC calculation is to find good trial wave functions. In the following, we will present the wave functions we have used in this thesis.

Our trial wave function is a product of the Slater determinant ψ_D and a correlation part ψ_C that considers the electron-electron repulsion. The second part is also called the *Jastrow factor*[15]. The general trial wave function is

$$\psi_T = \psi_D \psi_C \tag{2.112}$$

The trial wave functions are always dependent on the coordinates of every electron in some way. The Jastrow factor is generally written as

$$\psi_C = \prod_{i<j}^{n} \exp\left(\frac{a r_{ij}}{1 + \beta r_{ij}}\right) \tag{2.113}$$

where a is a factor that takes into account the spin of electron i and j, the *electron-electron cusp conditions*. When the particles have equal spin we have $a = 1/4$, and when they are different $a = 1/2$ (for example the ground state of Helium). The Slater determinant part is given by (2.122).

We also want to determine the local energy E_L to approximate the ground state energy of the atom. The general expression for the local energy is

$$E_L = \frac{1}{\psi_T(\mathbf{R})} \hat{\mathbf{H}} \psi_T(\mathbf{R}) \tag{2.114}$$

From this expression, it's clear that $\hat{\mathbf{K}}$ is the only operator that changes the trial wavefunction when we calculate the local energy. Therefore, we must calculate the following quantities

$$\frac{1}{\psi_T}\hat{\mathbf{k}}_i\psi_T = -\frac{1}{2}\frac{\hat{\nabla}_i^2\psi_T}{\psi_T} \tag{2.115}$$

The product rule of differentiation gives us

$$\frac{\hat{\nabla}^2\psi_T}{\psi_T} = \frac{\hat{\nabla}^2\psi_D}{\psi_D} + 2\frac{\hat{\nabla}\psi_D}{\psi_D}\cdot\frac{\hat{\nabla}\psi_C}{\psi_C} + \frac{\hat{\nabla}^2\psi_C}{\psi_C} \tag{2.116}$$

So we need to calculate the four quantities

$$\frac{\hat{\nabla}^2\psi_D}{\psi_D} \quad \frac{\hat{\nabla}\psi_D}{\psi_D} \quad \frac{\hat{\nabla}\psi_C}{\psi_C} \quad \frac{\hat{\nabla}^2\psi_C}{\psi_C} \tag{2.117}$$

They are derived in the appendix.

The following wave functions are the ones we used in the implementation.

2.4.1 Slater Determinants

The Hamiltonian we use is invariant when you interchange two particles, and since electrons are *fermions*, it is required that the wave function is antisymmetric.

In practice, this means that for a system of two particles, 1 and 2, described by the wave functions ψ_a and ψ_b, we must have a total wave function on the form

$$\psi^{1,2} = \psi_a^1\psi_b^2 - \psi_b^1\psi_a^2 \tag{2.118}$$

This is a result that follows from the fact that particles are indistinguishable, and the minus sign comes from the antisymmetry condition. An exchange operator $\hat{\mathbf{P}}$ acting on an antisymmetric wave function, will therefore change the sign, thus the wave function is an eigenfunction of $\hat{\mathbf{P}}$ with eigenvalue $p = -1$.

For bosons however, the sign is positive. This means that they do not obey the *Pauli principle*, stating that two particles can not occupy the same state, which would imply $\psi^{1,2} = 0$.

If we rewrite equation (2.118), we see that this is actually a determinant

$$\psi^{1,2} = \begin{vmatrix} \psi_a^1 & \psi_b^1 \\ \psi_a^2 & \psi_b^2 \end{vmatrix} \tag{2.119}$$

that can be generallized to the N-particle case as

$$\psi(\mathbf{r}_1,\ldots,\mathbf{r}_n) = \frac{1}{\sqrt{n!}}\begin{vmatrix} \psi_1^1 & \cdots & \psi_n^1 \\ \vdots & \ddots & \vdots \\ \psi_1^n & \cdots & \psi_n^n \end{vmatrix} \tag{2.120}$$

These Slater determinants obey exchange operations, since switching two rows of the determinant changes the sign. In this thesis we will omit the factor in front of the determinant, as the important part is proportionality, and we will see later that constants get cancelled.

In this thesis we are studying spin-$\frac{1}{2}$ particles, and we must address this problem in the Slater determinants we will use. The electrons have either spin $\uparrow$ or $\downarrow$, so our determinant in (2.120) have elements labeled $\uparrow$ or $\downarrow$. However, we must also label them by their quantum numbers, so an element of the determinant will have the form

$$\phi_{j\sigma}(\mathbf{r}_i) = \phi^i_{n_x n_y n_z \sigma} \tag{2.121}$$

where j is the quantum numbers, σ is the spin, and i is the particle number. Determinants of matrices with elements like that for Helium, Beryllium and Neon, are unfortunately zero, since they are independent of spin (two and two columns would be equal). This can be bypassed by writing the determinant $|\hat{\mathbf{D}}|$ as a product of two smaller ones for each spin[15][23], since

$$\psi_D = |\hat{\mathbf{D}}| \propto |\hat{\mathbf{D}}|_\uparrow |\hat{\mathbf{D}}|_\downarrow \tag{2.122}$$

For a four particle system, the Slater determinant is on the form

$$\psi_D \propto \begin{vmatrix} \phi^1_{100\uparrow} & \phi^2_{1100\uparrow} & \phi^3_{100\uparrow} & \phi^4_{100\uparrow} \\ \phi^1_{100\downarrow} & \phi^2_{100\downarrow} & \phi^3_{100\downarrow} & \phi^4_{100\downarrow} \\ \phi^1_{200\uparrow} & \phi^2_{200\uparrow} & \phi^3_{200\uparrow} & \phi^4_{200\uparrow} \\ \phi^1_{200\downarrow} & \phi^3_{200\downarrow} & \phi^2_{200\downarrow} & \phi^4_{200\downarrow} \end{vmatrix} \tag{2.123}$$

$$= \begin{vmatrix} \phi^1_{100\uparrow} & \phi^2_{100\uparrow} \\ \phi^1_{200\uparrow} & \phi^2_{200\uparrow} \end{vmatrix} \begin{vmatrix} \phi^3_{100\downarrow} & \phi^4_{100\downarrow} \\ \phi^3_{200\downarrow} & \phi^4_{200\downarrow} \end{vmatrix} \tag{2.124}$$

Here we have used equality instead of proportionality, since these are the actual wave functions we will use. We call the wavefunctions in the determinant for the *single-particle wavefunctions*.

2.5 Efficient Computation of the Slater Determinant

For larger systems, the evaluation of the gradient and the Laplacian of the Slater determinant becomes increasingly numerically demanding to compute. Computing these quantities with brute force, leads to $N \cdot d$ operations to find the determinant, where d is the number of dimensions. This must be multiplied with our $O(N^3)$ operations. In the following, we derive a method that deals with this issue, and achieves a lower number of operations.

We can approximate the Slater determinant as

$$\Psi_{SD}(\mathbf{r}_1, ..., \mathbf{r}_N) \propto \det\uparrow \cdot \det\downarrow \tag{2.125}$$

where the spin determinants are the determinants which only depend on spin up and spin down respectively. This is true only if the Hamiltonian $|H\rangle\langle\ |$ is spin independent.

Then, $\det\hat{\mathbf{D}} = |\hat{\mathbf{D}}| = |\hat{\mathbf{D}}|_{\uparrow}\cdot|\hat{\mathbf{D}}|_{\downarrow}$, where the Slater matrices are dependent on the positions of the electrons. Each time we update the positions and differentiate the Slater determinant, the Slater matrix is changed, but by calculating the determinant from scratch each time, we will certainly do unnecessary computations.

This is solved by the following algorithm, that instead of calculating the determinant, updates the *inverse* of the Slater matrix suitably.

We first express (i,j) elements of the inverse of $\hat{\mathbf{D}}$ as

$$d_{ij}^{-1} = \frac{C_{ji}}{|\hat{\mathbf{D}}|} \tag{2.126}$$

where C_{ji} is the transposed cofactor-matrix element of $\hat{\mathbf{D}}$.

This motivates the ratio

$$R \equiv \frac{|\hat{\mathbf{D}}(\mathbf{r}^{new})|_{\uparrow}}{|\hat{\mathbf{D}}(\mathbf{r}^{old})|_{\uparrow}} = \frac{\sum_{j=1}^{N} d_{ij}^{new} C_{ij}^{new}}{\sum_{j=1}^{N} d_{ij}^{old} C_{ij}^{old}} \tag{2.127}$$

Every time we move particle i, the i-th row of $\hat{\mathbf{D}}$ changes, and we have to update the inverse. However, the i-th row of $\hat{\mathbf{C}}$ is independent[2] of the i-th row of $\hat{\mathbf{D}}$, which means that we must have

$$\hat{\mathbf{C}}_{ij}^{new} = \hat{\mathbf{C}}_{ij}^{old} = (d_{ji}^{-1})^{old}\cdot|\hat{\mathbf{D}}| \text{ for } j = 1,...,N \tag{2.128}$$

and using

$$\sum_{k=1}^{N} d_{ik}\, d_{kj}^{-1} = \delta_{ij} \tag{2.129}$$

The result is

$$R = \sum_{j=1}^{N} d_{ij}^{new}(d_{ji}^{-1})^{old} = \sum_{j=1}^{N} \phi_j(\mathbf{r}_i^{new})\, d_{ji}^{-1}(\mathbf{r}_i^{old}) \tag{2.130}$$

The algorithm for updating the inverse of the matrix when a new position is accepted is then

We can then calculate the gradient and laplacian as

$$\frac{\hat{\nabla}_k|\hat{\mathbf{D}}|}{|\hat{\mathbf{D}}|} = \sum_{j=1}^{N} \nabla_k\phi_j(\mathbf{r}_i^{new})\, d_{ji}^{-1}(\mathbf{r}_i^{old}) \tag{2.131}$$

$$\frac{\hat{\nabla}_k^2|\hat{\mathbf{D}}|}{|\hat{\mathbf{D}}|} = \sum_{j=1}^{N} \nabla_k^2\phi_j(\mathbf{r}_i^{new})\, d_{ji}^{-1}(\mathbf{r}_i^{old}) \tag{2.132}$$

[2]Since the cofactor-matrix elements c_{ij} is defined by removing i-th row and j-th column from a matrix $\hat{\mathbf{A}}$, and then taking the determinant of the remaining matrix.

Algorithm 2 Inverse of Slater Matrix

```
1: procedure UPDATE COLUMNS $j \neq i$
2:     for each column $i \neq j$ do
3:         $S_j = \sum_{l=1}^{N} d_{il}(\mathbf{r}^{new}) d_{lj}^{-1}(\mathbf{r}^{old})$
4:         $(d_{kj}^{-1})^{new} = (d_{kj}^{-1})^{old} - \frac{S_j}{R}(d_{ki}^{-1})^{old}$
5: procedure UPDATE COLUMN $i$
6:     $(d_{ki}^{-1})^{new} = \frac{1}{R}(d_{ki}^{-1})^{old}$
```

The time consuming part of the computation of the Slater determinant, is calculating the Laplacian and gradient. This makes the number of operations go from the already undesirable $\mathcal{O}(N^3)$ to $\mathcal{O}(d \times N^4)$, since there are $d \times N$ independent coordinates. Using the method above, the total calculation scales as $\mathcal{O}(d \times N^2)$.

2.6 Variation of Parameters with Steepest Descent

Finding the ground energy is done with Monte-Carlo methods, but to have a trial wavefunction that gives the best result we must find the optimal variational parameters α and β. The best trial wavefunction is the one that returns the lowest ground state energy. Therefore we wish to minimize the energy as a function of the variational parameters.

The *Steepest Descent*[28] method[3] tries to minimize a function by taking steps proportional to the negative of the gradient of the function at some point α_i. The derivative (gradient) of the local energy is

$$E'_\alpha \equiv \frac{\mathrm{d}\langle E_L[\alpha]\rangle}{\mathrm{d}\alpha} \tag{2.133}$$

and a constant step length is defined by γ. Then the next approximation for a minima is

$$\alpha_{i+1} = \alpha_i - \gamma E'_\alpha < \alpha_i \tag{2.134}$$

Either an approximation or an analytical expression of the gradient can be used. By using the chain rule on equation 2.133, we get

$$E'_\alpha = 2\left[\left\langle \frac{1}{\psi_T[\alpha]}\psi'_\alpha E_L[\alpha]\right\rangle - \left\langle \frac{1}{\psi_T[\alpha]}\psi'_\alpha\right\rangle \langle E_L[\alpha]\rangle\right] \tag{2.135}$$

where

$$\psi'_\alpha \equiv \frac{\mathrm{d}\psi_T[\alpha]}{\mathrm{d}\alpha} \tag{2.136}$$

[3]Also called *gradient descent*.

This is done for both α and β and we must initially guess on the first values of the parameters, α_0 and β_0. To perform the mimimization efficiently, we should perform the steepest descent on a small system with few Monte-Carlo cycles, then use the optimal parameters to find the best approximation to the ground state enegy.

2.7 The Blocking Method

The Monte-Carlo simulations are a set of computational *experiments* with statistical errors. In these experiments, we are interested in the mean value of the ground energies and the density distribution. The individual samples are not as interesting to us, and we would rather like to look at the variance of the mean values. The variance of the mean value is closely connected with the correlation in the individual samples.

Both our random walker with and without importance sampling, gives correlated samples. In our case, we use *blocking*[10][13] as a technique to find out have many steps the walker has to do to be as uncorrelated as possible in comparison with its first step. This means that for a set of random samples A, one sample A_i will be correlated to some other sample A_{i+n}.

If we we group our data (N samples) in blocks of size n_b, then the number of blocks is $m = N/n_b$. The average of one block i (the average of all samples between $in_b + 1$ and $(i+1)n_b$), is given by

$$\langle A \rangle_i = \frac{1}{n_b} \sum_{j=in_b+1}^{(i+1)n_b} A_j \tag{2.137}$$

We can then calculate the average of all the blocks as

$$\langle A \rangle = \frac{1}{m} \sum_{i=1}^{m} \left(\frac{1}{n_b} \sum_{j=in_b+1}^{(i+1)n_b} A_j \right) \equiv \frac{1}{m} \sum_{i=1}^{m} \langle A \rangle_i \tag{2.138}$$

and the variance as

$$\mathrm{Var}(A) = \frac{1}{m} \sum_{i=1}^{m} \langle A \rangle_i^2 - \left(\frac{1}{m} \sum_{i=1}^{m} \langle A \rangle_i \right)^2 \tag{2.139}$$

By plotting the variance of the mean as a function of block size we will see that it reaches a plateau. This plateau means that increasing the sample length will no longer change the variance in the mean significantly. This happens at some block size $n_b > K_0$, where K_0 is called the *correlation length*. This makes us able to more easily calculate variance in the mean because we know how small block sizes we can use.

Algorithm 3 Blocking Method

```
1: procedure COMPUTE VARIANCE OF MEAN
2:     Compute MC calculation, store samples in array.
3:     Loop over a set of block sizes $n_b$.
4:     For each $n_b$, calculate the mean of the block,
         and store these values in a new array.
5:     Take the mean and variance of this array.
6:     Store results.
```

Part II

Implementation

3 Program Structure

In computational science and in programming in general, the idea of good and readable code has bloomed in recent years. No longer can you write huge pieces of code that is hard to decipher with short, unclear or even ambiguous naming conventions and expect to get away with it. This is definitely a good thing. It is especially important in the natural sciences were the ability to reproduce results is key.

For those reasons, an important part of this thesis has been to create a good code framework. This means that it should be easy to implement new concepts and methods and the code should be clear and concise. This includes naming conventions as well as abstraction levels in the code and what it does. Finally, it means that testing is crucial. We have chosen to implement a lot of unit tests, as this makes the work of writing and maintaining the code a lot easier.

The source code in this thesis is written in C++ and some assistance code is written in Python and can be found here: [9]. Much of the key concepts in this section and the idea behind the programming is gotten from the book *Clean Code: A Handbook of Agile Software Craftsmanship*[21].

3.1 Clean Code

3.1.1 Naming Conventions

When programming was a new thing, restrictions in some programming languages made short and non-descriptive naming conventions the norm. Unfortunately, this bad habit did not completely go away when there were no restrictions on the length of variable and function names. This isn't strange, as it it definitely very tempting to name a variable `sum` instead of `sumOfIntegers`. When your dealing with a short script or your own program, this isn't even necesarilly a bad thing. But when this is done in a large code base with several different contributors, some which might come in at later stages of production, it is really, really bad. In the end it will slow down or hinder progress. Therefore we will follow some general rules in this thesis:

Descriptive names All variables, functions, classes and methods should have names that lets you know what they are or do. Variable names like `c` when you mean `electronCharge` is not okay. It is probably wrong if you need a comment to let someone know what a variable is.

Consistent naming Variables and objects should be on the form of substantives, i.e. `electronPosition` and `SquareWell`. Names that start with verbs are reserved for things that *do* stuff, like functions and methods. `setElectronPosition`, `getHamiltonian` or `computeSlaterDeterminant` are examples of good function names. However, if you have functions that do some calculation then switching between verbs is not good. Stick to one verb to describe a calculation, don't have two functions that are called `computeDerivative` and `calculateSum`.

How one chooses to implement the rules with respect to things like camel case, or _ separated names is not so important as following the rules and being consistent.

3.2 Testing

A program without tests is a scary program. Without tests one should fear making changes to a working prorgram, as anything you do might produce a new set of bugs. Even worse than programs without tests are programs with bad testing. If the test code is not maintained to the same standard as the production code, it is difficult to adjust the testing code when changes are made to the production code. The cost of working with bad test code can then be as large if not larger than having no testing at all.

3.2.1 Unit Testing

A unit is the smallest testable part of a program. The role of the unit test is not to test the program's ability to produce the correct results, but rather that each part of the program does it's job. If each bit of code is strictly controlled, it's much easier to validate the superior task of the program.

Unit testing is a fundamental part of modern programming. It should be implemented alongside, preferably right *before* the production code. This means that any new production code written should have tests making More specifically it is wise to follow the the three laws defined by Robert C. Martin[1]:

[1]Pages 32–36 of [22]

First Law You may not write production code until you have written a failing unit test.

Second Law You may not write more of a unit test than is sufficient to fail, and not compiling is failing.

Third Law You may not write more production code than is sufficient to pass the currently failing test.

These laws ensures a coding workflow that forces you to produce tests that are up to speed with the production code—as long as the standard of the test code is kept at the same level as the production code.

Here follows an example how we did unit testing in this thesis. All the unit tests were down with the open source software *UnitTest++*[18]. We create suites of tests for a class and then test each tiny component and quantity that can be calculated. See listing 3.1.

```
SUITE(DoubleHarmonicOscillatorWellPotentialTests) {
    class DoubleHarmonicOscillatorFixture {
    public:
        double alpha = 1;
        double omega = 1;
        double beta = 0;
        double normConst = 1.;

        bool Jastrow = false;
        bool analyticalKinetic = true;
        bool repulsion = false;

        int numberOfDimensions = 2;
        int numberOfParticles = 1;
        int my_rank = 0;

        vec L = vec(3);

        System* system = new System();

        void initiateVector(vec L) {
            L.fill(0.);
            L(0) = 5.;
        }

        void initiateSystem(System* system, vec L, int
            numberOfDimensions, int numberOfParticles, int my_rank,
            double omega,
                                bool analyticalKinetic, bool repulsion,
                                    double Jastrow, double normConst,
                                double alpha, double beta) {
            system->setInitialState(new RandomUniform(system,
                numberOfDimensions, numberOfParticles, my_rank));
            system->setHamiltonian(new DoubleHarmonicOscillator(system,
                L, omega, analyticalKinetic, repulsion));
            system->setWaveFunction (new ManyElectrons(system, alpha,
                beta, omega, normConst, Jastrow));
```

```cpp

        }

    };

    TEST_FIXTURE(DoubleHarmonicOscillatorFixture, ComputeLocalEnergyTest
        ) {

        initiateVector(L);

        DoubleHarmonicOscillator* potential = new
            DoubleHarmonicOscillator(system, L, omega, analyticalKinetic,
             repulsion);

        system->setNumberOfParticles(numberOfParticles);
        system->setNumberOfDimensions(numberOfDimensions);

        system->setHamiltonian(potential);
        system->setInitialState(new RandomUniform(system,
            numberOfDimensions, numberOfParticles, my_rank));
        system->setWaveFunction(new ManyElectrons(system, alpha, beta,
            omega, normConst, Jastrow));

        std::vector<double> energies = potential->computeLocalEnergy(
            system->getParticles());
    }
    TEST_FIXTURE(DoubleHarmonicOscillatorFixture,
        ComputeHermitePolynomialTest) {


        DoubleHarmonicOscillator* potential = new
            DoubleHarmonicOscillator(system, L, omega, analyticalKinetic,
             repulsion);
        potential->setAlpha(alpha);

        int x1 = 1;
        int x2 = -2;

        //Check the first Hermite polynomials for x1 = 1
        CHECK_EQUAL(1, potential->computeHermitePolynomial(0, x1));
        CHECK_EQUAL(2, potential->computeHermitePolynomial(1, x1));
        CHECK_EQUAL(2, potential->computeHermitePolynomial(2, x1));
        CHECK_EQUAL(-4, potential->computeHermitePolynomial(3, x1));
        CHECK_EQUAL(-20, potential->computeHermitePolynomial(4, x1));
        CHECK_EQUAL(-8, potential->computeHermitePolynomial(5, x1));

        //Check the first Hermite polynomials for x2 = -2
        CHECK_EQUAL(1, potential->computeHermitePolynomial(0, x2));
        CHECK_EQUAL(-4, potential->computeHermitePolynomial(1, x2));
        CHECK_EQUAL(14, potential->computeHermitePolynomial(2, x2));
        CHECK_EQUAL(-40, potential->computeHermitePolynomial(3, x2));
        CHECK_EQUAL(76, potential->computeHermitePolynomial(4, x2));
        CHECK_EQUAL(16, potential->computeHermitePolynomial(5, x2));
    }
}
```
Listing 3.1: A unit test example from Hamiltonian class.

3.3 The Monte-Carlo Program

The primary classes of the program are outlined below and specifics are further explained in the following subsections.

Particle
This is the class that keeps the position information of a *single* particle. It is responsible for making changes to the position as well as processing this information to the rest of the program. There is one **Particle** instance for each particle in the system.

System
System holds information about the currently used physical system. Most importantly the type of wavefunction employed and the Hamiltonian in use. It is also responsible for the Metropolis algorithm and the steps involved in this process.

Sampler
Monte-Carlo simulations requires *sampling* of the calculated energy after each MC step. This class is responsible for this sampling and the sampling of other relevant quantities. At the end of a simulation **Sampler** calculates the average energy.

WaveFunction
As the name suggests, this is the class that stores the properties of the wavefunction. The wavefunctions studied in this theses are many-body wavefunctions with corresponding *Slater determinants* and the gradients (first derivative) and Laplacians (second derivative) of the wavefunction. These quantities are calculated and updated here.
The ratio used in the Metropolis algorithm is also calculated here and passed to **System** which determines wheter or not a move is accepted.

Hamiltonian
The Hamiltonian of our systems are made up of a kinetic and a potential part, where the potential is again split into a non-interacting and interacting part. This class calculates everything related to the Hamiltonian of the system. The evaluation of the single-particle wavefunction (and gradient and Laplacian) is also evaluated here and passed to **WaveFunction** when needed. This is done for practical reasons more than intuitive ones.

InitialStates
To initiate a system, certain things must be in order. We need to create a set of particles and give them initial positions. These are objects of type **Particle** (as explained above) and **InitialStates** does the work of initiating them.

VariationMethods
This class calculates the variational parameters of the simulation. We have implemented the *steepest decent method* to calculate these parameters, but more advanced methods like *conjugate gradient* are possible to implement without complications.

3.3.1 Initiating a Simulation

All simulations are run from the main program, "main.cpp". We first initiate the constant quantities. For the physical system, the number of particles, the number of spatial dimensions and the harmonic oscillator frequency must be set. The Monte-Carlo simulation also has certain constants, most importantly the number of Monte-Carlo cycles and our intitial guesses for the variational paramters.

When the main program is run, the **InitialStates** class initiates an instance of **Particle** for *each* particle randomly spaced. If potential barriers are placed in the system, we try to distribute the particles evenly on each side.

After the particles are in place, the **Hamiltonian** and **WaveFunction** are instantiated. **System** keeps track of all the class instances and will procede to execute the Metropolis algorithm.

3.3.2 Metropolis Sampling

As mentioned above, the **System** class is responsible for performing the Metropolis sampling. The main concept of Metropolis sampling is performing the random walk. For each Metropolis step, we change the position of every particle in the system. Our implementation has two ways of doing this. The brute-force method is to just assign the particle a new position which has been randomly distributed. A more accurate method is to use importance sampling. With importance sampling, the new position is dependent on the quantum force. This means that the physical system impacts the new position.

The new proposed position is then evaluated by taking the ratio of the wavefunctions of the old and new particle position. If this number equals or exceeds a randomly generated number between 0 and 1, we accept the proposal. When importance sampling is applied, we also consider the ratio between the Green's function of the new and old position as discussed in section 2.3.4. If the step is rejected, we keep the system in the same state as before.

After the Metropolis step has been accepted or rejected, we sample the energy. This is done by the **Sampler** class. Energy is accumulated in each step and averaged after all the steps have finished. Since the system is not equilibrated initially, the first few percent of the steps are not sampled. See listing 3.2.

```
bool System::metropolisStepImpSampling(int currentParticle){
    // Change position of current particle randomly to create a trial
        state.
    setCurrentParticle(currentParticle);

    std::vector<double> positionChange(m_numberOfDimensions);
    double D = 0.5;

    // Keep old position for Greens function
    std::vector<double> positionOld = m_particles[currentParticle]->
        getPosition();

    // Change position of current particle
    for (int i=0; i < m_numberOfDimensions; i++){
        positionChange[i] = Random::nextGaussian(0., sqrt(m_dt)) + D*
            m_dt*quantumForce()[i];
    }

    double qratio = m_waveFunction->computeMetropolisRatio(m_particles,
        currentParticle, positionChange);

    // Keep new position for Greens function
    std::vector<double> positionNew = m_particles[currentParticle]->
        getPosition();

    // Evaluate Greens functions and find Metropolis-Hastings ratio:
    double GreensFunctionNew = calculateGreensFunction(positionNew,
        positionOld);

    for (int i=0; i < m_numberOfDimensions; i++){
        m_particles[currentParticle]->adjustPosition(-positionChange[i],
             i);
    }

    double GreensFunctionOld = calculateGreensFunction(positionOld,
        positionNew);
    qratio *= GreensFunctionNew / GreensFunctionOld;

    // If move is accepted give the random particle the new position,
        otherwise keep the old position
    if (Random::nextDouble() <= qratio){
        for (int i=0; i<m_numberOfDimensions; i++){
            m_particles[currentParticle]->adjustPosition(positionChange[
                i], i);
        }
        m_waveFunction->updateSlaterDet(currentParticle);
        return true;
    }
    m_waveFunction->updateDistances(currentParticle);
    m_waveFunction->updateSPWFMat(currentParticle);
    m_waveFunction->updateJastrow(currentParticle);

    return false;
}
```

Listing 3.2: The Metropolis step calculation with importance sampling.

3.3.3 More on the Hamiltonian and WaveFunction classes

In equation 2.75 we defined the *local energy*. It is dependent on the trial wavefunction and the Hamiltonian of our system. The **Hamiltonian** class is reponsible for calculating the local energy. Our Hamiltonian is made up of a kinetic part and a potential part, which is again split into an interacting and non-interacting part.

Virtual Functions and Potential Energy

The **Hamiltonian** and **WaveFunction** have so-called *virtual functions*. A method defined by a subclass would normally override an identically named method in the superclass. When methods are virtual, the original method is not overwritten, but instead act as a template. Then each subclass can have its own implementation of methods for calculating i.e. the Hamiltonian as we have different subclasses for different Hamiltonians. When we create an instance of a subclass, the same methods are used for calculating the Hamiltonian or the kinetic energy, but the implementation of the method is unique for each subclass. This streamlines the program by avoiding multiple methods with different names that do similar things.

Virtual functions can either be regularly virtual or purely virtual. The pure virtual functions requires its own implementation of the methods defined in the superclass, while the regular virtual functions can be generally implemented in the superclass and then inherited directly in the subclass. The regular virtual functions can be useful if not all subclasses need their own implementation as we avoid duplicating code. If each subclass needs its own unique implementation of a method, then pure virtual functions can be a useful testing tool, as the program won't compile if the method isn't implemented.

The virtual functions are defined in the superclass as shown in listing 3.3

```
virtual std::vector<double> computeLocalEnergy(std::vector<class
    Particle*> particles) = 0;
virtual std::vector<double> computeLocalEnergy(std::vector<class
    Particle*> particles)  { /* Default implementation of
    computeLocalEnergy */ }
```

Listing 3.3: A pure virtual function and a regular virtual function as they would be implemented in **Hamiltonian**. When set to zero, the virtual function is pure, and we require an implementation in each subclass. Otherwise, it is regular, and we must implement a default method. If a subclas has its own implementation of `computeLocalEnergy`, then default method is overwritten.

The calculation of the potential energy of the Hamiltonian is an example of how we use virtual functions. We first need to set *which* Hamiltonian we are to use. As **System** is the class which holds the information about Hamiltonian, we need to make an instance of the desired subclass of **Hamiltonian** and feed it to the system. For example the standard harmonic oscillator well would be set in the following manner

```
    system->setHamiltonian      (new HarmonicOscillatorElectrons(system,
        alpha, omega, analyticalKinetic, repulsion));
```

while the finite square well is set like this

```
    system->setHamiltonian      (new SquareWell(system, V0, distToWall,
       alpha, omega, analyticalKinetic, repulsion));
```

This sets `m_hamiltonian` of **System** to the desired subclass. We can then compute the local energy with the `computeLocalEnergy` of `m_hamiltonian`

```
    m_hamiltonian->computeLocalEnergy(m_particles);
```

In practice this is used in the **Sampler** class

In listing 3.4 and 3.5 we see the difference in how the potential energy is calculated in the respective subclass of **Hamiltonian**.

```
double potentialEnergy = 0;
double repulsiveTerm = 0;

for (int i=0; i < numberOfParticles; i++){
    double rSquared = 0;
    std::vector<double> r_i = particles[i]->getPosition();
    for (int k=0; k < numberOfDimensions; k++){
        rSquared += r_i[k]*r_i[k];
    }
    potentialEnergy += rSquared;

    for (int j=i+1; j < numberOfParticles; j++){
        double r_ijSquared = 0;
        std::vector<double> r_j = particles[j]->getPosition();
        for (int k=0; k < numberOfDimensions; k++){
                r_ijSquared += (r_i[k] - r_j[k]) * (r_i[k] - r_j[k]);
        }

        double r_ij = sqrt(r_ijSquared);
        repulsiveTerm += 1./r_ij;
    }
}
potentialEnergy *= 0.5*m_omega*m_omega;

if (m_repulsion) {
    potentialEnergy += repulsiveTerm;
}
```

Listing 3.4:
The potential energy calculation in subclass **HarmonicOscillatorElectrons** of **Hamiltonian**.

```
double potentialEnergy = 0;
double repulsiveTerm = 0;

for (int i=0; i < numberOfParticles; i++){
    std::vector<double> r_i = particles[i]->getPosition();
    bool isOutside = false;

    for (int d = 0; d < m_numberOfDimensions; d++) {
        if (abs(r_i[d]) >= m_distToWall) {
            isOutside = true;
        }
    }
    if (isOutside) potentialEnergy += m_V0;

    for (int j=i+1; j < numberOfParticles; j++){
        double r_ijSquared = 0;
        std::vector<double> r_j = particles[j]->getPosition();
        for (int k=0; k < m_numberOfDimensions; k++){
                r_ijSquared += (r_i[k] - r_j[k]) * (r_i[k] - r_j[k]);
        }

        double r_ij = sqrt(r_ijSquared);
        repulsiveTerm += 1./r_ij;
    }
}

if (m_repulsion) {
    potentialEnergy += repulsiveTerm;
}
```

Listing 3.5: The potential energy calculation in subclass **SquareWell** of **Hamiltonian**.

The Kinetic Energy

The kinetic energy is dependent on the double derivative of the trial wavefunction. We can calculate the derivative in a few ways and either analytically or numerically. We have implemented both, each with their own strengths and weaknesses. For instance, the numerical derivation is independent of the type of potential we have. Therefore it is implemented generally in the **Hamiltonian** superclass, see listing 3.6.

```
double Hamiltonian::computeKineticEnergy(std::vector<Particle*>
    particles){
    // Compute the kinetic energy using numerical differentiation.

    double numberOfParticles = m_system->getNumberOfParticles();
    double numberOfDimensions = m_system->getNumberOfDimensions();
    double h = 1e-4;

    // Evaluate wave function at current step
    double waveFunctionCurrent = m_system->getWaveFunction()->evaluate(
        particles);
    double kineticEnergy = 0;

    for (int i=0; i < numberOfParticles; i++){
        for (int j=0; j < numberOfDimensions; j++){

            // Evaluate wave function at forward step
            particles[i]->adjustPosition(h, j);
            m_system->getWaveFunction()->updateDistances(i);
            m_system->getWaveFunction()->updateSPWFMat(i);
            m_system->getWaveFunction()->updateJastrow(i);
            double waveFunctionPlus = m_system->getWaveFunction()->
                evaluate(particles);

            // Evaluate wave function at backward step
            particles[i]->adjustPosition(-2*h, j);
            m_system->getWaveFunction()->updateDistances(i);
            m_system->getWaveFunction()->updateSPWFMat(i);
            m_system->getWaveFunction()->updateJastrow(i);
            double waveFunctionMinus = m_system->getWaveFunction()->
                evaluate(particles);

            // Part of numerical diff
            kineticEnergy -= (waveFunctionPlus - 2*waveFunctionCurrent +
                 waveFunctionMinus);

            // Move particles back to original position
            particles[i]->adjustPosition(h, j);
            m_system->getWaveFunction()->updateDistances(i);
            m_system->getWaveFunction()->updateSPWFMat(i);
            m_system->getWaveFunction()->updateJastrow(i);
        }
    }
    // Other part of numerical diff. Also divide by evaluation of
        current wave function
    // and multiply by 0.5 to get the actual kinetic energy.
    kineticEnergy = 0.5*kineticEnergy / (waveFunctionCurrent*h*h);
    return kineticEnergy;
}
```

Listing 3.6: The implementation of the kinetic energy obtained by numerically differentiation.

The analytical computation of the Laplacian is done with the method `computeDoubleDerivative` in **WaveFunction** and needs to be implemented specifically for each Hamiltonian.

The analytical solutions to the wavefunctions of the harmonic oscillator well also includes the Hermite polynomials. They have a recursion relation that can easily be implemented as shown in listing 3.7. This is the straight forward, brute-force method. It works for any number of Hermite polynomials. However it requires several operations that can be avoided. In the optimization chapter we suggest a faster way of doing these calculations.

```
double HarmonicOscillatorElectrons::computeHermitePolynomial(int nValue,
     double position) {
    // Computes Hermite polynomials.
    double alphaSqrt = sqrt(m_alpha);
    double omegaSqrt = sqrt(m_omega);
    double factor = 2*alphaSqrt*omegaSqrt*position;

    double HermitePolynomialPP = 0;                    // H_{n-2}
    double HermitePolynomialP = 1;                     // H_{n-1}
    double HermitePolynomial = HermitePolynomialP;     // H_n

    for (int n=1; n <= nValue; n++) {
        HermitePolynomial = factor*HermitePolynomialP - 2*(n-1)*
            HermitePolynomialPP;
        HermitePolynomialPP = HermitePolynomialP;
        HermitePolynomialP = HermitePolynomial;
    }
    return HermitePolynomial;
}
```

Listing 3.7: Computes the Hermite polynomials recursively as described in equation 2.28.

Wavefunction Setup

We previously mentioned that **WaveFunction** keeps track of the *full* wavefunction. This means that it needs both the Slater determinant of the single-particle wavefunctions and the Jastrow factor when we include particle interaction.

Since the single-particle wavefunctions are implemented in **Hamiltonian**, the **WaveFunction** superclass has only two subclasses: **ManyElectrons** and **ManyElectrons_Coefficients**. As the name suggest, the lather handles the case where we use the expansion of the wavefunction in the harmonic oscillator basis. These implementations are mostly identical with the exception of the calculations of the single-particle wavefunctions used in the Slater determinant and their derivatives.

The Slater determinant consists of the single-particle wavefunctions for different energy levels for all the particles and the **Hamiltonian** subclasses have a method `evaluateSingleParticleWF` that calculates the wavefunctions analytically. The elements of the Slater matrices in **ManyElectrons** are then calculated as seen in listing 3.8 by calling the `evaluateSingleParticleWF` (listing 3.9 shows this method for the standard harmonic oscillator). The single-particle wavefunctions of **ManyElectrons_Coefficients** on the other hand are superpositions of the harmonic oscillator basis functions and is discussed later.

```
for (int j=0; j<m_halfNumberOfParticles; j++) {
    vec n(m_numberOfDimensions);
    for (int d = 0; d < m_numberOfDimensions; d++) {
        n[d] = m_quantumNumbers(j, d);
    }
    m_SPWFMat(i,j) = m_system->getHamiltonian()->
        evaluateSingleParticleWF(n, r_i, j);
    m_SPWFDMat(i,j) = m_system->getHamiltonian()->computeSPWFDerivative(
        n, r_i, j);
    m_SPWFDDMat(i,j) = m_system->getHamiltonian()->
        computeSPWFDoubleDerivative(n, r_i, j);
}
```

Listing 3.8: Implementation of how to update the elements of the Slater matrices when the single-particle wavefunction is calculated analytically in **ManyElectrons** which is a subclass of **WaveFunction**. `i` is the index of the moved particle with corresponding position `r_i`.

```
double HarmonicOscillatorElectrons::evaluateSingleParticleWF(vec n, std
    ::vector<double> r, int j) {
    // Calculates the single particle wave function.
    double waveFunction = m_expFactor;

    for (int d = 0; d < m_numberOfDimensions; d++) {
        int nd = n[d];
        waveFunction *= m_hermitePolynomials[nd]->eval(r[d]);
    }
    return waveFunction;
}
```

Listing 3.9: An example of how the `evaluateSingleParticleWF` method is implemented in the normal harmonic oscillator potential.

Before we start the Monte-Carlo simulation, we initiate the Slater matrices that makes up the many-body part of the wavefunction. There is one (square) matrix for the spin up particles and one for spin down, but both matrices are stored in `m_SPWFMat` so that the first half (of the columns) has spin up wavefunctions and the latter half has spin down in the following manner

$$\mathbf{S} = \text{join}(\mathbf{S}^{\uparrow}, \mathbf{S}^{\downarrow}) = \begin{bmatrix} \varphi_1(\mathbf{r}_1) & \varphi_1(\mathbf{r}_2) & \dots & \varphi_1(\mathbf{r}_N) \\ \varphi_2(\mathbf{r}_1) & \varphi_2(\mathbf{r}_2) & \dots & \varphi_2(\mathbf{r}_N) \\ \vdots & \vdots & \dots & \vdots \\ \varphi_{N/2}(\mathbf{r}_1) & \varphi_{N/2}(\mathbf{r}_2) & \dots & \varphi_{N/2}(\mathbf{r}_N) \end{bmatrix} \tag{3.1}$$

Each column of the matrix stores all the single-particle wavefunctions for one particle. This matrix is *not* square.

The gradient of this matrix is a matrix of vectors and is stored in `m_SPWFDMat`,

$$\nabla\mathbf{S} = \begin{bmatrix} \nabla\varphi_1(\mathbf{r}_1) & \nabla\varphi_1(\mathbf{r}_2) & \dots & \nabla\varphi_1(\mathbf{r}_N) \\ \nabla\varphi_2(\mathbf{r}_1) & \nabla\varphi_2(\mathbf{r}_2) & \dots & \nabla\varphi_2(\mathbf{r}_N) \\ \vdots & \vdots & \dots & \vdots \\ \nabla\varphi_{N/2}(\mathbf{r}_1) & \nabla\varphi_{N/2}(\mathbf{r}_2) & \dots & \nabla\varphi_{N/2}(\mathbf{r}_N) \end{bmatrix} \tag{3.2}$$

And the Laplacian is again a matrix and is stored in `m_SPWFDDMat`

$$\nabla^2\mathbf{S} = \begin{bmatrix} \nabla^2\varphi_1(\mathbf{r}_1) & \nabla^2\varphi_1(\mathbf{r}_2) & \dots & \nabla^2\varphi_1(\mathbf{r}_N) \\ \nabla^2\varphi_2(\mathbf{r}_1) & \nabla^2\varphi_2(\mathbf{r}_2) & \dots & \nabla^2\varphi_2(\mathbf{r}_N) \\ \vdots & \vdots & \dots & \vdots \\ \nabla^2\varphi_{N/2}(\mathbf{r}_1) & \nabla^2\varphi_{N/2}(\mathbf{r}_2) & \dots & \nabla^2\varphi_{N/2}(\mathbf{r}_N) \end{bmatrix} \tag{3.3}$$

Two optimizations are also initiated at this stage by the `setUpJastrowMat` and `setUpDistances` methods. We will discuss these in more detail in chapter 4.1. The forementioned matrices are updated with the methods `updateSlaterDet`, `updateDistances`, `updateSPWFMat` and `updateJastrow` of **WaveFunction**. This is needed when we perform the Metropolis step.

The total wavefunction is calculated with the `evaluate` method, see listing 3.10. The total wavefunction is only needed when we calculate the kinetic energy with the brute-force numerical method (done in **Hamiltonian**).

```
double ManyElectrons::evaluate(std::vector<class Particle*> particles) {
    // Evaluates the wave function using brute force.

    mat spinUpSlater;
    mat spinDownSlater;
    if (m_numberOfParticles == 1) {
        spinUpSlater = zeros<mat>(m_numberOfParticles,
            m_numberOfParticles);
        spinDownSlater = zeros<mat>(m_numberOfParticles,
            m_numberOfParticles);
        spinUpSlater(0,0) = m_SPWFMat(0,0);
    }
    else {
        spinUpSlater = zeros<mat>(m_halfNumberOfParticles,
            m_halfNumberOfParticles);
        spinDownSlater = zeros<mat>(m_halfNumberOfParticles,
            m_halfNumberOfParticles);

        for (int i=0; i < m_halfNumberOfParticles; i++) {

            for (int j=0; j < m_halfNumberOfParticles; j++) {
                spinUpSlater(i,j) = m_SPWFMat(i,j);//
                    evaluateSingleParticleWF(nx, ny, xSpinUp, ySpinUp);
                spinDownSlater(i,j) = m_SPWFMat(i+
                    m_halfNumberOfParticles,j);//evaluateSingleParticleWF
                    (nx, ny, xSpinDown, ySpinDown);
            }
        }
    }

    double beta = m_parameters[1];
    double exponent = 0;
    if (m_Jastrow) {
        for (int i=0; i < m_numberOfParticles; i++) {

            for (int j=i+1; j < m_numberOfParticles; j++) {
                exponent += m_JastrowMat(i,j);
            }
        }
    }

    double waveFunction;
    if (m_numberOfParticles == 1) {
        waveFunction = det(spinUpSlater)*exp(exponent);
    }
    else {
        waveFunction = det(spinDownSlater)*det(spinUpSlater)*exp(
            exponent);
    }

    return waveFunction;
}
```

Listing 3.10: Method for evaluating the full wavefunction. The Slater determinant is split into a spin up and spin down part. The case where we only have one particle is special, as one of the Slater matrices will be empty and we calculate the product of them. The solution is to choose either the spin up or spin down case. In this code snippet we choose spin up.

The gradients of the Jastrow factor and the Slater determinant are calculated with the `computeSlaterGradient` and `computeJastrowGradient` methods. The results are then combined to get the total gradient.

The Metropolis Ratio

The Metropolis ratio is calculated as the product of the ratio of the Slater determinant part and the Jastrow factor part. These are given by the following equations. For the Slater part we have

$$R_{SD} = \sum_{j=1}^{N} \varphi_j(\mathbf{r}_i^{\text{new}}) d_{ji}^{-1}(\mathbf{r}^{\text{old}}) \tag{3.4}$$

and for the Jastrow part

$$R_C = \exp\left(\sum_{i=1, i \neq k}^{N} f_{ik}^{\text{new}} - f_{ik}^{\text{old}} \right) \tag{3.5}$$

The quantities we need to calculate these ratios are stored in the matrices we discussed above, where φ are the single-particle wavefunctions, d_{ji}^{-1} are the elements of the inverse Slater matrix and

$$f_{ij} = \frac{ar_{ij}}{1 + \beta r_{ij}} \tag{3.6}$$

We then only need a few loops to calculate the Metropolis ratio, see listing 3.11.

```
double ManyElectrons::computeMetropolisRatio(std::vector<Particle *>
   particles, int currentParticle, std::vector<double> positionChange) {
    // Function for calculating the wave function part of the Metropolis
        ratio
    // both the Slater part and the Jastrow part.
    m_distancesOld = m_distances;
    m_JastrowMatOld = m_JastrowMat;
    for (int i=0; i<m_numberOfDimensions; i++) {
        particles[currentParticle]->adjustPosition(positionChange[i], i)
            ;
    }
    m_system->getWaveFunction()->updateDistances(currentParticle);
    m_system->getWaveFunction()->updateSPWFMat(currentParticle);
    m_system->getWaveFunction()->updateJastrow(currentParticle);

    int i = currentParticle;
    double ratioSlaterDet = 0;

    if (i < m_halfNumberOfParticles) {
        for (int j=0; j < m_halfNumberOfParticles; j++) {
            ratioSlaterDet += m_spinUpSlaterInverse(j,i)*m_SPWFMat(i,j);
        }
    }
    else {
        for (int j=0; j < m_halfNumberOfParticles; j++) {
            ratioSlaterDet += m_spinDownSlaterInverse(j, i-
                m_halfNumberOfParticles)*m_SPWFMat(i,j);
        }
    }
    double exponent = 0;

    if (m_Jastrow) {
        for (int j=0; j < i; j++) {
            exponent += m_JastrowMat(i,j);
            exponent -= m_JastrowMatOld(i,j);
        }
        for (int j=i+1; j < m_numberOfParticles; j++) {
            exponent += m_JastrowMat(i,j);
            exponent -= m_JastrowMatOld(i,j);
        }
    }
    double ratioJastrowFactor = exp(exponent);
    m_ratioSlaterDet = ratioSlaterDet;
    m_metropolisRatio = ratioSlaterDet*ratioJastrowFactor;

    return m_metropolisRatio;
}
```

Listing 3.11:
Method for calculating the Metropolis ratio. `m_spinUpSlaterInverse` and `m_spinDownSlaterInverse` are the inverses of the Slater matrices. `m_JastrowMat` and `m_JastrowMatOld` are the exponent of the Jastrow factor for the current trial state and the last accepted state. The loop runs over half the number of particles, as the spin matrices are split into two so the ratio only needs to sum over the particles with the same spin as the moved particle. The Metropolis ratio is calculated as the product of the Slater determinant and Jastrow factor parts.

The Inverse Slater Elements

To efficiently update the Slater determinant (and the gradient and Laplacian) we also need to update the inverse as explained in chapter 2.5. Each element is calculated using the old elements of the inverse Slater matrices, the current Slater matrices and the SD part of the Metropolis ratio. The code is implemented as seen in listing 3.12. We need to check if particle `i` in the current Metropolis step has spin up or down. Then we cut the number of calculations by splitting the sum over particles $j \neq i$ into two loops. The last loop calculates the element d_{ki}^{-1}.

```
int i = currentParticle;

if (i < m_halfNumberOfParticles) {
    mat spinUpSlaterInverseOld = m_spinUpSlaterInverse;

    for (int j=0; j < i; j++) {
        double sum = 0;

        for (int l=0; l <m_halfNumberOfParticles; l++) {
            sum += m_SPWFMat(i,l)*spinUpSlaterInverseOld(l,j);
        }
        for (int k=0; k < m_halfNumberOfParticles; k++) {
            m_spinUpSlaterInverse(k,j) = spinUpSlaterInverseOld(k,j)-(
                sum/m_ratioSlaterDet)*spinUpSlaterInverseOld(k,i);
        }
    }
    for (int j=i+1; j < m_halfNumberOfParticles; j++) {
        double sum = 0;

        for (int l=0; l <m_halfNumberOfParticles; l++) {
            sum += m_SPWFMat(i,l)*spinUpSlaterInverseOld(l,j);
        }
        for (int k=0; k < m_halfNumberOfParticles; k++) {
            m_spinUpSlaterInverse(k,j) = spinUpSlaterInverseOld(k,j)-(
                sum/m_ratioSlaterDet)*spinUpSlaterInverseOld(k,i);
        }
    }
    for (int k=0; k < m_halfNumberOfParticles; k++) {
        m_spinUpSlaterInverse(k,i) = spinUpSlaterInverseOld(k,i)/
            m_ratioSlaterDet;
    }
}
```

Listing 3.12: Loop to calculate the inverse Slater matrix elements.

Single-Particle Wavefunctions in the Harmonic Oscillator Basis

How we approximate the single-particle wavefunctions of a general potential using the harmonic oscillator functions as basis functions was discussed in chapter 2.2.2. We use the approximation in equation 2.23 where the overlap coefficients are calculated as in equation 2.24. This is used to compute the matrix elements of the Slater determinant in the **ManyElectrons_Coefficients** subclass of **WaveFunction**. Now the single-particle wavefunctions are sums over the coefficients and the harmonic oscillator basis functions. These coefficients are computed before the Monte-Carlo simulation is run using the diagonalization program, see chapter 3.4. The loop for performing the updates to the Slater matrices are seen in in listing 3.13.

```
for (int j=0; j<m_halfNumberOfParticles; j++) {
    vec n(m_numberOfDimensions);
    for (int d = 0; d < m_numberOfDimensions; d++) {
        n[d] = m_quantumNumbersPrime(j, d);
    }

    m_SPWFMat(i,j) = 0;
    m_SPWFDMat(i,j) = zeros(m_SPWFDMat(i,j).size());
    m_SPWFDDMat(i,j) = 0;

    for (int eig = 0; eig < m_numberOfCoeffEigstates; eig++) {
        double term = 1;
        vec termD(m_numberOfDimensions);
        double termDD = 0;

        double coefficients = 1;
        vec qNums = conv_to<vec>::from(m_quantumNumbers.row(eig));

        for (int d = 0; d < m_numberOfDimensions; d++) {
            term *= harmonicOscillatorBasis(r_i[d], qNums[d], d);
            termD[d] = harmonicOscillatorBasisDerivative(r_i, qNums, d);
            termDD += harmonicOscillatorBasisDoubleDerivative(r_i, qNums
                , d);

            coefficients *= m_cCoefficients(qNums[d], n[d], d);
        }
        m_SPWFMat(i,j) += coefficients*term;
        m_SPWFDMat(i,j) += coefficients*termD;
        m_SPWFDDMat(i,j) += coefficients*termDD;
    }
}
```

Listing 3.13: As each element of the Slater determinant is a sum over harmonic oscillator basis functions, the process of updating the various Slater matrices requires an extra loop compared to listing 3.8. `i` is the index of the moved particle with corresponding position `r_i`. The quantum numbers work as indices for the Slater matrix in both cases, but when we use the harmonic oscillator basis, they require their own set of quantum numbers.

3.3.4 Steepest Descent

In chapter 2.6 we discussed the implementation of the variational method steepest descent. After all the initial values have been set for the simulation, we perform a short-cycled Monte-Carlo simulation for each iteration of the steepest descent method. The loop breaks when we have reached some tolerance or when a preset number of iterations is reached. We sample the expectation values required in equation 2.135 to find the derivative with respect to the variational parameter we are optimizing. Then we calculate the α_{i+1} parameter and repeat. The derivatives are calculated using the `computeDerivativeWrtParameters` method of **WaveFunction**.

It's important to notice that when the simulation is run in parallel, we have to take the average of the derivatives over all nodes so that the next variational parameter is the same for all nodes. The whole method can be seen in listing 3.14

```cpp
void SteepestDescent::obtainOptimalParameter(std::vector<double>
    parameters, double tol, int maxIterations,
    // Count iterations so we can force quit after a given number max
        iterations
    int iteration = 0;
    int numberOfParameters = parameters.size();
    double diff = 0;
    std::vector<double> parametersNew = parameters;
    int my_rank = m_system->getMyRank();

    for (int i=0; i < numberOfParameters; i++) {
        m_derivativeAvg.push_back(0);
    }

    do{

        // Set up an initial state with the updated parameters
        m_system->getInitialState()->setupInitialState();
        for (int i=0; i < numberOfParameters; i++) {
            m_system->getWaveFunction()->adjustParameter(parameters[i],
                i);
        }
        m_system->getHamiltonian()->setAlpha(parameters[0]);
        //m_system->getHamiltonian()->setUpHermitePolynomials();

        // Run Monte Carlo simulation to find expectation values
        m_system->runMetropolisSteps(numberOfMetropolisSteps,
            importanceSampling, false, false);

        //derivative of local energy
        std::vector<double> derivative(numberOfParameters);

        // Expectation values needed to calculate derivative of local
            energy:
        double energy = m_system->getSampler()->getEnergy();
        std::vector<double> waveFuncEnergy(numberOfParameters);
        std::vector<double> waveFuncDerivative(numberOfParameters);

        for (int i=0; i < numberOfParameters; i++) {
            waveFuncEnergy[i] = m_system->getSampler()->
                getWaveFuncEnergyParameters()[i];
            waveFuncDerivative[i] = m_system->getSampler()->
                getWaveFuncDerivativeParameters()[i];
            derivative[i] = 2*(waveFuncEnergy[i] - energy*
                waveFuncDerivative[i]);
        }

        for (int i=0; i < numberOfParameters; i++) {
            MPI_Reduce(&derivative[i], &m_derivativeAvg[i], 1,
                MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
            if (my_rank==0) {
                derivative[i] = m_derivativeAvg[i]/m_system->getNumProcs
                    ();
            }
            MPI_Bcast(&derivative[i], 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
        }

        // Find new parameters
        diff = 0;
        for (int i=0; i < numberOfParameters; i++) {
            parametersNew[i] = parameters[i] - derivative[i]*
                m_stepLengthSD;
            diff += abs(parametersNew[i] - parameters[i]);
        }
        //m_stepLengthSD *= 0.8;

        parameters = parametersNew;    // Update parameters
        iteration++;
        std::string upLine = "\e[A";

        if (my_rank == 0) {
            cout << "Iterations: " << iteration << endl;
            for (int i=0; i < numberOfParameters; i++) {
                cout << "Parameter " << i+1 << ": " << parameters[i] <<
                    endl;
                upLine += "\e[A";
            }
            cout << upLine;
        }

    }while(diff > tol && iteration < maxIterations);
    // Loop ends when requested tolerance for optimal parameters has
        been reached or after max iterations.
```

Listing 3.14: The steepest descent method. Optimizes the variational parameters of the MC simulation with a small number of cycles prior to the actual full-scale simulation.

3.3.5 Statistical Analysis with Blocking

The blocking method was discussed in chapter 2.7. It is used to determine the statistical error of the Monte-Carlo simulation and is implemented as a Python program. The reason for this is that it's not a very computationally heavy procedure and the data can be directly plotted with the *Matplotlib* library.

We iterate over different sized blocks ranging from `minBlockSize` to `maxBlockSize` with increments of `deltaBlockSize`. For every block, we calculate the standard deviation of the energy. These values are used to create a plot of the standard deviation as a function of block size, see figure 3.1. The plateau seen in the figure describes the situation where we have a large enough block size such that the sampled energies are uncorrelated as described in chapter 2.7.

```
import numpy as np
import matplotlib.pyplot as plt
import sys

def readData(filename):

    infile = open("%s" %filename, 'r')
    energies = []
```

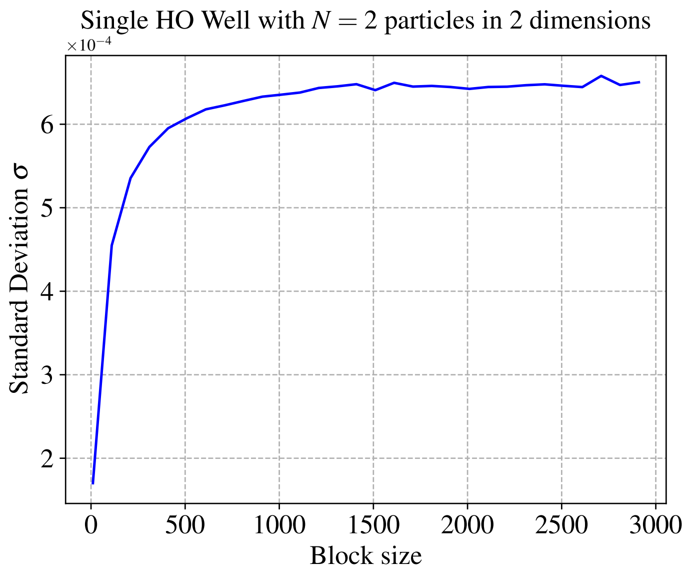


Figure 3.1: Example of blocking of the ground state energies of a MC simulation of the single harmonic oscillator with $N = 2$ particles in two dimensions. The VMC calculation was done with 1e7 MC cycles. We see that the standard deviation as function of block size reaches a plateau.

```

    for line in infile:
        energies.append(float(line))

    infile.close()
    return np.asarray(energies)

def blocking(energies, nBlocks, blockSize):

    meansOfBlocks = np.zeros(nBlocks)

    for i in range(nBlocks):
        energiesOfBlock = energies[i*blockSize:(i+1)*blockSize]
        meansOfBlocks[i] = sum(energiesOfBlock)/blockSize

    mean = sum(meansOfBlocks)/nBlocks
    mean2 = sum(meansOfBlocks**2)/nBlocks
    variance = mean2 - mean**2

    return mean, variance

if __name__ == "__main__":
    N = int(sys.argv[1])
    energies = readData(sys.argv[3])

    deltaBlockSize = 100
    minBlockSize = 10
    maxBlockSize = int(sys.argv[2])
    numberOfSizes = (maxBlockSize-minBlockSize)/deltaBlockSize + 1
    largestBlockSize = minBlockSize + (numberOfSizes-1)*deltaBlockSize #
```

```python
        9910

    #blockSizes = np.zeros(numberOfSizes)
    blockSizes = np.linspace(minBlockSize, largestBlockSize,
        numberOfSizes).astype(int)
    blockAmounts = len(energies)/blockSizes#np.zeros(numberOfSizes)
    means = np.zeros(numberOfSizes)
    variances = np.zeros(numberOfSizes)

    for i in range(numberOfSizes):
        #blockSize = minBlockSize + i*deltaBlockSize
        #blockAmount = len(energies)/blockSize
        #mean, variance = blocking(energies, blockAmount, blockSize)
        mean, variance = blocking(energies, blockAmounts[i], blockSizes[
            i])
        means[i] = mean
        variances[i] = variance

    standardDeviation = np.sqrt(abs(variances)/(blockAmounts-1.))

    font = {'family' : 'serif',
            'size'   : 15}

    plt.rc('font',**font)

    fig, ax = plt.subplots()

    ax.plot(blockSizes, standardDeviation,'b')

    ax.set_ylabel(r"Standard Deviation $\sigma$")
    ax.set_xlabel(r'Block size')

    ax.grid('on')
    ax.spines['right'].set_visible(False)
    ax.spines['top'].set_visible(False)
    plt.savefig('/home/alexanfl/uio/masterthesis/doc/part2/figures/
        blocking-example.eps')
    plt.show()
```

Listing 3.15: Blocking program implemented in Python. The program takes the number of particles, the maximum block size and the energy data file as command-line arguments.

3.4 Expansion in the Harmonic-Oscillator Basis

In chapter 2.2 we described a method for diagonalizing the one-particle problem for some potential. The goal was to obtain a set of discrete eigenfunctions for the given potential well, then expand these functions from the harmonic-oscillator basis functions.

The eigenvalue problem was set up and discretized as a tridiagonal matrix. We use an external library called *Armadillo*[30] to solve the eigenvalue problem. The

Armadillo function `eig_sym` returns the eigenfunctions and eigenvalues. Further we use the newly found eigenfunctions to calculate the *overlap coefficients*, which are the result of the innerproduct between the eigenfunctions and the harmonic-oscillator basis functions. These coefficients are then used to expand the eigenfunctions in the harmonic-oscillator basis.

The program that solves the problem at hand has a similar, but smaller structure as the forementioned VMC solver. Again, we have a **System** class responsible for handling the main mathods of the physical system. In this case, it is responsible for the diagonalization and solving the eigenvalue problem, as well as calculating the overlap coefficients and expanding the eigenfunctions in the harmonic-oscillator basis. Furthermore, each of the potentials modelled in this thesis are objects (subclasses) of the **WaveFunction** class. This is similar to the implementation in the VMC solver.

3.4.1 Details on the Diagonalization

The tridiagonal matrix eigenvalue problem described above is on the form of equation 2.22. Each type of potential will give a new set of matrix elements, and these are calculated by the corresponding subclass of **WaveFunction**. The matrix is set up by the `diagonalizeMatrix` method, see listing 3.16.

```
void System::diagonalizeMatrix(mat r, vec L, int N, cube &diagMat) {
    double Constant = 1./(m_h*m_h);
    mat V(N+1, m_numberOfDimensions);
    for (int d = 0; d < m_numberOfDimensions; d++) {
        V.col(d) = m_waveFunction->potential(r.col(d), L(d));
        //Set d_i elements of the matrix:
        diagMat.slice(d).diag(0) =  2*Constant + V.col(d).subvec(1,N-1);
        //Set e_i elements of the matrix:
        diagMat.slice(d).diag(1) = -Constant*ones(N-2);
        //Set e_i elements of the matrix:
        diagMat.slice(d).diag(-1)= diagMat.slice(d).diag(1);
    }
    return;
}
```

Listing 3.16: Setting up the tridiagonal matrix elements.

As described in chapter 2.2, the problem is solved for each dimensions separately, as we assume a spatially separable wavefunction. Therefore we must solve as many eigenproblems as dimensions, using `eig_sym`.

3.4.2 Details on Finding the Overlap Coefficients

The overlap coefficients are given by equation 2.24. This equation is implemented in the `findCoefficients` method of the **System** class (listing 3.17).

As equation 2.24 only gives a single overlap coefficient, we must loop over all n' as well as n. This will result in an overlap matrix for *one* dimension. Thus `findCoefficients` is called once for each dimension of the system.

```cpp
void System::findCoefficients(int nMax, int nPrimeMax, vec x, mat &C,
    int currentDim){
    cout << "Finding coefficients for dimension " << currentDim+1 << "
        of " << m_numberOfDimensions << endl;
    cout.flush();
    std::string upLine = "\033[F";
    for (int nPrime = 0; nPrime < nPrimeMax; nPrime++) {
        cout << "nPrime = " << nPrime << " of " << nPrimeMax-1 << endl;
        for (int nx = 0; nx < nMax; nx++) {
            cout << "[" << int(double(nx)/nMax * 100.0) << " %]\r";
            cout.flush();
            double innerprod = 0;
            for (int i = 0; i < m_N-1; i++) {
                innerprod += m_psi.slice(currentDim).col(nPrime)(i)*
                    m_waveFunction->harmonicOscillatorBasis(x, nx)(i);
            }
            C(nx, nPrime) = innerprod;
        }
        cout << upLine;
    }
    cout << upLine;
    C *= m_h;
}
```

Listing 3.17: The calculation of the overlap coefficient matrix. This functions is called once for every dimension and the result is stored in a three-dimensional array and passed to the VMC solver where it is used to expand the wavefunction in the harmonic-oscillator basis.

4 Optimization

An important part of this thesis has been to optimize the implementations of the methods we use. In this chapter we introduce three major optimization schemes that greatly reduced the computation time of the simulations.

The Monte-Carlo simulation can estimate the ground state energy of the system with good precision. However as the number of particles increase, we must decide between increased precision and a lower computation time. A first approach to an optimization is to identify quantities that are being calculated again and again. Finding a good way to store these values and reuse them is key in bringing down the number of *floating point operations per second (FLOPS)*.

Polymorphism in computer science is the idea that we have a general object that when instantiated will solve a specific task. In this thesis we have used polymorphism to remove the need for conditional statements when calculating Hermite polynomials.

As the Quantum Monte-Carlo methods are statistical (identical) experiments to handle large, multi-dimensional integrals, computations are apt for parallelization. In this thesis we use the *Open MPI* library[2] to compute each Monte-Carlo cycle in parallel.

4.1 Reuse of Calculated Quantities

4.1.1 Relative Distances Between Particles

The relative distance between two particles i and j are given by $r_{ij} = |\mathbf{r}_j - \mathbf{r}_i|$. This expression is involved in many of our calculations, mainly in the Jastrow factor and in the Coulomb electron-electron interaction. Instead of calculating the same expression every time we encounter it, we store it in a *relative-distances matrix*,

$$\mathbf{r} = \begin{bmatrix} 0 & r_{12} & \dots & r_{1N} \\ r_{21} & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ r_{N1} & \dots & \dots & 0 \end{bmatrix} \tag{4.1}$$

which is symmetric, $\mathbf{r} = \mathbf{r}^T$, since $r_{ij} = r_{ji}$. This symmetry means that when a particle moves, we only need to calculate $N-1$ distances on the upper or lower half and set the symmetric values equal to them.

In listing 4.1 we see the implementation of this optimization. For each particle that is moved, we call the method `updateDistances`.

```cpp

void ManyElectrons::updateDistances(int currentParticle) {
    /*
     * Function for updating the distances between electrons.
     */
    int i = currentParticle;
    std::vector<double> r_i = m_system->getParticles()[i]->getPosition()
       ;

    for (int j=0; j<i; j++) {
        std::vector<double> r_j = m_system->getParticles()[j]->
            getPosition();
        double r_ij = 0;

        for (int d = 0; d < m_numberOfDimensions; d++) {
            r_ij += (r_i[d]-r_j[d])*(r_i[d]-r_j[d]);
        }
        m_distances(i,j) = m_distances(j,i) = sqrt(r_ij);
    }

    for (int j=i+1; j<m_numberOfParticles; j++) {
        std::vector<double> r_j = m_system->getParticles()[j]->
            getPosition();
        double r_ij = 0;

        for (int d = 0; d < m_numberOfDimensions; d++) {
            r_ij += (r_i[d]-r_j[d])*(r_i[d]-r_j[d]);
        }
        m_distances(i,j) = m_distances(j,i) = sqrt(r_ij);
    }
}
```

Listing 4.1: The method that calculates the $N-1$ distances that change when a particle i is moved. The values on the opposite side of the diagonal is then set to the calculated value. The diagonal is skipped since it is always zero.

4.1.2 Single-Particle Wavefunctions

Another quantity frequently used in our calculations are the single-particle wavefunctions $\varphi_j(\mathbf{r}_i)$. In chapter 3.3.3 we explained how the concatenated spin matrix (equation 3.1) and its gradient and Laplacian (equations 3.2 and 3.3) were stored in the matrices `m_SPWFMat`, `m_DSPWFMat` and `m_DDSPWFMat`.

Since only the i'th column of these matrices are changed when particle i is moved, they were implemented as another optimization.

4.1.3 The Jastrow Factor

The Slater determinant is only one part of the product that make up the trial wavefunction, the other being the Jastrow factor Ψ_C. In this thesis, the Jastrow factor takes the form of equation 2.113. We have already optimized part of this expression using the relative distances matrix, but we can further improve it by also creating a matrix with the full expression in the sum of the Jastrow factor,

$$\frac{ar_{ij}}{1+\beta r_{ij}} \tag{4.2}$$

Each element ij of this Jastrow matrix will now be the fraction of the elements in the relative distances matrix. The matrix is symmetric.

As in the case of the Slater determinant part, we also need the gradient and Laplacian of the Jastrow factor. In chapter 2.4 we described how we needed the ratio between the gradient of the trial wavefunction and the trial wavefunction. The expression for the Jastrow factor was

$$\frac{\nabla_i \Psi_C}{\Psi_C} = \sum_{j\neq i} \frac{\mathbf{r}_{ij}}{r_{ij}} \frac{a}{(1+\beta r_{ij})^2} \tag{4.3}$$

The expression inside this sum is all we need to save time in calculating the ratios involving the gradient and Laplacian. Therefore, we create a matrix $d\mathbf{J}$ where each element should be defined as follows

$$d\mathbf{J}_{ij} = -d\mathbf{J}_{ji} \equiv \frac{\mathbf{r}_{ij}}{r_{ij}} \frac{a}{(1+\beta r_{ij})^2} \tag{4.4}$$

The matrix is antisymmetric $d\mathbf{J} = -d\mathbf{J}^T$, as there is only a sign change of the position vector $\mathbf{r}_{ij}$,

$$d\mathbf{J} = \begin{bmatrix} 0 & d\mathbf{J}_{12} & \dots & d\mathbf{J}_{1N} \\ -d\mathbf{J}_{21} & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ -d\mathbf{J}_{N1} & \dots & \dots & 0 \end{bmatrix} \tag{4.5}$$

The elements of this matrix are vectors with length equal to the number of spatial dimensions.

After a step in the Metropolis algorithm is performed, the new ratio of equation 4.3 is given by the sum over the elements of the matrix,

$$\frac{\nabla_i \Psi_C^{\text{new}}}{\Psi_C^{\text{new}}} = \sum_{j \neq i} d\mathbf{J}_{ij}^{\text{new}} \tag{4.6}$$

When a particle p is moved, one column and one row of $d\mathbf{J}$ changes, and equation 4.6 simplifies to

$$\frac{\nabla_{i \neq p} \Psi_C^{\text{new}}}{\Psi_C^{\text{new}}} = \sum_{j \neq i,p} d\mathbf{J}_{ij}^{\text{old}} + d\mathbf{J}_{ip}^{\text{new}} \tag{4.7}$$

$$\tag{4.8}$$

Thus, when this particle is *not* the same as particle k, we have the following expression

$$\begin{aligned}
\frac{\nabla_{i \neq p} \Psi_C^{\text{new}}}{\Psi_C^{\text{new}}} &= \sum_{j \neq i,p} d\mathbf{J}_{ij}^{\text{old}} + d\mathbf{J}_{ip}^{\text{new}} && (4.9) \\
&= \sum_{j \neq i,p} d\mathbf{J}_{ij}^{\text{old}} + d\mathbf{J}_{ip}^{\text{old}} - d\mathbf{J}_{ip}^{\text{old}} + d\mathbf{J}_{ip}^{\text{new}} && (4.10) \\
&= \frac{\nabla_{i \neq p} \Psi_C^{\text{old}}}{\Psi_C^{\text{old}}} - d\mathbf{J}_{ip}^{\text{old}} + d\mathbf{J}_{ip}^{\text{new}} && (4.11) \\
& && (4.12)
\end{aligned}$$

and we see that the new ratio is given by the old ratio and adding the two elements, $d\mathbf{J}_{ip}^{\text{new}} - d\mathbf{J}_{ip}^{\text{old}}$. When particle $i = p$, we must calculate the whole sum in 4.6.

The implementation of this optimization is done in the `updateJastrow` method, listing 4.2.

```cpp
void ManyElectrons::updateJastrow(int currentParticle) {

    int p = currentParticle;
    std::vector<double> r_p = m_system->getParticles()[p]->getPosition()
        ;
    double beta = m_parameters[1];
    m_dJastrowMatOld = m_dJastrowMat;

    for (int j=0; j<p; j++) {
        std::vector<double> r_j = m_system->getParticles()[j]->
            getPosition();
        double r_pj = m_distances(p,j);
        double denom = 1. + beta*r_pj;

        m_JastrowMat(p,j) = m_a(p,j)*r_pj / denom;
        m_JastrowMat(j,p) = m_JastrowMat(p,j);

        for (int d = 0; d < m_numberOfDimensions; d++) {
            m_dJastrowMat(p,j,d) = (r_p[d]-r_j[d])/r_pj * m_a(p, j)/(
                denom*denom);
            m_dJastrowMat(j,p,d) = -m_dJastrowMat(p,j,d);
        }
    }
    for (int j=p+1; j<m_numberOfParticles; j++) {
        std::vector<double> r_j = m_system->getParticles()[j]->
            getPosition();
        double r_pj = m_distances(p,j);
        double denom = 1. + beta*r_pj;

        m_JastrowMat(p,j) = m_a(p,j)*r_pj / denom;
        m_JastrowMat(j,p) = m_JastrowMat(p,j);

        for (int d = 0; d < m_numberOfDimensions; d++) {
            m_dJastrowMat(p,j,d) = (r_p[d]-r_j[d])/r_pj * m_a(p, j)/(
                denom*denom);
            m_dJastrowMat(j,p,d) = -m_dJastrowMat(p,j,d);
        }
    }

    m_JastrowGradOld = m_JastrowGrad;

    for (int d = 0; d < m_numberOfDimensions; d++) {
        m_JastrowGrad(p, d) = 0;

        for (int j=0; j<p; j++) {
            m_JastrowGrad(p, d) += m_dJastrowMat(p,j,d);
        }
        for (int j=p+1; j<m_numberOfParticles; j++) {
            m_JastrowGrad(p, d) += m_dJastrowMat(p,j,d);
        }
        for (int i=0; i<p; i++) {
            m_JastrowGrad(i, d) = m_JastrowGradOld(i,d) -
                m_dJastrowMatOld(i,p,d) + m_dJastrowMat(i,p,d);
        }
        for (int i=p+1; i<m_numberOfParticles; i++) {
            m_JastrowGrad(i, d) = m_JastrowGradOld(i,d) -
                m_dJastrowMatOld(i,p,d) + m_dJastrowMat(i,p,d);
        }
    }
}
```

Listing 4.2: The Method that updates the Jastrow matrices. Particle p is moved corresponds to a change of the p index of the matrices `m_JastrowMat` and `m_dJastrowMat`. All elements of the matrix `m_JastrowMat` must be changed when p is moved, but for the indices $i \neq p$, the optimization discussed is implemented. The diagonals of all matrices are skipped since it is always zero.

4.2 Hermite Polynomial Optimization

In chapter 3.3.3 we discussed the use of the virtual functions and subclasses of the **Hamiltonian** and **WaveFunction** classes. This is a type of polymorphism. We further discussed the different ways of implementing the Hermite polynomial calculations, and in this section we will discuss an optimization scheme based on this.

The **HermitePolynomials** class is a super class. Each degree of the Hermite polynomials has a sub class for the polynomial and its first and second derivative, each with a `eval` method. The `eval` method takes one argument, a position x, and returns the value that current subclass is responsible for. For example the subclass `HermitePolynomial_0` returns the zero degree Hermite polynomial for the given `x` position. The first derivative of this polynomial with position `x` is returned by the subclass `dell_HermitePolynomial_0`.

The quantum numbers, n_x, n_y, n_z determine which Hermite polynomial we are using. By representing each Hermite polynomial as an object of type `HermitePolynomials`, we can list them in ascending order in a list. This is implemented in our code as an array, `m_hermitePolynomials`, as a member of the **Hamiltonian** superclass. Then the desired polynomial is retrieved by inputing the index `nx` and the position `x` as follows

```
    double hermitePolynomial = m_hermitePolynomials[nx]->eval(x);
```

This also means that we must hardcode the desired number of Hermite polynomials and their derivatives. We have currently implemented 50 Hermite polynomials, which means 150 subclasses. The Hermite polynomials are easily calculated, it only takes time to do it by hand. Luckily, we can auto-generate the expressions for a desired number of polynomials with the Python package *SymPy*. It has the option to export the expressions directly as C++ code, which we can implement directly in our source code as subclasses of **HermitePolynomials**.

However, to make this optimization outperform the recursive method we must consider one more thing. Already at low degree Hermite polynomial the number of power expressions we must calculate is high. The eight order polynomial takes the form

$$H_8(x) = 256x^8 - 3584x^6 + 13440x^4 - 13440x^2 + 1680 \tag{4.13}$$

which can be implemented in our code as

```
  H_8 = 256.*pow(x, 8) - 3584.*pow(x, 6) + 13440.*pow(x, 4) - 13440.*
      pow(x, 2) + 1680;
```

In C++, the `pow` function is not very optimized by default. A way to circumvent this is by writing out the expression explicitly, for example x^4 can be written as `x8 = x*x*x*x*x*x*x*x;`. To further optimize, we should split the computation into pieces

```
    x2 = x*x;
    x4 = x2*x2
    x8 = x4*x4
```

so the number of FLOPS is reduced from seven to three. Though faster, we lose some precision, due to round-off error. It is not feasible to this manually for every Hermite polynomial, but using the compiler flag `ffast-math` in the `GCC` compiler will implement the optimization automatically to the `pow` function. Implementing this means we must take extra care in unit testing the Hermite polynomials so that round-off errors won't affect the results.

4.3 Parallelization

This optimization is perhaps the easiest of the ones implemented in this thesis. It is due to the nature of the Quantum Monte-Carlo methods. As each Monte-Carlo cycle is basically an independent experiment given a starting condition, we can divide the total number of cycles on a set of n nodes. Then the simulation run on each node is its own Monte-Carlo simulation, with a random initial configuration. It is a trivial parallelization as they nodes only need to communicate at the start and at the end when we average the result over all the nodes. This also means that the parallelization here scales approximately linearly. That is, given n nodes, we can expect the simulation to take $1/n$ the time of a single-node computation. The only time loss is due to the extra operations we must do to initiate the parallel simulation. To achieve meaningful results, the number of MC cycles on each simulation on a node must be large enough too support itself. If a simulation has too few cycles, it will ruin the simulation.

One of the things we are interested in studying in this thesis is the variatonal parameters $\boldsymbol{\alpha}$. The methods to find optimal variational parameters have not been parallelized. However, we run the variational methods on small systems so we already know the optimal parameters when we run larger simulations. Implementing parallelization of the variational methods would require mid-simulation communication between nodes.

In a modern personal computer we would typically find 4, 8 or 16 processors. As the implementation of the parallelization is rather simple, simulation of small systems that are feasible to run on a personal computer should always be run in parallel though the number of processors are small. A large-scale system should always be run in parallel on a super cluster.

5 Code Verification

In modern programming, *Test Driven Development (TDD)* is key. All programs have a task and testing is an important tool. This especially important in computational science. Physicists must be careful when setting up an experiment so that as few sources of error as possible are present. Therefore computer programs written for scientific purposes should be thoroughly tested to ensure that we can rely on the result. A large computational simulation has many variables that must be tested in order to verify the results. It is therefore important to have good analytical and theoretical results that we can test the computational results against. In this chapter we present the tests implemented in the various parts of the Monte-Carlo simulations. This includes unit tests and tests to verify parts of the results of the simulations. For the later, we focus mainly on the amount of harmonic oscillator basisfunctions needed to obtain a good ground state energy in the Variational Monte-Carlo simulations.

5.1 The Harmonic Oscillator

We will first go through the different ways we have tested and verified the results of the single and double harmonic oscillator potentials. This includes everything from the diagonalizaton of the Scrödinger equation to the VMC simulations with and without the harmonic oscillator basisfunctions. The first and most simple way we can test the VMC implementation is to compare the results to analytical solutions. These solutions are known for the harmonic oscillator well potentials in this thesis, in the non-interacting case.

5.1.1 Single Well

For a single harmonic oscillator potential, the single-particle wavefunctions of the Slater determinant is of course itself a HO basis function. Thus, the number of basisfunctions needed to obtain a perfect solution is the same as the sum of the number of basisfunctions that exists in the needed energy levels.

For one or two particles, we can fit them on the first energy level, where one particle has spin up and one has spin down. Therefore we only need *one* basisfunction, φ_{00}. In two dimensions, we need six particles to fill up the second energy level. Here there are two available solutions, φ_{10} and φ_{01}. We then put two particles on the first energy level and four on the second. The number of basisfunctions are then *three*.

We have listed the results with the corresponding analytical solution for two and three dimensions in table 5.1. The energy eigenvalue results from the diagonalization of the Schrödinger equation was also verified to the expected for each energy level before we used the basisfunctions in the VMC calculations. The diagonalization solution only gives the energy eigenvalues for the energy in each dimension (E_x, E_y, E_z) for each energy level and not the total ground state energy for the system. We can verify the total energy by summing the parts that should make up the lowest total energy.

# of Dimensions	# of Particles	E	E_{VMC}	# of Energy Levels	# of Basis Functions
2	1	1ω	1	1	1
	2	2ω	2	1	1
	6	10ω	10	2	3
	12	28ω	28	3	6
3	1	1.5ω	1.5	1	1
	2	3ω	3	1	1
	8	18ω	18	2	4
	20	60ω	60	3	10

Table 5.1: The results of VMC simulations with harmonic oscillator basis functions as the single-particle wavefunctions in the non-interacting closed-shell systems. The oscillator frequency was set to $\omega = 1$ for all simulations.

5.1.2 Double Well

In the double harmonic oscillator potential, there is a barrier between two wells that are equivalent to the single harmonic oscillator. The barrier is placed according to equation 2.5, with the minimum of the potentials at L_j where j is the the spatial dimensions we set the well. If the barrier is large enough, that is the potential for a particle to go from one side to the other is small, the situation is similar to having two single harmonic oscillators. Furthermore we will then have a situation where we can have two sets of particles that are in identical states of spin up and down, one on each side of the barrier. We assume this situation when we test our code.

This motivates the use of single-particle wavefunctions that are super positions of states in either side of the barrier[16]. Using these single-particle wavefunctions in our VMC simulation will give us a numerical benchmark for comparing our harmonic oscillator basis solution generated from the diagonalization. We will also be able to compare the results to the energies from the diagonalization solution. The resulting energy eigenvalues of the diagonalization (in two dimensions) can be seen in table 5.2. Here the energies in the x and y direction represent a well with a barrier in the x dimension. This is equivalent to two single harmonic oscillators. If we added another barrier in the y direction, we have four independent harmonic oscillators.

To fill up the first shell of a double harmonic oscillator with a barrier in the x direction, $L_x = 4$ and $L_y = 0$, we see that there are two combinations with total energy equal to 1, $E_{x,0} + E_{y,0} = 1$ and $E_{x,1} + E_{y,0} = 1$. Since a particle with a given energy eigenvalue can be either spin up or down, we have $E_{tot} = 4$, for four particles. Similarly there are eight combinations which give $E_x + E_y = 2$, the second lowest energy eigenvalue. The third level also has eight combinations with $E_x + E_y = 3$.

It could also be useful to mention that when we generate basisfunctions, the minimum amount of energy levels needed are given by equation 2.33. This means that the increase in computational time is quite large when we go from one to two to three dimensions in terms of generating basisfunctions

L_x	L_y	E_x^0	E_y^0	E_x^{diag}	E_y^{diag}
4	0	0.5	0.5	0.50000	0.50000
		0.5	1.5	0.50000	1.50000
		1.5	2.5	1.49999	2.50000
		1.5	3.5	1.50001	3.50000
		2.5	4.5	2.49989	4.49999
		2.5	5.5		

Table 5.2: When solving the two-dimensional single-particle problem by diagonalization in a double well, the resulting energy eigenvalues are degenerated in the dimension where we have a potential barrier. The first energies are the theoretical ground state energies in each state, while the second are the results from our diagonalization with 10^4 steps. The oscillator frequency was set to $\omega = 1$ for all simulations.

The next thing to verify is that the single-particle wavefunctions made from the harmonic oscillator basis we use as the Slater determinant part of the VMC calculations give a consistent result. In table 5.3 we present the results for the two and three dimensional cases. We see that the number of basisfunctions are rather high.

# of Dimensions	# of Particles	E	E_{VMC}	# of Energy Levels	# of Basis Functions
2	2	2ω	1.9996	23	276
	4	4ω	4.00034	25	325
	12	20ω	19.9998	27	378
	20	44ω	44.0003	35	630
3	2	3ω	3.00192	23	2300
	4	6ω	5.99929	25	2925
	16	36ω	36.0002	30	4960

Table 5.3: The results of VMC simulations with harmonic oscillator basis functions as the single-particle wavefunctions in the non-interacting closed-shell systems of a double harmonic oscillator potential with a barrier $L_x = 4$. The oscillator frequency was set to $\omega = 1$ for all simulations.

5.1.3 Verifying the Single-Particle Functions

To verify that we do indeed have a good representation of the single-particle wavefunctions in the HO basis given by equation 2.23, we should compare it to the original eigenfunction solution. The way we have chosen to do this comparison is to study the L_2 *norm* of these functions. Given two discretized functions $f(x)$ and $g(x)$, we define it as

$$||f(x) - g(x)||_2 = \sqrt{\sum_{i=0}^{N-1} |f(x_i) - g(x_i)|^2} \tag{5.1}$$

By letting $f(x)$ be the left-hand side of equation 2.23 and $g(x)$ right-hand side, we can check that the result of the L_2 norm is a small number. If that is the case, our coefficients $c_{n'n}$ of the approximation give a good representation of the eigenfunction $\psi_{n'}$. In addition, we will plot the curves of f and g to see if they coincide, see figure 5.1. A third test to see that the norm of the expansion, $||g|| = \sqrt{\langle g|g\rangle}$, is close to 1. Since we set $L_y = 0$, the norm of the y part of $\psi_{n'}$ will be practically 1, so the interesting part is seeing what the norm of the x component looks like. With two barriers such that $L_x = L_y$, the norm of the x and y part would be the same.

We expect to see that the number of basisfunctions used to represent $\psi_{n'}$ will have a great impact on the precision, that is how small the L_2 norm is. In addition, we expect that the higher-order wavefunctions, that is larger values of n', will require even more basisfunctions to achieve the same level of precision as lower n'.

By looking at table 5.4 and figure 5.1, we see that these assumptions seem to hold.

# of Basis Functions	n'	$\|\|\sum c_{n'n}\varphi_n^{\text{ho}}\|\|$	L_2 norm
300	0	0.999997	0.03190
	4	0.998727	0.10237
	7	0.995539	0.17560
171	0	0.998871	0.54428
	4	0.92183	0.12809
	7	0.86039	0.34298

Table 5.4: Comparison of the number of basisfunctions needed to achieve good precision for the approximate wavefunction $\psi_{n'}$ in the two dimensional double-well. The well constants are $L_x = 4$, $L_y = 0$ and the number of grid points $N = 1000$. It shows that the number of basisfunctions and the order of the energy eigenstate are variables that determine the precision of the wavefunction.

5.2 Finite Square Well

The finite square well has no analytical solution, but as we discussed in chapter 2.1.2, there is an analytical solution when the potential outside the well approaches infinity. We first present the results for the diagonalization of the Schrödinger equation for the "infinite" square well ($V_0 = 10^{10}$) in table 5.5. With relatively few steps (5000) we get satisfingly close to the analytical results. The accuracy falls with increasing energy level.

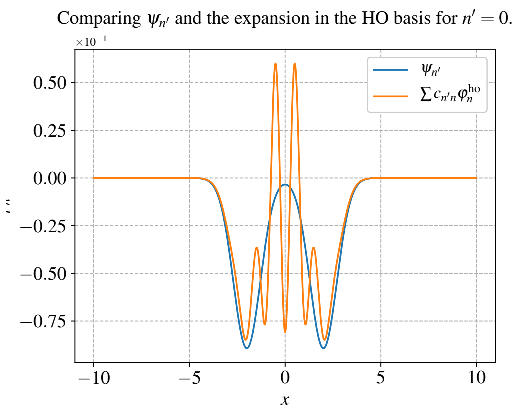
n	$E_n = \pi^2/(8L^2)$	E_{diag}
1	0.3084251	0.3084250
2	1.2337006	1.2337000
3	2.7758262	2.7758100
4	4.9348022	4.9347400

Table 5.5: Results for the diagonalization of the Schrödinger equation of the "infinite" square well with $5 \cdot 10^3$ steps. The potential outside the well was set to $V_0 = 10^{10}$ and the distance to the wall is $L = 2$.

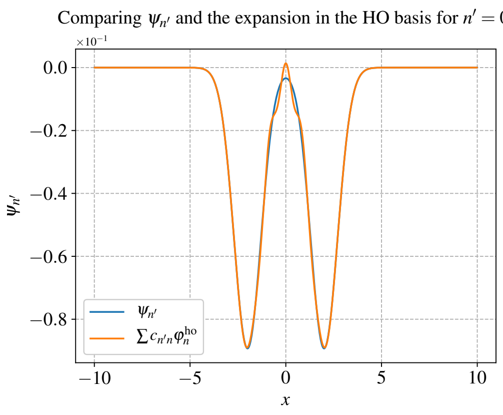
In [5], David L. Aronstein and C. R. Stroud Jr. present an approximation for the finite square well and use an iteration scheme to approximate benchmark energies for the one dimensional finite square well. To compare our results with theirs, we must introduce the *well-strength parameter*

$$P = \frac{\sqrt{2mV_0}}{\hbar}L \tag{5.2}$$

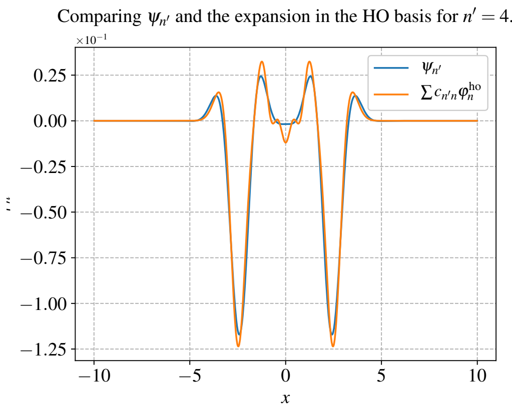
To replicate the conditions used in this paper, we need an expression for the potential



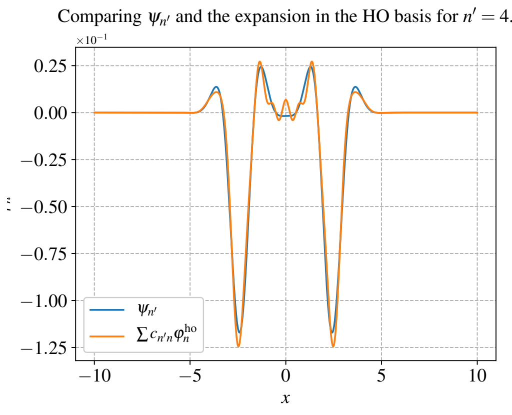
(a) 171 basisfunctions.



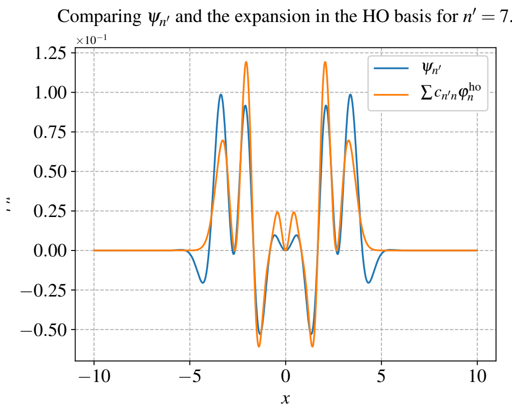
(b) 300 basisfunctions.



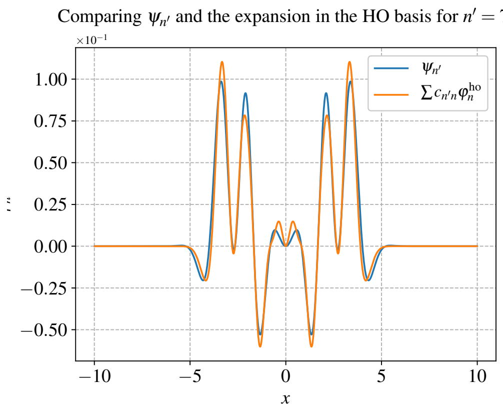
(c) 171 basisfunctions.



(d) 300 basisfunctions.



(e) 171 basisfunctions.



(f) 300 basisfunctions.

Figure 5.1: Plots of the double harmonic oscillator potentials wavefunctions $\psi_{n'}$ vs. the approximation using the single HO basisfunction expansion. We see that more basisfunctions means a better representation. We also see that we require more basisfunctions the higher order wavefunction (larger n' values).

given the well-strength parameter and the distance to the wall, L. We get

$$V_0 = \frac{1}{2}\frac{P^2}{L^2} \tag{5.3}$$

as we use natural units, $\hbar = m_e = 1$. We set the wall distance $L = 2$ for all simulations in this chapter. As an example, when $P = 1$, we have that $V_0 = 1/8 = 0.125$.

Therefore we use the well-strength parameter used in the article to determine the potential V_0. Note that they have used half the wall distance, $L/2$, as the variable, which make our expressions look slightly different from the expressions in the article. The results are listed in table 5.6. They show that for the one-dimensional case, our results are pretty close to the benchmark.

P	Energy Level	E	E_{diag}
1.00	0	1.0925	1.0984
4.50	0	3.2867	3.2893
	1	12.9179	12.9280
	2	27.8821	27.8941
6.00	0	3.6167	3.6198
	1	14.3518	14.3638
	2	31.7736	31.7995
	3	54.6214	54.6621

Table 5.6: Results for the diagonalization of the Schrödinger equation of the finite square well with 10^4 steps. The table shows the different energies of the energy levels when varying well-strength parameter $P = \sqrt{2V_0}L$. The wall distance was set to $L = 2$. Energies are in units of $\hbar^2/(m_e(2L)^2)$. As we use natural units, this implies that we multiply the energy resulting from the simulation by $4L^2 = 16$.

We will use the results of the diagonalization as benchmarks for verifying the VMC results of the finite square well in two and three dimensions when there are no interactions between particles. These results are of course not exact, and they are dependent on the precision, determined by the number of grid points we choose. This is shown in 5.7, we list the total energy for energy levels 0 and 14 in two dimensions with increasing number of grid points N.

Energy level	N	$E^{\text{diag}} = E_x + E_y$
0	100	0.354722
	1000	0.333872
	2000	0.332682
	3000	0.332284
	4000	0.332086
	5000	0.331966
	10000	0.331728
14	100	7.12142
	1000	7.21398
	5000	7.21400
	10000	7.21390

Table 5.7: Increasing the step numbers show that the precision for the ground state energy increases. In this table we list the two-dimensional ground state energy for the 0th and 14th energy level of the finite square well. The potential outside the well is set to $V_0 = 1$ and the distance to the well wall is $L = 2$. The increase in accuracy gets smaller and smaller as we further increase N. This is easily seen by looking at the jump in energy from $N = 100$ to $N = 1000$ compared to $N = 5000$ to $N = 10000$.

We now move on to the testing of the HO basisfunctions used as single-particle wavefunctions in the VMC calculations. These results depend heavily on the number of basisfunctions, as we see from table 5.8. In this table we also see that the number of Monte Carlo cycles is a contributing factor, but may be slightly less important. One of the most important aspects of the Monte Carlo cycles however, is that it is required to increase it for a higher number of particles. Otherwise our results will not be reliable.

N	E	E_{VMC}	# of Energy Levels	# of Basis Functions	# of Cycles
5000	0.663932	0.663081	10	55	1e4
		0.661318	20	210	
		0.663932	30	465	
		0.659647	5	15	1e4
		0.665143	5	15	1e5
		0.665083	5	15	1e7
10000	0.663456	0.661050	20	210	1e4

Table 5.8: Results for the finite square well in two-dimensions with varying number of grid points N, number of basisfunctions and Monte Carlo cycles. As we see, the "benchmark energy" E varies only with the number of grid points as discussed previously. The VMC energy varies with both the number of MC cycles and the number of basisfunctions, but note that the basisfunctions seems to be the most prominent factor. Also, there seems to be some discrepancy in the results when we compare the results. This could be due to the fact that our benchmarks are already not exact, and we should assume that improving the contributing factors should improve the results.

It should be obvious that increasing the number of particles yields an increase in the required number of basisfunctions. In table 5.9 we list the results for systems with different number of particles in two and three dimensions. We see that increasing the number of basisfunctions lower the VMC energy as assumed. However, it is important to note that while the results are consistent with the benchmarks, the benchmark is *not* a lower bound for the VMC energy. As discussed already this could be due to the benchmarks being approximate.

In table 5.10 we list the result for a system with 20 particles. Here the number of basisfunctions is very high, but still not enough to give a sufficiently good result.

# of Dim	# of Particles	E	E_{VMC}	# of Energy Levels	# of Basis Functions
2	2	0.663932	0.663081	10	55
			0.661177	23	276
	6	3.822220	3.828040	10	55
			3.808650	25	325
	12	11.168068	11.285200	40	820
			11.020500	50	1275
3	2	0.995898	1.008970	10	220
			0.996780	23	2300
	8	6.729228	6.748820	10	220
			6.677380	30	4960

Table 5.9: Results for the two and three dimensional finite square well. The potential inside the well is zero and $V_0 = 1$ outside. We have set the number of grid points is constant at $N = 5000$. As with the previous results, we see that since the benchmarks are approximate, it is challenging to conclude on the accuracy. The idea behind VMC is that the exact energy is a lower bound. That is, when we improve the wavefunctions, our VMC energy will approach the exact energy from above.

# of Dim	# of Particles	E	E_{VMC}	# of Energy Levels	# of Basis Functions
3	20	23.481330	23.706100	40	11480

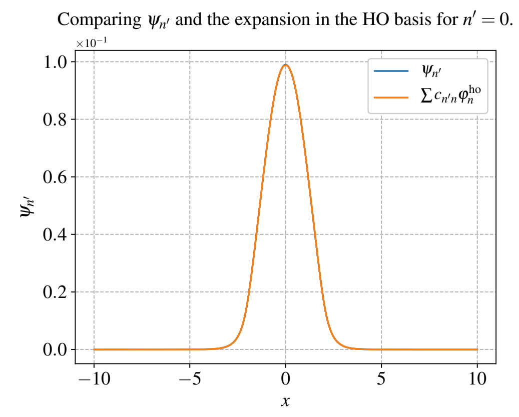
Table 5.10: Here we present the results of the VMC simulation of the three dimensional finite square well with a large amount of basisfunctions for 20 particles. As the number of particles is large, we are not able to get a result that is consistent with the benchmarks. Generating a large number of basisfunctions is very heavy computationally. Number of grid points $N = 11500$.

5.2.1 Verifying the Single-Particle Wavefunctions

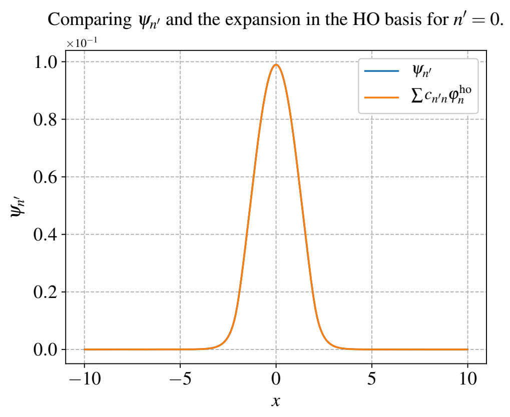
As for the double well, we want to know how good of a representation the harmonic oscillator basis expansion as an approximation to the single-particle wavefunction for the finite square well. We will therefore perform the same tests as we did previously. The results can be seen in table 5.11. Here the norm of the x and y part of the wavefunction are the same. This is due to the symmetric property of the finite square well. Again the assumptions hold: with a larger amount of basisfunctions, we get better accuracy. However we also see that for $n' = 0$, the results are pretty good, while for $n' = 4$ and $n' = 7$, we still have pretty bad accuracy for 300 basisfunctions. For 2550 basisfunction, the L_2 norm is still pretty poor, while the norm is getting pretty close to 1. See also the plots in figure 5.2.

# of Basis Functions	n'	$\|\|\sum c_{n'n}\varphi_n^{\text{ho}}\|\|$	L_2 norm
2550	0	0.99999980	0.00070980
	4	0.99976132	0.62519416
	7	0.99911307	0.60789965
300	0	0.99999836	0.00210450
	4	0.79762720	0.77804129
	7	0.82904940	0.91602723
171	0	0.99999586	0.00213014
	4	0.72055440	0.76978104
	7	0.75963737	0.91336362

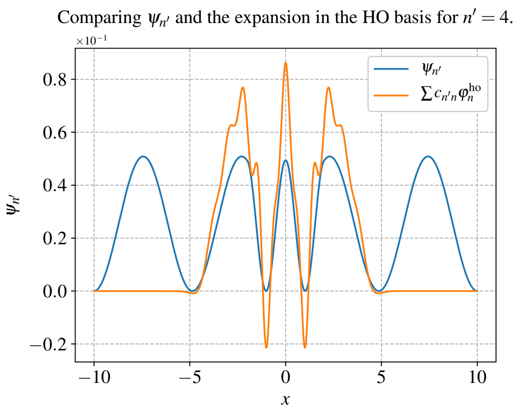
Table 5.11: Comparison of the number of basisfunctions needed to achieve good precision for the approximate wavefunction $\psi_{n'}$ in the two dimensional finite square well. Here the expansion in the y and x direction are the same, since the potential is symmetric. The number of grid points is $N = 1000$. It shows that the number of basisfunctions and the order of the energy eigenstate are variables that determine the precision of the wavefunction.



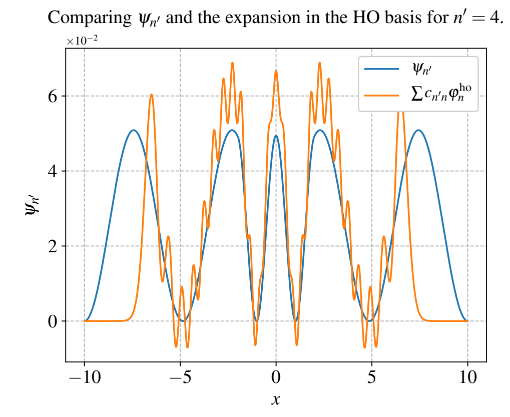
(a) 171 basisfunctions.



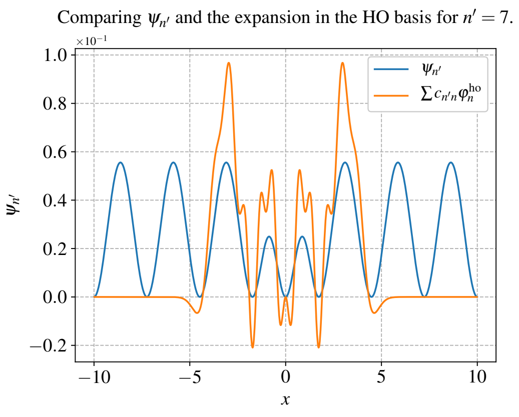
(b) 2550 basisfunctions.



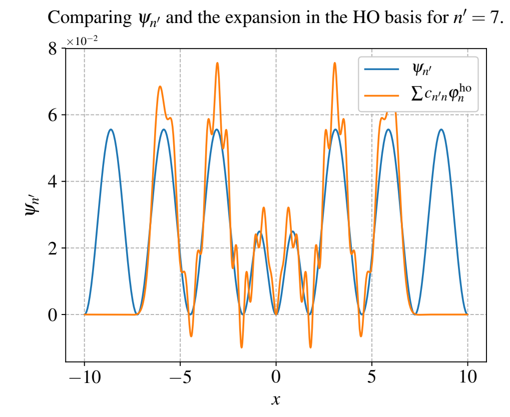
(c) 171 basisfunctions.



(d) 2550 basisfunctions.

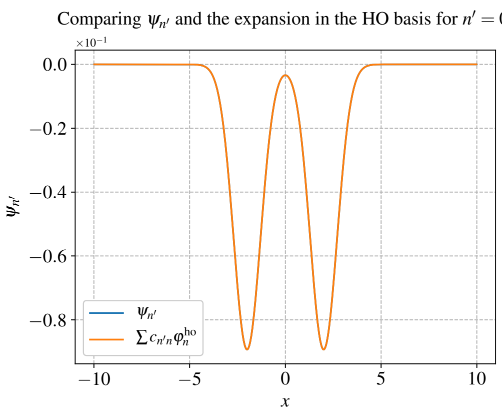


(e) 171 basisfunctions.

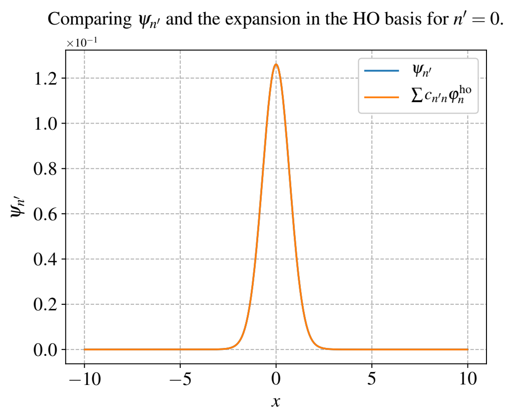


(f) 2550 basisfunctions.

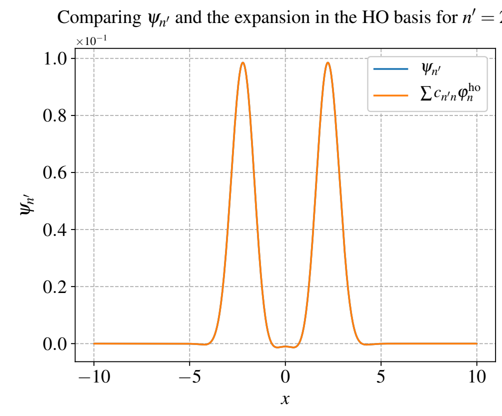
Figure 5.2: Plots of the finite square well wavefunctions $\psi_{n'}$ vs. the approximation using the single HO basisfunction expansion. We see that more basisfunctions means a better representation. We also see that we require more basisfunctions the higher order wavefunction (larger n' values). However not even 2550 basisfunctions is enough for the higher order wavefunctions to be good approximations.



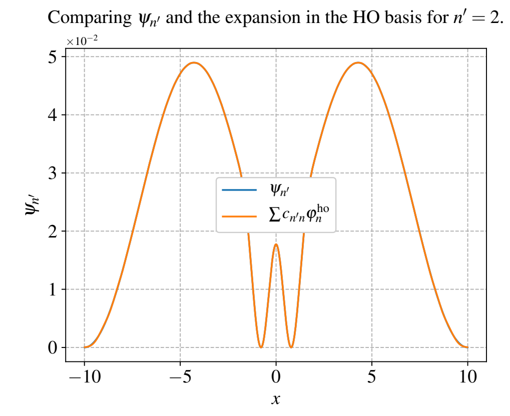
(a) Double Well, $n' = 0$



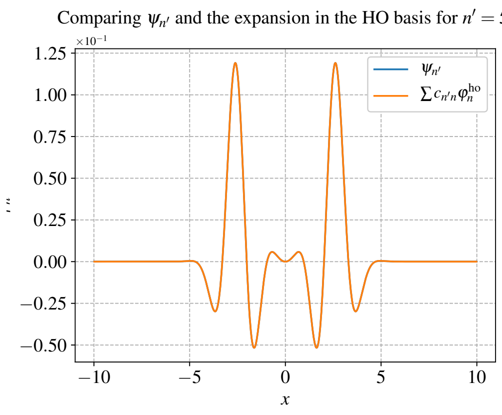
(b) Finite Square well, $n' = 0$



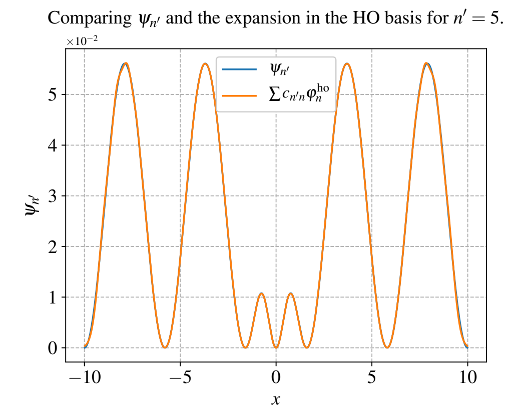
(c) Double Well, $n' = 2$



(d) Finite Square well, $n' = 2$



(e) Double Well, $n' = 5$



(f) Finite Square well, $n' = 0$

Figure 5.3: Plots of the approximation of $\psi_{n'}$ for both the double harmonic oscillator and finite square well. All the approximation were plotted using 5050 basisfunctions. We see that this is enough to very well approximate $\psi_{n'}$.

Part III

Results

6 Results

Through chapter 5 we have already seen some results of the VMC solver developed in this thesis. We will now list and comment systematically the results obtained for different systems with and without particle interaction. The results include the VMC energies with and without the harmonic oscillator basis functions as single-particle wavefunctions, the one-body densities of the system and the results of our optimization scheme.

6.1 Ground State Energies

6.1.1 Single Harmonic Oscillator Potential

Here we present the results of the VMC calculations of the ground state energy of the single harmonic oscillator. As the harmonic oscillator functions are already the single-particle wavefunctions used in the VMC computation, we would expect that the results should be the same in the two cases. However there is one difference between the two. That is the basis function expansion does *not* include an implicit expression for the variational parameter α when we include interactions between particles. This means that we have no way of varying it and it shows in the results. When we set $\alpha = 1$, the results are identical, even when vary the other variational parameter β. When we use the optimal α, the results are slightly different, which can be seen in the results below. A next step in improving the results is to perform the Hartree-Fock method on the basisfunctions which should bring out the α dependency explicitly. We have used statistical analysis in the form of blocking (chapter 2.7) and it is represented by the parenthesis behind the numbers in our results.

We have listed the results in two and three dimensions for varying ω values in table 6.1 and table 6.2. ω is also the changing parameter in our results is the oscillator frequency ω. It is interesting to note that lower ω values yield energies that are closer to each other. In other words: the ratio between E^0_{VMC} and E^1_{VMC} is closer to 1.

Our results are benchmarked against the results of another VMC calculation as well as Similarity Renormalization Group Theory, Coupled Cluster Theory and Full Configuration Interactions for the two-dimensional case and against VMC and DMC results in three dimensions. We could assume that the results of our VMC calculations should be exactly equal to the ones in [16] since this is also a VMC calculation. However, though we both have performed the same VMC calculation with similar trial wavefunctions, the method of finding the optimal variational parameters $\alpha,\ \beta$ is not the same. The actual implementation of the VMC may also produce small changes in the results.

N	ω	E^1_{VMC}	E^0_{VMC}	$E^{(a)}_{\text{ref}}$	$E^{(b)}_{\text{ref}}$	$E^{(c)}_{\text{ref}}$	$E^{(d)}_{\text{ref}}$
2	0.01	0.0754(3)	0.0745(2)	0.07406(5)	-	0.0738{23}	0.0783505{19}
	0.10	0.4460(3)	0.4428(2)	0.44130(5)	-	0.4408{23}	0.44079191{19}
	0.28	1.0283(3)	1.0245(3)	1.02215(5)	-	1.0217{23}	1.0216441{19}
	0.50	1.6658(3)	1.6614(4)	1.66021(5)	-	1.6599{23}	1.6597723{19}
	1.00	3.0034(4)	2.9984(4)	3.00030(5)	-	3.0002 {23}	3.0000001 {19}
6	0.10	3.602(3)	3.565(2)	3.5690(3)	3.49991 {18}	3.5805 {22}	3.551776 {9}
	0.28	7.658(3)	7.609(2)	7.6216(4)	7.56972 {18}	7.6254 {22}	7.599579 {6}
	0.50	11.853(3)	11.781(2)	11.8103(4)	11.76228 {18}	11.8055 {22}	11.785915 {6}
	1.00	20.243(3)	20.143(3)	20.1902(4)	20.14393 {18}	20.1734 {22}	20.160472 {8}
12	0.10	12.568(7)	12.282(3)	12.3162(5)	12.22533 {17}	12.3497 {21}	12.850344 {3}
	0.28	25.866(6)	25.642(4)	25.7015(6)	25.61084 {17}	25.7095 {21}	26.482570 {2}
	0.50	39.406(5)	39.152(4)	39.2343(6)	39.13899 {17}	39.2194 {21}	39.922693 {2}
	1.00	65.952(5)	65.666(4)	65.7905(7)	65.68304 {17}	65.7399 {21}	66.076116 {3}

Table 6.1: Results for the VMC simulations for varying ω in two dimensions. E^0_{VMC} is the regular VMC ground state energy. E^1_{VMC} is the VMC ground state energy using single-particle wavefunctions expanded in the harmonic oscillator basis. N is here the number of particles in the system. The number of basisfunctions used was obtained by inserting $m = N/2$ into equation 2.32 We vary the oscillator frequency ω. The benchmarks are from the following master's thesis'. (a) J. Høgberget [16] (VMC), (b) S. Reimann [25] (Similarity Renormalization Group theory), (c): C. Hirth [14] (Coupled Cluster Singles and Doubles), (d): V. K. B. Olsen [24] (Full Configuration Interaction).

N	ω	E^1_{VMC}	E^0_{VMC}	$E^{(a)}_{\text{ref}}$	$E^{(b)}_{\text{ref}}$
2	0.01	0.0800(2)	0.0790(1)	0.07939(3)	0.079206(3)
	0.10	0.5004(2)	0.4993(3)	0.50024(8)	0.499997(3)
	0.28	1.2006(2)	1.1993(3)	1.20173(5)	1.201725(2)
	0.50	1.9977(2)	1.9963(3)	2.00005(2)	2.000000(2)
	1.00	3.7257(3)	3.7242(3)	3.73032(8)	3.730123(3)
8	0.10	5.726(2)	5.699(2)	5.7130(6)	5.7028(1)
	0.28	12.210(2)	12.172(2)	12.2040(8)	12.1927(1)
	0.50	18.979(2)	18.921(2)	18.9750(7)	18.9611(1)
	1.00	32.685(2)	32.593(2)	32.6842(8)	32.6680(1)

Table 6.2: Results for the VMC simulations for varying ω in three dimensions. E^0_{VMC} is the regular VMC ground state energy. E^1_{VMC} is the VMC ground state energy using single-particle wavefunctions expanded in the harmonic oscillator basis. N is the number of particles in the system. The amount of basisfunctions used is determined by inserting $m = N/2$ into equation 2.33. Benchmarks (a) and (b) are obtained from the VMC and DMC calculations of [16].

6.1.2 Double Harmonic Oscillator Potential

One of the aims of this thesis is to study the amount of basisfunctions needed to create an expansion that can be used as good single-particle wavefunctions in the Slater determinant of equation 2.112 for different potentials. Therefore we must assume that the results presented in this chapter and the next differ from the VMC calculations with the standard single-particle wavefunctions of those potentials. The reason for this is that the equality of equation 2.23 only holds when $\Lambda \to \infty$.

For these results, we have chosen a double well potential with *one* barrier located such that $L_x = 1$, which means that the distance between the wells is 1. The importance of this distance becomes clear when we compare the results for the single and double well. For a system consisting of non-interacting particles, we could assume that a double well is comprised of two identical single wells side by side. The same is true when the distance between the wells is large, as the particle interaction would be weak. When the distance L decreases, we would have a system that more closely resembles the single well.

There is also a third factor, which is also interesting to note in our results. The oscillator frequency ω is part of the potential energy part of the Hamiltonian and thus plays a part in how strong the potential is. A larger ω translates to a stronger barrier between the wells. For that reason we expect lower ω values to yield ground state energies that looks more like the single well results.

What these things means in practice is that for two well centers that are far apart, we are basically looking at a system of $N/2$ particles in two independent single wells. This gives us an upper and lower bound of what the energies in the double well should be. If a system of N particles is in a double well with a well center distance $L_x \in (0, \infty)$, the total energy E_N^{dw} of the system must be

$$E_N^{\mathrm{dw}} \in (2 \cdot E_{N/2}^{\mathrm{sw}}, E_N^{\mathrm{sw}}) \tag{6.1}$$

where E_N^{sw} is the total energy of a single well system with N particles.

The results are presented in table 6.3 for two dimensions and in table 6.4 for three dimensions. As was the case for the single well, the VMC energies are listed as E^0_{VMC} for the standard VMC computation and E^1_{VMC} when we have used single harmonic oscillator expansion.

Unfortunately we found no benchmarks for the three dimensional case, but it is a good indication that our results are self-consistent.

N	ω	# of Basis Functions	E^1_{VMC}	E^0_{VMC}	E_{ref}
2	0.01	1	0.07358	0.0727(2)	-
	0.10	1	0.4106(9)	0.4018(7)	-
	0.28	6	0.860(1)	0.866(2)	-
	0.50	6	1.336(2)	1.339(2)	-
	1.00	15	2.326(2)	2.321(2)	2.3496(1)
4	0.10	3	1.5966(9)	1.5886(7)	-
	0.28	10	3.362(1)	3.22281	-
	0.50	55	4.958(2)	4.831(2)	-
	1.00	55	7.850(2)	7.850(2)	-
12	0.10	21	12.106(6)	12.034(6)	-
	0.28	21	23.771(6)	23.785(7)	-
	0.50	36	34.945(5)	34.945(6)	-
	1.00	45	55.165(6)	55.226(6)	-

Table 6.3: Results for the VMC simulations for varying ω in two dimensions for the double harmonic oscillator. E^0_{VMC} is the regular VMC ground state energy. E^1_{VMC} is the VMC ground state energy using single-particle wavefunctions expanded in the harmonic oscillator basis. N is the number of particles in the system. Benchmark is obtained from the double dot DMC calculations of [16].

N	ω	# of Basis Functions	E^1_{VMC}	E^0_{VMC}
2	0.01	1	0.0782(2)	0.0779(2)
	0.10	10	0.4639(4)	0.4652(5)
	0.28	10	1.0642(7)	1.0711(9)
	0.50	10	1.7324(8)	1.738(1)
	1.00	35	3.212(2)	3.194(2)
8	0.10	10	1.640(1)	1.642(1)
	0.28	56	3.518(1)	3.28382
	0.50	56	5.3858(9)	5.360(2)
	1.00	56	9.153(2)	9.106(2)

Table 6.4: Results for the VMC simulations for varying ω in three dimensions for the double harmonic oscillator. E^0_{VMC} is the regular VMC ground state energy. E^1_{VMC} is the VMC ground state energy using single-particle wavefunctions expanded in the harmonic oscillator basis. N is the number of particles in the system.

6.1.3 Finite Square Well

In this chapter we present the results of our VMC simulations of the finite square well. Since there are no direct explicit single-wave functions for the finite square well as discussed in chapter 2.1.2, we only use the harmonic oscillator basisfunction expansion as the single-particle wavefunction. Furthermore, we were not able to find any other results of the systems we have looked and hence we are left with no benchmarks except for the ones presented in section 5.2 in table 5.6 for the one-dimensional case without interactions. One of the aims of the thesis is to investigate the quantity of basisfunctions needed to achieve good results of the VMC calculation. We will therefore look at how many basisfunctions we need in order for the VMC ground state energy to converge, rather than verifying the results explicitly against known benchmarks.

Something to note is that how many basisfunctions we need is dependent on the physical properties of the harmonic oscillator well and the finite square well we use. How good the harmonic oscillator potential "fits" in the well determines how closely we can mimic the wavefunctions of the particles residing in the well.

In practice, the tuning of the wells is done by varying ω for the harmonic oscillator potential, and V_0 and the wall distance of the square well. We have typically used $V_0 = 1$ and the wall distance set to 2 in our calculations. A harmonic oscillator potential with $\omega = 1$ would then be more narrow than the square well and fit inside of it. However it would have room on each side, which means that it isn't able to represent all the possible wavefunctions in the square well. On the other hand, $\omega = 0.1$, would be much wider than the square well and also much less steep. This means it would have the possibility of representing some wavefunctions that the larger ω could not, especially near the bottom of the square well. Unfortunately it would not represent the top of the well very well as the potential strength would not be anywhere near $V_0 = 1$. These scenarios are shown in figure 6.1 (a) and (b).

The results are presented in table 6.5 for two dimenions and in table 6.6 for three dimensions. We would expect that the ground state energies are much closer for different ω for the finite square well than for the HO wells, since there is no dependency on ω in the finite square well. The results indicate that this is correct, but they are not exactly equal. Another thing to note is that we need more basisfunctions to find an optimal β value for the $\omega = 1$ case, but once we have found it, we need fewer basisfunctions for the energy to converge. For the three dimensional case, we of course need more basisfunctions to get good results. Thus the computational cost becomes very heavy with only two particles.

As discussed in the double well results chapter, the explicit α dependence is now completely lost without some sort of Hartree-Fock calculations on the coefficients of the single-particle wavefunction in equation 2.23. α is therefore set constant to 1 throughout all computations. We vary and find the best β with the steepest descent method.

N	ω	# of Basis Functions	E^1_{VMC}
2	0.10	15	1.33(2)
		15*	1.32(2)
		45	1.271(5)
		78	1.266(5)
		120	1.265(5)
	1.00	15	1.525(6)
		45	1.450(5)
		15*	1.296(2)
		78	1.281(2)
		120	1.281(2)
6	0.10	45	9.38(5)
	1.00	45	10.689(5)
		120	10.61(3)

Table 6.5: Results for the VMC simulations for varying ω in two dimensions for the finite square well. E^1_{VMC} is the VMC ground state energy using single-particle wavefunctions expanded in the harmonic oscillator basis. We have no known analytical solutions for the finite square well and hence we have only performed the VMC calculations with the HO basisfunction expansion of the single-particle wavefunction. N is the number of particles in the system.
*Here we have used 120 basisfunctions to perform the optimization of β, but 15 basisfunctions to perform the actual calculations.

N	ω	# of Basis Functions	E^1_{VMC}
2	0.01	165	1.412(5)
		364	1.411(5)
	1.00	165	1.465(3)
		364	1.463(3)

Table 6.6: Results for the VMC simulations for varying ω in three dimensions for the finite square well. E^1_{VMC} is the VMC ground state energy using single-particle wavefunctions expanded in the harmonic oscillator basis. We have no known analytical solutions for the finite square well and hence we have only performed the VMC calculations with the HO basisfunction expansion of the single-particle wavefunction. N is the number of particles in the system.

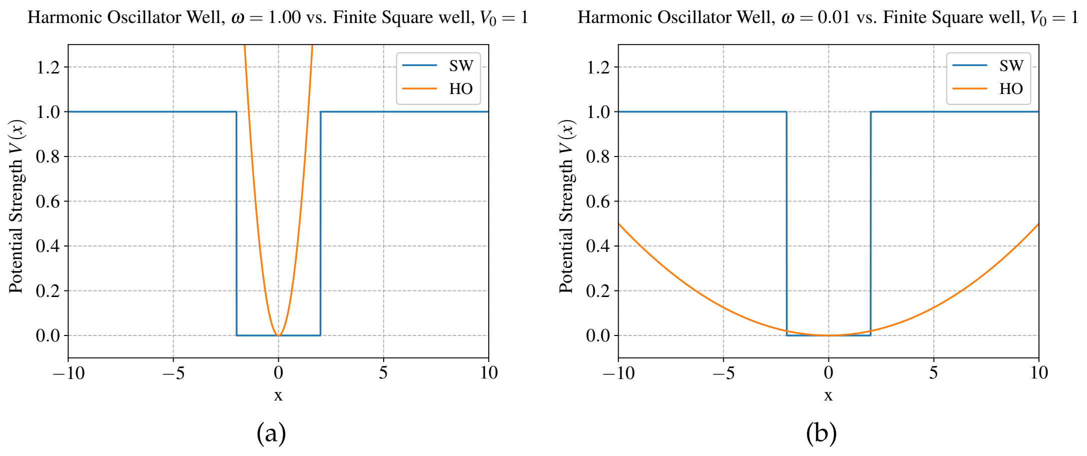


Figure 6.1: Plots of how well the harmonic oscillator fits the finite square well with two ω values. How well the wells fit determine the amount of basisfunctions needed to accurately calculate the ground state energy.

6.2 One-Body Densities

One-body densities are the positional distributions of the particles. That is, what is the most probable position for the particles to be in. This is visualized as either the distribution with respect to the radius around the center of the potential or with respect to the absolute position in the x, y or z direction. We have chosen to show the radial distribution, as well as a three dimensional plot of the distribution with respect to the positions x and y of the particles. The systems we will be looking at are the three first closed shells in two dimensions, which means that the number of particles will be $N = 2$, $N = 6$ and $N = 12$.

An important reason to study the one-body densities is to verify that we have a system that is probable. That is, we want to know that the particles tend to be where they should. The potential has its minimum point at $r = 0$, but the particles also repel each other when we include particle interactions. This means that we will have a ditribution of particles around the center of the potential at some distance r, but not in the absolute center. The more particles we put in the well, the stronger the interaction between them will be. Then the radius were we find the most particles will increase with the number of particles, as the well potential does not and the particles will find a stable radius where the forces are equal.

6.2.1 Single Harmonic Oscillator Potential

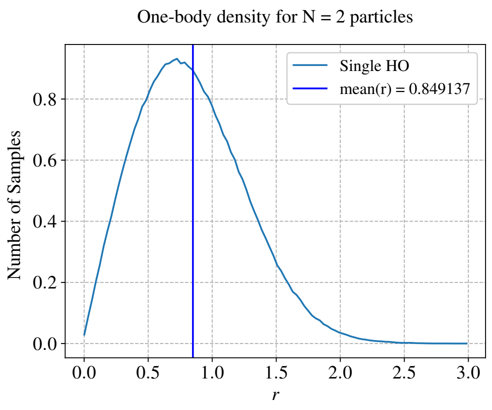
In figure 6.2 we show the one-body densities of the single harmonic oscillator. As predicted, the greater number of particles, the greater radius where the particles are most frequently found. We also see a slight change in the distribution curve. This means a higher distribution closer and farther away from the centre and indicates that the particles are on different energy levels. The surface plot is an even better visualization of the energy levels. Furthermore, the distribution with respect to the coordinates is almost perfectly radially symmetrical. This is due to the symmetry of the single harmonic oscillator. We will see another symmetry when examining the surface plots of the double well.

6.2.2 Double Harmonic Oscillator Potential

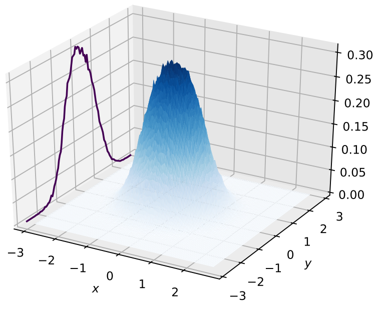
We move on to the double harmonic oscillator potential. The well is created by putting a barrier in the centre of the well such that $L_x = 2$. This sets each well centre at $r = 1$. The one-body densities for different number of particles can be seen in figure 6.3. Since the particles are divided by a barrier, we will have a slightly different dynamics. They will still interact with each other and the sum of the set of particles in one well will act on the other set. This will further increase the radius in which we will see the highest distribution of particles. In addition, since the well centers are not at origo, the push to higher r values will be even higher. For two particles we see that we get two distinct tops, while for six we get two tops with different height in each well. This indicates the two different energy levels that the particles are on. Also, opposed to the singel harmonic oscillator, we see that particles do not align radially symmetrically. When we move up to twelve particles, we see three peaks in each well, a total of six, representing the three first energy levels. It is interesting two see how the particles align themselves most efficiently when we change the forces acting on them by introducing more particles and a potential barrier separating them.

6.2.3 Finite Square Well

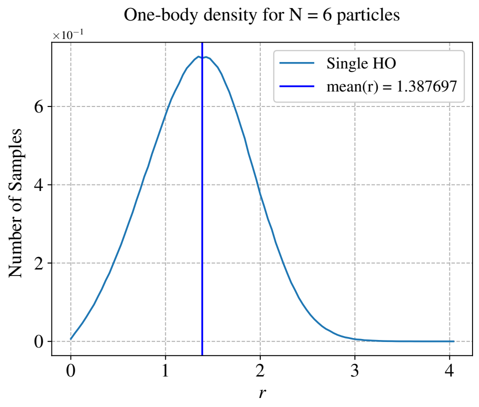
The finite square well is in some ways quite different from the two we've looked at. It is quadratic, and the well center is positioned at zero with the shortest distance to the walls being $r = 2$. The corners will thus be located at the $r = \sqrt{8}$. Our particles will therefore be trapped in this box and can't be any farther away from the centre.



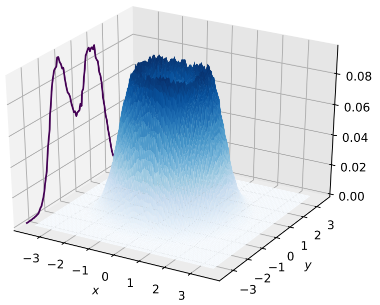
(a) N = 2 particles, 1e6 MC cycles



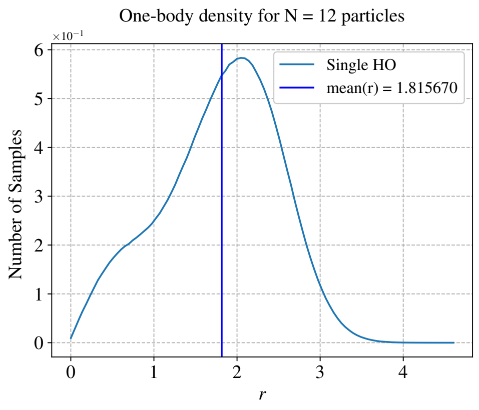
(b) N = 2 particles, 1e6 MC cycles



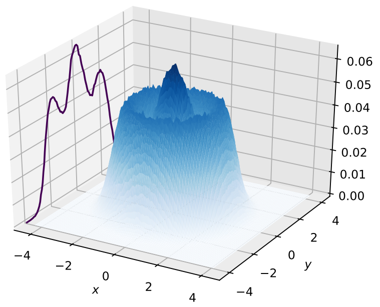
(c) N = 6 particles, 1e6 MC cycles



(d) N = 6 particles, 1e6 MC cycles



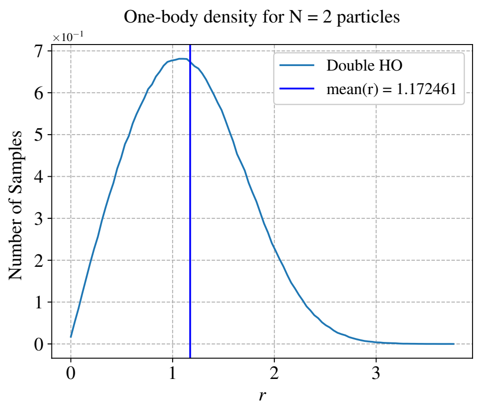
(e) N = 12 particles, 1e6 MC cycles



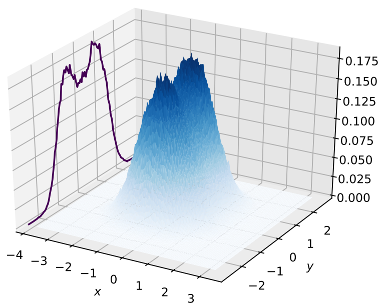
(f) N = 12 particles, 1e6 MC cycles

Figure 6.2: The left side shows the radial one-body density of the single harmonic oscillator well. The right side is the surface plot of the distribution with respect to the absolute position of the particles.

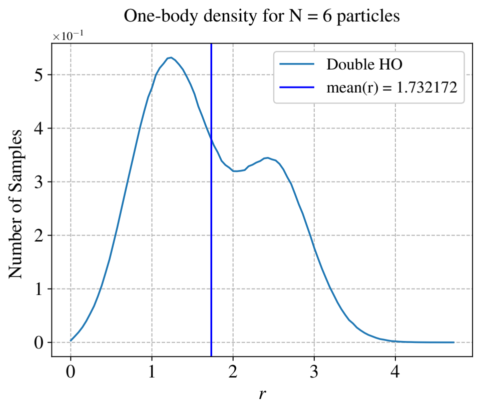
Since we do not have any analytic single-particle wavefunctions, we will investigate how the particles act for different amounts of basisfunctions. First we look at a system of $N = 2$ particles with varying numbers of basisfunctions. We see from figure 6.4 that



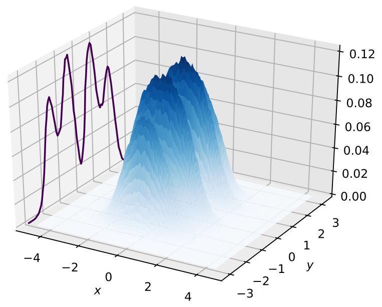
(a) N = 2 particles, 1e6 MC cycles



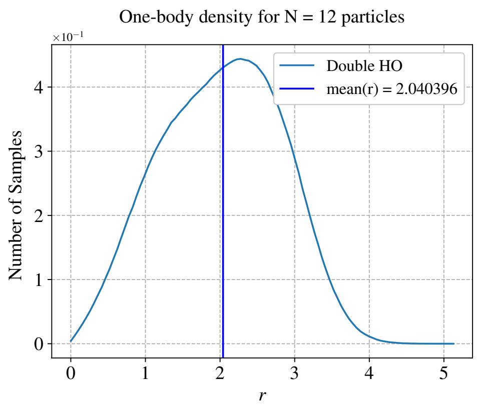
(b) N = 2 particles, 1e6 MC cycles



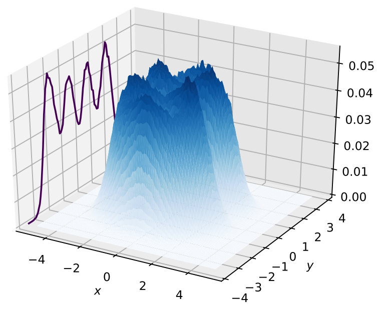
(c) N = 6 particles, 1e6 MC cycles



(d) N = 6 particles, 1e6 MC cycles



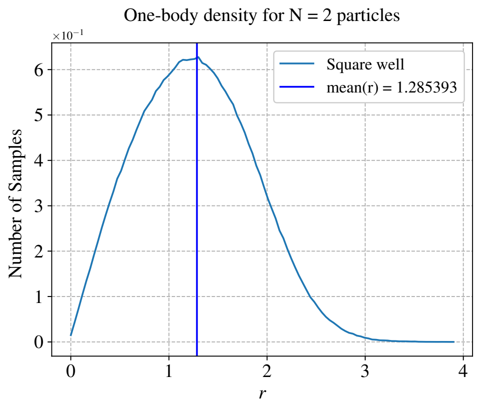
(e) N = 12 particles, 1e6 MC cycles



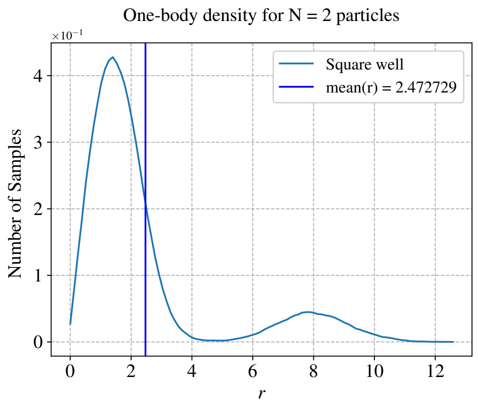
(f) N = 12 particles, 1e6 MC cycles

Figure 6.3: The left side shows the radial one-body density of the double harmonic oscillator well. The right side is the surface plot of the distribution with respect to the absolute position of the particles. For the twelve particle case, we see that we have three tops on each side of the x axis with $y = 0$, but the outline behind it only shows this as four tops since its silhouette is in the y, z plane.

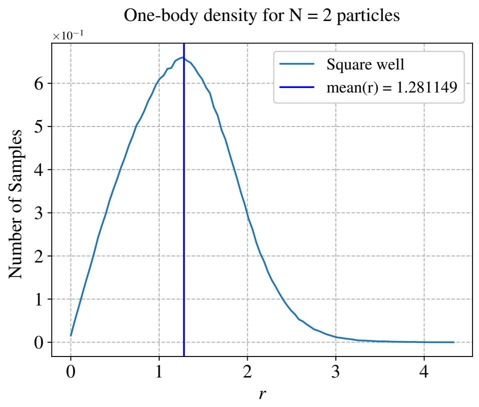
ω plays a part in how well the particles distribute. For $\omega = 0.10$, the particles are not completely bound. This means that we do not have a good representation of the single-particle wavefunction or the physical system. However we see that when we increase the number of basisfunctions, both the $\omega = 0.10$ and $\omega = 1.00$ are bound. In the final plot, figure 6.5, we see six particles in the finite square well. The particles tend to the corners of the well due to the repulsion between them.



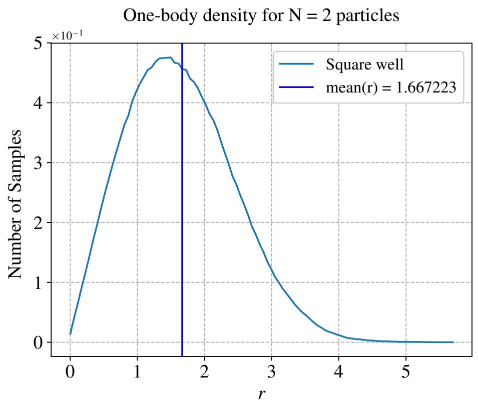
(a) 21 basisfunctions, 1e6 MC cycles, $\omega = 1$



(b) 21 basisfunctions, 1e6 MC cycles, $\omega = 0.10$

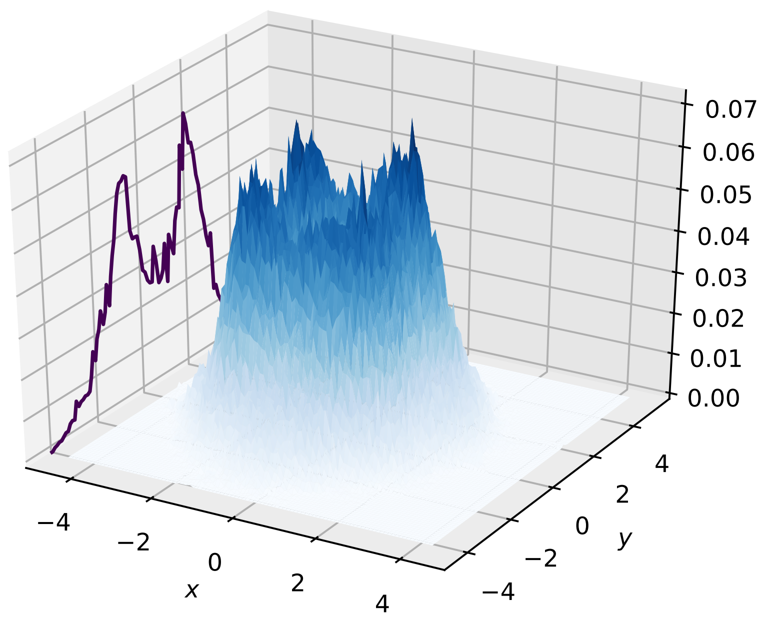


(c) 45 basisfunctions, 1e6 MC cycles, $\omega = 1$



(d) 45 basisfunctions, 1e6 MC cycles, $\omega = 0.10$

Figure 6.4: The density plots of two particles in a two dimensional finite square well. We vary the oscillator frequency ω and the number of basisfunctions.



(a) 45 basisfunctions, 1e5 MC cycles, $\omega = 1$

Figure 6.5: Six particles in the finite square well. We see how they tend towards the corners of the well.

6.3 Performance Optimization

In chapter 4 we presented the optimization schemes that we implemented in the VMC solver. To see how good the optimizations were, we performed two experiments were we tried combinations of the optimizations. Here we present the results of these experiments. We will see that we achieve a 30 times speed-up.

The first experiment was using 300 basisfunctions for a system of two particles in a double harmonic oscillator potential. We call this test 1. The second experiment (test 2) was on a system of 20 particles in the double harmonic oscillator potential, with the single-particle wavefunctions being a superposition of two single harmonic oscillator wavefunctions centered in each side of the barrier. Both tests were performed for a two-dimensional system. We used 1e4 MC cycles for both experiments and ran each of the experiments 10 times and averaged the time. Below we present each of the optimization and the results in table 6.7.

P, pow: The first optimization is to replace `pow` functions with the explicit products. That means writing `x*x` instead of `pow(x,2)`, or utilizing the `ffastmath` flag in the compiler. From the table we see that this gives only slight improvements when this is the only optimization. The reason for this is that when we are not using the Hermite polynomial optimization, the amount of power expressions is not that significant. It is even less effective for test 2. This is due to the superposition wavefunction only having two parts for each single-particle wavefunction. Hence the amount of power expressions is less.

D, relative distances matrix: By storing the relative distancs before particles in a matrix, we achieve no speed-up at all in test 1 as we only have two particles. However we see that we get an encouraging speed-up in test 2 where the number of particles is 20.

S, Slater matrices: Storing the Slater matrices provides a huge speed-up in test 1. This is due to the large expressions in the Slater matrices caused by the basisfunction expansions. They are calculated by a loop over all the functions for each Slater matrix element. Thus when we minimize these calculations we get a great speed-up. When we have few parts in the single-particle wavefunctions, the effect will be small as we see for test 2.

J, Jastrow matrices: The Jastrow part of the trial wavefunction increases with the amount of particles. Therefore we see no speed-up in test 1 and a large speed-up in test 2.

H, Hermite polynomials: By hardcoding the Hermite polynomials instead of recursively computing them, we save a significant amount of time. However, the speed-up is dimished when we aren't utilizing the power expression optimization. This is as expected and we mentioned this in the optimization chapter. Again, this optimization is not as effective in test 2, as we don't have as many parts in the single-particle wavefunctions.

Optimizations	Relative 1	Total 1	Relative 2	Total 2
-	1	1	1	1
P	1.15	1.15	1.02	1.02
P, D	1	1.15	1.1	1.12
P, D, S	6.30	7.25	1.68	1.85
P, D, S, J	1	7.25	5.14	9.52
P, D, S, J, H	4.2	30.45	1	9.52
D, S, J, H	0.13	4.02	0.98	9.35

Table 6.7: We performed two performance tests for two-dimensional systems. Test 1 was using a system of two particles in a double well with a single-particle wavefunction expanded by 300 HO basisfunctions. Test 2 was for a system of 20 particles in a double well, using a superposition wavefunction. The relative speed-up is the speed-up from the row above and the total speed-up is the accumulated speed-up of all optimizations.

In addition to implementing the optimizations above, we also parallelized the process. Since a VMV computation is just a large number of equal experiments, it can be parallelized almost perfectly. Thus if we divide the work on n processes, the runtime should be reduced by $1/n$. However this is not exactly the case. Even though we are able to perfectly split the workload, the process of splitting the work and the communication between the processes takes some time. As we see in table 6.8, the run that had a shorter duration has a less favorable speed-up than the longer one. It is due to the fraction of time that the program spends dividing workload and communicating between processes. This is called parallel overhead. When the runtime of the program approaches infinity, we should expect the the relative speed-up when doubling the number of processes approaches 2. Note also that the relative speed-up is lower for each doubling. This is due to there being more processors that needs to communicate.

# of Processors	Relative 1	Total 1	Relative 2	Total 2
1	1	1	1	1
2	1.926	1.926	1.939	1.939
4	1.859	3.580	1.883	3.651

Table 6.8: We performed two tests to see how good of a speed-up we would get by doubling the number of processors two times. Both tests were done for a double harmonic oscillator potential in two dimensions with 1265 basisfunctions. The first test was done with two particles and the second with four. We see that the second test, which has a longer runtime is closer to having a linear speed-up than the two-particle test. A longer runtime means that the time the program spends on dividing the workload and communicating is a smaller fraction than for a runtime with a shorter duration. Both tests were performed five times and the speed-up was averaged.

7

Conclusions

The aim of this thesis has been to further the knowledge of trapped fermionic quantum dots using various numerical methods to create a robust computational framework. A lot of effort was put into testing the framework. This includes unittesting and larger scale tests to verify the authenticity of the results. The code base was designed so that one can easily implement new methods and extensions to the physical systems. This is needed as to further the work from this thesis, there are several paths which will better the understanding of the systems we study and quantum dots in general.

Our first task was to solve the Schrödinger equation for a non-interacting system of different potentials were the wavefunctions were separable. This gave us the first way to verify the results to come. When we solved this equation, we got the eigenfunctions and energy eigenvalues of the Hamiltonian for each spatial coordinate. This let us find the different energy levels in which the non-interacting particles could take. Furthermore, we used these single-particle wavefunctions to create an expansion in the single harmonic oscillator basis. The use and study of this type of single-particle wavefunction has been the main focus of the thesis.

The by far largest part of the code base is the VMC solver. It is the implementation of the Variational Monte Carlo method used to find the ground state energy of quantum systems. We used it to reproduce the results of the ground state energies for the single harmonic oscillator with several different methods. In addition, we produced a set of new results for the double harmonic oscillator while also reproducing some, which seems to validate our implementation.

With this framework, we have studied the ground state energies of the single harmonic oscillator well, the double harmonic oscillator well and the finite square well. The single well was used to verify that our basisfunction expansions were valid. Then we spent our energy using the basisfunctions to expand the single-particle functions of the two other potentials. There are few benchmarks to be found for the finite square well, but we reproduced the results of the one-dimensional non-interacting case. Since there are no explicit analytical solutions of the single-particle wavefunctions of the finite square well, an approach like the basisfunction expansion is needed.

To verify the authenticity of the actual physical system we simulated, we plotted the one-body densities for all the potentials with varying number of particles and oscillator frequencies ω. This showed that ω plays a part in the number of basisfunctions needed to get good precision and that the particles distributed themselves as we would expect given what we know about the effects of particle-particle and particle-potential interactions.

As VMC is a variational method, we used the steepest descent method to optimize the variational parameters α and β. Without this we would not be able to effectively find good variational parameters which means the ground state energy would inaccurate. What we didn't first think of was that the α parameter is stated in the single-particle wavefunctions and thus it is not explicitly stated in our expansions. This effectively means that we had to set $\alpha = 1$ and vary β. When we did this we saw that we could reproduce the VMC results for a constant $\alpha = 1$ with the standard single-particle wavefunctions.

For both the double well and the finite square well we saw that a small amount of basisfunctions did not very well approximate the single-particle wavefunctions and they did not produce satisfactory results for the ground state energy. However, increasing the number of functions in the expansion resulted in a more or less identical wavefunction and the results also converged toward the benchmarks. However, this was very computationally expensive.

This brings us to the final part of the thesis which was to find ways to optimize the computations. We introduced several different optimizations and effectively cut the runtime of the program to 3.3%. We saw that the optimizations were uniquely effective on different types of systems. Some optimizations were dependent on the number of particles and some were dependent on the amount of basisfunctions used.

The Next Steps

To increase the accuracy of the ground state energies, extending the solver with a Diffusion Monte Carlo implementation is a natural next step. As our main focus has been the basisfunction expansions, there is a need to find a way to explicitly state the variational parameter α in our single-particle wavefunctions. This can be done by implementing the Hartree-Fock method on the coefficients of our expansions. The basisfunction approach is also very heavy computational-wise and even more optimization is definitely needed to study larger systems.

The finite square well has still a lot of interesting properties to study. Several different basisfunctions should be suitable to try to increase the accuracy and compare the results of the ground state energy to ours. A first step could be to try the infinite square well wavefunctions which we have mentioned. One could also try the Hydrogen-like functions, which are used in VMC calculations of atoms and molecules.

Appendices

A

Derivatives of Correlation Part of the Trial Wavefunction

In this appendix we will show the calculation of the correlation parts, given by the function in equation 2.113. That is, the derivatives of the Jastrow factor.

First off, we define

$$f_{ij} \equiv \frac{ar_{ij}}{1+\beta r_{ij}}$$

with the corresponding derivatives (with respect to r_{ij})

$$\begin{aligned} f'_{ij} &= \frac{a}{(1+\beta r_{ij})^2} \\ f''_{ij} &= \frac{-2a\beta}{(1+\beta r_{ij})^3} \end{aligned}$$

The gradient of the wavefunction, divided by the wavefunction, for particle k in the x-direction is then

$$\left[\frac{\hat{\nabla}_k \psi_C}{\psi_C}\right]_x = \frac{1}{\psi_C}\frac{\partial \psi_C}{\partial x_k}$$

If we look at the first derivative in the x-direction, we see that the parts of the wavefunction that is not dependent on k, will remain unaffected by the differentiation.

When we split the expression for $i < k$ and $k > i$, we get that

$$\begin{aligned}
\frac{\partial \psi_C}{\partial x_k} &= \prod_{i,j\neq k} g_{ij} \frac{\partial}{\partial x_k}\left[\prod_{i<k} g_{ik} \cdot \prod_{i>k} g_{ki}\right] \\
&= \prod_{i,j\neq k} g_{ij} \left[\prod_{i>k} g_{ki} \frac{\partial}{\partial x_k} \prod_{i<k} g_{ik} + \prod_{i<k} g_{ik} \frac{\partial}{\partial x_k} \prod_{i>k} g_{ki}\right] \\
&= \prod_{i,j\neq k} g_{ij} \left[\prod_{i>k} g_{ki} \sum_{i<k} \frac{\partial g_{ik}}{\partial x_k} \prod_{p\neq i} g_{pi} + \prod_{i<k} g_{ik} \sum_{i>k} \frac{\partial g_{ki}}{\partial x_k} \prod_{q\neq i} g_{iq}\right] \\
&= \prod_{i<j} g_{ij} \left[\frac{1}{\prod_{i<k} g_{ik}} \sum_{i<k} \frac{\partial g_{ik}}{\partial x_k} \prod_{p\neq i} g_{pi} + \frac{1}{\prod_{i>k} g_{ki}} \sum_{i>k} \frac{\partial g_{ki}}{\partial x_k} \prod_{p\neq i} g_{pi}\right] \\
&= \psi_C \left[\sum_{i<k} \frac{1}{g_{ik}} \frac{\partial g_{ik}}{\partial x_k} + \sum_{i>k} \frac{1}{g_{ki}} \frac{\partial g_{ki}}{\partial x_k}\right] \qquad \text{(A.1)}
\end{aligned}$$

Here we factorized the wavefunction outside the expression, and noticed that the only part that doesn't cancel is the *ik*-th and *ki*-th in the sums.

Dividing by the wavefunction, we get

$$\begin{aligned}
\left[\frac{\hat{\nabla}_k \psi_C}{\psi_C}\right]_x &= \sum_{i=1}^{k-1} \frac{1}{g_{ik}} \frac{\partial g_{ik}}{\partial x_k} + \sum_{i=k+1}^{N} \frac{1}{g_{ki}} \frac{\partial g_{ki}}{\partial x_k} \\
&= \sum_{i=1}^{k-1} \frac{\partial f_{ik}}{\partial x_k} - \sum_{i=k+1}^{N} \frac{\partial f_{ki}}{\partial x_i} \\
&= \sum_{i=1}^{k-1} \frac{x_k - x_i}{r_{ik}} \frac{\partial f_{ik}}{\partial r_{ik}} - \sum_{i=k+1}^{N} \frac{x_i - x_k}{r_{ki}} \frac{\partial f_{ki}}{\partial r_{ki}}
\end{aligned}$$

since g_{ij} is an exponential function, so $\partial g_{ij}/\partial x_i = g_{ij} \partial f_{ij}/\partial x_j$. We also used the fact that $\partial g_{ij}/\partial x_i = -\partial g_{ij}/\partial x_j$ to differentiate with respect to the second index in both of the sums. Finally, we have used the chain rule to attain an expression that is dependent on the distance between the two particles

$$\frac{\partial f_{ij}}{\partial x_j} = \frac{\partial f_{ij}}{\partial r_{ij}} \frac{\partial r_{ij}}{\partial x_j} = \frac{x_j - x_i}{r_{ij}} \frac{\partial f_{ij}}{\partial r_{ij}}$$

Thus

$$\begin{aligned}
\frac{\hat{\nabla}_k \psi_C}{\psi_C} &= \sum_{i=1}^{k-1} \frac{\mathbf{r}_{ik}}{r_{ik}} \frac{\partial f_{ik}}{\partial r_{ik}} - \sum_{i=k+1}^{N} \frac{\mathbf{r}_{ki}}{r_{ki}} \frac{\partial f_{ki}}{\partial r_{ki}} \\
&= \sum_{i=1}^{k-1} \frac{\mathbf{r}_{ik}}{r_{ik}} \frac{a}{(1+\beta r_{ik})^2} - \sum_{i=k+1}^{N} \frac{\mathbf{r}_{ki}}{r_{ki}} \frac{a}{(1+\beta r_{ki})^2} \\
&= \sum_{i\neq k} \frac{\mathbf{r}_{ik}}{r_{ik}} \frac{a}{(1+\beta r_{ik})^2}
\end{aligned}$$

From (A.1), we can gather that the double derivative part is described by

$$
\begin{aligned}
\left[\frac{\hat{\nabla}^2\psi_C}{\psi_C}\right]_x &= \frac{1}{\psi_C}\frac{\partial}{\partial x_k}\left(\psi_C\left[\sum_{i<k}\frac{\partial f_{ik}}{\partial x_k}+\sum_{i>k}\frac{\partial f_{ki}}{\partial x_k}\right]\right)\\
&= \left[\sum_{i<k}\frac{\partial^2 f_{ik}}{\partial x_k^2}+\sum_{i>k}\frac{\partial^2 f_{ki}}{\partial x_k^2}\right]+\frac{1}{\psi_C}\frac{\partial\psi_C}{\partial x_k}\left[\sum_{i<k}\frac{\partial f_{ik}}{\partial x_k}+\sum_{i>k}\frac{\partial f_{ki}}{\partial x_k}\right]\\
&= \sum_{i\neq k}\frac{\partial^2 f_{ik}}{\partial x_k^2}+\left[\sum_{i=1}^{k-1}\frac{\partial f_{ik}}{\partial x_k}-\sum_{i=k+1}^{N}\frac{\partial f_{ki}}{\partial x_i}\right]^2\\
&= \sum_{i\neq k}\frac{\partial}{\partial x_k}\left(\frac{\partial f_{ik}}{\partial r_{ik}}\frac{\partial r_{ik}}{\partial x_k}\right)+\left[\sum_{i\neq k}\frac{\partial r_{ik}}{\partial x_k}\frac{\partial f_{ik}}{\partial r_{ik}}\right]^2\\
&= \sum_{i\neq k}\left[\frac{\partial r_{ik}}{\partial x_k}\frac{\partial}{\partial x_k}\frac{\partial f_{ik}}{\partial r_{ik}}+\frac{\partial f_{ik}}{\partial r_{ik}}\frac{\partial^2 r_{ik}}{\partial x_k^2}\right]+\left[\sum_{i\neq k}\frac{\partial r_{ik}}{\partial x_k}f'_{ik}\right]\left[\sum_{j\neq k}\frac{\partial r_{jk}}{\partial x_k}f'_{jk}\right]\\
\left[\frac{\hat{\nabla}^2\psi_C}{\psi_C}\right]_x &= \sum_{i\neq k}\left[\left(\frac{\partial r_{ik}}{\partial x_k}\right)^2 f''_{ik}+f'_{ik}\frac{r_{ik}^2-(x_k-x_i)^2}{r_{ik}^3}\right]\\
&\quad+\sum_{j\neq k}\left[\frac{\partial r_{ik}}{\partial x_k}f'_{ik}\left(\frac{\partial r_{ik}}{\partial x_k}f'_{ik}+\sum_{j\neq k,i}\frac{\partial r_{jk}}{\partial x_k}f'_{jk}\right)\right]\\
\left[\frac{\hat{\nabla}^2\psi_C}{\psi_C}\right]_x &= \sum_{i\neq k}\left[\left(\frac{\partial r_{ik}}{\partial x_k}\right)^2 f''_{ik}+f'_{ik}\frac{r_{ik}^2-(x_k-x_i)^2}{r_{ik}^3}\right]\\
&\quad+\sum_{j\neq k}\left[\left(\frac{\partial r_{ik}}{\partial x_k}f'_{ik}\right)^2+\sum_{j\neq k,i}\frac{\partial r_{ik}}{\partial x_k}f'_{ik}\frac{\partial r_{jk}}{\partial x_k}f'_{jk}\right]\\
\left[\frac{\hat{\nabla}^2\psi_C}{\psi_C}\right]_x &= \sum_{i\neq k}\left[\left(\frac{x_k-x_i}{r_{ik}}\right)^2 f''_{ik}+f'_{ik}\frac{r_{ik}^2-(x_k-x_i)^2}{r_{ik}^3}\right]\\
&\quad+\sum_{j\neq k}\left[\left(\frac{x_k-x_i}{r_{ik}}f'_{ik}\right)^2+\sum_{j\neq k,i}\frac{(x_k-x_i)(x_k-x_j)}{r_{ik}r_{jk}}f'_{ik}f'_{jk}\right]\\
\left[\frac{\hat{\nabla}^2\psi_C}{\psi_C}\right]_x &= \sum_{i\neq k}\left[\left(\frac{x_k-x_i}{r_{ik}}\right)^2 f''_{ik}+f'_{ik}\frac{r_{ik}^2-(x_k-x_i)^2}{r_{ik}^3}\right]\\
&\quad+\sum_{i,j\neq k}\frac{(x_k-x_i)(x_k-x_j)}{r_{ik}r_{jk}}f'_{ik}f'_{jk}
\end{aligned}
$$

If we now sum up for all dimensions, we get

$$
\begin{aligned}
\frac{\hat{\nabla}^2\psi_C}{\psi_C} &= \sum_{i\neq k}\left[\frac{r_{ik}^2}{r_{ik}^2}f''_{ik}+f'_{ik}\frac{3r_{ik}^2-r_{ik}^2}{r_{ik}^3}\right]+\sum_{i,j\neq k}\frac{(\mathbf{r}_k-\mathbf{r}_i)(\mathbf{r}_k-\mathbf{r}_j)}{r_{ik}r_{jk}}f'_{ik}f'_{jk}\\
&= \sum_{i\neq k}\left[f''_{ik}+\frac{2}{r_{ik}}f'_{ik}\right]+\sum_{i,j\neq k}\frac{(\mathbf{r}_k-\mathbf{r}_i)(\mathbf{r}_k-\mathbf{r}_j)}{r_{ik}r_{jk}}f'_{ik}f'_{jk}
\end{aligned}
$$

Bibliography

[1] Jian-Sheng Wang A. D. Güçlü and Hong Guo. "Disordered quantum dots: A diffusion quantum Monte Carlo study." In: *Physical Review B* 68:035304 (2003). DOI: 10.1103/PhysRevB.68.035304. URL: https://doi.org/10.1103/PhysRevB.68.035304.

[2] R. L. Graham et al. "Open MPI: A High Performance, Flexible Implementation of MPI Point-to-Point Communications." In: *Parallel Process* Lett. 17, 79 (2007). DOI: 10.1142/S0129626407002880. URL: https://www.worldscientific.com/doi/abs/10.1142/S0129626407002880.

[3] O. F. de Alcantara Bonfim and David J. Griffiths. "Exact and approximate energy spectrum for the finite square well and related potentials." In: *Physics Faculty Publications and Presentations* 8 (2006).

[4] Cambridge Andrew James Williamson. Robinson College. "Quantum Monte Carlo Calculations of Electronic Excitations. PhD thesis." In: (1996). URL: http://www.tcm.phy.cam.ac.uk/~ajw29/thesis/thesis.html.

[5] David L. Aronstein and C. R. Stroud Jr. "General series solution for finite square-well energy levels for use in wave-packet studies." In: *American Journal of Physics* 68, 943 (2000). DOI: 10.1119/1.1285868. URL: http://aapt.scitation.org/doi/pdf/10.1119/1.1285868.

[6] F. Bolton. "Fixed-phase Quantum Monte-Carlo Method Applied to Interacting Electrons in a Quantum Dot." In: *Physical Review B* 54:4780 (1996). DOI: 10.1103/PhysRevB.54.4780. URL: https://doi.org/10.1103/%20PhysRevB.54.4780.

[7] W. T Coffey, Y. P Kalmykov, and J. T Waldron. *The Langevin Equation: With Applications to Stochastic Problems in Physics, Chemistry, and Electrical Engineering*. World Scientific, Singapore, 2004.

[8] C. J. Umrigar F. Pederiva and E. Lippiarini. "Diffusion Monte Carlo study of circular quantum dots." In: *Physica Review B* 62:8120 (2000). DOI: 10.1103/PhysRevB.62.8120. URL: https://doi.org/10.1103/PhysRevB.62.8120.

[9] Alexander Fleischer. *GitHub Repository: Quantum Monte Carlo*. GitHub, 2018. URL: http://www.github.com/alexanfl/QuantumMonteCarlo.

[10] H. Flyvbjerg and H. G. Petersen. “Error estimates on averages of correlated data.” In: *The Journal of Chemical Physics* 91:461 (1989). DOI: 10.1063/1.457480. URL: https://doi.org/10.1063/1.457480.

[11] C. W Gardiner. *Handbook of stochastic methods for physics, chemistry, and the natural sciences*. 3rd. Berlin: Springer-Verlag, 2004. ISBN: 3540208828 (acid-free paper). URL: http://www.loc.gov/catdir/enhancements/fy0818/2004043676-d.html;%20http://www.bibsonomy.org/bibtex/2982cb5cff63c31f2bc7d3f5880eb85e1/sidney.

[12] David Griffiths. *Introduction to Quantum Mechanics*. 2nd Edition. Pearson, 2005.

[13] B.L. Hammond, Jr. W. A. Lester, and P. J. Reynolds. *Monte Carlo Methods in Ab Initio Quantum Chemistry*. Ed. by S.H. Lin. World Scientific Publishing Co., 1994.

[14] C. Hirth. “Studies of quantum dots: Ab initio coupledcluster analysis using OpenCL and GPU programming. Master’s thesis, University of Oslo.” In: (2012). URL: http://urn.nb.no/URN:NBN:no-31657.

[15] Morten Hjorth-Jensen. “Computational Physics.” 2015.

[16] J. Høgberget. “Quantum Monte-Carlo Studies of Generalized Many-body Systems. Published Master’s thesis, University of Oslo.” In: (2013). URL: http://urn.nb.no/URN:NBN:no-38645.

[17] Jon Magne Leinaas. “Modern Quantum Mechanics.”

[18] N. Llopis and C. Nicholson. *GitHub Repository: UnitTest++*. GitHub, 2018. URL: https://github.com/unittest-cpp/unittest-cpp.

[19] M. Hjorth—Jensen et al M. L. Pedersen G. Hagen. “Ab initio computation of the energies of circular quantum dots.” In: *Physical Review B* 84:115302 (2011). DOI: 10.1103/PhysRevB.84.115302. URL: https://doi.org/10.1103/PhysRevB.84.115302.

[20] J. J. Mares et al. “Periodic reactions and quantum diffusion.” In: (). http://www.fzu.cz/ sestak/yyx/periodic%20reactions.pdf. URL: http://www.fzu.cz/~sestak/yyx/periodic%20reactions.pdf.

[21] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1st edition. Prentice Hall, 2008.

[22] R. C. Martin. “Professionalism and Test-Driven Development.” In: *IEEE Software* 24, 3 (May/June 2007). DOI: 10.1109/MS.2007.85. URL: http://ieeexplore.ieee.org/document/4163026/.

[23] Daniel Andres Nissenbaum. “The stochastic gradient approximation: an application to li nanoclusters.” In: (2008).

[24] V. K. B. Olsen. “Full Configuration Interaction Simulation of Quantum Dots. Master’s thesis, University of Oslo.” In: (2012). URL: http://urn.nb.no/URN:NBN:no-32952.

[25] S. Reimann. “Quantum-mechanical systems in traps and Similarity Renormalization Group theory. Master’s thesis, University of Oslo.” In: (2013). URL: http://urn.nb.no/URN:NBN:no38639.

[26] H. Risken and H. Haken. *The Fokker-Planck Equation: Methods of Solution and Applications Second Edition*. Springer, 1989.

[27] C.P. Robert and G. Casella. *Monte Carlo Statistical Methods*. Springer Verlag, 2004.

[28] S. Ruder. “An overview of gradient descent optimization algorithms.” In: (2016). URL: arXiv:1609.04747%20[cs.LG].

[29] J. J. Sakurai. *Modern Quantum Mechanics*. Revised ed. New York: Addison-Wesley, 1994.

[30] C. Sanderson and R. Curtin. *Armadillo: a template-based C++ library for linear algebra*. DOI: http://dx.doi.org/10.21105/joss.00026.

[31] Y. M. Wang. “Coupled-Cluster Studies of Double Quantum Dots. Published Master’s thesis, University of Oslo.” In: (2011). URL: http://urn.nb.no/URN:NBN:no-28527.